Resolving Inconsistencies and Redundancies in Declarative Process Models
Claudio Di Cicco, Fabrizio Maria Maggi, Marco Montali and Jan Mendling

8th International Workshop on Enterprise Modeling and Information Systems Architectures (EMISA 2017) Essen, Germany

claudio.di.cicco@wu.ac.at

Foreword

(Declarative) process discovery
Declarative constraints as automata
Process discovery

Event log

Process model
Mining flexible processes
Declarative process discovery

Objective: understanding the constraints that best define the allowed behaviour of the process behind the event log.
Declarative modelling of processes

- Usage of constraints
  - “Open model”
- Declare
  - state-of-the-art language

- Init(c)
  - c is always the first executed activity
- End(d)
  - d is always the last executed activity
- RespondedExistence(a,b)
  - If a is executed, b has to be executed
- Response(a,b)
  - If a is executed, b has to be executed afterwards
- ChainResponse(a,b)
  - If a is executed, b has to be executed immediately afterwards
- Precedence(a,b)
  - If b is executed, a must have been executed beforehand
- ChainPrecedence(a,b)
  - If b is executed, a has to be executed immediately beforehand
- NotChainSuccession(a,b)
  - If a is executed, b cannot be executed immediately afterwards
Subsumption hierarchy of relation Declare templates

(a) Existence templates

Cardinality templates
- Participation($x$)
- AtMostOne($x$)

Position templates
- Init($x$)
- End($x$)

(b) Relation templates

Backward-unidirectional relation templates
- RespondedExistence($y, x$)

Coupling templates
- CoExistence($x, y$)
- RespondedExistence($x, y$)
- NotCoExistence($x, y$)

Forward-unidirectional relation templates
- Precedence($x, y$)
- Succession($x, y$)
- Response($x, y$)
- NotSuccession($x, y$)

Negative templates
- AlternatePrecedence($x, y$)
- AlternateSuccession($x, y$)
- AlternateResponse($x, y$)

- ChainPrecedence($x, y$)
- ChainSuccession($x, y$)
- ChainResponse($x, y$)
- NotChainSuccession($x, y$)
Mining declarative processes: ingredients

“Submit draft”,  
“Write deliverable”,  
“Organise agenda”,  
...

Activities

Process alphabet

Event log

Declarative constraint templates
Mining declarative processes

RespondedExistence(a,b) ?
RespondedExistence(a,c) ?
...
Response(a,b) ?
Response(a,c) ?
...

- **Support**: fraction of cases fulfilling the constraint
- **Confidence**: support scaled by fraction of traces in which the activation occurs
- **Interest factor**: confidence scaled by fraction of traces in which the target occurs
Mining declarative processes

RespondedExistence(a,b) ?
RespondedExistence(a,c) ?
...
Response(a,b) ?
Response(a,c) ?
...

Support  Conf.  I.F.
Mining declarative processes

RespondedExistence(a,b) ✓
RespondedExistence(a,c) ?
...
Response(a,b) ?
Response(a,c) ✓
...

Support
Conf.
I.F.
Mining declarative processes

RespondedExistence(a, b)  ✓
RespondedExistence(a, c)  ?
...
Response(a, b)  ❌
Response(a, c)  ✓
...

Support  Conf.  I.F.
Mining declarative processes

RespondedExistence(a, b) ✓

RespondedExistence(a, c) ?

...

Response(a, b)

Response(a, c) ✓

...

Support | Conf. | I.F.

- Backward-unidirectional relation templates
- Coupling templates
- Forward-unidirectional relation templates
- Negative templates

- RespondedExistence(y, x)
- CoExistence(x, y)
- RespondedExistence(x, y)
- NotCoExistence(x, y)
- Precedence(x, y)
- Succession(x, y)
- Response(x, y)
- NotSuccession(x, y)
- AlternatePrecedence(x, y)
- AlternateSuccession(x, y)
- AlternateResponse(x, y)
- NotAlternateResponse(x, y)
- ChainPrecedence(x, y)
- ChainSuccession(x, y)
- ChainResponse(x, y)
- NotChainSuccession(x, y)
Mining declarative processes

RespondedExistence(a,b) ✔

RespondedExistence(a,c) ✗

...

Response(a,b)

Response(a,c) ✔

...

Backward-unidirectional relation templates

Coupling templates

Forward-unidirectional relation templates

Negative templates

RespondedExistence(y, x) ➔ CoExistence(x, y) ➔ RespondedExistence(x, y) ➔ NotCoExistence(x, y)

Precedence(x, y) ➔ Succession(x, y) ➔ Response(x, y) ➔ NotSuccession(x, y)

AlternatePrecedence(x, y) ➔ AlternateSuccession(x, y) ➔ AlternateResponse(x, y)

ChainPrecedence(x, y) ➔ ChainSuccession(x, y) ➔ ChainResponse(x, y) ➔ NotChainSuccession(x, y)
Mining declarative processes

RespondedExistence(a,b)
RespondedExistence(a,c)
and
Response(a,b)
Response(a,c)
and
...


From constraints-based model to FSA

**Regular Expression**

\[ [^a]^* ((a.*b.*) | (b.*a.*) [*[^a]^* \]

\[ \sigma \in \Sigma \setminus \{a, b\} \]

\[ \sigma \in \Sigma \setminus \{b\} \]

**Deterministic Finite State Automaton**

\[ [^a]^* (a.*c.*) [*[^a]^* \]

\[ \sigma \in \Sigma \setminus \{a\} \]

\[ \sigma \in \Sigma \setminus \{c\} \]

\[ a \]

\[ c \]
To be kept in mind

\[ \text{RespondedExistence}(a,b) \]
\[ \text{RespondedExistence}(a,c) \]
\[ \text{Response}(a,b) \]
\[ \text{Response}(a,c) \]
...
So far, so good

What is the problem?
While mining a real-life log...

- Support threshold: 0.85
- Confidence threshold: 0.25
- Interest factor threshold: 0.25
While mining a real-life log...

```plaintext
[submit draft] => {
    100.000% AlternatePrecedence(send draft, submit draft) 100.000%
    100.000% Response(submit draft, send deliverable) 100.000%
    100.000% NotChainSuccession(submit draft, send deliverable) 100.000%
    100.000% AlternateResponse(submit draft, send draft) 100.000%
    100.000% NotChainSuccession(submit draft, send draft) 100.000%
}
```
Time to challenge the X

```
[submit draft] => {
    100.000% AlternatePrecedence(send draft, submit draft) 100.000%
    100.000% Response(submit draft, send deliverable) 100.000%
    100.000% NotChainSuccession(submit draft, send deliverable) 100.000%
    100.000% AlternateResponse(submit draft, send draft) 100.000%
    100.000% NotChainSuccession(submit draft, send draft) 100.000%
    conf.: 0.250; int.'f: 0.250;
    conf.: 0.250; int.'f: 0.250;
    conf.: 0.250; int.'f: 0.250;
    conf.: 0.250; int.'f: 0.250;
    conf.: 0.250; int.'f: 0.250;
```
Time to challenge the X

Loading...
The result
The problems
1) inconsistency

- When support threshold is lower than 100%, constraints can be valid through most of the log, though being in conflict
- Example: an event log consists of two traces:
  1. \(<a, b, a, b, a, b, c>\)
  2. \(<a, b, a, b, a, c>\)
- Support threshold: 0.7
  - a is always the first
    \(\Rightarrow\) Init(a)
  - c is always the last
    \(\Rightarrow\) End(c)
  - In 6 cases over 8 (75%), a and c do not directly follow each other
    \(\Rightarrow\) NotChainSuccession(a,c)
  - In 5 cases over 7 (71.143%), b and c do not directly follow each other
    \(\Rightarrow\) NotChainSuccession(b,c)
The problems
1) inconsistency

- When support threshold is lower than 100%, constraints can be valid through most of the log, though being in conflict
- Example: an event log consists of two traces:
  1. \(<a, b, a, b, a, b, c>\)
  2. \(<a, b, a, b, a, c>\)
- Support threshold: 0.7
  - a is always the first
    \(\Rightarrow\) Init(a)
  - c is always the last
    \(\Rightarrow\) End(c)
  - In 6 cases over 8 (75%), a and c do not directly follow each other
    \(\Rightarrow\) NotChainSuccession(a,c)
  - In 5 cases over 7 (71.143%), a and b do not directly follow each other
    \(\Rightarrow\) NotChainSuccession(b,c)

- Question: what can be done right before c?
  \(\Rightarrow\) inconsistency!
The problems
1) inconsistency

- When support threshold is lower than 100%, constraints can be valid through most of the log, though being in conflict
- How to trust a discovery algorithm that can return inconsistent models?
The problems
2) redundancy

- Many constraints may be fulfilled 100% of times yet not add a bit of information to other already discovered ones

- Example: an event log consists of two traces:
  1. \(<a, b, a, b, a, b, c>\)
  2. \(<a, b, a, b, a, c>\)

  - \textit{a is always the first}
    \(\Rightarrow\) \text{Init}(a)
  - \textit{c is always the last}
    \(\Rightarrow\) \text{End}(c)
  - \textit{Before c, a precedes}
    \(\Rightarrow\) \text{Precedence}(a,c)
  - \textit{Before b, a precedes}
    \(\Rightarrow\) \text{Precedence}(a,b)
  - \textit{After a, c eventually follows}
    \(\Rightarrow\) \text{Response}(a,c)
  - \textit{After b, c eventually follows}
    \(\Rightarrow\) \text{Response}(b,c)
The problems
2) redundancy

- Many constraints may be fulfilled 100% of times yet not add a bit of information to other already discovered ones
- Example: an event log consists of two traces:
  1. \(<a, b, a, b, a, b, c>\)
  2. \(<a, b, a, b, a, c>\)
    - \textit{a is always the first}\n      \(\Rightarrow\) \text{Init}(a)
    - \textit{c is always the last}\n      \(\Rightarrow\) \text{End}(c)
    - \textit{Before c, a precedes}\n      \(\Rightarrow\) \text{Precedence}(a,c)
    - \textit{Before b, a precedes}\n      \(\Rightarrow\) \text{Precedence}(a,b)
    - \textit{After a, c eventually follows}\n      \(\Rightarrow\) \text{Response}(a,c)
    - \textit{After b, c eventually follows}\n      \(\Rightarrow\) \text{Response}(b,c)

\text{Of course! a is always the first}
The problems
2) redundancy

- Many constraints may be fulfilled 100% of times yet not add a bit of information to other already discovered ones
- Example: an event log consists of two traces:
  1. \(<a, b, a, b, a, b, c>\)
  2. \(<a, b, a, b, a, c>\)
     - **a is always the first**
       \(\Rightarrow\) Init(a)
     - **c is always the last**
       \(\Rightarrow\) End(c)
     - Before c, a precedes
       \(\Rightarrow\) Precedence(a,c)
     - Before b, a precedes
       \(\Rightarrow\) Precedence(a,b)
     - After a, c eventually follows
       \(\Rightarrow\) Response(a,c)
     - After b, c eventually follows
       \(\Rightarrow\) Response(b,c)

- Question: can't we avoid stating the obvious?
  \(\Rightarrow\) redundancy!

Of course! a is always the first
Of course! c is always the last
The problems
2) redundancy

- Many constraints may be fulfilled 100% of times yet not add a bit of information to other already discovered ones
- How to reduce the number of unnecessary returned constraints?
Automata-product *monoid*

*Algebraic structure with composition operator* \((\square)\) *holding the properties of*

- commutativity
- associativity

and bearing

- identity element \(\sigma \in \Sigma\)
- and absorbing element \(\rightarrow\)
The solution

Automata-product monoid
Rules of the game

- Intersect the **product automaton** with the newly visited constraints, one at a time

\[
\text{Init}(a) \times \text{Participation}(b) = \text{Product automaton}
\]

Init(a)  \hspace{1cm} Participation(b)  \hspace{1cm} Product automaton
Rules of the game

- Intersect the **product automaton** with the newly visited constraints, one at a time.
Exploiting formal properties

- We take advantage of
  1. associativity
    - allows for "storage" of results

\[
\text{Product automaton} = \text{ChainPrecedence}(a,b)
\]

Old product automaton

\[
\times
\]

Product automaton
Exploiting formal properties

- We take advantage of
  1. associativity
     - allows for "storage" of results
  2. commutativity
     - allows for priority sorting of constraints
Exploiting formal properties

- We take advantage of
  1. associativity
     - allows for "storage" of results
  2. commutativity
     - allows for priority sorting of constraints
Playing the game

- Newly visited constraints add information to the knowledge on the process model if they reduce the number of possible traces (accepted strings)
Playing the game

- Newly visited constraints add information to the knowledge on the process model if they reduce the number of possible traces (accepted strings)
Playing the game

- Newly visited constraints add information to the knowledge on the process model if they reduce the number of possible traces (accepted strings)
Inconsistency!

- Newly visited constraints add information to the knowledge on the process model if they reduce the number of possible traces (accepted strings)
- Conflict:
Inconsistency!

- Newly visited constraints add information to the knowledge on the process model if they reduce the number of possible traces (accepted strings)

- **Conflict:**
  - The product automaton becomes

  (empty language)
Inconsistency!

- Newly visited constraints add information to the knowledge on the process model if they reduce the number of possible traces (accepted strings)

- **Conflict:**
  - The product automaton becomes

  (empty language)
Inconsistency!

- Newly visited constraints add information to the knowledge on the process model if they reduce the number of possible traces (accepted strings)

**Conflict:**
- Remove the constraint, in case
Redundancy!

- Newly visited constraints add information to the knowledge on the process model if they reduce the number of possible traces (accepted strings)

- **Redundancy:**
Redundancy!

- Newly visited constraints add information to the knowledge on the process model if they reduce the number of possible traces (accepted strings)

- **Redundancy:**
  - The new product automaton accepts the same strings as before (language inclusion)
Redundancy!

- Newly visited constraints add information to the knowledge on the process model if they reduce the number of possible traces (accepted strings)

- **Redundancy:**
  - The new product automaton accepts the same strings as before (language inclusion)
Redundancy!

- Newly visited constraints add information to the knowledge on the process model if they reduce the number of possible traces (accepted strings)

- **Redundancy:**
  - Remove the constraint, in case
Objectives

- Remove inconsistencies
- Minimise redundancies
- Analyse each constraint once
The solution: Inconsistency detection

- Rationale:
  1. How to **find** inconsistencies among constraints?
     - Use the automaton-based model for constraints
     - Does the cross-product automaton recognise the empty language?
  2. How to **search** the inconsistencies?
     - Exploit:
       a) The product operation between automata
       b) The sorting of Declare templates

- Guideline:
  - Preserve the most meaningful constraints
  - The sorting prioritises constraints
The solution: Redundancy detection

- **Rationale:**
  1. **How to find** redundancies among constraints?
     - Use the automaton-based model for constraints
     - Does the cross-product automata recognise the **same language** as before?
  2. **How to search** the inconsistencies?
     - Exploit:
       a) The product operation between automata
       b) The sorting of Declare templates

- **Guideline:**
  - Preserve the most meaningful constraints
  - The sorting prioritises constraints
Conclusion

Which were the conflicting constraints in the log?
How does the redundancy removal perform?

What is more in the paper?

Limitations and future work
Which were the conflicting constraints in the log?

1. NotSuccession(send meeting, organize agenda)
2. NotChainSuccession(send draft, send deliverable)
3. Succession(send draft, submit report)
Redundancy reduction

The diagram illustrates the cumulative number of constraints for various types of checks and redundancies. The categories include:

- Position
- Cardinality
- Coupling
- Fw-uni-directional
- Bw-uni-directional
- Negative

The x-axis represents the different types of constraints, while the y-axis shows the cumulative number of constraints. The graph indicates significant reductions in redundancies, with values ranging from -5.56% to -61.94%.

Legend:
- Check
  - None
  - Single
  - Double

Types of Redundancy:
- +5.56%
- -30.32%
- -61.94%
- -73.88%
- -85.67%
- -71.61%
- -76.84%
- -87.58%
- -44.04%
- -57.30%

From the graph, it is evident that the highest reductions are observed in the higher ranges, indicating effective redundancy reduction strategies.
Conclusions, limitations and future work

We have presented an algorithm which automatically finds inconsistencies and redundancies in mined Declare models

- The checks are purely based on operations over automata (remember: monoids)
- [http://github.com/cdc08x/minerful](http://github.com/cdc08x/minerful)

More in the paper:

- The order in which the constraints are checked deeply affects the returned result
  - Comparative studies prove different sorting strategies to affect
    - computation time
    - fitness of the returned model
    - size

Limitations:

- Performances are heavily affected by the interplay of constraints

Future work:

- Users/analysts involvement
Resolving Inconsistencies and Redundancies in Declarative Process Models

Claudio Di Ciccio, Fabrizio Maria Maggi, Marco Montali and Jan Mendling

8th International Workshop on Enterprise Modeling and Information Systems Architectures (EMISA 2017) Essen, Germany

claudio.di.ciccio@wu.ac.at

Resolving Inconsistencies and Redundancies in Declarative Process Models

Claudio Di Ciccio, Fabrizio Maria Maggi, Marco Montali and Jan Mendling

8th International Workshop on Enterprise Modeling and Information Systems Architectures (EMISA 2017) Essen, Germany

claudio.di.ciccio@wu.ac.at

https://doi.org/10.1016/j.is.2016.09.005

Extra slides deck
Computational time complexity

\[ O\left( n \cdot \log n + \underbrace{n \cdot \bigcirc^T_{\mathcal{M}}} \right) \]

- \( n \cdot \log n \): sorting
- \( n \cdot \bigcirc^T_{\mathcal{M}} \): conflict and redundancy (single) check
- \( n^2 \cdot \bigcirc^T_{\mathcal{M}} \): redundancy double-check

Number of constraints

Automata product / language check