

The Fusemate Logic Programming System (System Description)

Peter Baumgartner^[0000–0002–6559–9654]

Data61/CSIRO and The Australian National University, Canberra, Australia
Peter.Baumgartner@data61.csiro.au

Abstract. Fusemate is a logic programming system that implements the possible model semantics for disjunctive logic programs. Its input language is centered around a weak notion of stratification with comprehension and aggregation operators on top of it. Fusemate is implemented as a shallow embedding in the Scala programming language. This enables using Scala data types natively as terms, a tight interface with external systems, and it makes model computation available as an ordinary container data structure constructor. The paper describes the above features and demonstrates them with a non-trivial use-case, the embedding of the description logic \mathcal{ALCIF} into Fusemate’s input language.

This version of the paper corrects an error in the published version, which used an unsuitable version of “blocking” in the \mathcal{ALCIF} embedding.

1 Introduction

Fusemate¹ is a logic programming system for computing possible models of disjunctive logic programs [23,24]. A Fusemate logic program consists of (typically) non-ground if-then rules with stratified default negation in the body [21]. Stratification entails that a true default-negated body literal remains true in the course of deriving new conclusions.

Fusemate was introduced in [7] for modelling systems that evolve over time and for analysing their current state based on the events so far. Such tasks are often subsumed under the terms of stream processing, complex event recognition, and situational awareness, and have been addressed (also) with logic-based approaches [2,9,4,5].

To my knowledge, Fusemate is unique among all these and other logic programming systems [12,1,13,26,16] (and theorem provers) in the way it is implemented. Fusemate is implemented by shallow embedding in a full-fledged programming language, Scala [25]. Essentially, the user writes a syntactically sugared Scala program utilizing familiar logic programming notation, and the program’s execution returns models. This has advantages and disadvantages. The main disadvantage is that it is more difficult to implement performance boosting measures like term indexing. The main advantage is that interfacing with data structure libraries and with external systems is easy, an aspect whose importance has been emphasized for virtually all of the above systems. In fact, Fusemate is motivated in parts by exploring how far the embedding approach can be pushed and to what benefit.

¹ Fusemate is available at <https://bitbucket.csiro.au/users/bau050/repos/fusemate/>.

The earlier Fusemate paper [7] focused on the model computation calculus with a belief revision operator as the main novelty. It utilized a certain notion of *stratification by time (SBT)* for making the calculus effective and useful in the intended application areas. This system description focuses on the advantages of the shallow embedding approach as boosted by new language features introduced here. These new language features are (a) non-standard comprehension and aggregation operators, among others, and (b) a weaker notion of *stratification by time and predicates (SBTP)*. In brief, SBTP is a lexicographic combination of stratification by time and the standard stratification in terms of the call-graph of the program. Section 5 has an example that demonstrates the need for (a) and (b) in combination, and Section 4 discusses the shallow embedding approach and its advantages on a more general level.

Here is an excerpt from a Fusemate program that previews some of the new features:

```

1 type Time = java.time.LocalDateTime
2 val allIds = 1 to 10
3 case class Change(time:Time, id:Int, color:String) extends Atom
4 case class State(time:Time, id:Int, color:String) extends Atom
5 case class FullState(time:Time, drive:Set[Int], stop:Set[Int]) extends Atom
6 State(time, id, color) :- Now(time), CHOOSE(id:Int, allIds), Change(t <= time, id, color)
7 FullState(time, drive.toSet, stop.toSet) :- State(time,_,_),
8   COLLECT(drive:List[Int], id STH State(time, id, "green")),
9   COLLECT(stop:List[Int], id STH (State(time, id, color), color=="red" || color=="yellow"))
10 MovingState(time) :- FullState(time, drive, stop), stop.size < drive.size
11 Faulty(time, id, since) :- State(time, id, "red"), Change(since < time, id, "green"),
12   NOT (Change(t, id, "yellow"), since < t, t < time)

```

The scenario comprises traffic lights identified by numbers 1 to 10 (line 2). In the course of time the traffic lights change their colors, and each such event is recorded as a corresponding Change atom (line 3). The rule on line 6 computes a State at a current time Now(time) as a snapshot of the current colors of all traffic lights. For that, the comprehension Change(t <= time, id, color) on line 6 finds the latest Change event before or at time for a fixed id chosen from allIds, and binds that time to the (unused) variable t. A FullState aggregates the separate State facts at a time partitioned as (Scala) sets of ids of “drive” and “stop” colors. In that, the **COLLECT** special form collects in a Scala List-typed variable the specified terms that satisfy the body behind **STH**. Notice that all atoms in FullState refer to the same time, yet the program is SBTP because State comes before FullState in predicate stratification. (Predicate stratification is computed automatically by Fusemate with Tarjan’s algorithm.) The rule on line 10 demonstrates the use of the Scala Set method size in the body. Line 11 demonstrates the use of default negation in combination with comprehension. When applied to a given sequence of Change events, Fusemate computes models, one-at-a-time, each as Scala set of atoms.

2 Fusemate Programs

For the purpose of this paper, a brief summary of the syntactic notions underlying Fusemate programs is sufficient; see [7] for details. Terms and atoms of a given signature

are defined as usual. Let $\text{var}(z)$ denote the set of variables occurring in an expression z . We say that z is *ground* if $\text{var}(z) = \emptyset$. We write $z\sigma$ for applying a substitution σ to z . The domain of σ is denoted by $\text{dom}(\sigma)$. A substitution γ is a *grounding substitution for z* iff $\text{dom}(\gamma) = \text{var}(z)$ and $z\gamma$ is ground. In this case we simply say that γ is for z .

Let \mathbb{T} be a countably infinite discrete set of *time points* equipped with a total strict ordering $<$ (“earlier than”), e.g., the integers. Assume that the time points, comparison operators $=$ and \leq , and a successor time function $+1$ are part of the signature and interpreted in the intended way. A *time term* is a (possibly non-ground) term over the sub-signature $\mathbb{T} \cup \{+1\}$.

The signature may contain other “built-in” predicate and function symbols for pre-defined types such as strings, arithmetic data types, sets, etc. We only informally assume that all terms are built in a well-sorted way and that built-in operators over ground terms can be evaluated effectively.

An *ordinary atom (with time term t)* is of the form $p(t, t_1, \dots, t_n)$ where p is an ordinary predicate (i.e., neither a time predicate nor built-in), t is a time term and t_1, \dots, t_n terms. A (*Fusemate*) *rule* is an implication written in Prolog-like syntax as

$$H :- b_1, \dots, b_k, \text{not } \vec{b}_{k+1}, \dots, \text{not } \vec{b}_n . \quad (1)$$

In (1), a rule *head* H is either (a) a disjunction $h_1 \vee \dots \vee h_m$ of ordinary atoms, for some $m \geq 1$, or (b) the expression **fail**.² In case (a) the rule is *ordinary* and in case (b) it is a *fail rule*. A rule *body* B , the part to the right of $:-$, is defined by mutual recursion as follows. A *positive body literal* is one of the following: (a) an ordinary atom, (b) a *comprehension atom (with time term x)* of the form $p(x \circ t, t_1, \dots, t_n) \text{sth } B$, where x is a variable, $\circ \in \{<, \leq, >, \geq\}$ and B is a body, (c) a built-in call, i.e., an atom with a built-in predicate symbol, or (d) a *special form* **let**(x, t), **choose**(x, ts), **match**(t, s) or **collect**($x, t \text{sth } B$) where x is a variable, s, t are terms, ts is a list of terms, and B is a body. A *positive body* is a list $\vec{b} = b_1, \dots, b_k$ of positive body literals with $k \geq 0$. If $k = 0$ then \vec{b} is *empty* otherwise it is *non-empty*. A *negative body literal* is an expression of the form **not** \vec{b} , where \vec{b} is a non-empty positive body. A *body* is a list $B = b_1, \dots, b_k, \text{not } \vec{b}_{k+1}, \dots, \text{not } \vec{b}_n$ comprised of a (possibly empty) positive body and (possibly zero) negative body literals. It is *variable free* if $\text{var}(b_1, \dots, b_k) = \emptyset$.

Let r be a rule (1). We say that r is *range-restricted* iff $\text{var}(H) \subseteq \text{var}(\vec{b})$. Compared to the usual notion of range-restrictedness [18], Fusemate rules may contain extra variables in negative body literals. For example, $p(t, x) :- q(t, x), \text{not}(s < t, r(s, x, y))$ is range-restricted in our sense with extra variables s and y . The extra variables are implicitly existentially quantified within the **not** expression. The example corresponds to the formula $q(t, x) \wedge \neg \exists s, y. (s < t \wedge r(s, x, y)) \rightarrow p(t, x)$. Semantically and operationally this will cause no problems thanks to stratification, introduced next.

Fusemate programs – sets of rules – need to be “stratified by time and by predicates” (SBTP). The standard notion of stratification by predicates means that the call graph of the program contains no cycles going through negative body literals. The edges of this call graph are the “depends on” relation between predicate symbols such that p

² This definition of head is actually simplified as Fusemate offers an additional head operator for belief revision, see [7]. This is ignored here.

positively (negatively) depends on q if there is a rule with a p -atom in its head and a q -atom in its positive (negative) body. For disjunctive heads, all head predicates are defined to depend positively on each other. Every strongly connected component of the call graph is called a stratum, and in predicate stratified programs negative body literals can occur only in strata lower than the head stratum.

SBTP is defined as follows: for every rule (1) in a given program, (a) there is a variable *time* that is the time term of some ordinary $b \in \vec{b}$, (b) if H is an ordinary head then every head literal must have a time term constrained to be \geq than *time*, and (c) for all rule bodies B occurring in the rule:

- (i) the time term of every ordinary or comprehension body literal in B must be constrained to be \leq than *time*, and
- (ii) for every negative body literal **not** \vec{b} in B (including the top-level body of (1)) and every ordinary or comprehension literal $b \in \vec{b}$, the time term of b must be constrained to be (i) $<$ than *time* or (ii) \leq than *time* and the predicate symbol of b is in a lower stratum than H .

For the purpose of this paper we only informally assume that all rules contain constraints for enforcing the required time ordering properties. There are similar stratification requirements for comprehension atoms and special forms so that their evaluation satisfies the counterpart of condition (ii) (see below for **collect**). A fully formal definition could be given by modifying the spelled-out definition of SBT in [7].

As an example, if r belongs to a lower stratum than p then the following five rules all are SBTP, while only the first two rules are SBT.

$$p(\text{time}, x) \text{ :- } q(\text{time}, x), r(t, y), t \leq \text{time} \quad (2)$$

$$p(\text{time}, x) \text{ :- } q(\text{time}, x), \text{not}(r(t, y), t < \text{time}) \quad (3)$$

$$p(\text{time}, x) \text{ :- } q(\text{time}, x), \text{not}(r(t, y), t \leq \text{time}) \quad (4)$$

$$p(\text{time} + 1, x) \text{ :- } q(\text{time}, x), \text{not}(r(t, y), t \leq \text{time}) \quad (5)$$

$$p(\text{time}, x) \text{ :- } q(\text{time}, x), (p(t < \text{time}, y) \text{ sth } q(t, y)), r(t, y) \quad (6)$$

Finally, a (*Fusemate*) *program* is a set of range-restricted rules that is SBTP.

3 Model Computation

The possible model semantics of disjunctive logic programs [23,24] associates to a given disjunctive program a certain set of normal programs (i.e., without disjunctive heads) and takes the intended model(s) of these normal programs as the possible models of the given program. These “split” programs represent all possible ways of making one or more head literals true, for every disjunctive rule. As a propositional example, the program $\{a \text{ :- } b, a \vee c \text{ :- } b, b \text{ :- } \}$ is associated to the split programs $\{a \text{ :- } b, b \text{ :- } \}$ and $\{a \text{ :- } b, c \text{ :- } b, b \text{ :- } \}$. The possible models, hence, are $\{a, b\}$ and $\{a, b, c\}$.

Fusemate computes possible models by bottom-up fixpoint computation and dynamic grounding the program rules in the style of hyper tableaux [8]. The model computation procedure is implemented as a variant of the well-known given-clause

algorithm, which seeks to avoid deriving the same conclusion from the same premises twice. It exhausts inferences in an outer loop/inner loop fashion according to the given program's stratification by time and by predicates. The main data structure is a set of paths, where each path represents a partial model candidate computed so far (see [7] for more details). Paths are selected, extended, split and put back into the set until exhausted, for a depth-first, left-to-right inference strategy. Paths carry full status information, which is instrumental for implementing incrementality, such that facts with current or later time can be added at any stage without requiring model recomputation from scratch. This, however, necessitated keeping already exhausted paths for continued inferences later.

The proof procedure's core operation is computing a *body matcher*, i.e., a substitution γ for a rule's positive body variables so that the rule body becomes satisfied in the current partial model candidate. Formally, let I be a set of ordinary ground atoms, representing the obvious interpretation that assigns true to exactly the members of I . Let B be a body. A *body matcher for B* is a substitution γ for the positive body of B , written as $I, \gamma \models B$, such that the following holds (b, B means the sequence of head b and rest body B):

- $I, \varepsilon \models \epsilon$ (ϵ is the empty body and ε is the empty substitution)
- $I, \gamma\sigma \models b, B$ iff γ is for b , $b\gamma \in I$ and $I, \sigma \models B\gamma$, with b ordinary atom
- $I, \gamma\sigma \models (p(x \stackrel{\leq}{<} t, t_1, \dots, t_n) \text{ sth } C), B$ iff γ is for $p(x, t_1, \dots, t_n)$ and
 - (1) $p(x, t_1, \dots, t_n)\gamma \in I$, $x\gamma < t$ and $I, \delta \models C\gamma$ for some δ ,
 - (2) there is no γ' for $p(x, t_1, \dots, t_n)$ and no δ such that
 - $p(x, t_1, \dots, t_n)\gamma' \in I$, $x\gamma < x\gamma' \stackrel{\leq}{<} t$ and $I, \delta \models C\gamma'$, and
 - (3) $I, \sigma \models B\gamma$
- $I, \sigma \models a, B$ iff a evaluates to true and $I, \sigma \models B$ where a is ground built-in
- $I, \gamma\sigma \models \text{let}(x, t), B$ iff $\gamma = [x \mapsto t]$ and $I, \sigma \models B\gamma$
- $I, \gamma\sigma \models \text{choose}(x, ts), B$ iff $\gamma = [x \mapsto t]$ and $I, \sigma \models B\gamma$ for some $t \in ts$
- $I, \gamma\sigma \models \text{match}(t, s), B$ iff γ is for t , $t\gamma = s$ and $I, \sigma \models B\gamma$
- $I, \gamma\sigma \models \text{collect}(x, t \text{ sth } C), B$ iff $\gamma = [x \mapsto \{t\delta \mid I, \delta \models C\}]$ and $I, \sigma \models B\gamma$
- $I, \sigma \models \text{not } \vec{b}, B$ iff there is no δ such that $I, \delta \models \vec{b}$, and $A, \sigma \models B$

A *comprehension atom* $p(x \circ t, t_1, \dots, t_n) \text{ sth } B$ stands for the subset of all ground p -instances in I such that B is satisfied and with a time x as close as possible to t wrt. $<$ or \leq . The cases for $>$ and \geq are dual and not spelled out above to save space. The **collect** special form collects in the variable x the set of all instances of term t such that the body C is satisfied in I . We require comprehension atoms and **collects** to be used in a stratified way, so that their results do not change later in a derivation when I is extended. The requirements are the same as with **not** and can be enforced by ordering constraints.

The definition above extends the earlier definition of body matchers in [7] with the new comprehension construct and the **let**, **choose**, **match**, **collect** operators. It now also enforces left-to-right evaluation of B because the new binding operators depend on a fixed order guarantee to be useful. An example is the (nonsensical) body **CHOOSE** (x : Int, List(1, 2, 3)), **LET**(xxx: Int, $3*x$), $xxx \% 2 == 0$ which relies on this order. Undefined cases, e.g., when evaluation of a non-ground built-in is attempted, or when a binder variable has already been used before are detected as compile time syntax errors.

4 Shallow Embedding in Scala

Fusemate is implemented as a shallow embedding into Scala [25]. It has three conceptual main components: a signature framework, a Scala compiler plugin, and an inference engine for fixpoint computation as explained in Section 3. The signature framework provides a set of Scala class definitions as the syntactical basis for writing Fusemate programs. It is parameterized in a type *Time*, which can be any Scala or Java type that is equipped with an ordering and an addition function for time increments, for example `Int` or `java.time.OffsetDateTime`. The programmer then refines an abstract class *Atom* of the *Time*-instantiated signature framework with definitions of predicate symbols and their (Scala-)sorted arities. See lines (3)–(5) in the program in the introduction for an example. These atoms then can be used in Fusemate rules, see lines (6)–(12) in the example.

While written in convenient syntax, rules are syntactically ill-formed Scala. This problem is solved by the compiler plugin, which intercepts the compilation of the input file at an early stage and transforms the rules into valid Scala source code.³ More precisely, a rule is transformed into a curried partial function that is parameterized in an interpretation context *I*. The curried parameters are Scala guarded pattern matching expression and correspond to the rule’s positive body literals, in order. For example, the *Faulty* rule on lines (11) and (12), with the condition `since < time` ignored, for simplicity, is (roughly) translated into the function *f*

```

1 (I: Interpretation) => { case State(time, id, "red") => {
2   case Change(since, id1, "green") if id == id1 &&
3     ({ case Change(t, id2, "yellow") if id == id2 && since < t && t < time => FAIL } failsOn I) =>
4     Faulty(time, id, since) } }
```

Notice the renaming of repeated occurrences of the *id* variable, which is needed for the correct semantics. Notice also that a Scala Boolean-valued expression in an ordinary body literal position (e.g., `t < time`) simply becomes a guard in a pattern.

The code above can be understood with body matcher computation in mind. Suppose the inference engine selects an interpretation *I* from the current set of paths. For exhausting *f* on *I*, the inference engine combinatorially chooses literals $l_1, l_2 \in I$ and collects the evaluation results of $f(I)(l_1)(l_2)$, if defined. Observe that by the transformation into Scala pattern matching, body matchers are only implicitly computed by the Scala runtime system. Each evaluation result, hence, is a body-matcher instantiated head.

The rule’s negative body literal is translated into the code on line (3) and conjoined to the guard of the preceding ordinary literal. In general, a negative literal **NOT** *body* is treated by translating **FAIL** :- **NOT** *body* and evaluating the resulting Scala code on *I* by means of the `failsOn` method. If **FAIL** is not derivable then **NOT** *body* is satisfied. Again, appropriate bindings for the variables bound outside of *body* are held implicitly by the Scala runtime system. The translation of the special forms and comprehension is not explained here for space reasons. Fusemate can show the generated code, though.

³ Early experiments showed it is cumbersome and error-prone to write the Scala code by hand, so this was not an option. The compiler plugin is written in Scala and operates at the abstract syntax tree level. This was conveniently be done thanks to a sophisticated quasiquote mechanism.

Properties and Advantages

The shallow embedding approach enables introspection capabilities and interfacing between the rule language and the host language beyond what is implemented in other systems. In Fusemate, the terms of the logical language are nothing but Scala objects. As a consequence, any available Scala type or library data structure can be used as a built-in without extending an “interface” to an extension language – simply because there is none. Dually, the embedding of the rule language into the host language Scala is equally trivial because rules, atoms and interpretations are Scala objects, too.

It is this “closed loop” that makes an aggregation operator (**collect**) possible that returns a list of Scala objects as specified by the programmer, e.g., a list of terms or atoms.⁴ This list can be further analysed or manipulated by the rules. See the description logic embedding in Section 5, which critically depends on this feature. This introspection capability stands out in comparison to the logic programming systems mentioned in the introduction. For instance, aggregation in systems like DLV [1], and IDB [12] is limited to predefined integer-valued aggregates for sum, count, times, max and min.

Most logic programming systems can be called from a (traditional) host programming language and can call external systems or utilize libraries for data structures. The DLV system, for instance, interfaces with C++ and Python [22], Prova [16] with Java, and IDP with the Lua scripting language. Systems based on grounding (e.g., DLV and IDP) face the problem of “value invention” by external calls, i.e., having to deal with terms that are not part of the input specification [10].

The main issue, however, from the Fusemate perspective is that these systems’ external interfaces are rather heavy-handed (boilerplate code, mapping logic terms to/from the host language, String representation of logic programs) and/or limited to a predefined set of data structures. In contrast, Fusemate’s seamless integration with Scala encourages a more integrated and experimental problem solving workflow. The following Scala program demonstrates this point with the traffic light example:

```

1 List("2020-07-02T10:00:00",1,green", ..., "2020-07-02T10:02:15",2,red")
2   .map { _.split(",") } // Split CSVs into triple, represented as Java array
3   .map { // Convert String triple to positive Change literals
4     case Array(date,id,color) => Change(LocalDate.parse(date), id.toInt, color) }
5   .saturate { rules } // saturate is the Fusemate call, computes all models of the rules
6   .head // Select the first model
7   .toList // Convert to Scala List because we want to sort elements by time:
8   .sortBy { _.time }
9   .flatMap { // Analyze literals in model and retain only Faulty ones as CSV
10    case Faulty(time, id, since) => List(s"$time,$id,$since")
11    case _ => List() }

```

From a workflow perspective, this program integrates Fusemate as a list operator (on a list of Change instances) in an otherwise unremarkable functional program.

⁴ Technically, this is possible because the current interpretation is available in the rule body through the parameter I (see the transformation example above). One could directly access I, e.g., as in **CHOOSE**(a: atom, I), **MATCH**(State(t,3,c), a), t>10, c!="red"

For a more realistically sized experiment I tried a combined Fusemate/Scala workflow for analysing the data of the DEBS 2015 Grand Challenge.⁵ The data comprises two millions taxi rides in New York City in terms of start/end times, and start/end GPS coordinates, among others. The problem considered was to detect anomalies where a taxi driver drives away from a busy hotspot without a passenger. Solving the problem required clustering locations by pickup/drop-off activity for determining hotspots, and then analysing driver behavior given their pickups/drop-offs at these hotspots.

Two million data points were too much for Fusemate alone and required Scala preprocessing, e.g., for filling a grid abstraction of New York coordinates, data cleansing and filtering out little active drivers. Fusemate was used for computing clusters with rules similar to transitive closure computation. Input to Fusemate calls were Scala precomputed point clouds. The computed clusters were used to analyze Scala prefiltered taxi rides for anomaly detection based on the clusters. This involved three moderately complex rules, for first identifying gaps and then analysing them. The comprehension operator was useful to find “the most recent ride predating a given start”, among others. The longest Fusemate run was 0.31sec for 64 rides (with 39 clusters fixed), most other runs took less than 0.15sec. Fusemate’s performance was perfectly acceptable in this experiment thanks to a *combined* workflow.

5 Embedding Description Logic \mathcal{ALCIF}

\mathcal{ALCIF} is the well-known description logic \mathcal{ALC} extended with inverse roles and functional roles. (See [3] for background on description logics.) This section describes how to translate an \mathcal{ALCIF} knowledge base to Fusemate rules and facts for satisfiability checking.

This is our example knowledge base, TBox on the left, ABox on the right:

Person \sqsubseteq Rich \sqcup Poor	Anne : Person \sqcap Poor
Person $\sqsubseteq \exists \text{father}.\text{Person}$	(Anne, Fred) : father
Rich $\sqsubseteq \forall \text{father}^{-1}.\text{Rich}$	Bob : Person
Rich \sqcap Poor $\sqsubseteq \perp$	(Bob, Fred) : father

The father role is declared as functional, i.e., as a right-unique relation, and father^{-1} denotes its inverse “child” relation. The third GCI says that all children of a rich father are rich as well. In all models of the knowledge base Fred is Poor. This follows from the given fact that his child Anne is poor, functionality of father and the third CGI. However, there are models where Bob is Rich and models where Bob is Poor.

Translating description logic into rule-based languages has been done in many ways, see e.g. [20,17,14,11]. An obvious starting point is taking the FOL version of a given knowledge base. Concept names become unary predicates, role names become binary predicates, and GCIs (general concept inclusions) are translated into implications. By polynomial transformations, the implications can be turned into clausal form (if-then rules over literals), except for existential quantification in a positive context, which

⁵ <http://www.debs2015.org/call-grand-challenge.html>

causes unbounded Skolem terms in derivations when treated naively (for example, the third CGI above is problematic in this sense). This is why many systems and also the transformation to Fusemate below avoid Skolemization.

The first GCI corresponds to the clause $\text{Person}(x) \rightarrow \text{Rich}(x) \vee \text{Poor}(x)$, and the second corresponds to the “almost” clause $\text{Person}(x) \rightarrow \exists y. (\text{father}(x, y) \wedge \text{Person}(y))$. Fusemate works with the reified rule versions of these, with an *IsA*-predicate for concept instances, and a *HasA*-predicate for role instances. For the whole TBox one obtains the following, where RN stands for “role name” and CN stands for “concept name”.⁶

```

1 IsA(x, Exists(RN("father"), CN("Person")), time) :- IsA(x, CN("Person"), time)
2 IsA(x, CN("Rich"), time) OR IsA(x, CN("Poor"), time) :- IsA(x, CN("Person"), time)
3 IsA(x, Forall(Inv(RN("father")), CN("Rich")), time) :- IsA(x, CN("Rich"), time)
4 FAIL :- IsA(x, CN("Poor"), time), IsA(x, CN("Rich"), time)
5 functionalRoles = Set(RN("father"))

```

Every GCI can be converted into rules like the above without problems. For that, starting from its NNF, \exists -quantifications in the premise of a rule can be expanded in place, and \forall -quantifications can be moved to the head as the \exists -quantification of the NNF of the negated formula. Similarly for negated concept names. See [20] for such transformation methods. The ABox is represented similarly. Its first element, for instance, is *IsA*(Name("Anne"), And2(CN("Person"), CN("Poor")), 0).

In addition, some more general “library” rules for the tableau calculus are needed:

```

1 IsA(x, c1, time) AND IsA(x, c2, time) :- IsA(x, And2(c1, c2), time)
2 IsA(x, c1, time) OR IsA(x, c2, time) :- IsA(x, Or2(c1, c2), time)
3 // Expansion rules for quantifiers
4 IsA(y, c, time) :- Neighbour(x, r, y, time), IsA(x, Forall(r, c), time)
5 HasA(x, r, rSuccOfx, time+1) AND IsA(rSuccOfx, c, time+1): @preds("TimePlus1") :-
6   IsA(x, Exists(r, c), time), ! (functionalRoles contains r),
7   NOT( Neighbour(x, r, y, time), IsA(y, c, time) ), NOT( Blocked(x, _, time) ),
8   LET(rSuccOfx: Individual, Succ(r, x))

10 HasA(x, r, rSuccOfx, time+1) AND IsA(rSuccOfx, c, time+1): @preds("TimePlus1") :- (
11   IsA(x, Exists(r, c), time), functionalRoles contains r,
12   NOT( Neighbour(x, r, y, time) ), NOT( Blocked(x, _, time) ),
13   LET(rSuccOfx: Individual, Succ(r, x))

15 IsA(y, c, time) :-
16   IsA(x, Exists(r, c), time), functionalRoles contains r,
17   Neighbour(x, r, y, time)

```

The expansion rules on lines 1 and 2 deal with the \mathcal{ALC} binary Boolean connectives And2 and Or2 in the obvious way. Supposing NNF of embedded formulas, no other cases can apply. The remaining rules can be understood best with the standard tableau algorithm for \mathcal{ALCIN} in mind, which includes blocking to guarantee termination. They follow the terminology in [6, Chapter 4]. The Neighbour relation abstracts from the HasA relation, left away for space reasons. The expansion rule for \exists comes for three cases. The first case (line 5), for example, applies to non-functional roles as per the Scala builtin test on line 6. The expansion of the given \exists -formula only happens if it

⁶ See the Fusemate web page for the full, runnable code.

is not yet satisfied and in a non-blocked situation (line 7). In this case the rule derives a Skolem object defined on line 8 for satisfying the \exists -formula. Notice the annotation `@preds("TimePlus1")` which makes sure that the head is on the highest stratum. This way, the rule will be applied after, in particular, the rules for blocking. Furthermore, with the time stamp `time + 1` the Skolem object is kept separate from the computations in the current iteration time. The blocking rules are defined as follows:

```

1 // Collect all concepts that an individual x isA, at a given time
2 Label(x, cs.toSet, time) :- IsA(x, _, time), COLLECT(cs: List[Concept], c STH IsA(x, c, time))
3 // Ancestor relation of Skolem objects introduced by exists-right
4 Anc(x, Succ(r, x), time) :- HasA(x, r, Succ(r, x), time)
5 Anc(x, Succ(r, z), time) :- HasA(z, r, Succ(r, z), time), Step(time, prev), Anc(x, z, prev)
6 // Blocked case 1: y is blocked by some individual x according to 'pairwise ancestor blocking'
7 Blocked(y, x, time) :- Anc(x, y, time), Label(y, yIsAs, time), Label(x, xIsAs, time), yIsAs == xIsAs,
8   HasA(y1, r, y, time), HasA(x1, r, x, time), Label(y1, y1IsAs, time), Label(x1, x1IsAs, time), y1IsAs == x1IsAs
9 // Blocked case 2: y is blocked by some ancestor
10 Blocked(y, x, time) :- Anc(x, y, time), Blocked(x, _, time)

```

Some additional rules are needed for dealing with basic inconsistencies and for carrying over `IsA` and `HasA` facts between iterations. They are not shown here.

The expansion rules and blocking rules follow the tableau calculus description in [6, Chapter 4]. One important detail is that the expansion rule for \exists must be applied with lowest priority. This is straightforward thanks to Fusemate's stratification and aggregation construct. Equally important is the access to (Scala) data structures via built-ins and using them as terms of the logical language. This made it easy to program Skolemization and the `Label` relation for collecting sets of concepts of an individual.

6 Conclusions

This paper described recent developments around the Fusemate logic programming system. It included new technical improvements for a weaker form of stratification, which enabled useful aggregation and comprehension language constructs. It also argued for the advantages of the tight integration with Fusemate's host language, Scala, in terms of data structures and usability.

Answer set solvers like DLV and SModels are designed to solve NP-complete or higher complexity search problems as fast as possible. Fusemate is not motivated as a competitive such system, it is motivated for "well-behaved" knowledge representation applications, similarly to description logic reasoners, whose (often) NExpTime complete solving capabilities are not expected to be typically needed. (Some more work is needed, though, e.g., on improving the current term indexing techniques to speed up model computation.) More specifically, the main intended application of Fusemate is for the runtime analysis of systems that evolve over time. The taxi rides data experiment explained in Section 4 is an example for that. It suggests that Fusemate is currently best used in a combined problem solving workflow if scalability is an issue.

As for future work, the next steps are to make the description logic reasoner of Section 5 callable from within Fusemate rules in a DL-safe way [19] and to embed a temporal reasoning formalism. The event calculus [15] seems to be a good fit.

Acknowledgements. I am grateful to the reviewers for their helpful comments.

References

1. Alviano, M., Faber, W., Leone, N., Perri, S., Pfeifer, G., Terracina, G.: The disjunctive datalog system DLV. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A.J. (eds.) *Datalog Reloaded - First International Workshop, Datalog 2010*, Oxford, UK, March 16-19, 2010. Revised Selected Papers. Lecture Notes in Computer Science, vol. 6702, pp. 282–301. Springer (2010). https://doi.org/10.1007/978-3-642-24206-9_17, https://doi.org/10.1007/978-3-642-24206-9_17
2. Artikis, A., Skarlatidis, A., Portet, F., Paliouras, G.: Logic-based event recognition. *Knowl. Eng. Rev.* **27**(4), 469–506 (2012). <https://doi.org/10.1017/S0269888912000264>, <https://doi.org/10.1017/S0269888912000264>
3. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): *Description Logic Handbook*. Cambridge University Press (2002)
4. Baader, F., Bauer, A., Baumgartner, P., Cregan, A., Gabaldon, A., Ji, K., Lee, K., Rajaratnam, D., Schwitler, R.: A novel architecture for situation awareness systems. In: Giese, M., Waaler, A. (eds.) *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2009)*. LNAI, vol. 5607, pp. 77–92. Springer (July 2009). https://doi.org/10.1007/978-3-642-02716-1_7, SAIL-TABLEAUX-09.pdf
5. Baader, F., Borgwardt, S., Lippmann, M.: Temporal conjunctive queries in expressive description logics with transitive roles. In: Pfahring, B., Renz, J. (eds.) *AI 2015: Advances in Artificial Intelligence - 28th Australasian Joint Conference, Canberra, ACT, Australia, November 30 - December 4, 2015, Proceedings*. Lecture Notes in Computer Science, vol. 9457, pp. 21–33. Springer (2015). https://doi.org/10.1007/978-3-319-26350-2_3, https://doi.org/10.1007/978-3-319-26350-2_3
6. Baader, F., Horrocks, I., Lutz, C., Sattler, U.: *An Introduction to Description Logic*. Cambridge University Press (2017), <http://www.cambridge.org/de/academic/subjects/computer-science/knowledge-management-databases-and-data-mining/introduction-description-logic?format=PB#17zVGwD2TZUeu6s.97>
7. Baumgartner, P.: Possible Models Computation and Revision – A Practical Approach. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) *International Joint Conference on Automated Reasoning*. LNAI, vol. 12166, pp. 337–355. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_19, possible-models-IJCAR-2020.pdf
8. Baumgartner, P., Furbach, U., Niemelä, I.: Hyper Tableaux. In: *Logics in Artificial Intelligence (JELIA '96)*. No. 1126 in *Lecture Notes in Artificial Intelligence*, Springer (1996), tableaux-jelia-11nscs.pdf
9. Beck, H., Dao-Tran, M., Eiter, T.: LARS: A logic-based framework for analytic reasoning over streams. *Artif. Intell.* **261**, 16–70 (2018). <https://doi.org/10.1016/j.artint.2018.04.003>, <https://doi.org/10.1016/j.artint.2018.04.003>
10. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence* **50**, 333–361 (08 2007). <https://doi.org/10.1007/s10472-007-9076-z>
11. Carral, D., Krötzsch, M.: Rewriting the description logic ALCHIQ to disjunctive existential rules. In: Bessiere, C. (ed.) *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. pp. 1777–1783. [ijcai.org](https://doi.org/10.24963/ijcai.2020/246) (2020). <https://doi.org/10.24963/ijcai.2020/246>, <https://doi.org/10.24963/ijcai.2020/246>
12. Cat, B.D., Bogaerts, B., Bruynooghe, M., Janssens, G., Denecker, M.: Predicate logic as a modeling language: the IDP system. In: Kifer, M., Liu, Y.A. (eds.) *Declarative Logic Programming: Theory, Systems, and Applications*, pp. 279–323. ACM / Morgan & Claypool (2018). <https://doi.org/10.1145/3191315.3191321>, <https://doi.org/10.1145/3191315.3191321>

13. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = ASP + control: Preliminary report. CoRR **abs/1405.3694** (2014), <http://arxiv.org/abs/1405.3694>
14. Grosz, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: combining logic programs with description logic. In: Hencsey, G., White, B., Chen, Y.R., Kovács, L., Lawrence, S. (eds.) Proceedings of the Twelfth International World Wide Web Conference, WWW 2003, Budapest, Hungary, May 20-24, 2003. pp. 48–57. ACM (2003). <https://doi.org/10.1145/775152.775160>, <https://doi.org/10.1145/775152.775160>
15. Kowalski, R.A., Sergot, M.J.: A Logic-based Calculus of Events. New Generation Computing **4**(1), 67–95 (1986). <https://doi.org/10.1007/BF03037383>
16. Kozlenkov, A., Peñaloza, R., Nigam, V., Royer, L., Dawelbait, G., Schroeder, M.: Prova: Rule-based java scripting for distributed web applications: A case study in bioinformatics. In: EDBT Workshops. LNCS, vol. 4254, pp. 899–908. Springer (2006)
17. Lukácsy, G., Szeredi, P.: Efficient description logic reasoning in prolog: The dlog system. Theory Pract. Log. Program. **9**(3), 343–414 (2009). <https://doi.org/10.1017/S1471068409003792>, <https://doi.org/10.1017/S1471068409003792>
18. Manthey, R., Bry, F.: SATCHMO: a theorem prover implemented in Prolog. In: Lusk, E., Overbeek, R. (eds.) Proceedings of the 9th Conference on Automated Deduction, Argonne, Illinois, May 1988. Lecture Notes in Computer Science, vol. 310, pp. 415–434. Springer (1988)
19. Motik, B., Sattler, U., Studer, R.: Query answering for owl-dl with rules. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) The Semantic Web – ISWC 2004. pp. 549–563. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
20. Motik, B., Shearer, R., Horrocks, I.: Hypertableau reasoning for description logics. J. Artif. Intell. Res. **36**, 165–228 (2009). <https://doi.org/10.1613/jair.2811>, <https://doi.org/10.1613/jair.2811>
21. Przymusiński, T.C.: Chapter 5 - on the declarative semantics of deductive databases and logic programs. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 193 – 216. Morgan Kaufmann (1988). <https://doi.org/https://doi.org/10.1016/B978-0-934613-40-8.50009-9>, <http://www.sciencedirect.com/science/article/pii/B9780934613408500099>
22. Redl, C.: The dlhex system for knowledge representation: Recent advances (system description). Theory and Practice of Logic Programming **16** (07 2016). <https://doi.org/10.1017/S1471068416000211>
23. Sakama, C.: Possible Model Semantics for Disjunctive Databases. In: Kim, W., Nicholas, J.M., Nishio, S. (eds.) Proceedings First International Conference on Deductive and Object-Oriented Databases (DOOD-89). pp. 337–351. Elsevier Science Publishers B.V. (North-Holland) Amsterdam (1990)
24. Sakama, C., Inoue, K.: An Alternative Approach to the Semantics of Disjunctive Logic Programs and Deductive Databases. Journal of Automated Reasoning **13**, 145–172 (1994)
25. The Scala Programming Language, <https://www.scala-lang.org>
26. Syrjänen, T., Niemelä, I.: The smodels system. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2173, pp. 434–438. Springer (2001). https://doi.org/10.1007/3-540-45402-0_38, https://doi.org/10.1007/3-540-45402-0_38