# Snorocket 2.0: Concrete Domains and Concurrent Classification

Alejandro Metke-Jimenez and Michael Lawley

The Australian e-Health Research Centre
ICT Centre, CSIRO
Brisbane, Queensland, Australia
{alejandro.metke,michael.lawley}@csiro.au
http://aehrc.com

**Abstract.** Snorocket is a high-performance ontology reasoner that supports a subset of the OWL EL profile. In the newest version, additional expressive power has been added to support concrete domains, enabling the classification of ontologies that use these constructs. Also, the reasoning algorithm has been modified to support concurrent classification. This feature is important because it enables the use of the full processing power available in modern multi-processor hardware.

**Keywords:** ontology, classification, concrete domains, concurrent

## 1 Introduction

This paper presents Snorocket 2.0, a high-performance ontology reasoner based on the CEL algorithm [6]. Snorocket was the first reasoner to provide ultra-fast classification of SNOMED CT and support incremental classification [1]. The initial version is used in the IHTSDO workbench to support SNOMED CT authoring and it is designed to work with a small heap footprint*. Snorocket 2.0 is now an open source project available at GitHub†.

Some biomedical ontologies, such as SNOMED CT, have been built using a subset of the OWL EL profile. Even though most of their content can be correctly modelled using this subset, some concepts cannot be fully modelled without concrete domains. For example, it is not possible to fully represent a "Hydrochlorothiazide 50mg tablet" without using a data literal to represent the quantity of the active ingredient. To overcome this limitation AMT v3, an extension of SNOMED CT used in Australia to model medicines, has recently introduced concrete domains. This has motivated the inclusion of concrete domains into the subset of constructs supported by Snorocket.

The development of extensions to SNOMED CT also means that ontology reasoners should be able to support classification of larger ontologies. This has motivated the implementation of a concurrent classification algorithm that allows using the extra processing power available in multi-processor machines.

*This was required to support 32-bit JVMs running on Windows machines.

†https://github.com/aehrc/snorocket

## 2 Background

The initial version of Snorocket was developed to support the fast classification of SNOMED CT and therefore only included support for a limited number of constructs. A table comparing the OWL EL constructs supported by Snorocket and other EL reasoners is available on the Snorocket website[‡].

Concrete domains are supported by several general tableaux-based reasoners such as FaCT++ [8] and HermiT [9]. The only specialised EL reasoner that currently supports concrete domains is ELK [4].

Concrete domains are used in AMT mainly to model quantities in the definition of medicines. An OWL version of AMT v3 can be obtained by using an updated version of the Perl script originally included in the SNOMED CT distribution. An example of a typical axiom found in AMT is available on the Snorocket website[§].

Most of the commonly used reasoners, including FACT++ [8], HermiT [9], CEL [6], and jCEL [7] are only capable of using a single processor. To our knowledge, the only reasoner that has successfully implemented a concurrent classification algorithm is ELK [4]. Because most modern hardware achieves better performance by providing more than one processor or core, it is important to be able to make use of this extra processing power.

## 3 Architecture

Figure 1 shows a high-level architecture diagram of Snorocket 2.0. The public Snorocket API, shown inside the **snorocket-core** module, enables third party applications to use the reasoner. The API uses a simple model to represent ontologies. This model is vastly simpler than other publicly available ontology models, such as OWL API, and excludes all the constructs not currently supported. This simplifies the usage of the public API. A more detailed description of this model is available on the Snorocket website[¶]. All external formats, such as OWL, RF1, and RF2, are transformed to and from this model. Additional ontology formats can be supported by adding new importer-exporter modules.

The Snorocket API defines an interface, IReasoner, to perform several reasoning functions. The reasoner interface defined by OWL API, OWLReasoner, is also implemented in the **snorocket-owlapi** module. Applications that want to use the reasoner can use either one. SNOMED CT-specific applications can use the RF1 and RF2 importer-exporter components to generate the axioms in our simple ontology format from the distribution files. It is also possible to create these axioms programatically. OWL ontologies are imported using OWL API and transformed into our simple model using the OWL importer-exporter component. A plugin for Protégé is also available in the **snorocket-protege** module.

[‡]http://aehrc.com/software/snorocket/index.html#constructs
[§]http://aehrc.com/software/snorocket/index.html#amtv3
[¶]http://aehrc.com/software/snorocket/index.html#model

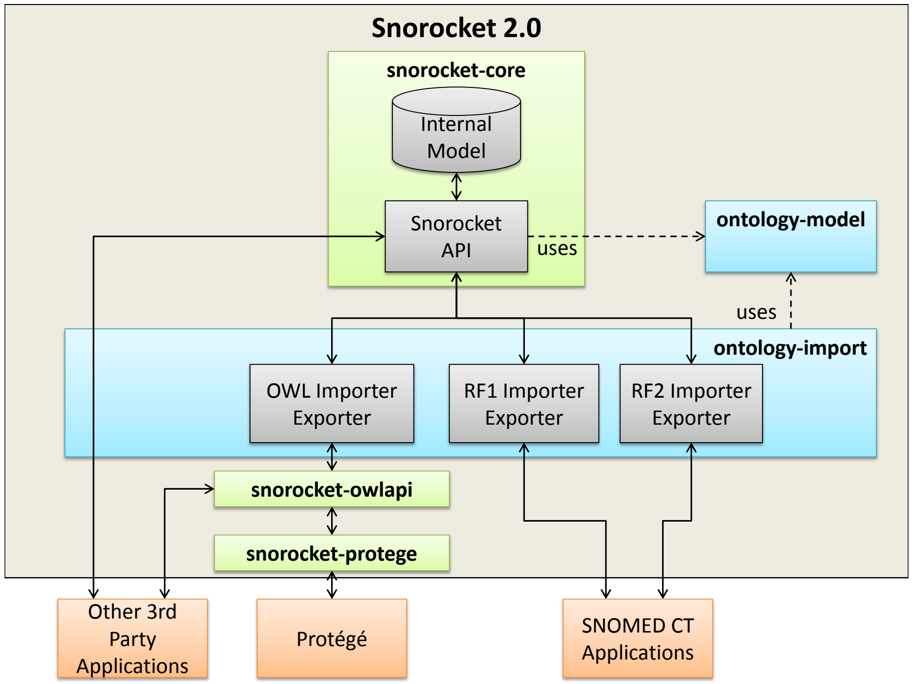


**Fig. 1.** Snorocket 2.0 architecture. The labels in bold refer to Maven modules.

## 4 Implementation

The current implementation of Snorocket is targeted at supporting the OWL EL profile. It is implemented using Java and built using Maven. The following sections describe the implementation details of the new features.

### 4.1 Concrete domains

In description logics a concrete domain is a construct that can be used to define new classes by specifying restrictions on attributes that have literal values (as opposed to relationships to other concepts). For example, children of age six can be defined by using the concrete domain expression $\exists hasAge.(=, 6)$. The class of individuals, in this case children of age six, is expressed as a restriction on the age attribute, which has a numeric value. The binary operators $<, <=, >, >=$ can also be used in a concrete domain expression, and attributes can have other types of literal values such as floating point numbers, string literals, and dates.

**Support for equality** An ontology can contain many complex axioms that include nested sub-expressions. The CEL algorithm works with normalised axioms and therefore creates a conservative extension of the original ontology containing

**Table 1.** Normal forms and completion rules.

| Normal form | Completion rules |
|---|---|
| $A_1 \sqcap A_2 \sqsubseteq B$ | **R1** If $A_1, A_2 \in S(X)$, $A_1 \sqcap A_2 \sqsubseteq B \in O$, and $B \notin S(X)$ then $S(X) := S(X) \cup \{B\}$ |
| $A \sqsubseteq \exists r.B$ | **R2** If $A \in S(X)$, $A \sqsubseteq \exists r.B \in O$, and $(X, B) \notin R(r)$ then $R(r) := R(r) \cup \{(X, B)\}$ |
| $\exists r.A \sqsubseteq B$ | **R3** If $(X, Y) \in R(r)$, $A \in S(Y)$, $\exists r.A \sqsubseteq B \in O$, and $B \notin S(X)$ then $S(X) := S(X) \cup \{B\}$ |
| $r \sqsubseteq s$ | **R4** If $(X, Y) \in R(r)$, $r \sqsubseteq s \in O$, and $(X, Y) \notin R(s)$ then $R(s) := R(s) \cup \{(X, Y)\}$ |
| $r \circ s \sqsubseteq t$ | **R5** If $(X, Y) \in R(r)$, $(Y, Z) \in R(s)$, $r \circ s \sqsubseteq t \in O$, and $(X, Z) \notin R(t)$ then $R(t) := R(t) \cup \{(X, Z)\}$ |
| $A \sqsubseteq \exists f.(o, v)$<br>$\exists f.(o, v) \sqsubseteq B$ | **R6** If $A \in S(X)$, $A \sqsubseteq \exists f.(o_1, v_1) \in O$, $\exists f.(o_2, v_2) \sqsubseteq B \in O$, $eval(o_1, v_1, o_2, v_2) = true$, and $B \notin S(X)$ then $S(X) := S(X) \cup \{B\}$ |

only axioms in normal form [2]. The normal forms and the corresponding completion rules R1 to R5 from the original CEL algorithm are shown in Table 1. The last two normal forms and completion rule R6 have been added to support concrete domains.

The normalised forms of concrete domain expressions are $A \sqsubseteq \exists f.(o, v)$ and $\exists f.(o, v) \sqsubseteq A$, where $f$ represents a feature, $o$ an operator, and $v$ a value. The original normalisation algorithm requires only minor changes to deal with these new constructs.

The classification algorithm does require significant changes to deal with the new concrete domain axioms. A new type of queue is introduced to deal with the queue entries of the form $A \sqsubseteq \exists f.(o, v)$ and it is initialised with these axioms. The entries are then processed in the following way:

1. The axioms of the form $\exists f.(o, v) \sqsubseteq B$ that match the feature $f$ of the data type in the queue entry are retrieved.
2. The data types are then compared using the $eval()$ function.
3. If only the equality operator needs to be supported then the two data types are considered to be matching if their literal value is equal.

**Support for other operators** It is known that supporting arbitrary combinations of different operators leads to intractability [3]. In this implementation no checks are made to ensure that the ontology being classified complies with the restrictions that guarantee tractability. If non-compliant axioms are found then the reasoning procedure will be sound but possibly incomplete.

Adding support for other operators requires a modification to the $eval()$ function that compares the data types when evaluating feature queue entries. The different combinations of operators and values have to be evaluated to determine if there is a match or not.

For example, consider the following axioms:

$$toddler \equiv person \sqcap \exists hasAge.(\leq, 3)$$
$$child \equiv person \sqcap \exists hasAge.(\leq, 17)$$

After the normalisation process these axioms are transformed into the following:

$$\begin{array}{ll} \exists hasAge.(\leq, 17) \sqsubseteq A & person \sqcap A \sqsubseteq child \\ child \sqsubseteq person & child \sqsubseteq \exists hasAge.(\leq, 17) \\ \exists hasAge.(\leq, 3) \sqsubseteq B & person \sqcap B \sqsubseteq toddler \\ toddler \sqsubseteq person & toddler \sqsubseteq \exists hasAge.(\leq, 3) \end{array}$$

These axioms allow us to infer that a toddler is also a child (but a child is not necessarily a toddler). This conclusion is derived when evaluating the expressions $toddler \sqsubseteq \exists hasAge.(\leq, 3)$ and $\exists hasAge.(\leq, 17) \sqsubseteq A$. The $eval()$ function in this case takes the arguments $(\leq, 3, \leq, 17)$ and returns a positive match because all the possible values of the first operator-value pair are covered by the possible values of the second operator-value pair. Whenever this is not the case the function returns false. Notice that this happens in some cases regardless of the literal values. For example, assuming we are dealing with integer values, $eval(x, <, y >)$ and $eval(x, >, y, <)$ will always return false because no matter what values are assigned to $x$ and $y$, the second operator-value pair will never be able to cover all the possible values expressed by the first pair.

### 4.2 Concurrent classification

This new version of Snorocket implements a multi-threaded saturation algorithm inspired by the algorithm used by ELK. The main idea of the algorithm is to split the computation into contexts that can be processed by workers independently while generating minimal locking overhead. Details of the original algorithm can be found in [5]. The main techniques in the algorithm can be applied in a straightforward manner to the CEL algorithm implemented by Snorocket.

## 5 Experimental results

Protégé was used to compare the performance of Snorocket against four other ontology reasoners: FaCT++, HermiT, jCel, and ELK. The previous version of Snorocket was also included. Two OWL ontologies were used in the tests: SNOMED CT and AMT v3. Both of these were derived from the RF2 distribution files using the corresponding Perl scripts.

The experiments were run in a computer equipped with a 3.3 GHz Intel i5 processor with 4 cores, 8 GB of physical memory, and running Windows 7. Protégé was run with Java 7 and a heap size of 4 GB. All the experiments use elapsed time as an indicator and use the external timing reported by Protégé. The multi-threaded reasoners (ELK 0.32 and Snorocket 2.0.1) were run using 4 threads.

Table 2 shows the profiles of the selected ontologies and Table 3 shows the classification times, in seconds, achieved by the reasoners, averaged over 5 runs.

**Table 2.** Profiles of the test ontologies.

| Ontology | #Classes | #Object Properties | #Data Properties | #Axioms Original | #Axioms Normalised |
|---|---|---|---|---|---|
| SNOMED CT | 296518 | 62 | 0 | 660610 | 1169913 |
| AMT | 61059 | 78 | 4 | 150750 | 561331 |

**Table 3.** Average classification times in seconds using various reasoners in Protégé on Windows averaged over 5 runs. *mem* indicates an OutOfMemory error.

| | SNOMED CT | AMT |
|---|---|---|
| FACT++ 1.6.2 | 330 | 4220 |
| HermiT 1.3.7 | 1567.3 | mem |
| jCel 0.15‡ | 761 | - |
| ELK 0.32 | 9.1 | 10.5 |
| Snorocket 1.3.4 | 33.8 | - |
| Snorocket 2.0.1 | 26 | 26.2 |

The results show that the performance of the tableaux-based reasoners was very poor when classifying AMT. On the other hand, the specialised EL reasoners were able to classify it in a fraction of the time. ELK currently provides the best performance, which is expected since Snorocket's multi-threaded implementation is based on the same techniques but has not been optimised. Also, Snorocket only runs the saturation phase concurrently, while the rest of the steps are still run sequentially.

## 6 Conclusions and future work

This paper presented Snorocket 2.0 and compared it against its previous version and four other reasoners using two large medical ontologies. Even though ELK obtained the fastest results, Snorocket 2.0 achieved competitive performance. Snorocket's built-in support for SNOMED CT distribution formats makes it an interesting alternative to ELK for SNOMED CT-centric applications.

Future work will include adding multi-threading to the whole classification process and incorporating the restrictions necessary to ensure tractability when dealing with concrete domains, either as a hard restriction or as a warning to the user.

‡The current version of the jCel plugin is 0.18.2 but version 0.15 was the most recent one that was compatible with our testing environment.

## References


1. Lawley, M. J., Bousquet, C.: Fast classification in Protégé: Snorocket as an OWL 2 EL reasoner. In: Proc. 6th Australasian Ontology Workshop (IAOA10). Conferences in Research and Practice in Information Technology, pp. 45–49. (2010)
2. Baader, F., Brandt, S., Lutz, C.: Pushing the EL envelope. In: International Joint Conference on Artificial Intelligence, p. 364 (2005)
3. Magka, D., Kazakov, Y., Horrocks, I.: Tractable Extensions of the Description Logic EL with Numerical Datatypes. In: Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2010). LNAI, vol. 6173, pp. 61–75. Springer (2010)
4. Kazakov, Y., Krötzsch, M., Simančk, F.: ELK Reasoner: Architecture and Evaluation. In: Proceedings of the 1st International Workshop on OWL Reasoner Evaluation, CEUR Workshop Proceedings, (2012)
5. Kazakov, Y., Kötzsch, M., Simančk, F.: Concurrent Classification of EL+ Ontologies. In: The Semantic Web ISWC 2011, pp. 305–320 (2011)
6. Baader, F., Lutz, C., Suntisrivaraporn, B.: Efficient reasoning in EL+. In: Proceedings of DL 2006, p.189 (2006)
7. Mendez, J. jcel: A Modular Rule-based Reasoner. In: Proc. of the 1st Int. Workshop on OWL Reasoner Evaluation (ORE12), pp. 130-135 (2012)
8. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. In: Proc. 3rd Int. Joint Conf. on Automated Reasoning (IJCAR 2006). LNCS, vol. 4130, pp. 292-297. Springer (2006)
9. Motik, B., Shearer, R., Horrocks, I.: HermiT: Hypertableau Reasoning for Description Logics. Journal of Artificial Intelligence Research 36, pp. 165–228 (2009)
10. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. J. of Web Semantics 5(2), pp. 51-53 (2007)