Mathematisch-
Naturwissenschaftliche Fakultät

Mario Frank

Axiom Relevance Decision Engine

Technical Report

Universität Potsdam

Mario Frank
Axiom Relevance Decision Engine

Mario Frank

Axiom Relevance Decision Engine

Technical Report

Universität Potsdam

Published online at the
Institutional Repository of the University of Potsdam:
URL http://opus.kobv.de/ubp/volltexte/2014/7212/
URN urn:nbn:de:kobv:517-opus-72128
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-72128

Axiom Relevance Decision Engine

Technical Report

Mario Frank

08.12.2012

Abstract

This document presents an axiom selection technique for classic first order theorem proving based on the relevance of axioms for the proof of a conjecture. It is based on unifiability of predicates and does not need statistical information like symbol frequency. The scope of the technique is the reduction of the set of axioms and the increase of the amount of provable conjectures in a given time. Since the technique generates a subset of the axiom set, it can be used as a preprocessor for automated theorem proving.

Contents

1 Introduction

Software and hardware are essential elements of the past, present and future in humans life and the more the time evolves, the more these elements become important. But while the complexity of hardware and software increases, the probability of errors increases, too. One concept to ensure the correctness of some properties of these elements is the concept of formal proof of decidable parts of their properties. This is where automated theorem provers (ATP) come into play. They try to prove some theorem (also called conjecture) which is a mathematical or logical model of a property by using some axioms or axiom-like formulas which are known to be true or assumed to be true. But even they have some theoretical and technical limits. Consider an intrusion detection system (IDS) which tries to recognise attack-attempts on a server. This system needs to be able to handle millions of events which occur in the network stack. Since most of the events are irrelevant for the decision, whether a query is an attack, the IDS must be able to filter the events.

This concept is an analogy to theorem provers with events being axiom-like formulas and the question "is this query an attack" being the conjecture. While the structure of events is quite simple, the structure of logical formulas can be complex. As a consequence, the proof of a logical formula is complex, too.

Theorem provers need to analyse all axioms and the conjecture and prove the conjecture by application of a logical calculus. The bigger the number of axioms becomes, the more inference steps are possible. This can lead to an explosion of time complexity and space complexity since memory is needed to store an axiom. When this happens, theorem provers reach their limits. Normally provable conjectures become unprovable in a specified time or with a given amount of memory.

In practice, not all axioms are needed to prove the conjecture. Thus, using all axioms is a waste of resources which needs to be avoided. This can be done by using only relevant axioms for the proof search. But for this we need to decide in acceptable time, which axioms are relevant. This process is commonly called relevance filtering and described in multiple documents like [Pud07], [SP07], [RPS09] and [PY03]. 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]. All these systems competed in the CADE Systems Competition (CASC J6) [Sut12] and were able to benefit from the filtering technique. leanCoP-ARDE was the only system in the Large Theorem Batch (LTB) division which used another filtering technique which is implemented by ARDE and used by the lean connection-based theorem prover **leanCoP**[Ott08]/[OB03].

This document addresses the filtering technique of ARDE and describes both theoretical aspects and implementation details and presents benchmarks with the current version of ARDE. Some information about the technique was already published in [Fra12].

2 Theoretical Aspects

To understand, what exactly ARDE does, we need some basic theoretical knowledge of classic first order logic, unification, connections and the normal form transformations. These concepts are described in-depth in [Bib92] and [BBJ07] but will also be introduced here. First, we will introduce classical first order logic (fol) and normal form transformations and after that, we will take a closer look at unification and connection.

2.1 First order logic and normal forms

All formulas in first order logic are defined by a syntax.

Definition 1 (Syntax of first order logic)**.** *A first order logic formula consists of sets of lower case named n-ary predicates (e.g. $p(x, Y)$), functions (e.g. $f(x)$) and constants (e.g. x) which are functions of zero arity and upper case variables (e.g. Z). These will also be called* ***simple terms****. Simple terms are chained by* ***logical connectives*** *like negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$) and equivalence ($\Leftrightarrow$). The logical connectives bind simple terms to* ***complex terms****.*

All variables may be free or bound by an existential quantifier ($\exists X : p(X)$) or an universal quantifier ($\forall X : p(X)$). They denote that there is a substitution for the variable X such that $p(X)$ holds and that $p(X)$ holds for all substitutions of X, respectively.

Surely, the arguments of functions may be simple terms, too. Let us take a look at the small set of first order formulas in Figure 1 conforming to the syntax definition.

$$
\begin{array}{|l|}
\hline
canFly(tweety) \\
\forall X : ((bird(X) \wedge \neg penguin(X)) \Rightarrow canFly(X)) \\
\exists X : (canFly(X) \Leftrightarrow (hasWings(X) \wedge bird(X))) \\
bird(tweety) \\
penguin(tux) \\
\neg penguin(tweety) \\
penguin(ralph) \\
\hline
\end{array}
$$

Figure 1: set of first order logic formulas

They say that Tweety is a bird and can fly and that Tux is a penguin. Furthermore, they say that all birds which are not penguins are able to fly. Also, there is (at least) one instance of X such that, if X is able to fly, then it is a bird with wings and vice versa.

These formulas can be transformed into different normal forms to simplify the expressions and make automated deduction easier. The most important normal forms are the negation normal form, the prenex normal form, the Skolem normal form and the clause normal form which will be introduced now.

The negation normal form is a form where all negations are located directly before the predicates and all logical connectives except the basic connectives (negation, disjunction and conjunction) are replaced by semantically equivalent expressions with the three basic ones.

The rules for semantic equivalence are (including De Morgan's laws):

- $A \Rightarrow B \equiv \neg A \vee B$
- $A \Leftrightarrow B \equiv (\neg A \vee B) \wedge (A \vee \neg B)$
- $\neg \forall A(F) \equiv \exists A(\neg F)$
- $\neg \exists A(F) \equiv \forall A(\neg F)$
- $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$
- $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$
- $\neg\neg A \equiv A$

where A and B denote some term. All predicates in the negation normal form of the formula get a polarity which is negative ("-") if the predicate is negated and positive ("+"), otherwise. The notion for polarities differs in different literature. For example, in [Smu95] T is used for positive polarity and F for negative.

After transforming a formula to negation normal form, it can be transformed into prenex normal form and Skolem normal form.

Definition 2 (rectified prenex normal form)**.** *A formula is in prenex normal form, iff it has the form $Q_1X_1...Q_nX_n(F)$ where $Q_i \in \{\exists, \forall\}$ and X_i being a variable and $i \in \mathbb{N}$. Furthermore, F is a term which does not contain any quantifiers and is called the* ***matrix*** *while the quantifier sequence is called* ***prefix****. A fol-formula is* ***rectified*** *if no two quantifiers bind the same variable and no variable occurs both bound and free in the matrix.*

Every fol-formula which is not in prenex normal form can be transformed into such one which was also proven in [BBJ07]. The transformation is done by shifting all quantifiers to the left end and replacing all variables which would become bound by them with a new variable.

Definition 3 (Skolem normal form)**.** *A closed fol-formula in rectified prenex normal form is in Skolem normal form iff it only contains universal quantifiers and no free variables.*

Free variables are bound by introduction of new universal quantifiers. As proven in [BEL01], every fol formula has a Skolem normal form which is satisfiability-equivalent ($\equiv_{sat}$) to the original formula.

The transformation is done by replacing every existentially quantified variable by a Skolem function which does not occur in the matrix. All universally quantified variables left to the existential quantified variable form the arguments of the new function. To illustrate this, let us consider the following two first order formulas in Figure 2.

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

Figure 2: Skolemization of fol-formulas

The skolemization of the first formula is done by replacing Y with sk_fun1 and defining X as its argument. The skolemization of the second formula is done by replacing X with sk_fun1. Since the second formula does not contain any universally quantified variables before the existential quantification, the Skolem function becomes a constant.

Most of the theorem provers use a clause normal form or the definitional normal form for the proof search. Thus, we will finally take a look at the clause normal form.

Definition 4 (Clause normal form)**.** *A fol-formula is in clause normal form, if it is a conjunction of disjunctions (****conjunctive normal form****) or a disjunction of conjunctions (****disjunctive normal form****).*

Every first order formula can be transformed into a clause normal form. While the naive transformation via distributivity axioms can yield exponential growth of the formula, the definitional transformation only yields linear growth in size of the original formula as noticed in [Ede92]. The definitional transformation is done by introducing a new name for every subformula of the original formula. By this, the formula is virtually flattened.

2.2 Unification and connections

Unification and the concept of connection are the core of ARDE's mechanism. Hence, these concepts need to be introduced.

Definition 5 (Substitution and unifiability)**.** *Let $\mathcal{V} = \{V_1, V_2, ..., V_n\}$ with $n \in \mathbb{N}$ be a set of variables and $\mathcal{T} = \{T_1, T_2, ..., T_n\}$ with $n \in \mathbb{N}$ a set containing functions and/or variables. The mapping $\sigma : \mathcal{V} \to \mathcal{T}$ 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 and $i \in \mathbb{N}$ being variables or functions, 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 the σ is called their unifier. σ is called the most general unifier (mgu), if there exists an unifier θ for every other unifier σ' such that $\sigma_b = \theta(\sigma_a)$. Precisely, every other unifier than the most general one is constructible by applying a substitution to the mgu. To prevent endless unification, the unification algorithm needs to implement an **occurs-check** which fails, if a function would substitute a variable while containing this variable itself.

The last instrument and most vital aspect of ARDE's concept which will be introduced is the connection. The decision whether a formula should be chosen is based on connections.

Definition 6 (Connectibility)**.** *A pair $\{p, \neg p\}$ of predicates which are unifiable is called a* ***connection****. In this document, a connection is seen as a conflict concerning the satisfiability of a formula.*

3 Concept

Many of the fundamental theoretical aspects introduced in the last section are not needed in depth but may improve the intuition of the filtering mechanism.

3.1 Basics

The concept involves some steps which will be described in the following order. First, we will take a loot at the initialization of the filtering mechanism. After that, the search engine and some of the properties of the engine will be described. Finally, some elaborated techniques, which do not alter the selection of the axioms, will be introduced. Axioms can be located in special files (axiom files) or in the file in which the conjecture is located, too (problem file).

Initialization To be able to select relevant axioms for a conjecture (or for a set of conjectures), all formulas need to be imported and analysed but are also saved in their original form. In especial, they at least need to be transformed into negation normal form, or at most into the Skolem normal form. While all axioms are considered to be true, the conjecture is considered to be false. Namely, the conjecture is negated and then transformed into normal form. While the transformation evaluates all negations by shifting them directly to the predicates, the Skolem normal form transformation eliminates existentially quantified variables. The resulting formula only contains universally quantified variables. Thus, the quantifiers may be omitted in the further processing. Conceptually, the mechanism sees every formula as a set of predicates which may be of positive or negative (negated) polarity. Hence, the form is not structure-preserving, which has some disadvantages. All predicates are aggregated into predicate classes which are sets of predicates sharing the name, polarity and arity (argument-count).

Search The search is the most important step and the implementation will be described in detail in the next section. Starting from the (negated) conjecture, connections are searched for every predicate of it (first iteration). Precisely, for every predicate of the conjecture, and for every predicate in the opposing predicate class (same name and arity but conflicting polarity) an unification is applied. If this unification succeeds, the unifier is disposed (the variable substitutions are discarded) and the axiom containing the now connected predicate is included into the axiom set if it was not already included. This is checked by comparing the id of the conjecture with an usage id i ($1 \leq i \leq |conjectures|$) of the axiom which is initially -1. The semantic of the usage id is that the axiom was connected in the $i-th$ conjecture. If the usage id is smaller than the id of the current conjecture, it is set to the new id. After processing the conjecture, every newly included axiom is processed in the same way (second iteration). Following this scheme, in every iteration, the formulas which were included in the last iteration are connected. After every iteration, the original form of the included axioms of every iteration and the conjecture are written to a file and the file is given to a theorem prover.

To make the search more intuitive, we will look again at the Tweety-Example in the last section. But this time, all formulas are transformed into Skolem normal form and the conjecture is negated, as it can be seen in Figure 3.

Conj:$\neg canFly(tweety)$
Ax1 :$\forall X : (\neg bird(X) \vee penguin(X) \vee canFly(X))$
Ax2 :$(\neg canFly(sk1) \vee (hasWings(sk1) \wedge bird(sk1)))$
$\wedge(\neg hasWings(sk1) \vee \neg bird(sk1) \vee canFly(sk))$
Ax3 :$bird(tweety)$
Ax4 :$penguin(tux)$
Ax5 :$\neg penguin(tweety)$
Ax6 :$penguin(ralph)$

Figure 3: set of first order logic formulas in Skolem normal form

In the second axiom, the variable was replaced by the new constant sk1.

When the search is done, a virtual connection graph is constructed which is shown in Figure 4. In the first iteration, the universally quantified axiom is selected via the connection

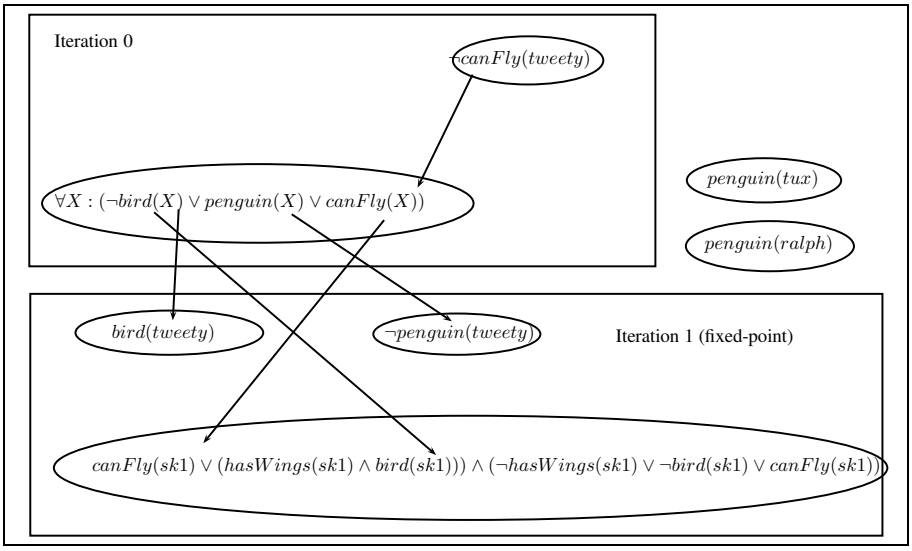

Figure 4: Visualization of the search

$\{\neg canFly(tweety), canFly(X)\}$. In the second iteration, multiple axioms are selected transitively. After the second iteration, no new axioms will be selected since no connections are possible to new axioms. Two of the axioms are completely irrelevant for the proof and will not be chosen. In fact, the skolemized axiom is not relevant for the proof and thus a false positive.

Elaborated techniques When using the same axioms for the proof of different conjectures, it makes sense to avoid doing the unifications multiple times. To understand this, consider two axioms which are connectible and relevant for the proof of more than one conjecture. The connectibility would be checked every time. Thus, we can try to cache these connections and reuse them which may save time in the search. There are two possible ways to cache connections:

1. Preconnection
2. Dynamic caching

Dynamic caching is done while searching the connections. If a connection was found, the fact that a predicate of one axiom is connectable to another predicate of another axiom (probably the same), is saved. Let $\mathbb{P}$ be the set of all Predicates and $p \in \mathbb{P}$ be a predicate. Then, a (partial) function $c_{cached} : \mathbb{P} \to \mathbb{P}^*$ "sais", that one predicate is connectible to a set of predicates. The function can yield three results:

$$c_{cached}(p) = \begin{cases} \{p_1, ..., p_n\}, \text{if connections were cached} \\ \{\}, \text{if no connections are possible} \\ \bot, \text{if not yet evaluated} \end{cases}$$

The set of cached connections for a predicate can be empty, if no connections were possible or non-empty, if connections were cached. If no connection attempt was started on the predicate, yet, the function is not defined for this predicate.

Preconnection can be seen as a brute-force caching which is applied after the import of all formulas. Connections are searched for every axiom in the set to any other axiom in the set. This is expensive in time and should not be used in competitions where time is limited. In fact, tests have shown that preconnection is not an adequate approach since the time consumption is enormous and took many minutes for only 55.000 axioms.

3.2 Properties

The concept has some theoretical properties which are described here. As stated, the theorem prover gets the original formulas as input or to be precise, the original formulas are written into the output of ARDE.

Idempotence and Injectivity The scope of the algorithm was to find relevant axioms for the proof of a conjecture. Furthermore, the algorithm should yield the same result, no matter how often it is used on the same set of formulas.

Sentence 1 (Idempotence and Injectivity)**.** *The function implemented by ARDE is idempotent and injective. I.e.,* $f_{ARDE}(f_{ARDE}(X)) = f_{ARDE}(X)$ *and* $X = Y \Rightarrow f_{ARDE}(X) = f_{ARDE}(Y)$ *holds where* X *and* Y *are sets of fol-formulas.*

Proof 1. *Let* X *be a set of first order logic formulas containing a conjecture* c *and a (possibly empty) set of axioms. If ARDE finds relevant axioms by connecting via unification, a subset of* X *is chosen. Thus,* $Y \subseteq X$ *is the output of ARDE. Let* $Z \subset X$ *be the set of Axioms which can be empty. Since ARDE outputs the relevant axioms and the conjecture,* $Y \setminus \{c\} \subseteq Z$ *holds. If*

Z is an empty set, only the conjecture is output since $Y \setminus \{c\}$ is empty, too. Applying ARDE again to the conjecture will yield the same result since ARDE cannot choose any axioms.

If Z is not empty, but ARDE cannot connect any axioms from Z, the output is the conjecture, again. If Z is not empty and ARDE can connect some axioms, the output is Y with $Y \setminus \{c\} \neq \emptyset$.

If f_{ARDE} was not idempotent, applying it on it's own output would yield another result. I.e. the output of ARDE would be $V \subset X$ with $V \neq Y$. But then, $|V| < |Y|$ would hold and there must be at least one axiom x which cannot be connected to the conjecture c or an axiom in V. Precisely, no predicate of x could be unifiable with a predicate in V. But then, it also could not be contained in Y since V is a subset of Y. This can only happen if the original formulas are changed. But since ARDE only searches connections on copies of the formulas and the unifier is discarded, changes on the original formulas are impossible.

The injectivity follows directly from the idempotence. If ARDE outputs a set of formulas X and is applied again to this set, then the set of formulas Y is created with $X = Y$. But this is a special case where all axioms in the set are relevant. Consider two sets of formulas X and Y with $X = Y$. Then, $f_{ARDE}(X) \subseteq X$ and $f_{ARDE}(Y) \subseteq Y$ are obvious. If $f_{ARDE}(X) = X_{ARDE} \neq Y_{ARDE} = f_{ARDE}(Y)$ was true, then $X_{ARDE} \setminus Y_{ARDE} \neq \emptyset$ or $Y_{ARDE} \setminus X_{ARDE} \neq \emptyset$ would follow and there would be at least one axiom which is not contained in both sets. Thus, this axiom could not be unifiable to any other axiom or the conjecture in one of the output sets. But then, $X \neq Y$ must be true which is a contradiction. □

Termination The most important property of a preprocessing algorithm is termination. Since the time for the theorem prover and ARDE is limited, the termination is obvious. But what if the time was unlimited? It needs to be shown that it is not possible to have infinite loops in the search.

Sentence 2 (termination)**.** *Let X be any finite set of first order formulas. Then, f_{ARDE} will terminate in finite time.*

The proof is not very complex and relies on the concept concerning usage and problem ids.

Proof 2. *Let c be a conjecture and A a non-empty, finite set of axioms. There are a few cases which can occur:*

1. $X = \emptyset$
2. $X = \{c\}$
3. $X = A$
4. $X = \{c\} \cup A$

The cases 1, 2 and 3 are trivial. If there is no conjecture or there are no axioms, ARDE cannot search initial connections, outputs the conjecture if there is one and terminates. If there is a conjecture and there are axioms, too, the search is done. Then, the conjecture has a problem id, which is it's index in the conjecture set (0) and every axiom has an usage id which is initially -1. If an axiom can be connected to the conjecture, it is taken, if it's usage id is less than the problem id and it's usage id is set to the problem id. After the conjecture was processed completely, the search is done for every connected axiom. Connections to other axioms are searched and all connectible axioms having a usage id of -1 are added and their usage id is set to 0. Thus, the set of axioms which have an usage id which is smaller than the problem id decreases after every successful connection with a new axiom. Finally, no new connections can be found since every

axiom which is connectible in the current iteration was already connected and, thus, has the usage id 0 or there are no more connections possible. Thus, the algorithm must terminate.

Now, let C be a finite set of conjectures which all have an problem id (index) $0 \leq i \leq |C|$. Since all conjectures are processed ordered, the search is first done with problem id 0, then with id 1 and so forth. Especially, according to the total ordering of the set of conjectures, the problem id is growing monotonously and thus, the usage id will always be less or equal to the current problem id. It should be easy to see, that every search terminates in finite time and thus, the algorithm must terminate in finite time, too. □

Correctness of caching This property concerns the caching algorithm. It was stated, that it does not alter the result of ARDE. Since this statement is not obviously true, it will be shown to be true.

Sentence 3 (Correctness of caching)**.** *Let c_1 and c_2 be conjectures and A a non-empty, finite set of axioms which must be used for both conjectures. If connections were cached for c_1, their use via (transitive) connectivity for c_2 cannot be wrong.*

Proof 3. *Let us take a look at what exactly is cached. In one search iteration for c_1, there is a connection between two axioms a_1 and a_2. In especial, there is a pair $\{p, \neg p\}$ such that $p \in a_1$ and $\neg p \in a_2$ and p and $\neg p$ are unifiable. Then, the predicate p contains the information, that it is connectable with $\neg p \in a_2$. The unifier (the variable substitution which leads to the equality of the two predicates) is discarded, as described in the last subsection. Thus, the original axioms and predicates are not altered. If the search algorithm which processes c_2 finds a connection to a_1, it will search all connectible axioms for a_1. Since a_1 was not altered by the unification which lead to the inclusion of it, the connection search will lead to the same set of connectible axioms as in the search for c_1 concerning axiom a_1. This set, in especial, contains the axiom a_2. Thus, using the cached connections for a_1 and the search for connections lead to the same set of connectible axioms for a_1.* □

This property can be used to alter the search strategy. When caching is applied, we can first check, if there are cached connections. If the function c_{cached} yields $\perp$, then connections are searched for the current predicate. If c_{cached} yields the empty set, we go on to the next predicate in the axiom or to the next axiom, if the predicate was the last one. If c_{cached} yields a non-empty set of predicates, we include all not yet included axioms which contain the predicates.

Weakness for equalities One problem which was not yet addressed, are equalities. Equalities can alter the structure of predicates in a disruptive manner. Consider an equality which states that two functions yield the same result but have a different name. For example, if we have an equality which sais, that $true = not(false)$, we can substitute the occurrences of $true$ by $not(false)$ and vice versa. This is a trivial example but should help to understand the problems with equalities. But if we search a connection between $\neg isBird(tweety, true)$ and $isBird(tweety, not(false))$, the unification will fail since the predicates cannot be made equal without applying the equality. This problem is not addressed by ARDE and can lead to **incomplete** axiom sets since some needed connections are not found.

Based on this concept, the implementation is described in the next section.

4 Implementation

In this section, the implementation of ARDE is described in detail. First, the initialization and the analysis of the formulas is covered. After that, the search and the caching of connections

is described. Finally the prover instantiation will be addressed. ARDE is implemented mostly in C++ (about 2500 lines of code) and Prolog (about 150 lines of code). The system is started by a bash script which parses the configuration file (batch file). The Prolog predicates are not interpreted but they are compiled into Assembler by GNU Prolog and linked with the C++ code. All parsing in the C++ part is done with Boost Spirit which is a parser generator. Linking the compiled Prolog predicates improves the performance significantly.

In this section, special names of objects defined for ARDE are written in CamelCase capitalization to make the differentiation between the objects and the file types clear.

4.1 Data formats

ARDE is able to read the TPTP syntax as described in [Sut09]. This syntax describes the form with which all formulas (including first order logic) need to comply. The TPTP contains about 20.000 Problems of which about 8.000 are first order logic problems. About 6800 of the first order problems are theorems and thus, true. A problem in TPTP is a file containing at least one conjecture and a (possibly empty) set of axioms which should be used for the proof of the conjectures. Every problem file may contain include directives which point to axiom files. These files contain further axioms. The TPTP contains more than 1.000.000 axioms which are divided into different axiom sets for different domains (e.g. software verification, common sense reasoning, etc.).

In the Large Theorem Batch division, problems are processed in batch. The configuration file for this divisions is a batch file which can contain multiple batches. Every batch consists of three sections. The first section (BatchConfiguration) contains information like the division and category tag, the wall clock time per problem (time limit), the wished output (proof/assurance/answer) and whether the problems should be processed ordered. The second section (BatchIncludes) contains include directives for axiom files which should be used in every problem. Those axiom files are called global in this document. The last section contains a the problem input files and output files. There is a global time limit for the complete batch file which is at least the sum of all problem time limits. In some cases, the global time limit is given but the time can be managed by the theorem prover since there is no problem time limit. In this case, the global time is equally distributed for the problems. If a problem is solves in the generated time limit, the remaining time is distributed equally on all remaining problems, again.

4.2 Import and analysis

The start script of ARDE is started with the (possibly absolute) path to the batch file and the optional global time limit. This script analyses the batch file and starts the ARDE binary with the information for each batch sequentially. Thus, all batches are processed in the order in which they occur in the batch file. The binary itself does the real work. First, all global axiom files are loaded and stored in AxiomFile objects which are marked as global.

The import itself is done in multiple steps. First, an axiom object is created for every first order formula in the axiom file. This object initially contains the original formula string and the usage id is set to -1. After this, the formula string is analysed and transformed by a Prolog term. Precisely, ARDE starts the Prolog term with a copy of the formula string and the term returns the list of predicates of the formula in Skolem normal form. The normal form transformation is done in two steps. First, the formula is rectified which means that all variables are renamed to make sure that no variable is bound by more then one quantifier. In this step, all implications and equivalences are transformed to combinations of disjunctions, conjunctions and negations. In the next step, the rectified formula is transformed into negation normal form and at the same

time in Skolem normal form. For every existentially quantified variable, a Skolem function is introduced.

The output of the Prolog term is parsed by ARDE and all predicates are inserted as predicate objects into the axiom object. Figure 5 shows the syntax of the output of the Prolog term.

$$
\begin{array}{l}
Axiom :== "[", Predicate^{+}, "]" \\
Predicate :== "[", Name, ",", Arity, ",", Polarity, ArgumentList, "]" \\
Name :== [a-z], [a-zA-Z0-9"\_""-"]^{*} \\
Arity :== [0-9]^{+} \\
Polarity :== 0|1 \\
ArgumentList :== (",", Argument, ArgumentList)|\epsilon \\
Argument :== Constant|Variable|Function \\
Constant :== Name \\
Variable :== [A-Z], [a-zA-Z0-9"\_""-"]^{*} \\
Function :== "'", Name, "(", ArgumentList, ")", "'"
\end{array}
$$

Figure 5: Axiom syntax in BNF-like syntax

In this syntax, all non-quoted colons have the semantic "followed by". The polarity 0 is positive ("+") and 1 is negative ("-") and the arity is the number of arguments of the predicate. While parsing the output of the Prolog term, ARDE instantiates all predicate objects and sets their arities, the polarities, the arguments and the argument types (constant, variable or function). At the same time, every predicate object gets it qualified name which is the concatenation of the polarity, the name and the arity. A negative predicate with an arity of 2 would get the qualified name "-name/2" and the same predicate with positive polarity would get the qualified name "+name/2".

After assigning the qualified name, the reference to the predicate object is added to it's predicate class which is the set of all predicate objects sharing the same qualified name. All predicate classes are held persistent in a hash table whose keys are the qualified names and the values are sets of pointers (references) to the predicate objects which are part of the predicate class. This hash table, the set of all predicate objects and a set of all axiom objects are saved in the AxiomFile object.

After importing the global axiom files, ARDE iteratively processes every problem conforming to the ordering in the batch file. The currently processed problem file needs to be analysed, too and the process is almost the same as for axiom files. But since the problem file contains a set of conjectures and include directives to axiom files, there are some special cases to cover. First of all, all axioms are imported exactly as they are for axiom files. The predicate class table is saved in a ProblemFile object and so are the set of predicate objects and the set of axiom objects. The conjectures are analysed and transformed into Skolem normal form, too. But the transformation is done with the opposite start polarity (1).

This implicitly simulates a negation of the conjectures. The conjecture objects are axiom objects which are held persistent in a special set in order to not mix up the structures. Furthermore, all include directives are read and all referenced axiom files which were not already imported as global files are imported and tagged as local files. The pointers to the local AxiomFile objects for the problem and the global AxiomFile objects are saved in the ProblemFile object. Finally, the ProblemFile object gets it's problem id which is the index of the underlying problem file in the batch file.

To sum things up, the schematic structure of the ProblemFile and AxiomFile objects is shown in Figure 6.

Both the AxiomFile and ProblemFile objects contain a predicate vector with the original

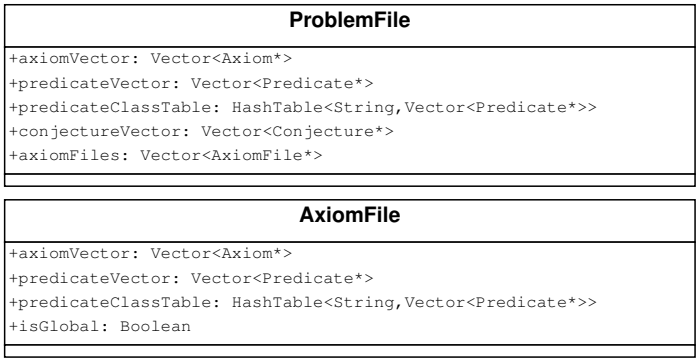

Figure 6: AxiomFile and ProblemFile object structure

predicate objects, the axiom vector with the original axiom objects and a predicate class table which holds references to the elements of the predicate vector. Also, the axiom objects only contain references to the elements in the predicate vector. By this, a level of structure sharing and thus less memory consumption is assured. In practice, ARDE assumes that a problem file contains only one conjecture. The support for multiple conjectures is a subject for further extensions.

4.3 Search

The search itself is done by the DecisionEngine object which is instantiated for every ProblemFile. The DecisionEngine works iteratively. In the first iteration, connections for the conjecture are searched. Precisely, for every predicate in the conjecture, connections are searched.

Basics The Figure 7 shows a schematic algorithm which is implemented by the DecisionEngine. The frontier vector is a vector of vectors and a model of the iterative approach

```
01 frontierVector.add(new Vector<Axiom>)
02 frontierVector.last.add(conjecture)
03 int iteration = 1
04 while (iteration ≤ 5 and !frontierVector.last.isEmpty())
05    frontierVector.add(new Vector<Axiom>)
06    for (Axiom formula : frontierVector.at(iteration - 1))
07       for (Predicate p : formula)
08          for (Predicate q : pClassTable.getOpposing(p))
09             if (p.unifiesWith(q))
10                if (q.getAxiom().getUsageId() < problemId)
11                   frontierVector.last.add(q.getAxiom())
12                   q.getAxiom().setUsageId(problemId)
13    new LeanCoPExecutor(frontierVector,iteration)
14    iteration = iteration + 1
```

Figure 7: Enlarge procedure

of ARDE. The first vector in the frontier vector contains the conjecture which is shown in lines one and two. In line four, the iteration is started with the iteration index 1. The search will end if the iteration index is bigger than 5 or when the last vector in the frontier vector is empty which can only be if no new connections could be found in the last iteration (fixed-point). The

iteration constraint was introduced to make sure that ARDE only starts a maximum count of prover instances. At the beginning of every iteration, a new vector, in which the new references to axiom objects are included, is added to the frontier vector. After that, for every predicate object of every formula in the last vector $(i-1)$ the opposing predicate class is searched (lines 06-08). Since the search is done as a lookup of a string (the opposing qualified name) in a hash table, the time complexity is constant. Then, the current predicate is unified with every predicate in the opposing class. If the unification succeeds, the axiom object which contains the unifiable predicate, is added to the new vector if it's usage id is smaller than the current problem id (lines 09-12). At the end of the iteration, a leanCoP instance is started with the conjecture and the frontier vector and the iteration index is increased (lines 13,14). ARDE tries to forward at most 15000 axioms to the prover. In some cases, some axioms are discarded to make sure that this threshold is not exceeded which may lead to incomplete axiom sets.

Details In order to make the scheme simple, some details were left out. These are the dynamic caching, the prover instantiation and the unification details which will be addressed now. The leanCoP (or any other prover) instance is, precisely said, a thread which runs parallel to the next iterations of the DecisionEngine and to all other already started prover instances. The input for the LeanCoPExecutor which encapsulates leanCoP are the frontier vector and the iteration in which the executor was started. Since the following iterations do not alter any vector than the last in the frontier vector, the axiom set which is used by the prover will not be altered. This approach makes parallel proof attempts possible. But the proof attempts need to be synchronized, too. if one instance of the prover finds a proof, the search should be terminated. Thus, if one thread finds a solution it locks a mutex (which blocks the access to the DecisionEngine for all other threads) and checks, whether a proof was already found (denoted by a boolean variable in the DecisionEngine). If not, it sets the variable to true, writes the proof into an output file and unlocks the mutex. This makes sure that if two instances of the prover find a proof at the same time, only one will output the proof. On the other hand, the DecisionEngine checks after every attempt to connect a predicate whether the boolean variable is true. In this case, it terminates the search and starts the axiom search for the next conjecture. The unification is done in two steps which are the signature-compatibility-check and the Prolog unification. The signature-compatibility-check compares the type of the predicate-argument pairs and in case of a constant-constant or function-function combination also the names of the constants or functions. The check fails with the result of non-unifiability in the following cases:

- constant-function
- function-constant
- constant1-constant2 and constant1≠constant2
- function1-function2 and functionName1≠functionName2

If this check is successful, copies of the two predicates (strings) are given as Prolog term objects (PlTerm) to an unification Prolog term which returns true or false and discards the unifier.

To make the dynamic caching approach clear we need some information which is not described in the algorithm scheme. In fact, the enlarge procedure is split in two parts. In every iteration, two searches are done. The first is the search in the axiom object set which is located in the ProblemFile object. The second is the connection search in all axiom object sets of all AxiomFile objects which are referenced in the ProblemFile object. But caching does not make sense for the ProblemFile since the concrete axiom objects in this file will not be used a second time. Also, caching does not make sense in the first iteration because these connections are

connections starting from a conjecture which will not be used a second time. Thus, the caching starts in the second iteration and only connections to axiom objects in the AxiomFile objects are cached. To save the connections, a few information are needed. Consider an predicate which can be connected to another predicate located in some AxiomFile. The fact, that predicate p is connectible with predicate q in the AxiomFile a is saved. Precisely, the predicate p contains a hash table (ConnectionTable) with the basename (the name of the file without directory prefixes) of the axiom file as key and a set of references of connectible predicates in the AxiomFile. As described in the concept section the set is empty if no connections are possible and the hash table entry for the AxiomFile is not existent if no connections were searched yet. Thus, in the second part of the iteration, first, a lookup in the ConnectionTable is done which has constant time complexity. If there is a set of connectible predicates, all not already included axioms containing those predicates are included and the next predicate is connected. If the set is empty, the next predicate is connected. If there is no entry for the AxiomFile, the connection-search is done.

5 Tests

This section describes all tests with ARDE, leanCoP and E. First, the test environment is described, followed by the latencies of ARDE. After that, the competition mode tests and the full benchmarks are reviewed.

5.1 Setup

All tests were done on a compute node at the University of Potsdam. The node had the following hardware:

- two Intel(R) Xeon(R) CPU X5355 with four cores, each (2,66 GHz, no HT)
- 16 GB of RAM (DDR2 , 667 MHz)
- Ubuntu 12.04 LTS 64 bit with Kernel 3.2.0-30
- SWI-Prolog 5.10.4

The ARDE version was 0.5.6 which is already able to cache connections, uses skolemization and had some major bug-fixes and speed optimizations. It was compiled statically with GNU Prolog 1.4.1 and G++ (GCC) 4.6.3 and used the Boost library 1.42. The optimization level for the GCC was O3. leanCoP 2.2 was used with the strategy selection as it was used in the CASC. E was used in version 1.6 (CASC version) and compiled with GCC 4.6.3, too.

5.2 Latency

One important property of ARDE is it's latency. The latency is the time interval which begins with the start of a conjecture (loading the problem file) and ends with the output of a depth file. The minimal and maximal latencies are rounded up since ARDE outputs the seconds as integers. Thus, the latencies can be seen in a "up to n seconds" way. All latencies are inclusive which means, that the latency for depth i includes the latency for depth $i-1$. To calculate the latency, leanCoP-ARDE (leanCoP 2.2 and ARDE 0.5.6) was started with the batch files from the CASC. Then, the minimal latency, the maximal latency and the average latency for the output of one depth was calculated. The average latency is the latency of all problems for one depth divided by the count of output files for the depth. The Table 1 shows the latencies for the Isabelle category which contains between 50 and 6000 Axioms and 75 problems.

depth	file-count	min	max	avg
0	62	1	3	1,62
1	59	1	32	4,27
2	52	1	44	11,88
3	46	1	44	12,82
4	13	1	36	7,38
5	7	2	5	3,00

Table 1: Latencies for the ISA category of the LTB division (60 sec. per Problem)

As the Table 1 shows, every depth can be created nearly instantly (in two seconds) and the average latency is always under 15 seconds. In some special cases, the latency is quite high (44 seconds). There are 13 Problems for which no axiom set can be generated due to inconnectibility caused by equalities or conjectures which say about themselves that they are false.

The next Table (2) shows the latency results for the Mizar category which contains 80 problems (20 per batch) and uses between 30 and about 70.000 axioms. While the first batch only uses the problem files, the other batches include axiom files. The last batch contains problems which mostly use between 30.000 and about 70.000 axioms and include nearly 1.000 axiom files. The wall clock time per problem was 30 seconds in the first batch, 60 seconds in the second, 90 in the third and 120 in the last.

batch	depth	file-count	min	max	avg
1	0	20	1	1	1,00
1	1	20	1	2	1,05
1	2	16	1	2	1,06
1	3	3	1	1	1,00
2	0	20	1	6	1,70
2	1	18	2	22	5,00
2	2	17	4	17	7,76
2	3	1	7	7	7,00
3	0	20	1	23	2.35
3	1	16	3	68	17.50
3	2	7	12	47	27.00
3	3	1	20	20	20.00
4	0	19	1	3	1.52
4	1	6	2	113	25.16

Table 2: Latencies for the MZR category of the LTB division

The latency of ARDE is minimal for the first batch and less then 10 percent of the time limit. In the second batch, the average latency is still less than 10 seconds. The last two batches show a growing latency but still less than 30 percent in average and many problems reach at least two depths. In the last batch, one problem cannot be addressed because it is the first and loads about 600 Axiom files which takes much time. Loading so many files could have been avoided if ARDE would manage all batches on it's own. The use of the shell script has the drawback that all loaded axiom files in the previous batches are lost. The calculation of the first depth needs too much time and ARDE gives up on this problem.

The last batch in the Large Theorem Batch division was the SUMO category which contains about 55.000 axioms. Every problem could be addressed for 60 seconds and every problem used

all 55.000 axioms which were located in one axiom file.

depth	file-count	min	max	avg
0	20	1	1	1.00
1	16	1	10	4.12

Table 3: Latencies for the SMO category of the LTB division (60 sec. per Problem)

As shown in the Table 3, an axiom set can be selected for every problem and the first depth comes in up to one second. A second depth can be generated for most of the problems in up to ten seconds.

The Mizar batch of the Turing competition (MRT, MIZAR@Turing) contains 400 problems with up to 5000 axioms. There is no wall clock time per problem but a global clock time (16.000 seconds) for the complete batch.

depth	file-count	min	max	avg
0	359	1	4	1.06
1	358	1	21	4.14
2	326	1	24	7.99
3	26	1	24	11.84

Table 4: Latencies for MRT, the MZR category of the Turing division (16000 sec. for the batch file)

An axiom set can be generated for 359 of the problems, as it can be seen in Table 4. The other problems cannot be addressed due to "false" conjectures and equalities. The maximum latency was 24 seconds but the average was 12 seconds.

These results imply that the algorithm is fast enough to be usable for preprocessing. To verify this, we still need to see how many problems become provable due to the filtering.

5.3 Provable problems

To determine, how many problems which a theorem prover cannot solve on it's own become provable, multiple benchmarks are required. First of all, the theorem prover must be tested on the original problems. After that, ARDE is started with the original problems and generates the restricted axiom sets and writes every search depth and the conjecture to files. Finally, the theorem prover is started with the now generated problem files. E was started with the option $--satauto$ to make sure that no relevance filtering with the implementation of the SinE-concept is done. The Table 5 shows the results of these tests for the theorem provers leanCoP (version 2.2) and E (Version 1.6).

The first block contains vital information about the categories like the problem count, the minimum and maximum count of axioms in this category, and the time limit per problem. The second block shows the count of provable conjectures for leanCoP 2.2 and E 1.6 (without SinE) on the original files with the time limit given in the second line as CPU time limit. E had the complete time limit for every problem in the LTB and 120 seconds per problem for the MRT problems. The reason for giving E more time on MRT is that spare time which arises due to fast solution of a problem is given uniformly distributed as additional time for the later problems. In the tests, the time for a problem increased to a maximum of 120 seconds. Thus, giving 120 seconds should show best how many problems are potentially solvable. Since leanCoP has different strategies to find a proof and uses multiple of them in a proof attempt, 19 strategies were chosen and every strategy had a time limit of 30 seconds for every problem. Thus, leanCoP had more time than E. But it was not the scope to show which prover would be better. The

	ISA	SMO	MZR1	MZR2	MZR3	MZR4	MRT
problem #	75	20	20	20	20	20	400
min. axiom #	518	55568	47	2951	10008	29085	27
max. axiom #	5249	55619	196	10434	19100	71136	4557
Time (sec.)	60	60	30	60	90	120	40 avg.
Proofs on original files							
E	44	0	12	0	0	0	108
leanCoP	19	0	2	0	0	0	223
Additional proofs							
E	1	6	1	3	3	1 (2)	32
leanCoP	4	5	0	0	1	1 (2)	16
In competition (CASC)							
EP-LTB	47	15	11	2	6	6	165
leanCoP-ARDE	17	2	2	0	0	0	7
In competition mode with ARDE 0.5.6							
E-ARDE	40/15	5/5	11/6	2/2	2/2	1/1	129/92
leanCoP-ARDE	18/13	4/4	2/1	0	0	1/1	116/38
E-ARDE-SinE	38/14	5/4	9/5	3/2	7/3	4/1	132/96

Table 5: Provable problems

goal was to find out how many proofs can be found, potentially. This first benchmark was used as a reference measure for the next block and the possible proof count can be seen as optimum. The next block shows the count of additional proofs which are possible due to the output from ARDE. Again, E had the full CPU time limit for each output file of ARDE (every depth of every problem) (and 120 seconds in the MRT category) and leanCoP was tested with each strategy with a 30 seconds CPU limit on every output of ARDE. The latency of ARDE was not considered. As the block shows, both provers are able to find more solutions and they also find additional solutions for SUMO problems and higher MZR problems. In the fourth batch of the MZR division, both provers could find two solutions but since one of the problems (the first) had a time out in ARDE due to many axiom files (500) which were needed for the problem only one solution is accepted. In fact the time out occurred because ARDE currently does not store axiom files from previous batches which will be fixed in upcoming releases.

The fourth block shows the count of proven conjectures in the CASC which was held in Manchester. E used the SinE concept in this competition and leanCoP-ARDE was the assembly of leanCoP 2.2 and ARDE 0.5.0 (which had many problems). In general, the current version of ARDE leads to a significant improvement for leanCoP in all divisions of the LTB category. For E on the other hand, ARDE can not yet compete with the SinE-selection as it is done in EP-LTB. This system was multi-threaded, too and thus could use different SinE heuristics to maximize the probability of a proof. In the CASC, no CPU limit was given. The given limits were real time which may involve loss of CPU time due to scheduling done by the operating system.

To compare the old and new ARDE version in competition like circumstances, leanCoP-ARDE, E-ARDE and E-ARDE-SinE were tested on the original batches and the time limits of the competition. E-ARDE was an assembly where E was started internally by ARDE with the option $--$ *satauto* on every depth and also on the original problem file. leanCoP also was additionally started on the original problem file. By this, the maximal count of provable problems should be determined. E-ARDE-SinE was the same system as E-ARDE but E was started with the option $--$ *auto* which allows E to use SinE-filtering. The results are shown in the last block for leanCoP-ARDE, E-ARDE and E-ARDE-SinE with the first number in a

cell being the count of proven problems and the second the count of proven problems by use of the output of ARDE. As it can be seen, the optimal proof count cannot be reached which is partially caused by using real time limits in competition mode. Also, in the output of ARDE, the formulas are ordered as the frontier vector is. I.e., every formula of iteration i is located before every formula of iteration $i+1$ and the formulas of iteration i may be randomly ordered due to the use of hash tables which contain the elements unordered. An inconvenient ordering of the formulas can lead to more needed time for the proof and in consequence to a time-out for the problem.

The assumption, enabling SinE-filtering would increase the count of provable problems significantly was not confirmed as it can be seen in the results for E-ARDE-SinE. In some cases, less problems are provable. Comparing the concept of EP-LTB and E-ARDE-SinE should make this clear. While E-ARDE-SinE starts a thread of E for every generated depth (iteration) and the original problem file and E has the possibility to use SinE-filtering with one chosen heuristic per thread, EP-LTB starts multiple threads of E with the original problem file and thus may use multiple heuristics or seeds in parallel. In fact, using the SinE-filtering on the output of ARDE may lead to incomplete axiom sets and make normally provable problems unprovable. Nevertheless, the last block shows that the use of the outputs of ARDE leads to faster proofs in many cases.

To complete the comparisons, it was determined which of the problems which are provable for E and leanCoP via ARDE were not provable (for E without/with SinE) before and how many axioms were selected.

The following Table 6 shows a subset of the problems which are normally not solvable by leanCoP and E(without SinE) and which become solvable. Furthermore it shows which prover now can solve these problems and how many axioms are selected by ARDE. The problems are only those of "hard" categories (SMO, MZR+2/3/4). In many cases less than 10 % are sufficient

problem	**depth**	**axioms**	**E**	**leanCoP**
SMO086+6	1	2150/55587	yes	no
SMO089+7	0	211/55591	yes	yes
SMO089+7	0	107/55591	yes	yes
SMO104+7	0	15/55591	yes	yes
SMO106+7	1	10001/55591	yes	yes
SMO108+7	0	4023/55591	yes	yes
SEU435+4	0	2956/71119	yes	yes

Table 6: Problems which become provable with ARDE

and in some cases even less then one per cent is selected.

There are some problems which are not even provable for E with SinE and also were not proven in the CASC. The Table 7 shows the problems which become provable by use of ARDE for E and in which minimum depth they could be proven. These problems could not be proven by EP-LTB in CASC. The problem CAT032 was only proven by Vampire in CASC and now

problem	**depth**	**axioms**
CAT032+3	0	625/11813
LAT304+2	0	431/2951

Table 7: Problems which are provable via ARDE but not with SinE for E

can be proven by E and leanCoP.

6 Related Work

Relevance filtering is no really new topic. In fact, the need to select relevant axioms (sometimes called premise or "sine qua non" - essential condition) is many years old. In the last years, some approaches were realized with mostly different goals.

The most comparable concept was described in [PY03]. This concept also uses unifications and searches for connections. Based on the connections, a relevance value (relevance distance) is defined for the axioms. Precisely, connections are searched for clauses which are sets of disjunctively bounded literals (e.g. predicates). If two clauses resolve (as in the resolution calculus), they are connected. If, for example two clauses are connected via another connection transitively, the relevance distance would be two. It should be clear that this concept relies on clause normal forms and is somehow tailored for resolution-like approaches. The ARDE-concept in contrast does not need clause normal forms and is not specialized for some calculus.

The following approaches rely on the existence of shared function or predicate symbols (names) in two axioms and thus are somehow probabilistic. The concept described in [RPS09] 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 axioms are reordered, the system Divvy selects a subset (initially the upper half and on failure varying percentages) of the axioms and tries to prove the problem. Thus, the concept initially halves the axiom set which, in large theorem batches would not be very helpful when taking 25.000 axioms from 50.000. Without further filtering, many theorem provers still could not handle the amount of axioms. The variation of the percentage increases the probability that the set contains the needed axioms and is not too big.

The approach described in [HV11] and implemented by SinE also uses shared symbols for the selection. But the filtering is somehow finer. First, the count of axioms in which a function symbol (or constant) or predicate symbol occurs, is determined. The more often a symbol occurs, the more probable it is that it is a common symbol like the predicate symbol "is_instance". This symbols should be penalized since they probably will not be helpful for filtering. If a symbol is rare but it is contained in the conjecture it should be favoured which is called a trigger. On this information, a relevance value is determined for the axioms and the most relevant are chosen. The approach had very good results in the last years and thus, is currently used in many theorem provers. On the contrary, the concept needs to select different seeds (symbols) to trigger the selections and can lead to incomplete axiom sets. ARDE, on the other hand would always create complete axiom sets in respect to provability if no equalities are contained.

The approaches described in [Pud07] and [SP07] (SRASS) use a semantic selection approach. While SRASS uses the relevance based ordering described in [RPS09] and computes models for the formulas (axioms and the negated conjecture), the concept of Petr Pudlák directly creates interpretations for the formulas by use of a model finder. Both approaches use a clause normal form.

The premise selection algorithm (PSA) which is used by the Naproche system [CKKS10] yet has some other approach. With the information about a proof for a conjecture given by a theorem prover which includes the steps needed for the proof, the PSA infers which axioms are likely to be used again in a later proof. Internally, it creates a connection graph which contains the axioms which were used in the proof as nodes. This graph contains the information which distance lies between the conjecture and some arbitrary axiom. Another used filtering method is a reimplementation of the Divvy concept which . Thus, this system can reuse informations from former proofs but it also needs them for the selection if no frequency bases filtering like Divvy is used.

7 Conclusion

Following the results of the CASC and the tests, the filtering technique implemented in ARDE can be seen as working. A restriction of the used axioms is done and more problems become solvable. Also, many problems are provable faster despite the latency introduced by ARDE. There are even some problems which can only be proven by E and leanCoP with ARDE but not with SinE. Not to forget, the preprocessing is usable for arbitrary theorem provers with any calculus and does not need clause normal form transformations. On the other hand, the restriction is sometimes not strong enough due to Weak Unification (unification without really substituting the variables) and loss of the formula structure. Also, the lack of equality handling can lead to an incomplete axiom set with respect to provability. The use of the shell script makes it impossible to easily reuse information from one batch like imported axiom files when batch files contain multiple batches. However, since the implementation is a prototype, there are many optimizations possible. Further development will include the refinement of the concept concerning equalities and use of the formula structure. The structure sharing is currently quite weak which needs to be improved. Also, the concept can be enhanced to be suitable for first order modal logic and intuitionistic logic including the use of prefix unification for modal logic. Furthermore, making ARDE easy to use for other theorem provers by introduction of a configuration file or some command line options would be reasonable.

References

[BBJ07] G.S. Boolos, J.P. Burgess, and R.C. Jeffrey. *Computability and Logic*. Cambridge University Press, 2007.

[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.

[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.

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

[Fra12] 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).

[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.

[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.

[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.

[Pud07] Petr Pudlák. 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.

[PY03] David A. Plaisted and Adnan H. Yahya. A relevance restriction strategy for automated deduction. *Artif. Intell.*, 144(1-2):59–93, 2003.

[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.

[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.

[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.

[Sut12] Geoff Sutcliffe, editor. *CASC-J6 Proceedings*, volume 11 of *EPiC Series*. EasyChair, 2012.

[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.