EasyChair Preprint

№ 126

Automated Theorem Proving by Translation to Description Logic

Negin Arhami and Geoff Sutcliffe

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

May 10, 2018

Automated Theorem Proving by Translation to Description Logic

Negin Arhami and Geoff Sutcliffe

University of Miami, Miami, Florida, U.S.A.
n.arhami@umiami.edu, geoff@cs.miami.edu

Abstract

Many Automated Theorem Proving (ATP) systems for different logics, and translators for translating different logics from one to another, have been developed and are now available. Some logics are more expressive than others, and it is easier to express problems in those logics. However, their ATP systems are relatively newer, and need more development and testing in order to solve more problems in a reasonable time. To benefit from the available tools to solve more problems in more expressive logics, different ATP systems and translators can be combined. Problems in logics more expressive than CNF can be translated directly to CNF, or indirectly by translation via intermediate logics. Description Logic (DL) sits between CNF and propositional logic. `Saffron` a CNF to DL translator, has been developed, which fills the gap between CNF and DL. ATP by translation to DL is now an alternative way of solving problems expressed in logics more expressive than DL, by combining necessary translators from those logics to CNF, `Saffron`, and a DL ATP system.

1 Introduction

A logic problem can be solved by either checking whether the conjecture of a logic problem is a theorem, or using proof by contradiction. Proof by contradiction checks if the set consisting of the axioms and the negated conjecture is unsatisfiable. *Automated Theorem Proving* (ATP) [9] is concerned with development of automatic techniques and computer programs that solve logic problems. An *ATP system* is a computer program that solves logic problems. Many logics have ATP systems available. Examples include `Konclude` [14] for Description Logic (DL), `iProver` [8] for Effectively Propositional Form (EPR), and `Alt-Ergo` [3] for Typed First order form-polymorphic (TF1).

Different logics with different levels of expressivity are available. Expressing a problem in a more expressive logic is easier because it is more natural and compact, but the ATP systems for more expressive logics are relatively newer, and need more development and testing to be able to solve more problems in a reasonable time.

It is possible to translate a problem in one logic to another logic. The translation from a source logic to a destination logic is *satisfiability preserving* means if a logic problem is satisfiable in the source logic, then the translated problem in the destination logic is satisfiable. The translation is *countersatisfiability preserving*[1] means if the conjecture of a problem is not a logical consequence of its axioms in the source logic, then the translated conjecture is not a logical consequence of the translated axioms in the destination logic. The translation is *countersatisfiability-satisfiability* preserving means if the conjecture of a problem is not a logical consequence of its axiom in the source logic, then the translated negated conjecture and axioms are satisfiable in the destination logic. A sound translation is satisfiability, countersatisfiability, and countersatisfiability-satisfiability preserving. Many sound translators for translating a

[1] The SZS ontology [16] supplies the definitions of status values such as "theorem", and "countersatisfiable".

problem in one logic to another have been implemented [18]. Examples of available translators are `ECNF` [11], which translates from First Order Form (FOF) to Conjunctive Normal Form (CNF), and `Why3-FOF` [4], which translates from TF1 to FOF.

As illustrated in Figure 1, a problem expressed in a logic can be either solved using an ATP system for that logic, or translated to a less expressive logic. If it is translated to a less expressive logic, again the same two options of solving by using an ATP system, and translating down (if possible) are available. This continues until no further translation is possible. Table 1 lists the translators and ATP systems used in this research, corresponding to the labels on the arrows in Figure 1. In Figure 1, the plain arrows indicate the process of translation, and the dashed arrows indicate the process of solving using an ATP system.

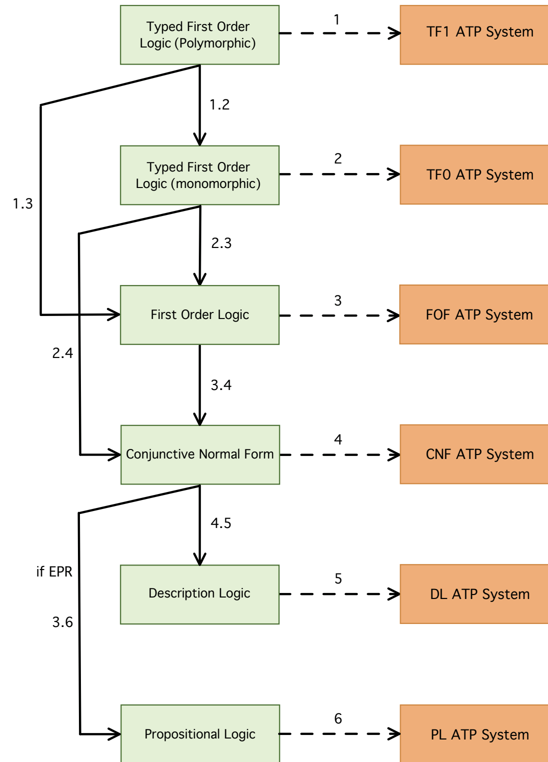

Figure 1: Different Combinations of Translators and ATP Systems to Solve a Problem

Problems in logics more expressive than CNF can be translated directly to CNF, or indirectly by translation via intermediate logics. No CNF to DL translator was available when this research was started. A sound translation from CNF to DL, and its implementation as `Saffron`, are provided in this research. `Saffron` was implemented in Prolog. `Saffron` fills the gap between CNF and DL. The translated DL problem can be solved using a DL ATP system. In this research, `Konclude`, a powerful DL ATP system, was used. `SaKo` is a new CNF ATP system which combines `Saffron` and `Konclude`. `SaKo` is now available on the `SystemOnTPTP` [15] website, `www.tptp.org/cgi-bin/SystemOnTPTP`.

The *Thousands of Problems for Theorem Provers* (TPTP) problem library [17] is a comprehensive library of test problems for ATP systems. It also includes software tools that facilitate using ATP systems. The TPTP was developed to make evaluating and comparing ATP systems

Label	Action			Tool	
1	TF1	→	Proof	Alt-Ergo 0.95.2	[3]
1.2	TF1	→	TF0	Why3-TF0 0.85	[4]
1.3	TF1	→	FOF	Why3-FOF 0.85	[4]
2	TF0	→	Proof	CVC4 1.4 TFF-1.5	[2]
2.3	TF0	→	FOF	Monotonox-2FOF e3c1636	[5]
2.4	TF0	→	CNF	Monotonox-2CNF e3c1636	[5]
3	FOF	→	Proof	Vampire 4.0	[10]
3.4	FOF	→	CNF	ECNF 1.8	[12]
4	CNF	→	Proof	Vampire 4.0 or iProver-1.4	[8]
4.5	CNF	→	DL	Saffron 4.5	[1]
4.6	EPR	→	DL	EGround 1.8	[13]
5	DL	→	Proof	Konclude 0.6.1	[14]
6	PL	→	Proof	MiniSAT 2.2.0	[7]

Table 1: ATP Systems and Translators used in this research

possible by providing general guidelines, outlining the requirements for ATP system evaluation, and standards for input and output for ATP systems. The TPTP provides syntaxes for many logics. The TPTP syntax is compact and natural, and uses only ASCII characters. Propositional Logic (PL), Conjunctive Normal Form (CNF), Effectively Propositional Form (EPR), First Order Form (FOF), Typed First order form-monomorphic (TF0), and Typed First order form-polymorphic (TF1) are the logics supported by the TPTP. Description Logic Form (DLF) is the new TPTP syntax for DL. DLF was designed as a part of this research, and is described in Appendix A.

2 CNF to DL Translation

The translation of a CNF problem to DL is based on translating each CNF clause of the problem to a set of DL formulae that has equivalent semantics. Constants, unary predicates, and binary predicates in CNF correspond to the individuals, classes, and roles that are the building blocks of DL ontologies. Clauses with only constants, unary predicates, and binary predicates might be translated directly to DL. It might not be possible to translate some such clauses to DL, because there might not be equivalent DL semantics for the clauses. The current version of `Saffron` supports translation of CNF clauses with a wide range of literal structures, which have equivalent semantics in DL.

A *splitting technique* is applied in `Saffron` to preprocess problems with propositions, and hence remove the propositions. Considering each proposition in turn, the proposition is assigned `TRUE` and then `FALSE`. In each case, the resultant clause set is simplified by removing clauses that contain `TRUE` or `~FALSE`, and removing `FALSE` and `~TRUE` literals from the other clauses. Once all propositions have been removed, the resultant clause sets, i.e., proposition free versions of the problem, are translated to DL. If there are n propositions, there are 2^n sets to translate. If any one of the translated sets is satisfiable, the original CNF is satisfiable. If all of the translated sets are unsatisfiable, the original CNF is unsatisfiable. (Most readers will recognize this approach from the DPLL algorithm [6].)

Definition 1. CNF clauses that can be translated to DL using `Saffron` are referred to as *DL-able clauses*. Since DL-able clauses have no functions of arity greater than zero, they are

EPR.

Definition 2. CNF problems that includes only DL-able clauses are referred to as *DL-able problems*. Since these problems have no functions of arity greater than zero, they are EPR.

A problem with one or more CNF clauses that are not DL-able can be partially translated to DL, i.e, less than 100% of its clauses can be translated to DL. The unsatisfiability of a partially translated problem might be preserved, because an unsatisfiable subset of the clauses has been translated. The experiments in Section 5 show how successful this translator is in preserving the unsatisfiability of partially translated problems.

3 Translation Procedure

Definition 3. The translation function $saffron : c \longmapsto saffron(c)$ translates a CNF clause c in a CNF problem to a set of DL formulae $saffron(c)$. If c is not DL-able, $saffron(c)$ is the empty set. For each constant, unary predicate, and binary predicate in c, an individual, a class, and a role are correspondingly defined in $saffron(c)$ with the same name.

Every DL formula defines a characteristic of an individual, a class, a role, or the default class *Thing*. Some of the characteristics of individuals, classes, roles, and the class *Thing* that are covered in this translation are illustrated in Table 2. The CNF clause c is in TPTP syntax, and the corresponding DL formulae $saffron(c)$ are in DL mathematical syntax. As examples... The CNF clause in row 3 of Table 2 expresses that at least one of `p_i(a)` or `~q_j(a)` is `TRUE`. The DL formula expresses that the individual a belongs to the union of the classes p_i and the complements of the classes q_j. The CNF clause in row 9 is equivalent to the FOF formula `![X]: (p(X) => q(X))`, and expresses that for all `X` if `p(X)` then `q(X)`. The DL formula expresses that the class p is a subclass of the class q. The CNF clause in row 15 is equivalent to the FOF formula `![X,Y]: (r(X,Y) => s(Y,X))`, and expresses that for all `X` and `Y` if `r(X,Y)` then `s(Y,X)`. The DL formula expresses that the role r is a subrole of the inverse of the role s. The CNF clause in row 20 expresses that for all `X`, `X` is equal to at least one of the constants `a_i`. The DL formula expresses that the class *Thing* consists of individuals a_i.

4 Implementation of `Saffron`

`Saffron` has two implementations. One implementation outputs the resultant DL problem in RDF/XML, and the other implementation outputs the resultant DL problem in DLF. Both implementations consists of three translation related modules, and several support modules that are responsible for reading the input problem and saving the clauses in a processable format. The three translation related modules are named `translation`, `gather`, and `generate-output`.

In both implementations, the `translation` module translates the CNF problem clause-by-clause. Each clause is translated to a set of DL formulae. In the RDF/XML implementation the output from the `translation` module is passed to the `gather` module for further processing, and then the output of the `gather` module is passed to the `generate-output` module. In the DLF implementation the output from the `translation` module that needs further processing is passed to the `gather` module, and the rest of the output is passed directly to the `generate-output` module. In both implementations, the `generate-output` module uses the data received from the other module(s) to generate the output DL problem.

An output DL problem of `Saffron` is written to a file, which consists of a definition header and the body of the problem. The definition header includes information such as definitions

Table 2: CNF Clauses and Equivalent DL Formulae

#	Clause c	$saffron(c)$
1	`p(a)`	$\{ a : p \}$
2	`~p(a)`	$\{ a : \neg p \}$

3	`p_1(a) | ... | p_2(a) | p_n(a) |`
	`~q_1(a) | ~q_2(a) | ... | ~q_m(a)`
	where $n \geq 0$, $m \geq 0$, and $(m+n) \geq 2$
		$\{a : (\bigsqcup_{i=1}^{n} p_i \sqcup \bigsqcup_{j=1}^{m} \neg q_j)\}$

4	`r(a,b)`	$\{ r(a,b) \}$
5	`~r(a,b)`	$\{ \neg r(a,b) \}$
6	`a=b`	$\{ a = b \}$
7	`a!=b`	$\{ a \neq b \}$
8	`p(X)`	$\{ p = Thing \}$
9	`~p(X)|q(X)`	$\{ p \sqsubseteq q \}$
10	`~p(X)|r(X,Y)`	$\{ p \sqsubseteq \exists r.Thing \}$
11	`p(X)|~r(X,Y)`	$\{ \exists r.Thing \sqsubseteq p \}$
12	`p(X)|~q(Y)|~r(X,Y)`	$\{ \exists r.q \sqsubseteq p \}$

13	`p_1(X) | p_2(X) | ... | p_n1(X) |`
	`~q_1(X) | ~q_2(X) | ... | ~q_n2(X) |`
	`~r_1(X,Y) | ~r_2(X,Y) | ... | ~r_m(X,Y)`
	where $(n1 + n2) \geq 1$, and $m \geq 1$
		$\{\bigsqcap_{k=1}^{m} \exists r_k.(\bigsqcap_{i=1}^{n1}(\neg p_i) \sqcap \bigsqcap_{j=1}^{n2} q_j)$
		$\sqsubseteq Nothing \}$

14	`~r(X,Y)|s(X,Y)`	$\{ r \sqsubseteq s \}$
15	`~r(X,Y)|s(Y,X)`	$\{ inv_s = s^{-},$ $r \sqsubseteq inv_s \}$
16	`~r_1(X,Y)|~r_2(Y,Z)|s(X,Z)`	$\{ s \sqsupseteq (r_1 \circ r_2) \}$
17	`~r_1(X,Y)|~r_2(X,Z)|s(Y,Z)`	$\{ inv_s = s^{-},$ $inv_r_2 = r_2^{-},$ $inv_s \sqsupseteq (inv_r_2 \circ r_1) \}$
18	`~p(X)`	$\{Thing = \neg p\}$

19	`p_1(X) | p_2(X) | ... | p_n(X) |`
	`~q_1(X) | ~q_2(X) | ... | ~q_m(X) |`
	where $m \geq 0$, $n \geq 0$, and $(m+n) \geq 2$
		$\{ Thing = (\bigsqcup_{i=1}^{n} p_i \sqcup \bigsqcup_{j=1}^{m} \neg q_j) \}$

20	`X = a_1 | X = a_2 | ... | X = a_n` where $n \geq 1$	$\{ Thing = \{a_1, a_2, \ldots, a_n\} \}$
21	`r(X,a_1) | r(X,a_2) | ... | r(X,a_n)` where $n \geq 1$	$\{ Thing = \exists r.\{a_1, a_2, \ldots, a_n\} \}$

22	`p_1(X) | p_2(X) | ... | p_n(X) |`
	`~q_1(X) | ~q_2(X) | ... | ~q_m(X) |`
	`r(X,a_1) | r(X,a_2) | ... | r(X,a_n)`
	where $(m1 + m2) \geq 1$, and $n \geq 1$,
		$\{ Thing = (\bigsqcup_{i=1}^{n} p_i \sqcup \bigsqcup_{j=1}^{m} \neg q_j \sqcup$
		$\{a_1, a_2, \ldots, a_n\}) \}$

of schemas, namespaces and the ontology. The body of the problem in RDF/XML syntax is expressed as a set of RDF/XML tags. An RDF/XML tag might include sub-tags. The body of the problem in DLF syntax is expressed as a set of DLF formulae.

4.1 The `translation` Module

For each CNF clause, the `translation` module tries to translate the clause. If the clause is not DL-able, it is kept as an untranslated clause to be used later for statistics purposes, such as calculating the translation percentage. The translation module uses a translation table that maps clause structures to their translation in DL. The translation table used in the implementation of `Saffron` is of the form of Table 3, with over 30 rows. Table 3 illustrates the translation of DL-able clauses with no constants, no predicates, two or three binary predicates, and two or three variables.

To translate a CNF clause c, all the constants, unary predicates, binary predicates, and distinct variables of the clause c are extracted. The clause c is then filtered out if it is obvious that c is not DL-able. The filtering is done by checking the sizes of the sets of constants, unary predicates, binary predicates, and distinct variables of the clause. If the sizes of these sets does not match any of the DL-able clause structures in the translation table, then c is filtered out, and kept as untranslated. The filtering process narrows down the search space for a possible translation of the clause c.

The CNF clauses of Example 1 have no constants, no unary predicates, and two binary predicates. Clause 1.a has four distinct variables, and other clauses have two distinct variables. There is no DL-able clause structure with four distinct variables, so the clause 1.a is filtered out. However, clauses 1.b, 1.c, and 1.d are kept for the next step because the sizes of their sets of constants, unary predicates, binary predicates and variables match the DL-able clause structures in the first two rows of Table 3.

Example 1.

1.a. `dad(X,Y) | child(Z,T)`

1.b. `~dad(X,X) | child(Y,Y)`

1.c. `~dad(X,Y) | child(X,Y)`

1.d. `~dad(X,Y) | child(Y,X)`

Table 3: Examples of CNF and DL Equivalent Formulae

<table>
<tr><th>Clause c</th><th>Constants</th><th>Unary Predicates</th><th>Binary Predicates</th><th>Variables</th><th>$saffron(c)$</th></tr>
<tr><td><code>~r(X,Y) | s(X,Y)</code></td><td>{}</td><td>{}</td><td>{r,s}</td><td>{X,Y}</td><td>$\{ r \sqsubseteq s \}$</td></tr>
<tr><td><code>~r(X,Y) | s(Y,X)</code></td><td></td><td></td><td></td><td></td><td>$\{ inv_s = s^-,$
$r \sqsubseteq inv_s \}$</td></tr>
<tr><td><code>~r1(X,Y) | ~r2(Y,Z) | s(X,Y)</code></td><td>{}</td><td>{}</td><td>{r1,r2,s}</td><td>{X,Y,Z}</td><td>$\{ s \sqsupseteq (r_1 \circ r_2) \}$</td></tr>
<tr><td><code>~r1(X,Y) | ~r2(X,Z) | s(Y,Z)</code></td><td></td><td></td><td></td><td></td><td>$\{ inv_s = s^-,$
$inv_r_2 = r_2^-,$
$inv_s \sqsupseteq (inv_r_2 \circ r_1) \}$</td></tr>
</table>

The last step of the translation is to determine if there is an exact match for c. All features of the clause c, such as polarity of its literals, and the order of appearance of variables and constants in the arguments of binary predicates, are taken into account when finding a match.

In Example 1, the clauses 1.b, 1.c and 1.d share same sets of constants, unary predicates, binary predicates, and distinct variables. However, the clause 1.b is not DL-able, and the clauses 1.c and 1.d match the clause structures in the first and second rows of the Table 3. Examples 2.a and 2.b are the translation for clauses 1.c and 1.d correspondingly.

Example 2.

2.a. $\{dad \sqsubseteq child\}$

2.b. $\{inv_child = child^{-},$
$dad \sqsubseteq inv_child\ \}$

In the RDF/XML implementation, the output of the `translation` module on a DL-able clause c is a set of *pairs*. The first argument of the pair contains information that uniquely specifies an RDF/XML tag. The second argument of the pair is a DL formula expressed as a sub-tag. All the pairs are passed to the `gather` module to create completed RDF/XML tags.

In the DLF implementation, the output of the `translation` module on a DL-able clause c is a set of DL formulae and the list of the names of the constants, unary predicates, and binary predicates of c. DL formulae expressing that an individual belongs to a class, along with the names of constants, unary predicates and binary predicates of the clause c, are passed to the `gather` module. All other DL formulae are created in DLF syntax, and are passed directly to the `generate-output` module.

4.2 The `gather` and `generate-output` Modules

The characteristics of a specific individual, class or role, and the description of the class *Thing* are distributed in a CNF problem. After the `translation` module has translated all the DL-able clauses, the `gather` module processes the data received from the `translation` module.

In the RDF/XML implementation, the `gather` module collects all the resultant pairs from the `translation` module. It merges the sub-tags of all the pairs whose first arguments specify the same RDF/XML tag, to create a completed RDF/XML tag that describes the individual, class, role, or *Thing*. The output of the `gather` module is a set of RDF/XML tags that are passed to the `generate-output` module.

In the DLF implementation, the `gather` module collects all the DL formulae and the list of names received from the `translation` module. A type formula needs to be created for each individual, each class, and each role in DLF syntax. If an individual belongs to more than one class, then the type of the individual is the intersection of those classes. If an individual does not belong to any class, the type of the individual is `&thing` (the TPTP DLF syntax for *Thing*). The types of classes are always `$tType` (the kind of types in the typed TPTP languages). The types of roles are always `&thing * &thing > $o` (`$o` is the TPTP boolean type for formulae), because no CNF clause is translated in a way to provide the domain or range of a role. The output of the `gather` module is a set of DLF type formulae that is passed to the `generate-output` module.

The `generate-output` module creates an appropriate definition header for the DL problem, depending on the desired syntax of the output DL problem. The definition header and the set of either RDF/XML tags or DLF formulae are written to an output file.

5 Experiments

An experiment was carried out to compare solving problems by translation to DL, and solving by other possible combinations of translators and ATP systems. The comparison is done in terms of the number of problems solved, and the average time of the whole reasoning process through the path (including translations) over all solved problems. It is also interesting to note how many problems are solved exclusively by translation to DL.

The source of the problems for the experiment was the TPTP. Three sets of problems were attempted: (i) all DL-able CNF problems in the TPTP, (ii) EPR **S**oft**W**are **V**erification (SWV) problems that are DL-able after applying the splitting technique, and (iii) the set of all TF1 problems in the TPTP. The time limit for each stage of the reasoning process was 300 seconds, and the experiment was done on a machine with the following specifications.

NumberOfCPUs : 4
CPUModel: Intel(R) Xeon(TM) CPU 2.80GHz
RAMPerCPU: 756MB
OS: Linux 2.6.32.26-175.fc12.i686.PAE

The results of the experiments over each set of problems are discussed in Sections 5.1, 5.2, and 5.3. In the result tables "Total Available" is the number of problems available for the final translation step (or the ATP run, if there is no further translation) of the path. "Success" is the number of problems solved, and the percentage with respect to "Total Available". "Time" is the average CPU time for the whole reasoning process for solved problems. "Common Time" is the average CPU time for the common problems solved through all paths up to and including that row. "Translation Time" is the average CPU time for the translation steps for solved problems. "Solving Time" is the average CPU time for the ATP system to solve the (translated) problems. "Timeout" is the number of problems for which the last translation step or the ATP system timed out, and the percentage with respect to "Total Available". "Failed" is the number of problems for which the last translation step or the ATP system failed, and the percentage with respect to "Total Available".

5.1 DL-able Problems

The first set of problems was the DL-able CNF problems in the TPTP. This set was chosen to experiment with solving by translation to DL, without the effects of splitting, partial CNF to DL translation, and the translation from logics more expressive than CNF to CNF. There are 23 unsatisfiable and 3 satisfiable DL-able problems in the TPTP. Table 4 shows the results of the experiment.

Since all these problems are EPR, `iProver` is used to solve the problems in their original CNF form. As illustrated in this table, `iProver` can solve more problems than `EGround > MiniSAT`, and `EGround > MiniSAT` can solve more problems than `Saffron > Konclude`.

The results of the experiment suggest that solving an EPR problem by `Saffron > Konclude` is an alternative, despite the fact that `iProver` is a well-tuned and powerful ATP system for EPR problems. The average time of the translation of all problems by `Saffron` is 0.43 second, but `Konclude` timed out with the time limit of 300 seconds over the "Timeout" problems.

5.2 Software Verification Problems

The second set of problems was all the EPR SWV problems in the TPTP that were DL-able after applying splitting technique. This set was chosen to experiment with solving by translation to DL with the splitting technique, but without the effects of partial CNF to DL translation,

Table 4: Results of Comparing Paths from EPR over 26 DL-able Problems

Path	Tools	Success	Time	Translation Time	Solving Time	Timeout	Failed
EPR	`iProver`	26 (100%)	34.76	0.00	34.76	0 (0%)	0 (0%)
EPR.PL	`EGround` > `MiniSAT`	13 (50%)	1.40	0.00	1.40	0 (0%)	13 (50%)
EPR.DL	`Saffron` > `Konclude`	12 (46%)	0.54	0.00	0.54	14 (54%)	0 (0%)

and the translation from logics more expressive than CNF. There are 40 unsatisfiable such SWV problems. They all have only one proposition. Splitting technique produces two unsatisfiable proposition free versions of each problem. The problem is considered solved by translation to DL if the unsatisfiability of the both proposition free versions of the problem is confirmed by `Konclude`.

The two versions of the problem can be solved in parallel if more than one CPU is available. Table 5 shows the results of the experiment. In Table 5, EPR.DL(serial) and EPR.DL(parallel) show the statistics for solving the two versions of SWV problems in serial and parallel through the path EPR.DL. The success rate of solving by translation to DL is 80%. This results suggest that solving by translation to DL is an alternative way of solving EPR SWV problems.

Table 5: SWV Problems with Splitting Technique

Path	Tools	Success	Time	Translation Time	Solving Time	Timeout	Failed
EPR	`iProver`	40 (100%)	13.87	0.00	13.87	0 (0%)	0 (0%)
EPR.DL (parallel)	`Saffron` > `Konclude`	32 (80%)	22.00	2.11	19.89	8 (20%)	0 (0%)
EPR.DL (serial)	`Saffron` > `Konclude`	32 (80%)	29.92	4.21	25.71	8 (20%)	0 (0%)
EPR.PL	`EGround` > `MiniSAT`	3 (8%)	0.38	0.27	0.12	24 (60%)	13 (33%)

5.3 Typed First Order From - polymorphic Problems

The last set of problems was all the TF1 problems without arithmetic in the TPTP. There are 538 TF1 problems. This set was chosen to experiment with solving problems expressed in logics more expressive than CNF by translation to DL. TF1 problems can be translated to TF0 or FOF. If a problem is translated to TF0, then it can be translated to FOF or CNF. If a problem is translated to FOF, then it can be translated to CNF. None of the problems translated to CNF are EPR.

Table 6 shows the results of the experiment. The results suggest that solving by translation down to FOF or CNF solves more problem than than using the ATP system for TF1. The

reason is that Vampire, the ATP system for FOF and CNF problems, is a very powerful ATP system. The TF1 problems are mostly very hard problems, and the best approach can solve 44% of the problems within the time limit of 300 seconds. The best translation to DL approach, through the path TF1.TF0.CNF.DL, can solve 3% of the problems. The other two translation to DL approaches can solve 2% and 1% of the problems. All the translated problems to CNF are partially translated to DL.

Table 6: Results of Comparing Paths from TF1

Path	Tools	Total Available	Success	Time	Common Time	Translation Time	Solving Time	Timeout	Failed
TF1.FOF	Why3-FOF > Vampire	538	238 (44%)	9.34	9.34	0.08	9.26	288 (54%)	12 (2%)
TF1.TF0.FOF	Why3-TF0 > Monotonox-2FOF > Vampire	538	234 (43%)	2.95	3.77	0.12	2.82	225 (42%)	79 (15%)
TF1.TF0.FOF.CNF	Why3-TF0 > Monotonox-2FOF > ECNF > Vampire or iProver	495	233 (47%)	7.72	8.47	0.19	7.52	225 (45%)	37 (7%)
TF1.FOF.CNF	Why3-FOF > ECNF > Vampire or iProver	538	231 (43%)	7.51	3.70	0.09	7.42	303 (56%)	4 (1%)
TF1.TF0.CNF	Why3-TF0 > Monotonox-2CNF > Vampire or iProver	538	208 (39%)	7.48	5.82	0.14	7.35	257 (48%)	73 (14%)
TF1	Alt-Ergo	538	197 (37%)	0.79	0.25	0.00	0.79	158 (29%)	183 (34%)
TF1.TF0	Why3-TF0 > CVC4	538	184 (34%)	11.00	5.09	0.25	10.75	295 (55%)	59 (11%)
TF1.TF0.CNF.DL	Why3-TF0 > Monotonox-2CNF > Saffron > Konclude	486	14 (3%)	0.68	0.71	0.11	0.56	0 (0%)	472 (97%)
TF1.TF0.FOF. CNF.DL	Why3-TF0 > Monotonox-2FOF > ECNF > Saffron > Konclude	495	8 (2%)	0.71	0.71	0.15	0.56	0 (0%)	487 (98%)
TF1.FOF.CNF.DL	Why3-FOF > ECNF > Saffron > Konclude	538	3 (1%)	0.66	0.67	0.12	0.54	0 (0%)	535 (99%)

When more than one CPU is available, different approaches can be executed on one problem simultaneously to increase the chance of the problem being solved. Therefore, with N processors, the N paths that provide the maximum number of problem solved in this experiment are chosen. Table 7 shows which path is the best to be added to maximize the number of problems solved when increasing the number of CPUs. Out of the 538 problems, 300 problems can be solved through all paths together. The five paths shown in Table 7 can solve all these problems, so using more than five CPUs will not increase the chance of a problem being solved. It shows that using multiple CPUs significantly increases the chance of a problem being solved. Using the five approaches in Table 7 solves 300 problems, which is an over 26% increase in the number of problems solved. The two translation to DL approaches can exclusively solve 2% of

the problems. This result is interesting since the TF1 ATP system cannot solve any problems exclusively.

Table 7: Paths from TF1 in Parallel

N	Path	Exclusive Problems	Total Solved	Average Time
1	TF1.FOF	238	238	9.34
2	TF1.TF0.FOF.CNF	55	293	7.18
3	TF1.TF0.CNF.DL	4	297	7.03
4	TF1.FOF.CNF	2	299	7.47
5	TF1.FOF.CNF.DL	1	300	7.45

6 Conclusion

There are different ways of solving a problem expressed in a logic. It can be solved directly by using the ATP system for that logic, or it can be translated to a less expressive form. When it is translated to a less expressive form, again the same two options are available.

DL is more expressive than PL, and less expressive than EPR. There used to be a gap between DL and more expressive logics because no CNF to DL translator was available before. In this research, `Saffron`, a CNF to DL translator was developed. DL, as a specific logic for expressing ontologies was not a part of TPTP world. DLF is a TPTP syntax was designed in this research. Now, DL can be embedded in the TPTP world. The successful application to software verification problems introduces another application for DL.

Experiments have been carried out to evaluate solving problems by translation to DL. The results of the experiments suggest that solving problems by translation to DL is an alternative way of solving DL-able CNF problems, CNF SWV problems, and TF1 problems. Moreover, solving by translation to DL is the only way of solving some TF1 problems.

Future work includes adding more features to `Saffron` to translate CNF clauses with wider range of clause structures. DLF and DLF tools can be integrated into the TPTP world. This is possible by developing tools for importing problems in other DL syntaxes into the TPTP in the DLF syntax, and adding features to TPTP tools for exporting problems in other DL syntaxes. DL problems can be collected and released in the TPTP problem library. Solutions for DL problems can be collected and released in the TSTP solution library. Eventually, DL ATP systems and tools can be added to the `SystemOnTPTP`.

A Description Logic Form

Description Logic Form (DLF) is the new TPTP syntax for the DL SROIQ. The design of DLF is a first step toward adding DL to the TPTP world, so the DL community will be able to benefit from the TPTP tools. DLF, which is compact, natural and easily processable by both human and computers, was designed in this research. A BNF for DLF was designed to enable the construction of parsers for DLF problems and solutions in other languages other than Prolog.

Like all TPTP problem files, DLF problem files include up to three sections: the header, include directives, and the annotated formulae. The header and include directives are the same

as all TPTP problems. The annotated formulae of a DL problem can include *definitions*, *type formulae*, and *logic formulae*.

Definitions are used to set aliases for repeatedly-used identifiers, such as namespaces and schemas. Example 3 shows the definition of the OWL namespace and the default class *Thing* in the OWL namespace. The aliases `&owl` and `&thing` can be used in the problem instead of their long URIs.

Example 3.

```
dlf(owl_defn,definition,
    owl := 'http://www.w3.org/2002/07/owl' ).
dlf(thing_defn,definition,
    thing := &owl#'Thing' ).
```

DLF type formulae are used to declare individual, classes, and roles. In addition to the declaration, the type formulae for individuals specify the classes that the individual belongs to, and the type formulae for roles specify their domain and range. In Example 4, `student` is a class, `joe` is an individual belonging to the class `student`, and `register` is a role with the domain `student` and the range `&thing`.

Example 4.

```
dlf(student_type,type,
    student:  $tType ).
dlf(joe_type,type,
    joe:  student ).
dlf(register_type,type,
    register:  ( student * &thing ) > $o ).
```

The DLF logic formulae are used to express any DL formula other than type formulae, such as DL axioms and conjectures. In Example 5, `math101` is declared to be an individual belonging to the class `&thing`. The DLF logic formula `joe_registered_math101` is a DL axiom expressing that `joe` is registered in `math101`.

Example 5.

```
dlf(math101_type,type,
    math101:  &thing ).
dlf(joe_registered_math101,axiom,
    register(joe,math101) ).
```

References

[1] N. Arhami. The Efficiency of Automated Theorem Proving by Translation to Less Expressive Logics. Master's thesis, Department of Computer Science, University of Miami, Miami, USA, 2014.

[2] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Confer-*

ence on Computer Aided Verification, number 6806 in Lecture Notes in Computer Science, pages 171–177. Springer-Verlag, 2011.

[3] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Assia Mahboubi, Alain Mebsout, and Guillaume Melquiond. A Simplex-based extension of Fourier-Motzkin for solving linear integer arithmetic. In Bernhard Gramlich, Dale Miller, and Ulrike Sattler, editors, *IJCAR 2012: Proceedings of the 6th International Joint Conference on Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pages 67–81, Manchester, UK, June 2012. Springer.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[5] K. Claessen, A. Lilliestrom, and N. Smallbone. Sort It Out with Monotonicity - Translating between Many-Sorted and Unsorted First-Order Logic. In N. Bjorner and V. Sofronie-Stokkermans, editors, *Proceedings of the 23rd International Conference on Automated Deduction*, number 6803 in Lecture Notes in Artificial Intelligence, pages 207–221. Springer-Verlag, 2011.

[6] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[7] N. Eén and N. Sörensson. MiniSat - A SAT Solver with Conflict-Clause Minimization. In F. Bacchus and T. Walsh, editors, *Posters of the 8th International Conference on Theory and Applications of Satisfiability Testing*, 2005.

[8] K. Korovin. iProver - An Instantiation-Based Theorem Prover for First-order Logic (System Description). In P. Baumgartner, A. Armando, and D. Gilles, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, number 5195 in Lecture Notes in Artificial Intelligence, pages 292–298, 2008.

[9] D.W. Loveland. *Automated Theorem Proving : A Logical Basis*. Elsevier Science, 1978.

[10] A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.

[11] S. Schulz. E: A Brainiac Theorem Prover. *AI Communications*, 15(2/3):111–126, 2002.

[12] S. Schulz. System Abstract: E 0.81. In M. Rusinowitch and D. Basin, editors, *Proceedings of the 2nd International Joint Conference on Automated Reasoning*, number 3097 in Lecture Notes in Artificial Intelligence, pages 223–228. Springer-Verlag, 2004.

[13] S. Schulz and G. Sutcliffe. System Description: GrAnDe 1.0. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, number 2392 in Lecture Notes in Artificial Intelligence, pages 280–284. Springer-Verlag, 2002.

[14] Andreas Steigmiller, Thorsten Liebig, and Birte Glimm. Konclude: System description. *Web Semantics: Science, Services and Agents on the World Wide Web*, 0(0), 2014.

[15] G. Sutcliffe. SystemOnTPTP. In D. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, number 1831 in Lecture Notes in Artificial Intelligence, pages 406–410. Springer-Verlag, 2000.

[16] G. Sutcliffe. The SZS Ontologies for Automated Reasoning Software. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics*, number 418 in CEUR Workshop Proceedings, pages 38–49, 2008.

[17] 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.

[18] G. Sutcliffe. The TPTP World - Infrastructure for Automated Reasoning. In E. Clarke and A. Voronkov, editors, *Proceedings of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, number 6355 in Lecture Notes in Artificial Intelligence, pages 1–12. Springer-Verlag, 2010.