The Fusemate Logic Programming System

Peter Baumgartner
Data61|CSIRO and ANU, Canberra, Australia
Fusemate - Language and Model Computation Overview
Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)
Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

\[ R(a,b) \]
\[ R(X,Y) :- R(Y,X) \]
\[ R(X,Z) :- R(X,Y), r(Y,Z) \]

Models

\[ R(a,b) \]
\[ R(b,a) \]

Bottom-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, ...)


Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

\[
\begin{align*}
R(a,b) \\
R(X,Y) & : - R(Y,X) \\
R(X,Z) & : - R(X,Y), \ r(Y,Z)
\end{align*}
\]

Models

\[
\begin{align*}
R(a,b) \\
R(b,a)
\end{align*}
\]

Application:
Situational awareness
= model computation

Bottom-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, ...)

Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)

Models

R(a,b)
R(b,a)

Default negation: stratification “by time”

GoodSleep(time) :-
    WakeUp(time),
    GoToBed(t), t <= time - 8,
    not (s < time, t < s, WakeUp(s))

Application:
Situational awareness
= model computation

Bottom-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, ...)

Application:
Situational awareness
= model computation
Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)

Default negation: stratification “by time”

GoodSleep(time) :-
    WakeUp(time),
    GoToBed(t), t <= time - 8,
    not (s < time, t < s, WakeUp(s))

Models
R(a,b)
R(b,a)

Application:
Situational awareness
= model computation

Bottom-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, ...)

“not” subgoals must be strictly earlier “<“ than current time
Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)

Models
R(a,b)  R(b,a)

Default negation: stratification “by time”

GoodSleep(time) :-
    WakeUp(time),
    GoToBed(t), t <= time - 8,
    not (s < time, t < s, WakeUp(s))

Application:
Situational awareness
= model computation

Bottom-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, …)

“not” subgoals must be strictly earlier “<“ than current time

unhappy(time) :- Now(time), not winLottery(time+7)
Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), R(Y,Z)

Default negation: stratification “by time”

GoodSleep(time) :-
  WakeUp(time),
  GoToBed(t), t <= time - 8,
  not (s < time, t < s, WakeUp(s))

Models

R(a,b)
R(b,a)

Application:
Situation awareness
= model computation

Bottom-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, ...)

“not” subgoals must be strictly earlier “<” than current time

unhappy(time) :- Now(time), not winLottery(time+7)
Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

\[ R(a,b) \]
\[ R(X,Y) :- R(Y,X) \]
\[ R(X,Z) :- R(X,Y), r(Y,Z) \]

Default negation: stratification “by time”

\[
\text{GoodSleep}(\text{time}) :- \\
\quad \text{WakeUp}(\text{time}), \\
\quad \text{GoToBed}(\text{t}), \text{t} \leq \text{time} \,-\, 8, \\
\quad \textbf{not} (\text{time}, \text{time} \leq \text{s}, \text{WakeUp}(\text{s}))
\]

Disjunctions: possible model semantics [Sakama 90]

\[
\text{Thirsty}(\text{time}) \textbf{ or } \text{Hungry}(\text{time}) :- \text{GoodSleep}(\text{time})
\]

Application:
Situational awareness
= model computation

Bottom-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, …)
Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

\[
R(a,b) \\
R(X,Y) :- R(Y,X) \\
R(X,Z) :- R(X,Y), R(Y,Z)
\]

Models

\[
R(a,b) \\
R(b,a)
\]

Default negation: stratification “by time”

GoodSleep(time) :-
    WakeUp(time),
    GoToBed(t), t <= time - 8,
    not (s < time, t < s, WakeUp(s))

Disjunctions: possible model semantics [Sakama 90]

Thirsty(time) or Hungry(time) :- GoodSleep(time)

Models

\[
\text{Thirsty}(10) \quad \text{Hungry}(10) \quad \text{Hungry}(10) \quad \text{Thirsty}(10)
\]

Application:
Situational awareness
= model computation

Bottom-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, …)

“not” subgoals must be strictly earlier “<“ than current time

unhappy(time) :- Now(time), not winLottery(time+7)

Inclusive reading of “or”
Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

\[ R(a,b) \]
\[ R(X,Y) :\neg R(Y,X) \]
\[ R(X,Z) :\neg R(X,Y), R(Y,Z) \]

Default negation: stratification “by time”

\[ \text{GoodSleep}(time) :\neg \]
\[ \quad \text{WakeUp}(time), \]
\[ \quad \text{GoToBed}(t), t \leq time - 8, \]
\[ \quad \neg (s < time, t < s, \text{WakeUp}(s)) \]

Disjunctions: possible model semantics [Sakama 90]

\[ \text{Thirsty}(time) \text{ or } \text{Hungry}(time) :\neg \text{GoodSleep}(time) \]

Belief revision

\[ \text{fail}(+ \text{GoToBed}(time - 8)) :\neg \]
\[ \quad \text{WakeUp}(time), \]
\[ \quad \neg (\text{GoToBed}(t), t \leq time - 8) \]

Application:
Situational awareness
= model computation

Bottom-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, ...)

“not” subgoals must be strictly earlier “<“ than current time

\[ \text{unhappy}(time) :\neg \text{Now}(time), \neg \text{winLottery}(time+7) \]

Inclusive reading of “or”

\[
\begin{array}{ccc}
\text{Models} \\
\hline
\text{Thirsty}(10) & \text{Hungry}(10) & \text{Hungry}(10) \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{Models} \\
\hline
\text{Thirsty}(10) & \text{Hungry}(10) & \text{Thirsty}(10) \\
\end{array}
\]

2
Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

\[ R(a,b) \]
\[ R(X,Y) :- R(Y,X) \]
\[ R(X,Z) :- R(X,Y), r(Y,Z) \]

Default negation: stratification “by time”

\[ \text{GoodSleep}(\text{time}) :- \]
\[ \quad \text{Wakeup}(\text{time}), \]
\[ \quad \text{GoToBed}(t), t \leq \text{time} - 8, \]
\[ \quad \text{not} (s < \text{time}, t < s, \text{Wakeup}(s)) \]

Disjunctions: possible model semantics [Sakama 90]

\[ \text{Thirsty}(\text{time}) \text{ or } \text{Hungry}(\text{time}) :- \text{GoodSleep}(\text{time}) \]

Belief revision [IJCAR 2020]

\[ \text{fail}(+ \text{GoToBed}(\text{time} - 8)) :- \]
\[ \quad \text{Wakeup}(\text{time}), \]
\[ \quad \text{not} (\text{GoToBed}(t), t \leq \text{time} - 8) \]

Application:
Situational awareness
= model computation

\[ R(a,b) \quad R(b,a) \]

Botton-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, ...)

“not” subgoals must be strictly earlier “<“ than current time

\[ \text{unhappy}(\text{time}) :- \text{Now}(\text{time}), \text{not} \text{winLottery}(\text{time+7}) \]

Inclusive reading of “or”

<table>
<thead>
<tr>
<th>Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thirsty(10)</td>
</tr>
</tbody>
</table>

Application:
Situational awareness
= model computation
Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)

Default negation: stratification "by time"

GoodSleep(time) :-
  WakeUp(time),
  GoToBed(t), t <= time - 8,
  not (s < time, t < s, WakeUp(s))

Disjunctions: possible model semantics [Sakama 90]

Thirsty(time) or Hungry(time) :- GoodSleep(time)

Belief revision [IJCAR 2020]

fail(+ GoToBed(time - 8)) :-
  WakeUp(time),
  not (GoToBed(t), t <= time - 8)

Application:
Situational awareness
= model computation

Bottom-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, ...)

Models

\begin{align*}
\text{R}(a,b) \\
\text{R}(b,a) \\
\end{align*}

"not" subgoals must be strictly earlier "<" than current time

unhappy(time) :- Now(time), not winLottery(time+7)

Inclusive reading of "or"

Models

\begin{align*}
\text{Thirsty(10)} & \quad \text{Hungry(10)} & \quad \text{Hungry(10)} \\
\text{Thirsty(10)} & \quad \text{Hungry(10)} \\
\end{align*}

What's special?

What's new?
What’s Special?

Implementation language: Scala

• Scala combines **object-oriented** and **functional programming**
  
  ```scala
def qsort(l: List[Int]): List[Int] = 
  l match {
    case Nil           => Nil
    case pivot :: tail => qsort(tail filter {_ < pivot}) ::: pivot :::
          qsort(tail filter {_ >= pivot})
  }
  ```

• Access to **huge ecosystem of libraries**

• Runs on JVM; compiled or in data-analysis style **interactive workbooks** (Jupyter)
What’s Special?

Implementation language: Scala

• Scala combines **object-oriented** and **functional programming**

  ```scala
def qsort(l: List[Int]): List[Int] =
  l match {
    case Nil           => Nil
    case pivot :: tail => qsort(tail filter {_ < pivot}) ::: pivot :::
                        qsort(tail filter {_ >= pivot})
  }
```

  **Pattern matching**

• Access to **huge ecosystem of libraries**

• Runs on JVM; compiled or in data-analysis style **interactive workbooks** (Jupyter)
What’s Special?

Implementation language: Scala

• Scala combines **object-oriented** and **functional programming**

```scala
def qsort(l: List[Int]): List[Int] = 
l match {
  case Nil           => Nil
  case pivot :: tail => qsort(tail filter {_ < pivot}) ::: pivot :::
                     qsort(tail filter {_ >= pivot})
}
```

• **Pattern matching**

• Access to **huge ecosystem of libraries**

• Runs on JVM; compiled or in data-analysis style **interactive workbooks** (Jupyter)

Implementation technique: shallow embedding

• Logic program **translated into** Scala program that is executed for model computation

• AFAIK Fusemate is the only logic programming system implemented that way

• **Q: what are the advantages/disadvantages of this approach?**

  E.g. in terms of capitalizing on / integrating the **above features** of Scala
Shallow Embedding Into Scala

- User writes Scala program with rules embedded into it

```scala
type Time = Int
case class GoodSleep(time: Time) extends Atom
@rules
...
GoodSleep(time) :-
    WakeUp(time),
    GoToBed(t), t <= time - 8,
    not (s < time, t < s, WakeUp(s))
```
Shallow Embedding Into Scala

- User writes Scala program with rules embedded into it

```scala
type Time = Int
case class GoodSleep(time: Time) extends Atom
@rules
...
GoodSleep(time) :-
    WakeUp(time),
    GoToBed(t), t <= time - 8,
    not (s < time, t < s, WakeUp(s))
```
Shallow Embedding Into Scala

- User writes Scala program with rules embedded into it

```scala
type Time = Int
case class GoodSleep(time: Time) extends Atom
@rules
...
GoodSleep(time) :-
  WakeUp(time),
  GoToBed(t), t <= time - 8,
  not (s < time, t < s, WakeUp(s))
```
Shallow Embedding Into Scala

- User writes Scala program with rules embedded into it

```
import scala.language.experimental.macros

type Time = Int

case class GoodSleep(time: Time) extends Atom

@rules
...

GoodSleep(time) :-
  WakeUp(time),
  GoToBed(t), t <= time - 8,
  not (s < time, t < s, WakeUp(s))
```

- The rules are macro expanded into Scala curried partial functions

```
(I: Interpretation) => {
  case WakeUp(time) => {
    case GoToBed(t) if t <= time - 8 && I.failsOn("body of not") => GoodSleep(time)
  }
}
```
Shallow Embedding Into Scala

• User writes Scala program with rules embedded into it

\[
\begin{align*}
\text{type Time} & = \text{Int} \\
\text{case class GoodSleep(time: Time) extends Atom} \\
& @rules \\
& \ldots \\
& \text{GoodSleep(time)} : - \\
& \quad \text{WakeUp(time)}, \\
& \quad \text{GoToBed(t), t} \leq \text{time} - 8, \\
& \quad \text{not (s < time, t < s, WakeUp(s))}
\end{align*}
\]

• The rules are macro expanded into Scala curried partial functions

\[
(I: \text{Interpretation}) \Rightarrow \{
\begin{align*}
\text{case WakeUp(time)} & \Rightarrow \{
\text{case GoToBed(t) if t} \leq \text{time} - 8 \&\& I\text{.failsOn("body of not")} \Rightarrow \text{GoodSleep(time)}
\}
\}
\}
\]
Shallow Embedding Into Scala

- User writes Scala program with rules embedded into it

\[
\begin{align*}
\text{type } & \text{ Time } = \text{ Int} \\
\text{case class } & \text{ GoodSleep} (\text{time}: \text{ Time}) \text{ extends Atom} \\
\text{@rules} & \\
\text{...} & \\
\text{GoodSleep(\text{time}) :- } & \\
& \text{WakeUp(\text{time}),} \\
& \text{GoToBed(t), } t <= \text{ time} - 8, \\
& \text{not } (s < \text{ time}, t < s, \text{WakeUp(s)})
\end{align*}
\]

- The rules are macro expanded into Scala curried partial functions

\[
\begin{align*}
(I: \text{ Interpretation}) & \Rightarrow \{ \\
\text{case WakeUp(\text{time})} & \Rightarrow \{ \\
\text{case GoToBed(t)} & \text{if } t <= \text{ time} - 8 \&\& I.\text{failsOn(\text{“body of not”})} \Rightarrow \text{GoodSleep(\text{time})} \\
\} & \\
\}
\end{align*}
\]

Function application mimics rule evaluation

Recursive call of model computation
User writes Scala program with rules embedded into it

```
val Time = Int

case class GoodSleep(time: Time) extends Atom

@rules
... 
GoodSleep(time) :- 
    WakeUp(time),
    GoToBed(t), t <= time - 8,
    not (s < time, t < s, WakeUp(s))
```

Rules

- The rules are macro expanded into Scala curried partial functions

```
(I: Interpretation) => {
    case WakeUp(time) => {
        case GoToBed(t) if t <= time - 8 && I.failsOn("body of not") => GoodSleep(time)
    }
}
```

- Given-clause loop operating on rules-as-partial-functions and interpretations (tableaux)
Shallow Embedding Into Scala

- User writes Scala program with rules embedded into it

\[\text{type } \text{Time} = \text{Int}\]
\[\text{case class } \text{GoodSleep}(\text{time}: \text{Time}) \text{ extends } \text{Atom}\]

@rules

\[
\text{GoodSleep}(\text{time}) \text{ :- } \\
\text{WakeUp}(\text{time}), \\
\text{GoToBed}(\text{t}), \text{ t } \leq \text{ time } - 8, \\
\text{not (s } < \text{ time, t } < \text{ s, WakeUp(s)})
\]

- The rules are macro expanded into Scala curried partial functions

\[
(I: \text{Interpretation}) \Rightarrow \{ \\
\text{case WakeUp(} \text{time} ) \Rightarrow \{ \\
\text{case GoToBed(} \text{t} ) \text{ if } \text{t } \leq \text{ time } - 8 \&\& \text{I}.failsOn(\text{"body of not") } \Rightarrow \text{GoodSleep(} \text{time} ) \\
\}
\}
\]

- Given-clause loop operating on rules-as-partial-functions and interpretations (tableaux)

All logic notions are Scala

- “Interpretation” available as term
- Trivial interface to/from Scala
- Type checking/inference for free

Every Scala term is a term of the logic

Logic | Scala
--- | ---
Pred/Fun signature | Class declaration
Atom/Term | Class instance
Interpretation | Set of class instances
Variable | Variable
Rule | Partial function
Matching subst | Pattern matching

Function application mimics rule evaluation

Recursive call of model computation

Σ
Rules
What’s New? (1)
What’s New? (1)

General aggregation operator
What’s New? (1)

General aggregation operator

• Many LP systems (DLV, IDP, Gringo, …) support aggregation ops \#count \#sum \#times \#min \#max

\[2 < \#count \{ \text{time} : \text{GoodSleep}(\text{time}) \}\]
What’s New? (1)

General aggregation operator

• Many LP systems (DLV, IDP, Gringo, ...) support aggregation ops \texttt{#count \#sum \#times \#min \#max}

\[
2 < \#\text{count} \left\{ \text{time} : \text{GoodSleep} (\text{time}) \right\}
\]

• Fusemate implements a more general stratified \texttt{collect} operator

\[
\texttt{collect}(\text{gsTimes, time sth GoodSleep(time)})
\]

Semantics: \texttt{gsTimes} = \{ \text{time} | I \models \text{GoodSleep(time)} \}
What’s New? (1)

General aggregation operator

• Many LP systems (DLV, IDP, Gringo, …) support aggregation ops #count #sum #times #min #max

2 < #count { time : GoodSleep(time) }

• Fusemate implements a more general stratified collect operator

collect(gsTimes, time sth GoodSleep(time))

Semantics: gsTimes = { time | I ⊨ GoodSleep(time) }
What’s New? (1)

General aggregation operator

• Many LP systems (DLV, IDP, Gringo, ...) support aggregation ops \#count \#sum \#times \#min \#max

\[ 2 < \#\text{count} \{ \text{time} : \text{GoodSleep}(\text{time}) \} \]

• Fusemate implements a more general stratified collect operator “Logic term = Scala term”

\[
\text{collect}(\text{gsTimes}, \text{time sth GoodSleep}(\text{time}))
\]

Semantics: \[ \text{gsTimes} = \{ \text{time} | I \models \text{GoodSleep}(\text{time}) \} \]

• Recover standard aggregation functionality \#... with Scala operator

\[ 2 < \text{gsTimes}.\text{size} \]
What’s New? (1)

General aggregation operator

- Many LP systems (DLV, IDP, Gringo, …) support aggregation ops \#count \#sum \#times \#min \#max

  \[ 2 \ < \ \#count \ \{ \ \text{time} : \ \text{GoodSleep}(\text{time}) \} \]

- Fusemate implements a more general stratified \texttt{collect} operator  

  \texttt{collect(gsTimes, \text{time sth GoodSleep(time))}}

  Semantics: \texttt{gsTimes = \{ \text{time} \ \mid \ I \models \text{GoodSleep}(\text{time}) \}}

- Recover standard aggregation functionality \#… with Scala operator

  \[ 2 \ < \ \texttt{gsTimes.size} \]

- But is more expressive

  \( (\texttt{gsTimes map \{ _ \ % \ 24 \} foldLeft(0) \{ _ + _ \}) / \texttt{gsTimes.size} \)
What’s New? (1)

General aggregation operator

• Many LP systems (DLV, IDP, Gringo, …) support aggregation ops #count #sum #times #min #max

\[ 2 < \#\text{count} \{ \text{time} : \text{GoodSleep}(\text{time}) \} \]

• Fusemate implements a more general stratified collect operator “Logic term = Scala term”

\[
\text{collect}(\text{gsTimes}, \text{time sth GoodSleep}(\text{time}))
\]

Semantics: \( \text{gsTimes} = \{ \text{time} | I \models \text{GoodSleep}(\text{time}) \} \)

• Recover standard aggregation functionality #… with Scala operator

\[ 2 < \text{gsTimes}.\text{size} \]

• But is more expressive

\[
(\text{gsTimes} \map \{ _ \% 24 \} \text{foldLeft}(0) \{ _ + _ \}) / \text{gsTimes}.\text{size}
\]

Comprehension operator

\[
\text{choose}(t < \text{time sth GoodSleep}(t))
\]

“The most recent \( t \) before \( \text{time} \) such that \( \text{GoodSleep}(t) \)”

• Useful for analysing “current state” in situational awareness application
What’s New? (1)

General aggregation operator

• Many LP systems (DLV, IDP, Gringo, ...) support aggregation ops \#count \#sum \#times \#min \#max

  \[
  2 \prec \#\text{count} \{ \text{time} : \text{GoodSleep}(\text{time}) \} \\
  \]

• Fusemate implements a more general stratified collect operator

  \[
  \text{collect}(\text{gsTimes}, \text{time sth GoodSleep}(\text{time})) \\
  \]

  Semantics: \text{gsTimes} = \{ \text{time} \mid I \not\models \text{GoodSleep}(\text{time}) \} \\

• Recover standard aggregation functionality \#… with Scala operator

  \[
  2 \prec \text{gsTimes}.\text{size} \\
  \]

• But is more expressive

  \[
  (\text{gsTimes map \{ _ \% 24 \}} \text{foldLeft}(0) \{ _ + _ \}) / \text{gsTimes}.\text{size} \\
  \]

Comprehension operator

These operators are user-definable

\[
\text{choose}(t \prec \text{time sth GoodSleep}(t)) \\
\]

  “The most recent \text{t before time such that GoodSleep}(\text{t})”

• Useful for analysing “current state” in situational awareness application
What’s New? (2)

Stratification by predicates and by time (SBTP)

- Stratification disallows definitonal loop through “not <body>” literal
- Stratification renders “not <body>” evaluation monotonic

\[
p(time) :- q(time), \text{not } r(time-1) \\
r(time) :- p(time)
\]

\[
q(time) :- p(time), \text{not } s(time)
\]

SBTP = lexicographic combination of “by time” and “by predicates”
What’s New (1) - (2) Showcase - Fusemate as Description Logic Reasoner

Description logic ALCIF

Person ⫋ Rich ⫄ Poor
Person ⫋ ∃father.Person
Rich ⫋ ∀father⁻¹.Rich
Rich ⟪ Poor ⫡ ⊥

Anne : Person ∩ Poor
(Anne, Fred) : father
Bob : Person
(Bob, Fred) : father

father is functional

5 Embedding Description Logic

ALCIF

ALCIF is the well-known description logic ALC extended with inverse roles and functional roles. See [S] for background on description logics. This section describes how to translate an ALCIF knowledge base to fusemate rules and facts for satisfiability checking.

This is our example knowledge base:

\[
\begin{align*}
    \text{Person} & \sqsubseteq \text{Rich} \sqcup \text{Poor} \\
    \text{Person} & \sqsubseteq \exists \text{father}.\text{Person} \\
    \text{Rich} & \sqsubseteq \forall \text{father}^{-1}.\text{Rich} \\
    \text{Rich} \cap \text{Poor} & \sqsubseteq \bot
\end{align*}
\]

Anne : Person \cap Poor
(Anne, Fred) : father
Bob : Person
(Bob, Fred) : father

father is functional

5 Embedding Description Logic

ALCIF

ALCIF is the well-known description logic ALC extended with inverse roles and functional roles. See [S] for background on description logics. This section describes how to translate an ALCIF knowledge base to fusemate rules and facts for satisfiability checking.

This is our example knowledge base:

\[
\begin{align*}
    \text{Person} & \sqsubseteq \text{Rich} \sqcup \text{Poor} \\
    \text{Person} & \sqsubseteq \exists \text{father}.\text{Person} \\
    \text{Rich} & \sqsubseteq \forall \text{father}^{-1}.\text{Rich} \\
    \text{Rich} \cap \text{Poor} & \sqsubseteq \bot
\end{align*}
\]

Anne : Person \cap Poor
(Anne, Fred) : father
Bob : Person
(Bob, Fred) : father

father is functional


Description logic ALCIF

father is functional
What’s New (1) - (2) Showcase - Fusemate as Description Logic Reasoner

Description logic ALCIF

Person ⊑ Rich ⊔ Poor
Person ⊑ ∃father.Person
Rich ⊑ ∀father⁻¹.Rich
Rich ⊓ Poor ⊑ ⊥

Anne : Person ⊓ Poor
(Anne, Fred) : father
Bob : Person
(Bob, Fred) : father

father is functional

Iterative algorithm
Uses SBTP
Uses aggregation
Paper has details
What’s New (1) - (2) Showcase - Fusemate as Description Logic Reasoner

Description logic ALCIF

Person ⊑ Rich ⊔ Poor
Person ⊑ ∃father.Poor
Rich ⊑ ∀father⁻¹.Rich
Rich ⊓ Poor ⊑ ⊥

father is functional

Anne : Person ⊓ Poor
(Anne, Fred) : father
(Bob, Fred) : father

Bob : Person

As a logic program

\[
\text{IsA}(x, \text{Exists}(\text{RN}("father"), \text{CN}"Person"), \text{time}) :- \\
\text{IsA}(x, \text{CN}"Person", \text{time})
\]

Iterative algorithm

Uses SBTP
Uses aggregation
Paper has details
What’s New (1) - (2) Showcase - Fusemate as Description Logic Reasoner

Description logic ALCIF

Person ⊑ Rich ⊔ Poor
Person ⊑ ∃father.Person
Rich ⊑ ∀father⁻¹.Rich
Rich ⊓ Poor ⊑ ⊥

Anne : Person ⊑ Poor
(Anne, Fred) : father
Bob : Person
(Bob, Fred) : father

father is functional

As a logic program

IsA(x, Exists(RN("father"), CN("Person")), time) :-
IsA(x, CN("Person"), time)

ALCIF satisfiability = LP satisfiability”

• LP encodes standard tableau construction [Baader et al 2017]
  • “Time” is quantifier expansion depth
  • TBox -> rules, ABox -> facts
  • Some general library rules
  • Requires model inspection for “double blocking”

Iterative algorithm

Uses SBTP

Uses aggregation

Paper has details
What's New (1) - (2) Showcase - Fusemate as Description Logic Reasoner

Description logic ALCIF

Person ⊑ Rich ⊉ Poor
Person ⊑ ∃father.Person
Rich ⊑ ∀father⁻¹.Rich
Rich ⊑ Poor ⊑ ⊥

Anne : Person △ Poor
(Anne, Fred) : father
Bob : Person
(Bob, Fred) : father

father is functional

As a logic program

IsA(x, Exists(RN("father"), CN("Person")), time) :-
IsA(x, CN("Person"), time)

ALCIF satisfiability = LP satisfiability

- LP encodes standard tableau construction [Baader et al 2017]
- “Time” is quantifier expansion depth
- TBox -> rules, ABox -> facts
- Some general library rules
- Requires model inspection for “double blocking”

Iterative algorithm

Uses SBTP
Uses aggregation
Paper has details

Label(x, cs, time) :-
IsA(x, _, time),
COLLECT(cs, c STH IsA(x, c, time))

// Pairwise blocking
// y is blocked by x if ...
Blocked(y, x, time) :-
// ... x is an ancestor of y,
Anc(x, y, time),
// ... the labels of y and x are the same
Label(y, yIsAs, time),
Label(x, xIsAs, time),
yIsAs ≡ xIsAs,
// ... y and x are r-successors of some y1 and x1, for some r
HasA(y1, r, y, time),
HasA(x1, r, x, time),
// ... the labels of y1 and x1 are the same
Label(y1, y1IsAs, time),
Label(x1, x1IsAs, time),
y1IsAs = x1IsAs
What's New (1) - (2) Showcase - Fusemate as Description Logic Reasoner

Description logic ALCIF

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>Rich ⊑ Poor</td>
</tr>
<tr>
<td>Person</td>
<td>∃father.Person</td>
</tr>
<tr>
<td>Rich</td>
<td>∀father⁻¹.Rich</td>
</tr>
<tr>
<td>Rich ⊓ Poor</td>
<td>⊥</td>
</tr>
</tbody>
</table>

Anne : Person ⊓ Poor (Anne, Fred) : father
Bob : Person (Bob, Fred) : father

father is functional

As a logic program

\[
\text{IsA}(x, \text{Exists(RN("father"), CN("Person")), time}) :- \\
\text{IsA}(x, \text{CN("Person"), time}) \\
\]

ALCIF satisfiability = LP satisfiability

- LP encodes standard tableau construction [Baader et al 2017]
- “Time” is quantifier expansion depth
- TBox -> rules, ABox -> facts
- Some general library rules
- Requires model inspection for “double blocking”

Iterative algorithm

Uses SBTP
Uses aggregation
Paper has details

Label(x, cs, time) :-
ISA(x, _, time),
COLLECT(cs, c STH ISA(x, c, time))

// Pairwise blocking
// y is blocked by x if ...
Blocked(y, x, time) :-
// ... x is an ancestor of y,
Anc(x, y, time),
// ... the labels of y and x are the same
Label(y, yIsAs, time),
Label(x, xIsAs, time),
yIsAs ≡ xIsAs,
// ... y and x are r-successors of some y1 and x1, for some relation r
HasA(y1, r, y, time),
HasA(x1, r, x, time),
// ... the labels of y1 and x1 are the same
Label(y1, y1IsAs, time),
Label(x1, x1IsAs, time),
y1IsAs ≡ x1IsAs

Textbook 1-to-1


Description logic ALCIF

ALCIF satisfiability = LP satisfiability

• LP encodes standard tableau construction [Baader et al 2017]
• “Time” is quantifier expansion depth
• TBox -> rules, ABox -> facts
• Some general library rules
• Requires model inspection for “double blocking”
What’s New (3) - Usability and Workflow

Case study for combined Scala / logic programming workflow

2 Million taxi rides in New York City
Ride(taxi,license,from,to,start,end,fare)

(1) Rules for gaps, pickup/dropoff clustering and concave hull
(2) Rules for anomaly detection

---
driver license-3568
---
taxi-3568 license-3568 2013-01-01T22:10 2013-01-01T22:38 28m 5.7km
pickup anomaly from: hotspot-15

<table>
<thead>
<tr>
<th>hour</th>
<th>pickups</th>
<th>dropoffs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>34</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>35</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>73</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td>48</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>17</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>22</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>28</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>44</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>43</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>116</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>76</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>76</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>83</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>74</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>70</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>76</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>36</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>13</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>34</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
<td>18</td>
</tr>
</tbody>
</table>
What’s New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```scala
val gaps42 = rides filter {
  _.license ≡ "42"
} saturateFirst {
  Gap(taxi, license, prevEnd, start, prevTo, from) :- (
    Ride(taxi, license, start, end, _, _, from, _, _, _),
    Ride(taxi, license, _, prevEnd, _, _, _, prevTo, _, _),
    start isAfter prevEnd,
    NOT ()
      Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
      (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
  )
} collect {
  case g:Gap ⇒ g
}
```
What's New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```scala
val gaps42 = rides filter {
  _.license ≡ "42"
} saturateFirst {
  Gap(taxi, license, prevEnd, start, prevTo, from) :- ( 
    Ride(taxi, license, start, end, _, _, from, _, _, _),
    Ride(taxi, license, _, prevEnd, _, _, _, prevTo, _, _),
    start isAfter prevEnd,
    NOT ( 
      Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
      (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
    )
  )} collect {
  case g:Gap => g
}
```
What’s New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```scala
val gaps42 = rides filter {
  _.license == "42"
} saturateFirst {
  Gap(taxi, license, prevEnd, start, prevTo, from) :- (  
    Ride(taxi, license, start, end, _, _, from, _, _, _),  
    Ride(taxi, license, _, prevEnd, _, _, _, prevTo, _, _),  
    start isAfter prevEnd,
    NOT (  
      Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),  
      (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
    )
  )
} collect {
  case g:Gap ⇒ g
}
```
What’s New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```scala
val gaps42 = rides filter {
  _.license ≡ "42"
} saturateFirst {
  Gap(taxi, license, prevEnd, start, prevTo, from) :- ( 
    Ride(taxi, license, start, end, __, __, from, __, __, __),
    Ride(taxi, license, __, prevEnd, __, __, __, prevTo, __, __),
    start isAfter prevEnd,
    NOT ( 
      Ride(taxi, license, otherStart, otherEnd, __, __, __, __, __, __),
      (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
    )
  } } collect {
  case g:Gap ⇒ g
}
```
What’s New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```scala
val gaps42 = rides filter {
  _.license == "42"
} saturateFirst {
  Gap(taxi, license, prevEnd, start, prevTo, from) :- ( 
    Ride(taxi, license, start, end, _, _, from, _, _, _), 
    Ride(taxi, license, _, prevEnd, _, _, _, prevTo, _, _),
    start isAfter prevEnd,
    NOT ( 
      Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
      (start isAfter otherStart) \ (otherStart isAfter prevEnd) )
  ) } collect {
  case g:Gap => g
}
```
What’s New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```scala
val gaps42 = rides filter {
  _.license == "42"
} saturateFirst {
  Gap(taxi, license, prevEnd, start, prevTo, from) :- (  
    Ride(taxi, license, start, end, _, _, from, _, _, _),  
    Ride(taxi, license, _, prevEnd, _, _, _, prevTo, _, _),  
    start isAfter prevEnd,  
    NOT (  
      Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),  
      (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)  
    )
  )} collect {
  case g:Gap ⇒ g
}
```
What’s New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```scala
val gaps42 = rides filter {
  _.license == "42"
} saturateFirst {
  Gap(taxi, license, prevEnd, start, prevTo, from) :- (
    Ride(taxi, license, start, end, _, _, from, _, _, _),
    Ride(taxi, license, _, prevEnd, _, _, _, prevTo, _, _),
    start isAfter prevEnd,
    NOT (
      Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
      (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
    )
  )} collect {
  case g:Gap ⇒ g
}
```
What’s New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```scala
val gaps42 = rides filter {
  _.license == "42"
} saturateFirst {
  Gap(taxi, license, prevEnd, start, prevTo, from) :- ( 
    Ride(taxi, license, start, end, __, __, from, __, __, __),
    Ride(taxi, license, __, prevEnd, __, __, __, prevTo, __, __),
    start isAfter prevEnd,
    NOT ( 
      Ride(taxi, license, otherStart, otherEnd, __, __, __, __, __, __),
      (start isAfter otherStart) ∧ (otherStart isAfter prevEnd) 
    )
  )
} collect {
  case g:Gap ⇒ g
}
```
What’s New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```scala
val gaps42 = rides.filter {
  _.license == "42"
} saturateFirst {
  Gap(taxi, license, prevEnd, start, prevTo, from) :- (
    Ride(taxi, license, start, end, _, _, from, _, _, _),
    Ride(taxi, license, _, prevEnd, _, _, _, prevTo, _, _),
    start isAfter prevEnd,
    NOT (  
      Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
      (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
    )
  )
} collect {
  case g:Gap => g
}
```
**What’s New (3) - Usability an Workflow**

**From Scala to logic program and back**

Scala is both extension language and scripting language

```scala
val gaps42 = rides.filter {
  _.license == "42"
} saturateFirst { Fusemate invocation
  Gap(taxi, license, prevEnd, start, prevTo, from) :- ( 
    Ride(taxi, license, start, end, _, _, from, _, _, _),
    Ride(taxi, license, _, prevEnd, _, _, _, prevTo, _, _),
    start isAfter prevEnd,
    NOT ( 
      Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
      (start isAfter otherStart) ∧ (otherStart isAfter prevEnd) 
    ) )
} collect {
  case g:Gap ⇒ g
}
```

**Functional + Logic programming (in a new way?)**
What’s New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```scala
val gaps42 = rides.filter { 
  _.license == "42"
} saturateFirst { 
  Gap(taxi, license, prevEnd, start, prevTo, from) :- ( 
    Ride(taxi, license, start, end, __, __, from, __, __, __), 
    Ride(taxi, license, __, prevEnd, __, __, __, prevTo, __, __), 
    start isAfter prevEnd, 
    NOT ( 
      Ride(taxi, license, otherStart, otherEnd, __, __, __, __, __, __), 
      (start isAfter otherStart) ∧ (otherStart isAfter prevEnd) 
    ) 
  ) } collect { 
  case g:Gap ⇒ g
}
```

Functional + Logic programming

(in a new way?)
Conclusions

**Fusemate is implemented by shallow embedding into Scala**

- New operators for aggregation and comprehension
- Atoms and interpretations are first-class citizens
- Light-weight interface logic programming <-> Scala
  
  Workflow: logic programming = operator on collections of objects (case classes)

**Efficiency**

- SAT problem for propositional possible models of stratified DLPs is NP-complete
- Atoms indexed by time then indexed by predicate symbols
  
  Helps a lot, in particular “comprehension”

- OK for slow-running processes

  Bigger data sets currently need combined workflow (taxi example)

**Availability**

https://bitbucket.csiro.au/users/bau050/repos/fusemate/