Should I Stay or Should I Go
A New Reasoner for Description Logic

Aaron Eberhart[1,2], Joseph Zalewski[1], Pascal Hitzler[1]

[1] DaSe Lab, Kansas State University, Manhattan KS 66506, USA
[2] metaphacts GmbH, 69190 Walldorf, Germany
aaron.eberhart@gmail.com

Abstract. We present the Emi reasoner, based on a new interpretation of the tableau algorithm for reasoning with Description Logics with unique performance characteristics and specialized advantages. Emi turns the tableau inside out, solving the satisfiability problem by adding elements to expressions rather than adding expressions to element node labels. This strategy is inspired by decidable reasoning algorithms for Horn Logics and $\mathcal{EL}^{++}$ that run on a loop rather than recursive graph-based strategies used in a tableau reasoner. Because Emi solves the same problem there will be a simple correspondence with tableaux, yet it will feel very different during execution, since the problem is inverted. This inversion makes possible many unique and straightforward optimizations, such as paralellization of many parts of the reasoning task, concurrent ABox expansion, and localized blocking techniques. Each of these optimizations contains a design trade-off that allows Emi to perform extremely well in certain cases, such as instance retrieval, and not as well in others. Our initial evaluations show that even a naive and largely unoptimized implementation of Emi is performant with popular reasoners running on the JVM such as Hermit, Pellet, and jFact.

1 Introduction

Knowledge graph schema are complex artifacts that can be very useful, but are often difficult and expensive to produce and maintain. This is especially true when encoding them in OWL (the Web Ontology Language) as ontologies for data management or reasoning. The high expressivity of OWL is a boon, in that it makes it possible to describe complex relationships between classes, roles,[3] and individuals in an ontology. At the same time, however, this high expressivity is often an obstacle to its correct usage that can limit adoption, and can hamper any practical reasoning applications by adding complexity to the reasoning process.

To manage the complexity of OWL some prefer to accept hardness as it is and develop tools to manage or simplify the tricky parts. However, it occasionally is the case that problems appear difficult when they are actually quite intuitive when expressed differently. When this happens it can be helpful to start again

[3] We refer to properties as **roles**, unless a distinction is relevant, as this is the standard description logic term. These include both object properties and data properties.

from the beginning and re-imagine what is possible by trying something entirely different. This will of course not alter the fundamental proven computational complexity of any reasoning or modeling problem. But it can lead to algorithms and design patterns that are more easily understandable, and potentially uncover unique use cases and optimizations.

In this spirit we have used a new API for OWL called (f OWL) [2] for our experiment that can represent and better facilitate ontology data for the reasoning tasks we want it to perform. A custom reasoning algorithm called Emilia[4] (Emi) was implemented specifically to leverage the unique advantages of the new API, and with time the two will be able to seamlessly work within a single framework. The Emi reasoner will be able solve all of the same problems that current reasoners are able to solve, but its execution follows an iterative rule-like path rather than recursive tableau.

This type of experiment may seem equivalent and redundant to logicians and mathematicians, and looking at it only in a purely formal sense this can seem to be the case, however Emi works entirely differently from current systems and presents many opportunities for new research. The reasoner presented here is a prototype which demonstrates that the technique is valid and no less efficient than other comparable Java Virtual Machine (JVM) reasoners – the potential for new unique research directions and optimization techniques using this method will be the subject of future works. Initial testing is underway of Emi, and it is already performing competitively with other state-of-the-art systems in use today, and is particularly efficient with instance retrieval, despite being a naive prototype system written in a high level language with only the most basic and straightforward of optimizations.

2 $\mathcal{ALCH}$

The Emi algorithm currently supports $\mathcal{ALCH}$ reasoning, and the syntax and semantics of that logic are given below. Future extensions will likely expand expressivity for additional description logics.

2.1 Syntax

The signature Σ for the Description Logic $\mathcal{ALCH}$ is defined as $\Sigma = \langle N_I, N_C, N_R \rangle$ where:

- N_I is a set of individual element names.
- N_C is a set of class names that includes $\top$ and $\bot$.
- N_R is a set of role names.
- N_I, N_C, N_R are pairwise disjoint

Expressions in $\mathcal{ALCH}$ use the following grammar:

[4] **E**ager **M**aterializing and **I**terating **L**ogical **I**nference **A**lgorithm

$\mathbf{R} ::= N_R$
$\mathbf{C} ::= N_C \mid \neg\mathbf{C} \mid \mathbf{C} \sqcap \mathbf{C} \mid \mathbf{C} \sqcup \mathbf{C} \mid \exists\mathbf{R}.\mathbf{C} \mid \forall\mathbf{R}.\mathbf{C}$

2.2 Semantics

An interpretation $\mathcal{I} = \left(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}\right)$ maps $N_I,\ N_C,\ N_R$ to elements, sets, and relations in $\Delta^{\mathcal{I}}$ with function $\cdot^{\mathcal{I}}$. An axiom A in $\mathcal{ALCH}$ is satisfiable if there is an interpretation where $\cdot^{\mathcal{I}}$ maps all elements, sets, and relations in A to $\Delta^{\mathcal{I}}$. An ontology O is a set of axioms formed from $\mathcal{ALCH}$ expressions, and is satisfiable if there is an interpretation that satisfies all axioms it contains, this interpretation being a model for O. $\cdot^{\mathcal{I}}$ is defined in Table 1 below.

Table 1. $\mathcal{ALCH}$ Semantics

Description	Expression	Semantics
Individual	x	$x^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
Top	$\top$	$\Delta^{\mathcal{I}}$
Bottom	$\bot$	$\emptyset$
Class	B	$B^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
Role	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Negation	$\neg B$	$\Delta^{I} \setminus B^{\mathcal{I}}$
Conjunction	$B \sqcap C$	$B^{\mathcal{I}} \cap C^{\mathcal{I}}$
Disjunction	$B \sqcup C$	$B^{\mathcal{I}} \cup C^{\mathcal{I}}$
Existential Restriction	$\exists R.B$	$\{\ x \mid$ there is $y \in \Delta^{\mathcal{I}}$ such that $(x,y) \in R^{\mathcal{I}}$ and $y \in B^{\mathcal{I}}\ \}$
Universal Restriction	$\forall R.B$	$\{\ x \mid$ for all $y \in \Delta^{\mathcal{I}}$ where $(x,y) \in R^{\mathcal{I}}$, we have $y \in B^{\mathcal{I}}\ \}$
Class Assertion	$B(a)$	$a^{\mathcal{I}} \in B^{\mathcal{I}}$
Role Assertion	$R(a,b)$	$(a,b) \in R^{\mathcal{I}}$
Negated Role Assertion	$\neg R(a,b)$	$(a,b) \notin R^{\mathcal{I}}$
Class Subsumption	$B \sqsubseteq C$	$B^{\mathcal{I}} \subseteq C^{\mathcal{I}}$
Class Equivalence	$B \equiv C$	$B^{\mathcal{I}} = C^{\mathcal{I}}$
Role Subsumption	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$

3 (f OWL)

We use (f OWL)[2] because it includes many novel optimizations that streamline ontology and reasoner development. True to the functional paradigm (f OWL) uses functions and shuns classes. Indeed all OWL ontology objects can be created directly with (f OWL) functions. And since these functions are not bound up in an arbitrary class hierarchy, they can operate on independent data structures that are internally typed to represent OWL semantics. A (f OWL) ontology

itself is simply another data structure made of collections of smaller similar structures. This means that in most cases an expression, an axiom, even an ontology can be traversed recursively as-is without writing more functions. The use of immutable data structures by (f OWL) in standard Clojure style also permits straightforward implementation of concurrent processes that operate on ontology data.

3.1 (f OWL) and Emi

Emi makes use of many features of (f OWL), some of which are uniquely advantageous in that they have no comparable alternative in other reasoners or APIs. One major advantage with using (f OWL) is the ontology-as-data-structure approach, which means that once (f OWL) has loaded the ontology into memory, Emi can efficiently traverse and manipulate expressions in the ontology without worrying about concurrency, since a (f OWL) ontology is immutable. Once the ontology is processed and elements in the signature are assigned mutable memory for use during reasoning, it is straightforward to define a partially paralellizable reasoning process with Clojure built-in functions that can be precisely lazy or eager when needed. The optimal configuration for when to choose each strategy is subject to many design trade-offs and does not have an objective best answer, though we believe we have developed a decent strategy. Emi processes axioms eagerly, but sets of axioms lazily and in parallel when possible due to the general observation that many non-synthetic ontologies contain a large number of axioms, most of which are rather small. This can affect performance on synthetic datasets where the occurrence of large or complex axioms is potentially higher and a different strategy may work better.

4 Emi Reasoner

As mentioned in the introduction, the new reasoner we are developing works iteratively without a graph, and is more like reasoning in $\mathcal{EL}^{++}$ or datalog where the process runs on a loop that eventually terminates rather than an expansive graph traversal. To reason iteratively we effectively need to turn the tableau inside out so we can work directly with axioms instead of a graph of elements. This means we will need to look at each axiom individually and add elements to or remove elements from expressions according to the tableau expansion rules as if they were backwards. This process slowly builds up a model for the ontology by partially realizing expressions until they become satisfiable and no longer need to be modified. Since every element is inspected for every axiom and the inverted tableau rules are followed in expressions we know this will be consistent. We assume that all expressions in our algorithm will be converted into Negation Normal Form (NNF); it is important to emphasize that absolutely no other logical preprocessing is used in the algorithm. In this section we assume that readers are familiar with the basics of tableau reasoning, if not they can consult [1] for further information.

4.1 Partial Interpretations

This algorithm works by directly attempting to realize an ontology and eventually build a consistent interpretation using the axioms as they are written in NNF, by adding and removing elements until all axioms are satisfiable or the ontology is shown to be unsatisfiable. We refer to the changing states of a program as partial interpretations.

Before continuing to the definition, it is important to stop and emphasize the subtle difference between partial interpretations and standard interpretations. A partial interpretation *may* be equivalent to some interpretation, and indeed when Emi terminates and determines we have a satisfiable ontology this final partial interpretation is a model. However partial interpretations are not necessarily models and represent the states of the program as different assignments are attempted in order to find a model. This difference may be confusing at first for logicians who are used to only seeing standard interpretations, but the distinction is important not just for showing a simple proof but also for describing the actual implementation of the algorithm as well. Our terminology deliberately corresponds much more closely to how an actual implemented algorithm, rather than a theoretical proof of one, would function by avoiding whenever possible notions of infinity or concepts that would need to be represented by global variables that are pervasive in proofs and inconvenient or impossible to implement. We do this intentionally so that it can be understood by programmers as well as logicians; we hope not just to show correctness but that it is clear to anyone how they could actually write a reasoner such as this.

Definition 1. *A* partial interpretation *representing the current state of the algorithm when a function is called is* $\mathcal{I}^* = (\mathcal{R}^{\mathcal{I}^*}, \cdot^{\mathcal{I}^*})$ *where* $\cdot^{\mathcal{I}^*}$ *is a function that maps elements, sets, and relations in an ontology* O *to a realization* $\mathcal{R}^{\mathcal{I}^*}$ *of* O*.*

Definition 2. *A* realization *of ontology* O *is a set of assertions that states for every class name* A *and element* x *in* O *that* $x \in A^{\mathcal{I}^*}$ *or* $x \notin A^{\mathcal{I}^*}$ *and for every role name* R *and element pair* (x, y) *in* O *that* $(x, y) \in R^{\mathcal{I}^*}$ *or* $(x, y) \notin R^{\mathcal{I}^*}$*.*

Each partial interpretation in this algorithm corresponds to some realization with a function $\cdot^{\mathcal{I}^*}$. Note that there is no restriction against inconsistent realizations and partial interpretations, only that they must be complete. A partial interpretation $\mathcal{I}^*$ with realization $\mathcal{R}^{\mathcal{I}^*}$ of ontology O is said to be a model of O iff there is an interpretation $\mathcal{I}$ of $O \cup \mathcal{R}^{\mathcal{I}^*}$.

As mentioned previously, while $\mathcal{I}$ often denotes a single consistent interpretation, $\mathcal{I}^*$ denotes the *current interpretation* when a function is evaluated in the algorithm, and as such may change over time and contain information that is later proved to be incorrect. Frequently we will say things like "Add x to $E^{\mathcal{I}^*}$", and this indicates that the partial interpretation $\mathcal{I}^*$ will henceforth be modified in the program state to represent the described change to $\cdot^{\mathcal{I}^*}$ that produces the desired realization.

4.2 Inverted Tableau Expansion Rules

Inverting the tableau rules is straightforward when we turn the standard tableau terminology, such as occurs in [1,6], on its head and assume that "labels" represent the action of the $\cdot^{\mathcal{I}^*}$ function on class and role names in an ontology to produce a realization, and that for any class or role name E that $E^{\mathcal{I}^*} = \mathcal{L}(E)$. To connect this idea with labelling terminology in a graph we use a labelled object, which can represent both notions.

Definition 3. *A* labelled object E *is an object that is associated with some set* $\mathcal{L}(E)$*, or* label*.*

First we note that nodes and labels do not necessarily need to be connected in a graph to represent the semantics of $\mathcal{ALCH}$, and can be reinterpreted as freely associating labelled objects. A labelled object can be a node in a tableau graph or simply an isolated named object with no inherent graph connections. This means that a label for a complex expression can be defined recursively to match the semantics of a partial interpretation $\mathcal{I}^*$, e.g. when an expression $C \sqcup D$ is satisfiable after applying the inverted tableau rules we have $\mathcal{L}(C \sqcup D) = (C \sqcup D)^{\mathcal{I}^*} = C^{\mathcal{I}^*} \cup D^{\mathcal{I}^*}$. We can represent axioms equivalently as labelled objects without actually needing to make a graph as long as all names in the signature are not duplicated in expressions but actually reference the same labelled objects.

For this algorithm we consider the inverted $\mathcal{ALC}$ tableau rules sufficient for deciding $\mathcal{ALCH}$ satisfiability, since the addition of the $\mathcal{H}$ fragment does not permit complex role expressions and they do not require expansion to check. The RBox axioms that are not included here will be described in Table 4 with the TBox axioms. In Table 2 we show the tableau expansion rules for $\mathcal{ALC}$ [1] and assume the standard notion of blocking for them, which can be found with the original proofs, then show the inversions in Table 3 where blocking is defined in Subsection 4.4.

	the following expansion rules apply to element x when x is not blocked
$\sqcap$-rule	if 1. $(C \sqcap D) \in \mathcal{L}(x)$
	2. $\{C, D\} \notin \mathcal{L}(x)$
	then $\mathcal{L}(x) \to \mathcal{L}(x) \cup \{C, D\}$
$\sqcup$-rule	if 1. $(C \sqcup D) \in \mathcal{L}(x)$
	2. $\{C, D\} \cap \mathcal{L}(x) = \emptyset$
	then either $\mathcal{L}(x) \to \mathcal{L}(x) \cup \{C\}$
	or else $\mathcal{L}(x) \to \mathcal{L}(x) \cup \{D\}$
$\exists$-rule	if 1. $\exists R.C \in \mathcal{L}(x)$
	2. there is no y s.t. $\mathcal{L}((x, y)) = R$ and $C \in \mathcal{L}(x)$
	then create a new node y and edge (x, y) with $\mathcal{L}(y) = \{C\}$ and $\mathcal{L}((x, y)) = R$
$\forall$-rule	if 1. $\forall R.C \in \mathcal{L}(x)$
	2. there is some y s.t. $\mathcal{L}((x, y)) = R$ and $C \notin \mathcal{L}(x)$
	then $\mathcal{L}(y) \to \mathcal{L}(y) \cup \{C\}$

Table 2. $\mathcal{ALC}$ Tableau Expansion Rules

$\sqcap$-rule	if 1.	$x \in \mathcal{L}(C \sqcap D)$
	2.	$x \notin \mathcal{L}(C) \cap \mathcal{L}(D)$
	then	$\mathcal{L}(C) \rightarrow \mathcal{L}(C) \cup \{x\}$
		$\mathcal{L}(D) \rightarrow \mathcal{L}(D) \cup \{x\}$
$\sqcup$-rule	if 1.	$x \in \mathcal{L}(C \sqcup D)$
	2.	$x \notin \mathcal{L}(C) \cup \mathcal{L}(D)$
	then either	$\mathcal{L}(C) \rightarrow \mathcal{L}(C) \cup \{x\}$
	or else	$\mathcal{L}(D) \rightarrow \mathcal{L}(D) \cup \{x\}$
$\exists$-rule	if 1.	$x \in \mathcal{L}(\exists R.C)$
	2.	there is no y s.t. $(x, y) \in \mathcal{L}(R)$ and $y \in \mathcal{L}(C)$
	3.	there is no z s.t. x is blocked by z
	then	create a new element y with $y \in \mathcal{L}(C)$ and $(x, y) \in \mathcal{L}(R)$
$\forall$-rule	if 1.	$x \in \mathcal{L}(\forall R.C)$
	2.	there is some y s.t. $(x, y) \in \mathcal{L}(R)$ and $y \notin \mathcal{L}(C)$
	then	$\mathcal{L}(C) \rightarrow \mathcal{L}(C) \cup \{x\}$

Table 3. Inverted $\mathcal{ALC}$ Tableau Expansion Rules

The labelling terminology is used in this subsection to show tableau correspondence, though in general we avoid it since it is more obvious what is happening to non-logicians when we show our algorithm as an implementable process that produces interpretations, rather than a convoluted mathematical abstraction. Specifically, we believe it is more natural to think of a predicate that represents a set as a labelled object where the label represents the set associated with the predicate, than it is to invert this and represent all the elements as labels for connected sets of predicate names.

4.3 ABox Expansion

One of the more useful optimizations that this algorithm permits is the expansion of ABox axioms into many additional, unstated, expressions which must hold in every possible model. This expansion begins while the algorithm evaluates the ABox and detects any immediate clashes. Later while evaluating the TBox the expansion detects any clashes that cannot possibly be fixed to allow the algorithm to terminate quickly, and it can be done concurrently whenever the algorithm adds or removes elements from expressions. In Definition 4 we maintain the assumption that all expressions are NNF and use $\neg$ to simplify the appearance of equivalent expressions.

Definition 4. *For partial interpretation $\mathcal{I}^*$, expression E, and element/pair x:*

1. *A* clash *means $x \in E^{\mathcal{I}^*}$ and $x \in \neg E^{\mathcal{I}^*}$*
2. *x* must be in *E iff there is no $\mathcal{I}^*$ where modifying $\cdot^{\mathcal{I}^*}$ to obtain $x \in \neg E^{\mathcal{I}^*}$ does not cause a clash*
3. *x* must not be in *E iff there is no $\mathcal{I}^*$ where modifying $\cdot^{\mathcal{I}^*}$ to obtain $x \in E^{\mathcal{I}^*}$ does not cause a clash*
4. *An* unresolvable clash *means x must be in $E^{\mathcal{I}^*}$ and x must not be in $E^{\mathcal{I}^*}$*

The set of all elements that must be in an expression $E^{\mathcal{I}^*}$, denoted $E_m^{\mathcal{I}^*}$, is a subset of the elements of $E^{\mathcal{I}^*}$, and the set of all elements that must not be in $E^{\mathcal{I}^*}$, written $E_{\overline{m}}^{\mathcal{I}^*}$, is a subset of the elements of $\neg E^{\mathcal{I}^*}$. Like the expressions themselves, the exact members of $E_m^{\mathcal{I}^*}$ and $E_{\overline{m}}^{\mathcal{I}^*}$ are unknown at the beginning of the algorithm, so $E_m^{\mathcal{I}^*}$ and $E_{\overline{m}}^{\mathcal{I}^*}$ indicate the additional knowledge of constraints on satisfiability that grow as the algorithm runs and are thus referred to with the $\mathcal{I}^*$ as well. When we talk about element(s) being "added" to these sets (nothing is ever removed), it is to indicate the change in $\cdot^{\mathcal{I}^*}$ that must hold in all future $\mathcal{I}^*$. Sets of elements that must (not) be in a complex expression can be computed in a straightforward way from the components of the expression they are in, for example $(B \sqcap C)_m^{\mathcal{I}^*} \equiv B_m^{\mathcal{I}^*} \sqcap C_m^{\mathcal{I}^*}$ and $(B \sqcap C)_{\overline{m}}^{\mathcal{I}^*} \equiv B_{\overline{m}}^{\mathcal{I}^*} \sqcup C_{\overline{m}}^{\mathcal{I}^*}$ (note the use of DeMorgan for $\overline{m}$).

Expansion occurs during TBox evaluation by, whenever possible, propagating the elements that must (not) be in expressions from antecedent to consequent, e.g. if we have axioms $\{A(x), A \sqsubseteq B\}$ then $x \in A_m$, and when we check $A \sqsubseteq B$ it is often possible to conclude $x \in B_m$. This is not always possible for expressions containing disjunction and negation except in certain very specific cases. Regardless the effect is powerful. Maintaining these sets allows us to explore only potentially correct solutions by preemptively avoiding actions that will be inconsistent in every model and also detect unresolvable clashes so the evaluation can terminate more quickly.

4.4 Local Blocking

The notion of blocking is also required for termination due to the new elements[5] created by the existential function. Our notion of blocking corresponds to the usual definition in that it references the same cyclic patterns in roles, however Emi cannot use blocking in reference to the global role hierarchy which is not computed explicitly, so cycles are detected locally by tracing the dependency paths that emerge as new elements are created to satisfy expressions.

Definition 5. *A new element x was* created to satisfy *expression $\exists R.B$ in an algorithm A if there are partial interpretations $\mathcal{I}^*, \mathcal{I}^{*\prime}$ for A where for some y we have $(y, x) \in R^{\mathcal{I}^*}, x \in B^{\mathcal{I}^*}, y \in \exists R.B^{\mathcal{I}^*}$ and $y \notin \exists R.B^{\mathcal{I}^{*\prime}}$ if $x \notin \Delta^{\mathcal{I}^{*\prime}}$.*

Definition 6. *For expression $\exists R.B^{\mathcal{I}^*}$, an element x is* blocked *by element z if $(z, x) \in R^{\mathcal{I}^*}$ and x was created to satisfy $\exists R.B$, or if for $(z, y_0) \in R^{\mathcal{I}^*}$ where y_0 was created to satisfy $\exists R.B$ we have $n \geq 0$ pairs of elements such that $\bigcup_{k=1}^{n}(y_{k-1}, y_k) \cup \{(y_n, x)\} \subseteq \Delta^{\mathcal{I}^*} \times \Delta^{\mathcal{I}^*}$.*

Fortunately this restriction is rather intuitive to implement: we simply need the function for an existential to not create new elements when checking new elements that exist as a result of a prior application of the same function on

[5] When we say 'new element' this is equivalent to a fresh element symbol. It is written this way because we are avoiding terminology that refers to an infinite set of names and instead refer to what a program really does here, i.e. create something new

the same existential, since this will induce a cycle. It is effectively as if elements have a 'history' represented by the chain of pairs that they inherit from the element that created them and which also contains their own origin. An element will therefore not generate new elements in a function with the existential that created it, and not if the function for the existential created an element that it depends on for its existence, or that is in its 'history'.

4.5 Termination Condition

The final component necessary for the algorithm to work is a termination condition. Unlike a tableau, this algorithm will run on a loop and will actually attempt to directly realize the ontology. How then do we know that the algorithm has correctly realized all of the axioms? This is actually rather simple. If the algorithm begins an iteration of checking all axioms with initial state $\mathcal{I}_a^*$ and ends this iteration with state $\mathcal{I}_b^*$ without encountering any unresolvable clashes, then we know it can terminate successfully if $\mathcal{I}_a^* = \mathcal{I}_b^*$. Since we know that each $\mathcal{I}^*$ represents a possible interpretation, it is clear that $\mathcal{I}^*$ is a model when no clash occurs after every axiom is checked and all elements remain the same in all expressions.

Definition 7. *A partial interpretation $\mathcal{I}^*$ is said to* equal *partial interpretation $\mathcal{I}^{*\prime}$, or $\mathcal{I}^* = \mathcal{I}^{*\prime}$, iff $\forall E \in N_C \cup N_R$ we have $E^{\mathcal{I}^*} = E^{\mathcal{I}^{*\prime}}$.*

Equality can be verified by checking if the names in the signature have not changed any of their memberships. If an element has been added to the signature, or has moved into or out of any class or role, the algorithm must check the axioms one more time to see if these changes cause side effects. Otherwise the algorithm will have checked every axiom and found them to be satisfiable without making any changes and has produced a model.

4.6 Algorithm Definition

An outline of Emi is shown in Algorithm 1 that uses functions from Table 4.

Backtracking Explicit backtracking in this algorithm is handled by 'reporting' clashes and unresolvable clashes.

Definition 8. *A* report *for Algorithm 1 immediately terminates execution of whatever process is occurring and returns to the function that handles this report.*

A report is meant to act like a programming exception. For example, when an unresolvable clash is directly reported while evaluating the ABox the behavior is clear, the ontology is unsatisfiable so all reasoning stops and the algorithm immediately exits by returning False.

Standard clashes occur when the inverted tableau rules require an action that is known to be inconsistent, but which could potentially be resolved by making changes elsewhere. In this case the immediate action is to stop trying the

obviously inconsistent task and instead do something different. This type of clash is normally handled in Emi by the code that solves the inverted tableau rules in the same way that a standard tableau might. However, there are cases where the *sat* function must intervene, for instance when the antecedent of an axiom can neither lack nor contain an element and both assignments have been attempted to exhaustion. In this case we have indirectly found an unresolvable clash. If a clash is reported but is able to be resolved, the algorithm merely proceeds along as normal, and any problems that a removal causes will themselves be reported and dealt with in the same way. Emi internally ensures that backtracking does not return to an identical previous program state so that termination is not a concern. ABox expansion greatly impacts this functionality by automatically removing many impossible solutions from consideration, allowing these reports to happen faster.

Parallelization Because we are not working with one single expanded concept representing the entire set of axioms like a standard tableau, it is possible in this algorithm to parallelize many operations on separate axioms. In the simplest case, it is unproblematic for the algorithm to process more than one axiom at the same time so long as they do not share any class or role names. Clashes that occur as a result of two independent axioms will in either case still be detected elsewhere and are dealt with regardless of the ordering. Additionally, this idea can be extended in the implementation to allow for axioms that share names to be evaluated concurrently as well, so long as shared names are not modified in ways that can cause a clash and begin backtracking. Emi makes use of the simplest case already and the implementation of more complex parallelization is in development.

Table 4. *sat* function behavior for $\mathcal{ALCH}$ satisfiability of ontology O, class name A, class expressions B, C, and roles R, S

Axiom	Action
$A(a)$	If $a \in A_{\overline{m}}^{\mathcal{I}^*}$ report an unresolvable clash, otherwise add a to $A^{\mathcal{I}^*}$ and $A_m^{\mathcal{I}^*}$
$\neg A(a)$	If $a \in A_m^{\mathcal{I}^*}$ report an unresolvable clash, otherwise add a to $A_{\overline{m}}^{\mathcal{I}^*}$
$B(a)$	Add a new class A and the axiom $A \sqsubseteq B$ to O and replace $B(a)$ with $A(a)$
$R(a,b)$	If $(a,b) \in R_{\overline{m}}^{\mathcal{I}^*}$ report an unresolvable clash, otherwise add (a,b) to $R^{\mathcal{I}^*}$ and $R_m^{\mathcal{I}^*}$
$\neg R(a,b)$	If $(a,b) \in R_m^{\mathcal{I}^*}$ report an unresolvable clash, otherwise add (a,b) to $R_{\overline{m}}^{\mathcal{I}^*}$
$B \sqsubseteq C$	For all $x \in B^{\mathcal{I}^*}$ add x to C using the inverted tableau expansion rules. If a clash is reported, instead bactrack to remove x from B. If both report a clash, report an unresolvable clash.
$B \equiv C$	Do $sat(B \sqsubseteq C)$ and $sat(C \sqsubseteq B)$ and report any unresolvable clashes.
$R \sqsubseteq S$	For all $(x,y) \in R^{\mathcal{I}^*}$ add (x,y) to S. If a clash is reported, instead backtrack to remove (x,y) from R. If both report a clash, report an unresolvable clash.

Algorithm 1: $\mathcal{ALCH}$ satisfiability for ontology O

Result: A realization that is a model of O when $True$, otherwise $False$ when an unresolvable clash occurs

```
// Initialize an empty set to represent a realization for O
```
$\mathcal{I}^* \leftarrow \{\}$;

```
// Assume all E/E_m/E_m̄ are empty and add them to I*
```
for $E \in N_C \cup N_R$ **do**
  $E \leftarrow \{\}$;
  $E_m \leftarrow \{\}$;
  $E_{\overline{m}} \leftarrow \{\}$;
  $\mathcal{I}^* \leftarrow \mathcal{I}^* \cup \{E, E_m, E_{\overline{m}}\}$;

```
// Obtain NNF of all axioms
```
$\mathcal{F} \leftarrow$ NNF of ABox of O;
$\mathcal{A} \leftarrow$ NNF of TBox of O;

```
// Process ABox
```
for $F \in \mathcal{F}$ **do**
  $sat(F)$;
  if *an unresolvable clash is reported* **then**
    return *False*

```
// Process TBox
```
repeat
  $\mathcal{I}^*_{old} \leftarrow \mathcal{I}^*$;
  for $A \in \mathcal{A}$ **do**
    $sat(A)$;
    if *an unresolvable clash is reported* **then**
      return *False*
until $\mathcal{I}^* = \mathcal{I}^*_{old}$;
return *True*

4.7 Correctness

The correctness of this algorithm follows from the direct correspondence between the tableau rules and their inversions along with a few minor details. First we will examine the inverted tableau rules. Each rule contains two distinct states in both the standard and inverted rules, the antecedent 'if' clause where certain conditions must hold, and the consequent 'then' clause where changes are made to the state in order to find a model. Thus it will be sufficient to show that each 'if' and 'then' clause from the inverted tableau rules is re-writable into the corresponding clause in the standard tableau rules.

$\sqcap$-rule

'if' This state in the inverted tableau rules corresponds to the standard tableau rules, since in each case we know x should be in $C \sqcap D$, but it is either not in C or D or both.

'then' The change made in response to the 'if' clause is also equivalent, since the inversion adding x to $C^{\mathcal{I}^*}$ and $D^{\mathcal{I}^*}$ represents the notion that we now have $x \in C^{\mathcal{I}^*}$ and $x \in D^{\mathcal{I}^*}$ which is identical to the standard tableau rules adding $\{C, D\}$ to $\mathcal{L}(x)$.

$\sqcup$-rule

'if' This state in the inverted tableau rules corresponds to the standard tableau rules, since in each case we know x should be in $C \sqcup D$, but it is neither in C nor in D.

'then' The change made in response to the 'if' clause is also equivalent, since the inversion adding x to $C^{\mathcal{I}^*}$ or $D^{\mathcal{I}^*}$ represents the notion that we now have $x \in C^{\mathcal{I}^*}$ or $x \in D^{\mathcal{I}^*}$ which is identical to the standard tableau rules either adding $\{C\}$ to $\mathcal{L}(x)$ or adding $\{D\}$ to $\mathcal{L}(x)$.

$\exists$-rule

'if' This state in the inverted tableau rules corresponds to the standard tableau rules, since in each case we know x should be in $\exists R.C$, but the current state means that there is either no $(x, y) \in R$ such that $y \in C$ or there is no $y \in C$ such that $(x, y) \in R$. The blocking condition (3) only applies when x is blocked because it is a new individual that exists as a (possibly indirect) result of an element previously created to satisfy this expression. When x is blocked in this way it is clear that we have completed the first iteration of a cycle and do not need to continue.

'then' The change made in response to the 'if' clause is also equivalent, since the inversion creating some y such that $y \in C^{\mathcal{I}^*}$ and $(x, y) \in R^{\mathcal{I}^*}$ represents the notion that $x \in \exists R.C$, which is identical to the standard tableau rules creating a new node and edge for y with $\mathcal{L}(y) = \{C\}$ and $\mathcal{L}((x, y)) = R$.

$\forall$-rule

'if' This state in the inverted tableau rules corresponds to the standard tableau rules, since in each case we know x should be in $\forall R.C$, but the current state means that there is some y such that $(x, y) \in R$ and $y \notin C$.

'then' The change made in response to the 'if' clause is also equivalent, since the inversion adding every y to $C^{\mathcal{I}^*}$ wherever $(x, y) \in R^{\mathcal{I}^*}$ represents the notion that $x \in \forall R.C$, which is identical to the standard tableau rules adding $\{C\}$ to any y so that $\mathcal{L}(y) = \{C\}$.

The only remaining thing to discuss is that Algorithm 1 and the *sat* function are sound and complete. For this it is simple to outline without a lengthy proof, since as we mentioned previously, the sat functions will test all elements in all expressions with the tableau expansion rules, exactly as you would have in a tableau. It is unnecessary and in fact unhelpful in this algorithm to combine all expressions into a connected whole since each axiom is itself evaluated consistently.

The subsumption axioms are solved in such a way that they implicitly compute a class and role hierarchy, so any inference dependent on a hierarchy will

still be entailed. Next, the stop condition of Algorithm 1 when the interpretation has not changed is identical with a completed tableau that no longer requires expansion or backtracking since both are models. And of course all cases where unresolvable clashes can occur are sound because they indicate instances where an element simultaneously must be in and must not be in an expression so the algorithm should terminate. As for completeness, it is again clear that whenever an unresolvable clash occurs it is identical with a case when a tableau discovers a clash but cannot backtrack to correct it.

5 Evaluation

The implementation of the Emi reasoner is evaluated against other popular JVM reasoners such as Hermit[6], Pellet[7], and jFact[8] [4,7,8] using Leiningen[9], which can run Clojure as well as Java programs. 500 ontologies were randomly sampled, half from ODPs and Biomedical ontologies in [3], and half from the Ontology Reasoner Evaluation 2015 competition dataset. These files were altered by removing any axiom that is not expressible in $\mathcal{ALCH}$ for testing. Among the 500 ontologies, 56 of them either timed out on all reasoners or caused our evaluation program to crash in some unexpected way due to lack of heap. It is not entirely clear from our logs which reasoner may be breaking in this way so we omit them, though we are confident that Emi has no heap issues or memory leaks (it is quite difficult to even *intentionally* create memory leaks with Clojure) and it can load all files by itself without error.

All testing was done on a computer running Ubuntu 20.04.1 64-bit with an Intel Core i7-9700K CPU@3.60GHz x 8, 47.1 GiB DDR4, and a GeForce GTX 1060 6GB/PCIe/SSE2.

5.1 Results

Testing and development is ongoing for the Emi reasoner and these results are a first example of the potential for this type of algorithm. A preliminary example of the reasoning time can be found in Figure 1. In this example, 3 tests are performed and timed on each reasoner: initialization from a file in memory shown in Figure 2, satisfiability and consistency checking shown in Figure 3, and retrieval of the elements in all non-empty class and role names after reasoning has completed shown in Figure 4.[10] Each test was given a timeout of 5 minutes, and when a test timed out more than 5 tests in a row for a reasoner it was prevented from testing again on the same file to save time in the evaluation. Across all tests, when we compare the percent difference between the reasoner time and the average time for all reasoners we see that Emi is overall 9.8% faster than

[6] Hermit Version 1.4.5.519
[7] Openllet Version 2.6.5
[8] jFact Version 5.0.3
[9] `https://leiningen.org/`
[10] Raw data and charts are available for inspection `https://tinyurl.com/kgswc2023`

average, Hermit is 4.2% slower than average, Pellet is 42.3% faster than average, and jFact is 55.9% slower than average. No reasoner was ever more than 100% better than average or 300% worse than average. Emi, Hermit, and Pellet each had 2 files when they failed due to an exception, while jFact failed on 70 files, though this is largely due to an issue it has with anonymous individuals in the ORE dataset.

As you can see in Figures 2 and 3, the Emi reasoner has a relatively slow initialize time and time for satisfiability on small ontologies. This is partially due to the fact that Emi solves all ABox axioms without complex expressions while it loads the ontology. This allows it to preemptively reject any naively unsatisfiable ABox, and the cost of this is usually only a few milliseconds. Emi seems to scale better than other systems and its performance improves in comparison as the number of axioms increases. Also, Emi is definitively faster in every single test when asked to compute the elements of all non-empty classes and roles, except the two tests where it had a timeout when computing satisfiability. Emi finishes reasoning with this information already computed, it is in effect computing both things at once, and only needs a variable amount of time to answer in our tests because it has to sort out the anonymous elements from every expression for comparison with the other reasoners where these elements are hidden by the OWL API [5]. Otherwise it could answer this in constant time.

Looking more closely at the data, there are three large clusters at 163, 325, and approximately 5500 axioms due to the synthetic ontologies in the ORE dataset. As you can see, all reasoners appear to behave similarly across multiple files of the same size in the clusters. Except for these clusters there appears to be a mostly stochastic distribution in the performance of each reasoner across different files with jFact doing great on small files but not scaling well, while Hermit, Pellet, and Emi start off a bit slower and seem to scale much better on very large files.

An interesting pattern we have noticed in the evaluation is that Emi usually outperforms other reasoners when it checks large ontologies with empty or nearly-empty ABoxes. This case makes sense when you notice that in $\mathcal{ALCH}$, as long as all axioms do not contain either negation or Top in the antecedent, there will always be a model where every predicate is empty. This is the default state when Emi starts, so it simply checks everything and the initial realization turns out to be fine after the first iteration.

6 Future Work

In the future there is quite a bit of work left to finalize and sufficiently verify Emi. These improvements will be ongoing as part of any new extensions. Parallelization in particular will likely prove difficult to formally pin down, though empirically its use is not as yet seeming to be problematic when comparing against the behavior of other reasoners. Some obvious extensions that are planned for the near future are inverse roles, nominals, and cardinality expressions which can work quite directly with some of the already existing code. Role chains are

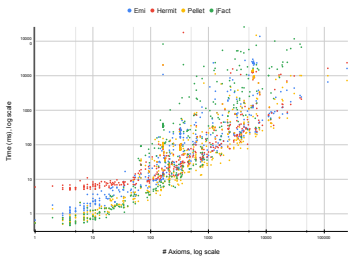

Fig. 1. All Reasoning Tests

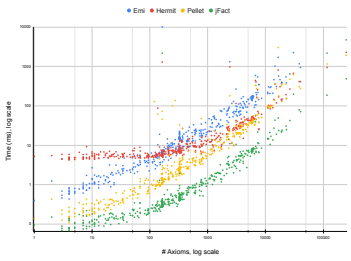

Fig. 2. Initialize

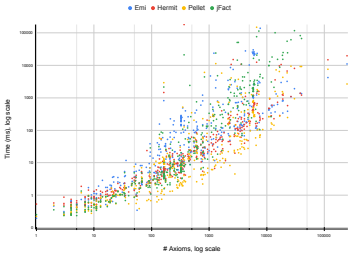

Fig. 3. Satisfiability

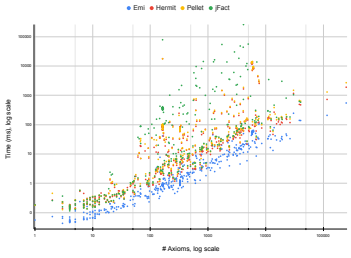

Fig. 4. Non-Empty Classes and Roles

another interesting and useful addition we are considering, though these will be difficult to implement efficiently so we will be cautious.

An interesting observation we have made while considering extensions is that some of the common difficulty with nominals in reasoning may turn out to be trivial in many cases for the Emi algorithm because nominals are not necessarily connected to anything as long as we maintain sufficient information about (in)equality. Once this is implemented and our hypothesis is checked it would also be straightforward to extend nominals to nominal schemas, since Emi does not normalize or pre-process away the original axioms. Initial testing suggest there is an intuitive way to bind nominal variables within axioms to solve this directly as Emi reasons.

Acknowledgement This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-18-1-0386.

References

1. Baader, F., Calvanese, D., McGuinness, D., Patel-Schneider, P., Nardi, D.: The description logic handbook: Theory, implementation and applications. Cambridge university press (2003)
2. Eberhart, A., Hitzler, P.: A functional API for OWL. In: Taylor, K.L., Gonçalves, R.S., Lécué, F., Yan, J. (eds.) Proceedings of the ISWC 2020 Demos and Industry Tracks: From Novel Ideas to Industrial Practice co-located with 19th Inter-

national Semantic Web Conference (ISWC 2020), Globally online, November 1-6, 2020 (UTC). CEUR Workshop Proceedings, vol. 2721, pp. 315–320. CEUR-WS.org (2020), http://ceur-ws.org/Vol-2721/paper580.pdf
3. Eberhart, A., Shimizu, C., Chowdhury, S., Sarker, M.K., Hitzler, P.: Expressibility of owl axioms with patterns. In: Verborgh, R., Hose, K., Paulheim, H., Champin, P.A., Maleshkova, M., Corcho, O., Ristoski, P., Alam, M. (eds.) The Semantic Web. pp. 230–245. Springer International Publishing, Cham (2021)
4. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: Hermit: An owl 2 reasoner. J. Autom. Reason. **53**(3), 245–269 (oct 2014). https://doi.org/10.1007/s10817-014-9305-1, https://doi.org/10.1007/s10817-014-9305-1
5. Horridge, M., Bechhofer, S.: The OWL API: A Java API for working with OWL 2 ontologies. In: Proceedings of the 6th International Conference on OWL: Experiences and Directions – Volume 529. p. 49–58. OWLED'09, CEUR-WS.org, Aachen, DEU (2009)
6. Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible sroiq. In: Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning. p. 57–67. KR'06, AAAI Press (2006)
7. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical owl-dl reasoner. Web Semant. **5**(2), 51–53 (jun 2007). https://doi.org/10.1016/j.websem.2007.03.004, https://doi.org/10.1016/j.websem.2007.03.004
8. Tsarkov, D., Horrocks, I.: Fact++ description logic reasoner: System description. In: Proceedings of the Third International Joint Conference on Automated Reasoning. p. 292–297. IJCAR'06, Springer-Verlag, Berlin, Heidelberg (2006). https://doi.org/10.1007/11814771_26, https://doi.org/10.1007/11814771_26