Solving the Projection Problem with OWL2 Reasoners: Experimental Study

Wael Yehia[1] and Mikhail Soutchanski[2]

[1] Department of Computer Science and Engineering York University, 4700 Keele Street, Toronto, ON, M3J 1P3, Canada
`w2yehia@cse.yorku.ca`
[2] Department of Computer Science, Ryerson University, 245 Church Street, ENG281, Toronto, ON, M5B 2K3, Canada `mes@scs.ryerson.ca`

Abstract. We evaluate HERMIT, an OWL2 reasoner, on a set of test cases that emerge from an unusual but very practical way of using Description Logics (DLs). In the field of reasoning about actions, the projection problem is an elemental problem that deals with answering whether a certain formula holds after doing a sequence of actions starting from some initial states represented using an incomplete theory. We consider a fragment of the situation calculus and Reiter's basic action theories (BAT) such that the projection problem can be reduced to the satisfiability problem in an expressive description logic $\mathcal{ALCO}^U$. We adapt an approach called regression where an input query is equivalently transformed until it can be directly checked against the initial theory supplemented with the unique name axioms (UNA) without any consideration to the rest of the BAT. To study regression in practice, we implemented it in C++, defined an XML SCHEMA to describe a BAT and queries, created 7 domains some of which are inspired from well-known planning competition domains, and invented generators that can create random but meaningful instances of the projection problem. The formula resulting from regressing a projection query, together with the initial theory and UNA, is fed to an OWL2 reasoner to answer whether the regressed query holds given the initial theory and UNA. We measure the input formula using a number of metrics, such as a number of $\mathcal{U}$-role occurrences and a number of individuals, and evaluate the performance of HERMIT on formulas along each dimension.

Keywords: random generator, Description Logics, projection problem, situation calculus, Action Theory

1 Introduction

We study a new class of formulas that arise from using Description Logics to solve one of the reasoning problems known as the projection problem (PP) which occurs naturally in the area of reasoning about actions. We consider a sufficiently broad sub-case of PP that can be formulated in a fragment of the situation calculus (SC) such that PP becomes decidable. We will explain later what the PP is, but what's important to note is that it is a prerequisite to few other important reasoning problems like planning, which makes it an interesting problem to tackle.

The reasoning about actions community hasn't made solutions to the problem very practical to use so far. There are no practical implementations that can solve the PP in a logical language that is not purely propositional, and when initial knowledge is incomplete, i.e., in realistic settings without the closed world assumption (CWA) and without the domain closure assumption (DCA). Using our approach of transforming the PP into the satisfiability problem in DL is one way we can start the ball rolling (in the DL direction). Our goal is to provide automatically generated test cases for the DL community to work on and speed up reasoning time on. These test cases can be used for stress testing the DL reasoners. Indeed, our testing clearly shows that the majority of the time spent on solving one instance of the PP is in the DL reasoner.

Our test cases usually originate from a single common setting. There are two formulas: a premise formula (translated into a DL concept representing an initial theory) and a regressed query formula (translated into another DL concept). The goal is to check if the regressed query holds given the premise. The two formulas are translated into concepts in a DL called $\mathcal{ALCO}^U$ that includes nominals ($\mathcal{O}$), the universal role ($\mathcal{U}$), and constructs from the well-known logic $\mathcal{ALC}$. In addition, each test case may contain a set of individuals that belong to certain concepts, and might be related to each other using certain roles. We also make the Unique Name Assumption (UNA) for individuals by stating they are pairwise unequal.

Finally, the test cases are based on some practical domains such as Logistics and Assembly from the International Planning Competitions [6, 2], and on other domains invented by us to illustrate the expressivity of our language $\mathcal{L}$. We provide 7 domains (Logistics, Assembly, Airport, Turing Machine Addition, Turing Machine Successor, Scheduling World, and Hemophilia from biology) in which projection queries can be formulated, and provide query and action sequence generators for the first four domains. A thorough description of the domains can be found in [5].

We have developed the C++ program that transforms a PP instance into an instance of the $\mathcal{ALCO}^U$ satisfiability problem. Using all our tools, we are able to generate random PP instances, and subsequently transform them into $\mathcal{ALCO}^U$ concepts given as input to OWL2 reasoners.

There are at least two directions that can be taken from here in terms of testing. One is to generate test cases by varying the complexity of the projection problem such as initial state complexity/size, query size, and length of the action sequence. This will lead to investigation of the projection problem in terms of reasoning time. Second is to generate arbitrary but simple PP instances, and study the final transformed formulas based on their structure. That will provide new test suites for DL reasoners, and shed light on the areas where they need improvement. We take the second approach in this paper, because the first was studied in [1] where a comparison between two approaches to solving the projection problem is carried out. In this paper, we will measure the reasoning time for HERMIT[3], an OWL2 reasoner, on a randomly generated pool of test cases. We group the test cases based on (a) a number of individuals in the domain, for similar queries, and (b) a number of $\mathcal{U}$-role occurrences, then study the behavior of HERMIT on each group.

We will now explain the nature of the problem we are solving, and how it leads us to using DL reasoners.

[3] `http://www.hermit-reasoner.com/publications.html`

2 Background

In the field of reasoning about actions, an important problem is answering whether a formula holds after executing a sequence of actions starting from some initial state. This problem is known as the projection problem, and there are many flavors of the problem that differ by the restrictions on the query, a set of initial states, language at hand, and other properties. In description logics (DLs) and earlier terminological systems, this problem was formulated using roles to represent transitions and concept expressions to represent states. This line of research as well as earlier applications of DLs to planning and plan recognition are discussed and reviewed in [3]. Reiter showed that the projection problem in the situation calculus can be solved using a method called *regression* [7]. The problem is undecidable in the general SC, but by limiting the expressivity to a fragment of SC one can get decidability. We propose a fragment P based on a language $\mathcal{L}$ in which we solve the projection problem using regression. Regression (explained below) is a method of transforming the projection problem into a satisfiability problem in some language. This language is critical, as the fact whether SAT in the language is decidable will lead to the decidability of the projection problem. Our test case generation and testing is based on P, and the language used to solve the satisfiability problem after regression can be mapped to the DL $\mathcal{ALCO}^U$. This is how we end up using the OWL reasoners because they can easily handle $\mathcal{ALCO}^U$ concepts. The executability problem is another common problem where an action's precondition axioms are checked for satisfiability in a given state. This problem is important as mechanisms like regression depend on the assumption that the action sequence in the query is executable starting from the initial situation.

In this paper, all constants start with upper case, and all variables with lower case letters. The free variables in formulas are assumed to be $\forall$-quantified at front. We use the standard first order logic (FOL) definitions of well-formed formulas and terms.

2.1 Basic Action Theories and Language $\mathcal{L}$

This approach is based on Basic Action Theories in SC. We will be brief in our description of BATs, but the interested readers can find formal definitions in [8]. In general, a BAT $\mathcal{D}$ consists of the pre-condition axioms (PAs) $\mathcal{D}_P$, the successor state axioms (SSAs) $\mathcal{D}_{SS}$, which characterize the effects and non-effects of actions; an incomplete initial theory $\mathcal{D}_{S_0}$ about the initial situation S_0; a set of domain independent foundational axioms; and axioms for the unique-name assumption (UNA). It is also possible to augment the BAT with a TBox [9, 4]. In SC, a *situation* is a sequence of actions. The truth values of some of the predicates can vary with respect to the state pointed out by the sequence of actions. The *fluents* are the predicates with a situation term as their last argument. You can think of them as "dynamic" predicates whose truth value can change based on the situation argument. Other predicates (with no situation argument) are called "static" predicates, because their truth value does not depend on situation. We say that a SC formula $\phi(s)$ is *uniform* in s, if s is the only situation term mentioned in $\phi(s)$, and $\phi(s)$ has no quantifiers over situation variables.

Subsequently, we consider only BATs with relational fluents, and no other function symbols except $do(a, s)$ and action functions are allowed. Terms of sort object can only be constants or variables. Action functions can have any number of object arguments.

The following is a short and informal definition of the language $\mathcal{L}$ that is used to construct formulas that are allowed in our restricted BATs. The precise definition is provided in [9]. $\mathcal{L}$ is a set of FOL formulas that is divided into two symmetric subsets $\mathcal{L}_x$ and $\mathcal{L}_y$. It uses a finite set of auxiliary variables usually denoted by z_1, z_2, and so on (whose sole purpose is to be replaced by constants during regression), as well as the x and y variables. The main restriction is that z-variables cannot be quantified over, hence they are always free in $\mathcal{L}$ formulas, while the x and y variables can be quantified, but only one of them can be free in a particular $\mathcal{L}$ formula. The $\mathcal{L}_x$ set includes all formulas with free x, and $\mathcal{L}_y$ includes all formulas with free y. The $\mathcal{L}$ formulas without occurrences of x, y, and the $\mathcal{L}$ formulas where all occurrences of x and y variables are bound belong to both $\mathcal{L}_x$ and $\mathcal{L}_y$. Informally, an $\mathcal{L}_x$ formula ($\mathcal{L}_y$ formulas are symmetric) has one of these syntactic forms:

1. atomic formulas such as $true$, $false$, $x = t_1$, $A(t_1)$, and $R(t_1, t_2)$,
   where A and R are unary and binary predicate symbols respectively, t_1 and t_2 can be either constants or z-variables, and t_1 can also be x.
2. non atomic formulas such as:
   (a) $A(x) \wedge B(x)$, $A(x) \vee B(x)$, and $\neg A(x)$, where $A(x)$ and $B(x)$ are $\mathcal{L}_x$ formulas.
   (b) $\forall t_1.D(t_1)$, $\exists t_1.D(t_1)$, $\forall y.R(t_2, y) \supset C(y)$, and $\exists y.R(t_2, y) \wedge C(y)$,
   where $D(t_1)$ is a $\mathcal{L}_x$ or $\mathcal{L}_y$ formula depending on whether t_1 is x or y, $C(y)$ is $\mathcal{L}_y$ formula, R is a binary predicate symbol, and t_2 can be either x, a constant, or a z-variable.

Note any z_i variable other than x and y has to be free in a formula from $\mathcal{L}$. The intuition behind the definition of $\mathcal{L}$ is that $\mathcal{L}$ formulas with z_i variables instantiated with constants should be $\mathcal{ALCO}^U$ representable.

Lemma 1. *For any formula $\phi \in \mathcal{L}$ with all z-variables instantiated with constants, there exist a translation to an $\mathcal{ALCO}^U$ concept with no more than a linear increase in the size of ϕ. The inverse also holds, i.e. any $\mathcal{ALCO}^U$ concept can be translated into an $\mathcal{L}$ formula without z-variables.*

The variables z_i are important for our purposes because they serve as arguments of actions in axioms. Thanks to them, we can consider BATs where actions may have any number of arguments, thereby increasing expressivity of BATs that can be formulated. This becomes important when benchmark planning domains (considered as FOL specifications) have to be represented as our BATs. Notice that $\mathcal{L}$ formulas do not always have equivalent $\mathcal{ALCO}^U$ concepts; only $\mathcal{L}$ formulas **without z-variables** do have. But this is exactly what is required for our purposes of regression. The final regressed formula is guaranteed to be **z-variable free**, since in the projection formula uniform in S all arguments of actions mentioned in the situation term S must be instantiated with constants. As a consequence, all z variables become instantiated as well. The Lemma 1 is the reason why we can use the DL $\mathcal{ALCO}^U$ to solve any PP instance in our class P of BATs. Lemma 1 is proved using the standard translation between DLs and First Order Logic (FOL); the proof is similar to the proof of Lemma 1 in [4]. Notice that using the standard translation between FOL and a DL, the formula $\forall y.R(t_2, y) \supset C(y)$ translates as $\forall R.C$, the formula $\exists y.R(t_2, y) \wedge C(y)$ becomes $\exists R.C$, and the formulas $\forall t_1.D(t_1)$,

$\exists t_1.D(t_1)$ can be translated as $\forall U.D$ and $\exists U.D$, respectively. The latter formulas are handy for our purposes because in our axioms we need unguarded quantifiers. As another example, a z-free $\mathcal{L}$ formula $\exists x.(Box(x) \wedge x \neq B_1 \wedge ready(x))$, some box distinct from B_1 is ready, can be translated to $\mathcal{ALCO}^U$ as $\exists U.(Box \sqcap \neg\{B_1\} \sqcap ready)$, where B_1 is a nominal. Notice that the nominals are handy to translate exceptions.

We proceed to describing how $\mathcal{L}$ is used in the BAT, and how it expands the expressivity of the BAT to beyond DLs, while maintaining complexity of deciding the PP in the range acceptable from a DL perspective. To facilitate understanding of each part of the BAT, we will use examples from one of our domains, the Logistics domain.

The domain describes objects (boxes or luggage) that can be loaded and unloaded from vehicles (trucks or airplanes) and transported from one location to another. Trucks can be driven from one location to another in the same city, while airplanes can fly between airports (which are locations) in different cities. Let the logical language include:

1. Four **action functions**: $load(obj, vehicle, location)$, $unload(obj, vehicle, location)$, $drive(truck, locFrom, locTo, city)$, $fly(airplane, locFrom, locTo)$.
2. Three **relational fluents**: $loaded(obj, vehicle, s)$, $at(x, loc, s)$, and $ready(obj, s)$ (where x can be an object or a vehicle).
3. **Static unary predicates** (i.e. predicates with no situation term) to describe each type of entity, and one static binary predicate $in\_city(loc, city)$.

For brevity, let a vector $\boldsymbol{x}$ of object variables denote either x, y, or $\langle x, y\rangle$, and let $\boldsymbol{z}$ denote a vector of place holder variables $\langle z_1, z_2, ...\rangle$.
Action precondition axioms (PAs) $\mathcal{D}_{AP}$: the preconditions that have to hold before an action can be executed. There is one axiom per action $Act(\boldsymbol{z})$ of the following form:

$$(\forall \boldsymbol{z}, s).\ \ Poss(A(\boldsymbol{z}), s) \equiv \mathit{\Pi}_A(\boldsymbol{z}, s),$$

where $Poss$ (derived from 'possible') is a special binary predicate that occurs on the left hand side of PAs only, and $\mathit{\Pi}_A$ is an $\mathcal{L}$ formula whose only free variables are the place holder z_i variables. The formula $\mathit{\Pi}_A(\boldsymbol{z}, s)$ is uniform in s. For example, the PA for action $load(Object, Vehicle, City)$ is the following:

$$\begin{aligned}&Poss(load(z_1, z_2, z_1), s) = \\ &\ (obj(z_1) \wedge veh(z_2) \wedge loc(z_3) \wedge ready(z_1, s) \wedge at(z_1, z_3, s) \wedge at(z_2, z_3, s))\end{aligned}$$

Definition 1. *Let ϕ_x and ϕ_y be $\mathcal{L}_x$ and $\mathcal{L}_y$ formulas, respectively, such that they are uniform in s (ϕ_x, ϕ_y are called context conditions). Let $\boldsymbol{u}$ be a vector of variables at most containing $\boldsymbol{z}$ and an optional x variable, $\boldsymbol{v}$ be a vector of variables at most containing $\boldsymbol{z}$, and optionally x or y variables. Let $Act(\boldsymbol{u})$ and $Act(\boldsymbol{v})$ be action terms, and a be an action variable. A CC formula has one of the following two forms:*

$$\begin{array}{ll}\exists \boldsymbol{z}.a = Act(\boldsymbol{u}) \wedge \phi_x(x, \boldsymbol{z}, s) & \textit{SSA for an unary fluent}\\ \exists \boldsymbol{z}.a = Act(\boldsymbol{v}) \wedge \phi_x(x, \boldsymbol{z}, s) \wedge \phi_y(y, \boldsymbol{z}, s) & \textit{SSA for a binary fluent}\end{array}$$

Successor state axioms (SSAs) $\mathcal{D}_{SS}$: Define the direct effects and non-effects of actions. There is one SSA for each fluent $F(\boldsymbol{x}, s)$ of the following syntactic form:

$$F(\boldsymbol{x}, do(a, s)) \equiv \gamma_F^+(\boldsymbol{x}, a, s) \vee F(\boldsymbol{x}, s) \wedge \neg\gamma_F^-(\boldsymbol{x}, a, s), \tag{1}$$

where each of the γ_F's are disjunctions of CC formulas. For example, the SSA for fluent *loaded* is as follows:

$$\begin{aligned}
&loaded(x, y, do(a, s)) = \\
&[\exists z_1.a = load(x, y, z_1) \wedge obj(x) \wedge veh(y) \wedge loc(z_1) \wedge ready(x, S) \wedge at(x, z_1, s)] \\
&\vee [loaded(x, y, s) \wedge \neg[\exists z_1.a = unload(x, y, z_1)]]
\end{aligned}$$

TBox axioms $\mathcal{D}_T$:
These are TBox axioms for unary predicates, where the right hand side is an $\mathcal{L}$ formula without z variables. For example:

$$\begin{aligned}
veh(x) &\equiv truck(x) \vee airplane(x) \\
loc(x) &\equiv street(x) \vee airport(x)
\end{aligned}$$

Initial Theory $\mathcal{D}_{S_0}$: The $\mathcal{D}_{S_0}$ is an $\mathcal{L}$ sentence without z variables, i.e. it can be transformed into an $\mathcal{ALCO}^U$ concept. For example:

$$\begin{aligned}
&city(Toronto) \wedge airport(YYZ) \wedge in\_city(YYZ, Toronto) \wedge street(Yonge) \wedge \\
&in\_city(Yonge, Toronto) \wedge box(B1) \wedge at(B1, Yonge, S0) \wedge \\
&mail\_truck(T1) \wedge at(T1, Yonge, S0) \wedge \forall x(obj(x) \supset ready(x, S0)) \wedge \\
&box(B2) \wedge (loaded(B2, T1, S0) \vee \neg\exists x(loaded(B2, x, S0) \wedge vehicle(x)))
\end{aligned}$$

Finally, a projection query is an $\mathcal{L}$ sentence, without z variables, and there is a ground situation term S representing the action sequence after which the formula should hold or not. For example, given the above initial theory, the action sequence represented by situation $do(drive(T1, Yonge, YYZ, Toronto), S_0)$ is executable and the following query answers true:

$$at(T1, YYZ, do(drive(T1, Yonge, YYZ, Toronto), S_0))$$

2.2 Regression for Solving the Projection Problem

In the context of BATs and SC, *regression* is a recursive transformation converting a formula uniform in situation $do(a, s)$ into a logically equivalent formula uniform in s (that is one action shorter down the situation term) by making use of the SSAs. A modified regression operator $\mathcal{R}$ is defined to guide the regression process in our class P of BATs, and it is defined recursively on formulas of the underlying language at hand, $\mathcal{L}$ in our case. We do not define the modified operator here due to space limitations, but interested readers can see [4] for more details about regression in a language that is similar to $\mathcal{L}$ (but that is a proper subset of $\mathcal{L}$). The idea is that all static predicates are not affected by regression, and hence remain the same after the regression operator is applied. Fluents ("dynamic" predicates) on the other hand are transformed by $\mathcal{R}$. On each step, the regression operator $\mathcal{R}$ replaces each fluent formula uniform in situation $do(a, s)$ by the right hand side (RHS) of the SSA for the fluent (recall that the CC formulas on the RHS are uniform in s). Subsequently, regression continues until all fluents have S_0 as the only situation term. Consider the following example query:

$$loaded(B1, T1, do(load(B1, T1, Yonge), S_0))$$

Also, let the above $\mathcal{D}_{S_0}$ be the initial theory against which this query is checked. First, replace the fluent (with its constant arguments and situation term) by the right hand side of the SSA, to get:

$$\big(\exists z_1.load(B1,T1,Yonge) = load(B1,T1,z_1) \wedge obj(B1) \wedge veh(T1) \\ \wedge\, loc(z_1) \wedge ready(B1,S_0) \wedge at(B1,z_1,S_0)\,\big) \vee \\ \big(loaded(B1,T1,S_0) \wedge \neg\exists z_1.load(B1,T1,Yonge) = unload(B1,T1,z_1)\,\big)$$

By applying UNA – similar action names denote the same action and similar constant names denote the same object in the world – we get a shorter FOL formula:

$$\big(\,\exists z_1.B1{=}B1 \wedge T1{=}T1 \wedge Yonge{=}z_1 \wedge obj(B1) \wedge veh(T1) \wedge loc(z_1) \wedge \\ ready(B1,S_0) \wedge at(B1,z_1,S_0)\big) \vee \big(loaded(B1,T1,S_0) \wedge \neg\exists z_1.false\,\big)$$

Further simplifications yield the formula, which is the result of one step of regression:

$$\big(obj(B1) \wedge veh(T1) \wedge loc(Yonge) \wedge ready(B1,S_0) \wedge at(B1,Yonge,S_0)\big) \vee \\ loaded(B1,T1,S_0)$$

It is clear that the first disjunct holds given the above $\mathcal{D}_{S_0}$. Hence, the answer to this projection problem is true.

Note that the resulting formula is uniform in S_0. In general, a query whose situation term mentions n ground actions, requires n consecutive regression steps to bring it down to situation S_0. The benefit of regression is that the final regressed formula is logically equivalent to the original query, but now we do not need to consider the whole BAT to answer the query, just $\mathcal{D}_{S_0}$ and UNA. Thereby, the projection problem is transformed from solving whether $BAT \models Query$ to solving whether $UNA \cup \mathcal{D}_{S_0} \models \mathcal{R}[Query]$, where $\mathcal{R}[Query]$ is the formula resulting from regression of the query. Since $\mathcal{D}_{S_0}$, UNA and $\mathcal{R}[Query]$ are z-free $\mathcal{L}$ formulas, they can be converted into $\mathcal{ALCO}^U$ concepts, and the above entailment problem can be transformed into the satisfiability problem of the $\mathcal{ALCO}^U$ concept (abusing notation) $\mathcal{D}_{S_0} \sqcap UNA \sqcap \neg\mathcal{R}[Query]$.

3 Test Case Generation

As a means of representing a PP instance, we used XML to represent each part of the BAT and designed an XML SCHEMA to characterize the representation. After performing regression on the query of the PP instance, the input to the reasoner was represented in OWL Manchester syntax. A PP test case contains a fixed part consisting of the SSAs, PAs, and TBox axioms for a particular domain, and a varying part consisting of (1) the initial theory, (2) the query and (3) the action sequence. We obviously need to generate the variable part. Due to the non-trivial expressivity of the language at hand, it is hard to generate useful test cases. We looked in the literature and found no precedence for such an attempt, i.e. generating random projection problem test cases. Planning domains [2] had some test case generation involved, but still on purely propositional level, so that inconsistencies in input data can be easily avoided. In contrast, generating random $\mathcal{ALCO}^U$ formulas usually yields meaningless queries or initial theories that are inconsistent. Building formulas from patterns is one step forward towards generating good test cases, but it suffers from the fact that generated formulas might be similar and consequently this approach does not provide the extensive coverage that random formula generation does. We tried mixing patterns with a bit of randomness.

We have 7 domains in our disposal, and we created query generators and action sequence generators for 4 of them (lack of time is the only obstacle for the other 3). For every domain, we created a few patterns to guide in creating the query formula (6-10 per domain). Some of these patterns can take a human input, or can generate their random input if none given. A pattern for the Logistics domain might for example describe the question of whether there are any boxes on a truck X in some city Y, where X and Y is the input to the pattern. The objective is to have as versatile patterns as possible but keeping them simple because we make use of them in the next step. Next, we generate a random propositional DNF formula made up of, say n, unique literals. Then, use the patterns with random input (or guided input if used with action sequence generators) to generate n atomic queries, and replace every literal in the DNF formula with one of those queries (we map each of the n literals to a specific atomic query to avoid propositional inconsistency). The motivation for this approach is that having randomness at the propositional level is good, but not at the FOL level.

To generate random but executable action sequences, we used patterns again. But now a pattern is a generic description of a sequence of actions necessary to satisfy a goal. The pattern is represented by an algorithm that computes an executable sequence of actions based on the provided pattern. For instance, one action sequence in the Logistics domain describes the process of gathering all known boxes in a particular city and transporting them to another city. The choice of the cities is random, and picking the transportation vehicle is random as well. Basically, the generator extract all the information it can from the given initial theory, and picks its random input from the gathered information. Of course, we could have tried picking random actions with random ground arguments and check the executability of these actions, but most likely they would fail to be executable. In fact, picking an executable ground action is related to conjunctive query answering which is a totally different and nontrivial problem. Besides that, solving the executability problem will incur an expensive running time for test case generation.

It is important to note that the query and action sequence generators use the initial theory as input, so that they can generate meaningful queries and executable actions. One limitation is that only literals are used from the initial theory, assuming the initial theory is a conjunction containing some literals. This assumption simplifies significantly solving the executability problem, because the preconditions of an action can be easily verified using these literals (again, assuming the preconditions are simple enough to be verified with the information from the literals, which is true in our case).

Finally, the initial theory (IT) contains both static and dynamic incomplete knowledge. Due to lack of time and the nature of some of our metrics, we decided to manually create ITs, and did that for only one domain: Logistics. Hence, we did not make use of the generators for the other 3 domains. We created 55 unique ITs, with number of individuals per IT varying between 5 and 60 individuals. We built them starting from a small IT of size 5, and incrementally added an individual to the IT to create the next bigger IT. This way we have a monotonically ascending order of IT size. This is important because the projection queries generated from a smaller initial theory can then be run against a bigger initial theory simply because the preconditions of an action are satisfied in the bigger IT if they are satisfied in the smaller IT. This way we can better measure the effect of varying the number of individuals on the same query.

Using the ITs, and the two generators, we were able to generate at least 10 unique combinations of query + action sequence (QAS) for each IT. And by reusing QASs from smaller initial theories, we generated a pool of around 12000 test cases.

The next step is to classify the pool of test cases based on several metrics.

3.1 Classification

We use 2 metrics to classify our generated formulas, in an effort to show how the variation of values in each metric affects the reasoning time in HERMIT. The metrics are: (a) number of constants in the initial theory, and (b) number of $\mathcal{U}$-role occurrences. Metric (b) is measured using the initial theory and the regressed query formula combined, while metric (a) can be measured using the initial theory $\mathcal{D}_{S_0}$ only because $\mathcal{D}_{S_0}$ (together with UNA) defines all the individuals allowed in a PP instance. Neither the query nor regression can add new individuals to the domain, and the query and action sequence would use individuals mentioned in the initial theory.

The number of individuals in the domain is an interesting factor because in practice it would be useful to know the effect of adding more individuals on the run time of answering a projection query. Note that the regression of a QAS is the same regardless of the IT, but the ITs are increasing in size in each test case, which enables us to see the direct affect of having more individuals in a domain. For this metric, we create groups of test cases, such that each group has a single query common to all test cases, and the number of individuals in the initial theory of each test case increases monotonically.

We picked the second metric, number of $\mathcal{U}$-role occurrences, because we noticed that even test cases that have few occurrences of $\mathcal{U}$-roles slow down solving SAT for a concept. Out of curiosity, we also tried to replace all occurrences of $\mathcal{U}$-roles with some ordinary role R (we know semantics drastically change), and the reasoning time dropped by a factor of 10 or more.

There could be other possible metrics, such as the number of disjunctions, or the depth of propositional connectives, or the depth of quantifiers in the formula, but we didn't use them due to lack of time.

One last observation we made, is that the initial theory in $\mathcal{L}$ usually contains a lot of assertive formulas, i.e. of the form $A(c)$ and $R(c, b)$ for some unary predicate A, binary predicate R, and constants/individuals c and b. For instance, in the Logistics example above, we have formulas such as $city(Toronto)$ and $at(Mt1, Main, S0)$, and they appear as conjuncts in the initial theory $\mathcal{D}_{S_0}$. Note that these assertive formulas can be fluents, not just static predicates. Instead of representing these formulas from the initial theory as an $\mathcal{ALCO}^U$ concept, we may represent them as concept and role assertions in the test case. While doing this, we may expect speed-up in reasoning time as this is the more natural way of representing this sort of information in OWL. For this reason, we created yet another set of test cases which are ABox'ed versions of the original set of test cases. We call a test case ABoxed if it represents its initial theory as

OWL assertions. Note that this representation does not leave the initial theory empty, but only shortens it by removing those assertive formulas from it, and keeping the other formulas untouched. In the next section, we deal with these two sets of test cases, where SAT test cases are the regular test cases where the initial theory is represented as a complex $\mathcal{ALCO}^U$ concept with all assertive formulas included. Finally, to get UNA in OWL2 we declare all individuals to be pairwise different (using the OWL2 construct `differentIndividuals`).

4 Results

For all testing we used a machine with the following specs: Intel® Core™ 2 Duo E8400 CPU with a clock frequency of 3.00 GHz, and 4 GB of RAM. We used JVM version 1.7.0 and HermiT 1.3.6. We used a cutoff time of 30 sec. All results and test-cases can be found at: `http://www.cse.yorku.ca/~w2yehia/ORE_results.html`

4.1 Number of Constants

We already explained how we measure this metric, and how our choice and construction of ITs is suitable for the purpose of testing this metric. In the two graphs below, we show the reasoning time taken by HERMIT as a function of the number of individuals in the initial theory.

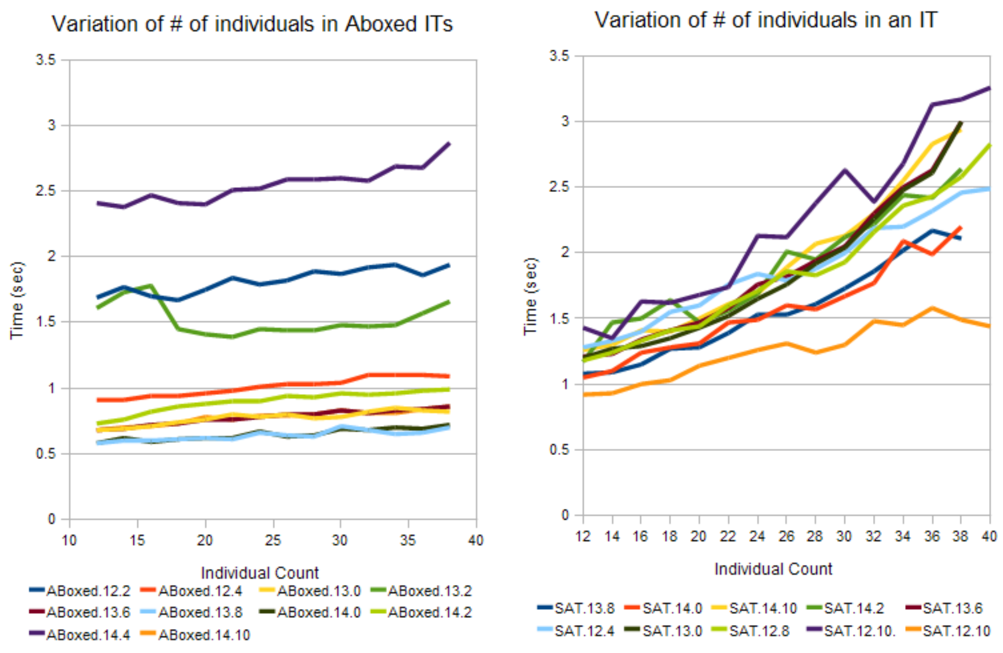

Fig. 1.

The left graph shows the running time of ABoxed test cases, and the right graph shows the regular non-ABoxed test cases. We chose to graph the part of our testing results where all test cases in a group finished in under the 30 seconds cutoff time. The groups of test cases shown below are labeled with the prefix 'ABoxed' and 'SAT', followed by the individual count in the initial theory that spawned the QAS for that particular group, and the last number is just an index of the QAS for the particular individual count.

4.2 Number of $\mathcal{U}$-role Occurrences

For this metric, we counted the number of $\mathcal{U}$-role occurrences in a test case (tc). Then, we grouped the test cases into sets, with a range of allowed number of $\mathcal{U}$-role occurrences per set. For better granularity, the ranges are narrow for small number of $\mathcal{U}$-roles, and widen as the number grows. Table 1 shows the ranges that we used in the first column, the number of test cases tested for that range (some ranges had > 1000 test cases, so we picked the first 100 for each range) in the second column, and the number of test cases that was answered in under 30 sec in the third. The fourth column simply shows the success rate for that range. The next three columns show the same as the previous three but for ABoxed test cases.

Table 1. Testing Results based on $\mathcal{U}$-role number of occurrences

# of U-role occur.	# of SAT tcs tested	# of successful SAT tcs	success rate (%)	# of ABoxed tcs tested	# of successful ABoxed tcs	success rate (%)
0-5	0	0	-	100	98	98
6-10	0	0	-	100	89	89
11-15	10	10	100	100	60	60
16-20	25	25	100	100	67	67
21-25	57	57	100	100	13	13
26-30	100	100	100	100	33	33
31-35	100	100	100	80	31	39
36-40	100	100	100	100	82	82
41-50	100	99	99	100	44	44

It is important to note that for all test cases, doing regression alone takes in most cases 1 second or less, and at most 2 seconds. Thus, most of the time spent on solving a PP is in HERMIT.

5 Discussion and Future Work

For the number of constants in the initial theory, it is clear that ABoxed test cases are less affected by the increase in individual count as compared to the non-ABoxed test cases. This shows that ABox-ing an initial theory is the more efficient way of representation when it comes to HERMIT.

For the number of $\mathcal{U}$-role occurrences, we believe that the reason why ABoxed test-cases seem to run much slower on average (lower success rate), is because of the way we translate an assertion into a concept - we use $\mathcal{U}$-roles. So, the ABoxed version will contain less $\mathcal{U}$-roles coming from the initial theory, than the regular version, simply because the assertions turned into concepts using $\mathcal{U}$-roles in the regular version, become OWL assertions in the ABoxed version. As a consequence, the remaining $\mathcal{U}$-roles come from the regressed formula so that the total number of $\mathcal{U}$-roles is same (recall the number of $\mathcal{U}$-role occurrences is counted both in the initial theory and in the regressed formula). For example, if a test-case had 10 $\mathcal{U}$-roles coming from the initial theory and 5 $\mathcal{U}$-roles from the regressed formula, then the ABoxed version will contain 5 $\mathcal{U}$-roles only, because the initial theory $\mathcal{U}$-roles disappear as a result of ABoxing, but SAT (non-ABoxed) version will contain 15 (5+10) $\mathcal{U}$-roles. This means that you cannot compare directly a SAT test-case with n occurrences of $\mathcal{U}$-roles to an ABoxed test-case with n occurrences of $\mathcal{U}$-roles because they represent different test-cases. You should expect that the ABoxed one will run slower because it has a bigger regressed formula to handle (this is where the extra $\mathcal{U}$-roles come from).

Finally, the long reasoning time in HERMIT can be attributed to either (a) inefficient representation of the regression formula when outputted by our regression program, or (b) slow performance of HERMIT when it comes to dealing with $\mathcal{U}$-roles.

Future work: We started with 7 domains, and wrote generators for 4 of them, and manually created initial theories for one domain. So there is some work left to be done, at least to make use of the generators for the 3 domains. We only managed to measure the performance of HERMIT, but the generated test cases are in Manchester syntax, and any reasoner using OWL API can run those test cases. We mentioned already that more metrics can be measured, and the effects of ABox'ing can be studied in more details.

References

1. Baader, F., Lippmann, M., Liu, H., Soutchanski, M., Yehia, W.: Experimental Results on Solving the Projection Problem in Action Formalisms Based on Description Logics. In: Description Logics W/sh DL-2012 (Accepted). (2012)
2. Bacchus, F.: The AIPS '00 planning competition. AI Magazine **22**(3) (2001) 47–56
3. Devanbu, P.T., Litman, D.J.: Taxonomic plan reasoning. Artif. Intell. **84**(1-2) (1996) 1–35
4. Gu, Y., Soutchanski, M.: A Description Logic Based Situation Calculus. Ann. Math. Artif. Intell. **58**(1-2) (2010) 3–83
5. Kudashkina, E.: An Empirical Evaluation of the Practical Logical Action Theory (Undergraduate Thesis CPS40A/B, Fall 2010 - Winter 2011). Department of Computer Science, Ryerson University, Toronto, Ontario, Canada (2011)
6. McDermott, D.V.: The 1998 AI Planning Systems Competition. AI Magazine **21**(2) (2000) 35–55
7. Reiter, R.: The projection problem in the situation calculus: A soundness and completeness result, with an application to database updates. In: In Proceedings First International Conference on AI Planning Systems, Morgan Kaufmann (1992) 198–203
8. Reiter, R.: Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems. The MIT Press (2001)
9. Yehia, W., Soutchanski, M.: Towards an Expressive Logical Action Theory. In: Proc. of the 25th Intern. Workshop on Description Logics (DL-2012), Rome, Italy (2012) to appear