Verification of Relational Data-Centric Dynamic Systems with External Services

Marco Montali
Joint work with: B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch

KRDB Research Centre for Knowledge and Data
Free University of Bozen-Bolzano, Italy

PODS 2013
New York, USA
Introduction

This talk is about verification of systems merging data and processes. See yesterday’s keynote by Diego Calvanese.
Introduction

Data-Centric Dynamic Systems (DCDSs)

An abstract, pristine framework to formally describe processes that manipulate data.

- Captures virtually all existing approaches to data-aware processes, such as the artifact-centric paradigm.

- Data layer: relational database (with constraints).

- Process layer: condition-action rules (include service calls that input new data).
DCDS

A DCDS $S$ is a pair $\langle \mathcal{D}, \mathcal{P} \rangle$.

Data layer $\mathcal{D}$

- Relational schema.
- Constraints (equality constraints/domain-independent FOL).
- Initial DB.

Process layer $\mathcal{P}$

- Services to introduce new data into the system - results taken from a countably infinite domain.
- Actions with parameters, specified in terms of effects that:
  1. query the current DB (with UCQs + domain-independent FO filters);
  2. transfer the obtained answers, together with service call results, into facts that constitute the new DB.
- Declarative description of the process with condition-action rules.
  ▶ Condition: (domain-independent) FO query.
  ▶ Each rule queries the current DB and determines the executability of the corresponding action with params.
DCDS

A DCDS $S$ is a pair $\langle D, P \rangle$.

**Data layer $D$**

- Relational schema.
- Constraints (equality constraints/domain-independent FOL).
- Initial DB.

**Process layer $P$**

- **Services** to introduce new data into the system - results taken from a *countably infinite domain*.
- **Actions** with parameters, specified in terms of effects that:
  1. query the current DB (with UCQs + domain-independent FO filters);
  2. transfer the obtained answers, together with service call results, into facts that constitute the new DB.
- Declarative description of the process with condition-action rules.
  - Condition: (domain-independent) FO query.
  - Each rule queries the current DB and determines the executability of the corresponding action with params.
An Example: Hotels and Price Conversion

Data Layer: Info about hotels and room prices

\[
\text{Cur} = \langle \text{UserCurrency} \rangle \quad \text{CH} = \langle \text{Hotel, Currency} \rangle \quad \text{PEntry} = \langle \text{Hotel, Price, Date} \rangle
\]

Process Layer/1

User selection of a currency.

- Process: \( \text{true} \rightarrow \text{ChooseCur}() \)
- Service call for currency selection: \( \text{uInputCurr()} \)
  - Models \textit{user input} with \textit{non-deterministic} behavior: same-argument calls possibly return different values at different time moments.
- \( \text{ChooseCur}() : \begin{cases} 
\text{true} \rightarrow \text{Cur(} \text{uInputCurr()} \text{)} \\
\text{CH}(h, c) \rightarrow \text{CH}(h, c) \\
\text{PEntry}(h, p, d) \rightarrow \text{PEntry}(h, p, d) 
\end{cases} \)
An Example: Hotels and Price Conversion

Data Layer: Info about hotels and room prices

\[ \text{Cur} = \langle \text{UserCurrency} \rangle \quad \text{CH} = \langle \text{Hotel, Currency} \rangle \quad \text{PEntry} = \langle \text{Hotel, Price, Date} \rangle \]

Process Layer/2

Price conversion for a hotel.

- Process: \( \text{Cur}(c) \land \text{CurHotel}(h, c_h) \land c_h \neq c \longmapsto \text{ApplyConv}(h, c) \)

- Service call for currency selection: \( \text{CONV}(\text{price, from, to, date}) \)
  - Models historical conversion with deterministic behavior: same-argument calls always return the same value along a run.

- \( \text{ApplyConv}(h, c) : \)
  \[
  \begin{align*}
  \{ & \text{PEntry}(h, p, d) \land \text{CH}(h, c_{\text{old}}) \land \text{Cur}(c) \longmapsto \text{PEntry}(h, \text{CONV}(p, c_{\text{old}}, c, d), d) \\
  & \text{PEntry}(h', p, d) \land h' \neq h \longmapsto \text{PEntry}(h', p, d) \\
  & \text{CH}(h, c_{\text{old}}) \longmapsto \text{CH}(h, c) \\
  & \text{CH}(h', c') \land h' \neq h \longmapsto \text{CH}(h', c') \\
  & \text{Cur}(c) \longmapsto \text{Cur}(c) \}
  \end{align*}
  \]
### Run

<table>
<thead>
<tr>
<th>HC</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( h_1 )</td>
<td>eur</td>
<td></td>
</tr>
<tr>
<td>( h_2 )</td>
<td>eur</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PEntry</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( h_1 )</td>
<td>95</td>
</tr>
<tr>
<td>( h_1 )</td>
<td>80</td>
</tr>
<tr>
<td>( h_1 )</td>
<td>80</td>
</tr>
<tr>
<td>( h_2 )</td>
<td>80</td>
</tr>
</tbody>
</table>

ChooseCur(): uInputCurr() = usd

\[
\text{ApplyConv}(\text{Conv}(95, \text{eur}, \text{usd}, \text{apr}-25)) = 115
\]

\[
\text{ApplyConv}(\text{Conv}(80, \text{eur}, \text{usd}, \text{sep}-18)) = 95
\]
Run

ChooseCur(): `uINPUTCURR() = ?`

<table>
<thead>
<tr>
<th>HC</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$</td>
<td>eur</td>
<td></td>
</tr>
<tr>
<td>$h_2$</td>
<td>eur</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PEEntry</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$</td>
<td>95</td>
<td>apr-25</td>
</tr>
<tr>
<td>$h_1$</td>
<td>80</td>
<td>sep-18</td>
</tr>
<tr>
<td>$h_2$</td>
<td>80</td>
<td>sep-18</td>
</tr>
</tbody>
</table>

ChooseCur() ApplyConv($h_1$, usd): 
\[ \text{conv}(95, \text{eur}, \text{usd}, \text{apr-25}) = 115 \]

ChooseCur() ApplyConv($h_2$, usd): 
\[ \text{conv}(80, \text{eur}, \text{usd}, \text{sep-18}) = 95 \]

ChooseCur() ApplyConv($h_2$, usd): 
\[ \text{conv}(80, \text{eur}, \text{usd}, \text{sep-18}) = 95 \]
Run

ChooseCur(): \texttt{uINPUTCURR()} = \texttt{usd}

ChooseCur() \rightarrow \texttt{ApplyConv(} h_2, \texttt{usd) \rightarrow \texttt{ChooseCur() \rightarrow \texttt{ApplyConv(} h_2, \texttt{usd) \rightarrow \texttt{conv(} 80, \texttt{eur,usd,sep-18) = 95 \texttt{}} \texttt{}} \texttt{)}\texttt{}}
ChooseCur(): \texttt{uInputCurr()} = \texttt{usd}

ApplyConv(h_1, \texttt{usd})

ApplyConv(h_2, \texttt{usd})

ChooseCur()
Run

ChooseCur(): \texttt{uInputCurr()} = \texttt{usd}

ApplyConv(h_1, usd):

\begin{align*}
\text{CONV}(95, \text{eur}, \text{usd}, \text{apr-25}) &= ? \\
\text{CONV}(80, \text{eur}, \text{usd}, \text{sep-18}) &= ?
\end{align*}
ChooseCur(): \texttt{uInputCurr()} = \texttt{usd}

ApplyConv(h1, usd):
\[
\begin{align*}
\text{CONV}(95, \text{eur}, \text{usd}, \text{apr-25}) &= 115 \\
\text{CONV}(80, \text{eur}, \text{usd}, \text{sep-18}) &= 95
\end{align*}
\]
ChooseCur(): \texttt{uINPUTCURR()} = \textit{usd}

ChooseCur()

ApplyConv(h_2, \textit{usd})

ApplyConv(h_1, \textit{usd}):

\begin{align*}
\text{CONV}(95, \textit{eur}, \textit{usd}, \text{apr-25}) &= 115 \\
\text{CONV}(80, \textit{eur}, \textit{usd}, \text{sep-18}) &= 95
\end{align*}
Run

ChooseCur(): \textbf{uINPUTCRR() = usd}

ApplyConv(h_2, usd): 
\texttt{CONV(80,eur,usd,sep-18) = 95}

ApplyConv(h_1, usd):
\texttt{CONV(95,eur,usd,apr-25) = 115}
\texttt{CONV(80,eur,usd,sep-18) = 95}

\begin{tabular}{|l|l|}
\hline
HC & \\
\hline
h_1 & eur \\
\hline
h_2 & eur \\
\hline
\end{tabular}

\begin{tabular}{|l|l|}
\hline
PEntry & \\
\hline
h_1 & 95 apr-25 \\
\hline
h_1 & 80 sep-18 \\
\hline
h_2 & 80 sep-18 \\
\hline
\end{tabular}

\begin{tabular}{|l|l|}
\hline
Cur & \\
\hline
usd & \\
\hline
\end{tabular}

\begin{tabular}{|l|l|}
\hline
HC & \\
\hline
h_1 & usd \\
\hline
h_2 & usd \\
\hline
\end{tabular}

\begin{tabular}{|l|l|}
\hline
PEntry & \\
\hline
h_1 & 115 apr-25 \\
\hline
h_1 & 95 sep-18 \\
\hline
h_2 & 95 sep-18 \\
\hline
\end{tabular}

\begin{tabular}{|l|l|}
\hline
Cur & \\
\hline
usd & \\
\hline
\end{tabular}
Execution Semantics

Transition system accounting for all possible runs of the DCDS:

- **States**: each linked to a DB - instance of the data layer;
- **Transitions**: *legal* applications of action+params+service call evals.
  - **Action+params**: executable according to the process rules.
  - **Deterministic services** behave consistently with the previous results.

We obtain a possibly infinite-state (relational) transition system:

- from the initial DB;
- by applying transitions in all possible ways.
Sources of Unboundedness/Infinity

In general: service calls cause...

- **Infinite branching** (due to all possible results of service calls).
- **Infinite runs** (usage of values obtained from unboundedly many service calls).
- **Unbounded DBs** (accumulation of such values).

```
ApplyConv(h_1, usd): exchange rate = 1.2
ApplyConv(h_1, usd): exchange rate = 1.23
ApplyConv(h_1, usd): exchange rate = 1.3
ApplyConv(h_1, usd): exchange rate = ...
```
Verification of DCDSs

Verification
Given a DCDS $S$ (with transition system $\Upsilon_S$), and a temporal/dynamic property $\Phi$, check whether

$$\Upsilon_S \models \Phi$$

Requirements for temporal/dynamic properties:

- to capture data $\leadsto$ first-order queries;
- to capture dynamics $\leadsto$ temporal modalities;
- to capture evolution of data $\leadsto$ quantification across states.
Verification of DCDSs

Verification

Given a DCDS $S$ (with transition system $\Upsilon_S$), and a temporal/dynamic property $\Phi$, check whether

$$\Upsilon_S \models \Phi$$

Requirements for temporal/dynamic properties:

- to capture data $\leadsto$ first-order queries;
- to capture dynamics $\leadsto$ temporal modalities;
- to capture evolution of data $\leadsto$ quantification across states.

Our goal

Investigate “robust” conditions on decidability of verification:

- for sophisticated branching- and linear-time temporal properties;
- exploiting conventional, finite-state model checking via construction of a faithful (sound and complete), finite-state abstraction.
Design Space

We employ variants of first-order $\mu$-calculus ($\mu L_{FO}$):

$$\Phi ::= Q \mid \neg \Phi \mid \Phi_1 \land \Phi_2 \mid \exists x. \Phi \mid \langle - \rangle \Phi \mid Z \mid \mu Z. \Phi$$

- Employs fixpoint constructs to express sophisticated properties defined via induction or co-induction.
- Subsumes virtually all logics used in verification, such as LTL, CTL, CTL*.

\[
\begin{align*}
\mu L_{FO} & \uparrow \\
\mu L & \uparrow & \uparrow \\
LTL & \rightarrow & PDL & \rightarrow & CTL
\end{align*}
\]
Design Space

We employ variants of first-order $\mu$-calculus ($\mu\mathcal{L}_{FO}$):

$\Phi ::= Q \mid \lnot \Phi \mid \Phi_1 \land \Phi_2 \mid \exists x. \Phi \mid \langle - \rangle \Phi \mid Z \mid \mu Z. \Phi$

- Employs fixpoint constructs to express sophisticated properties defined via induction or co-induction.
- Subsumes virtually all logics used in verification, such as LTL, CTL, CTL*.

Problem 1

Unrestricted first-order quantification: no hope of reducing verification to finite-state model checking.

See:

$\exists x_1, \ldots, x_n \cdot \bigwedge_{i \neq j} x_i \neq x_j \land \bigwedge_{i \in \{1, \ldots, n\}} \langle - \rangle Q(x_i)$

$\leadsto$ We need to consider fragments of $\mu\mathcal{L}_{FO}$ with controlled quantification.
### Design Space

We employ variants of first-order \( \mu \)-calculus (\( \mu L_{FO} \)):

\[
\Phi ::= Q \mid \neg \Phi \mid \Phi_1 \land \Phi_2 \mid \exists x. \Phi \mid \langle - \rangle \Phi \mid Z \mid \mu Z. \Phi
\]

- Employs fixpoint constructs to express sophisticated properties defined via induction or co-induction.
- Subsumes virtually all logics used in verification, such as LTL, CTL, CTL*.

### Problem 1

Unrestricted first-order quantification: no hope of reducing verification to finite-state model checking.

See:

\[
\exists x_1, \ldots, x_n \ . \ \land_{i \neq j} x_i \neq x_j \land \land_{i \in \{1, \ldots, n\}} \langle - \rangle Q(x_i)
\]

\( \rightsquigarrow \) We need to consider fragments of \( \mu L_{FO} \) with controlled quantification.

### Problem 2

Verification is undecidable for simple propositional CTL \( \cap \) LTL properties.

\( \rightsquigarrow \) We need to pose restrictions on DCDSs.
History-Preserving $\mu$-calculus ($\mu\mathcal{L}_A$)

Active-domain quantification: restricted to those individuals *present in the current database*.

$$\exists x. \Phi \leadsto \exists x. \text{LIVE}(x) \land \Phi$$

where $\text{LIVE}(x)$ states that $x$ is present in the current active domain.

---

**Example**

$$\nu X. (\forall x. \text{LIVE}(x) \land \text{Stud}(x) \rightarrow \mu Y. (\exists y. \text{LIVE}(y) \land \text{Grad}(x, y) \lor \langle - \rangle Y) \land [ - ] X)$$

Along every path, it is always true, for each student $x$, that there exists an evolution eventually leading to a graduation of the student (with some final mark $y$).
Persistence-Preserving $\mu$-calculus ($\mu\mathcal{L}_P$)

In some cases, objects maintain their identity only if they persist in the active domain (cf. business artifacts and their IDs).

$\exists x. \Phi; \exists x. \text{live}(x) \land \Phi; \text{live}(\vec{x}) \land \langle-\rangle \Phi(\vec{x}); \text{live}(\vec{x}) \land \langle-\rangle X)$

Example (persistence)

$\nu X. (\forall x. \text{live}(x) \land \text{Stud}(x) \rightarrow \mu Y. (\exists y. \text{live}(y) \land \text{Grad}(x, y) \lor (\text{live}(x) \langle-\rangle Y)) \land \langle-\rangle X)$
Persistence-Preserving $\mu$-calculus ($\mu\mathcal{L}_P$)

In some cases, objects maintain their identity only if they persist in the active domain (cf. business artifacts and their IDs).

$\mu\mathcal{L}_P$ restricts $\mu\mathcal{L}_A$ to quantification over persisting objects only, i.e., objects that continue to be LIVE.

$\exists x. \Phi \leadsto \exists x. \text{LIVE}(x) \land \Phi$

$\langle - \rangle \Phi(x) \leadsto \text{LIVE}(x) \land \langle - \rangle \Phi(x)$

$[ - ] \Phi(x) \leadsto \text{LIVE}(x) \land [ - ] \Phi(x)$

Example (persistence)

Along every path, it is always true, for each student $x$, that there exists an evolution in which she eventually graduates.
Persistence-Preserving $\mu$-calculus ($\mu\mathcal{L}_P$)

In some cases, objects maintain their identity only if they persist in the active domain (cf. business artifacts and their IDs).

$\mu\mathcal{L}_P$ restricts $\mu\mathcal{L}_A$ to quantification over persisting objects only, i.e., objects that continue to be LIVE.

$$
\exists x. \Phi \rightsquigarrow \exists x. \text{LIVE}(x) \land \Phi
$$

$$
\langle - \rangle \Phi(x) \rightsquigarrow \text{LIVE}(x) \land \langle - \rangle \Phi(x)
$$

$$
[-] \Phi(x) \rightsquigarrow \text{LIVE}(x) \land [-] \Phi(x)
$$

Example ("strong persistence")

$$
\nu X. (\forall x. \text{LIVE}(x) \land \text{Stud}(x)) \rightarrow \\
\mu Y. (\exists y. \text{LIVE}(y) \land \text{Grad}(x, y) \lor (\text{LIVE}(x) \land \langle - \rangle Y)) \land [-] X
$$

Along every path, it is always true, for each student $x$, that there exists an evolution in which $x$ persists in the database until she eventually graduates.
Persistence-Preserving $\mu$-calculus ($\mu\mathcal{L}_P$)

In some cases, objects maintain their identity only if they persist in the active domain (cf. business artifacts and their IDs).

$\mu\mathcal{L}_P$ restricts $\mu\mathcal{L}_A$ to quantification over persisting objects only, i.e., objects that continue to be LIVE.

$\exists x.\Phi \leadsto \exists x.\text{LIVE}(x) \land \Phi$

$\langle - \rangle \Phi(x) \leadsto \text{LIVE}(x) \land \langle - \rangle \Phi(x)$

$[\neg] \Phi(x) \leadsto \text{LIVE}(x) \land [\neg] \Phi(x)$

Example ("weak persistence")

$\nu X. (\forall x. \text{LIVE}(x) \land \text{Stud}(x) \rightarrow$

$\mu Y. (\exists y. \text{LIVE}(y) \land \text{Grad}(x, y) \lor (\text{LIVE}(x) \rightarrow \langle - \rangle Y)) \land [\neg] X)$

Along every path, it is always true, for each student $x$, that there exists an evolution in which either $x$ does not persist, or she eventually graduates.
Bisimulations

We introduce two novel notions of bisimulation to account for $\mu\mathcal{L}_A/\mu\mathcal{L}_P$.

These bisimulation relations capture:
• **dynamics** $\sim$ standard notion of bisimulation;
• **data** $\sim$ DB isomorphism;
• **evolution of data** $\sim$ compatibility of the bijections witnessing the isomorphisms along a run.
Bisimulations

History-preserving bisimulation requires each isomorphism to be witnessed by a bijection that extends the bijection used in the previous step.

**Theorem**

If $\Upsilon_1$ and $\Upsilon_2$ are history-preserving bisimilar, then for every $\mu \mathcal{L}_A$ closed formula $\Phi$, we have:

$$\Upsilon_1 \models \Phi \quad \text{if and only if} \quad \Upsilon_2 \models \Phi.$$
Bisimulations

History-preserving bisimulation requires each isomorphism to be witnessed by a bijection that extends the bijection used in the previous step.

Theorem

If $\Upsilon_1$ and $\Upsilon_2$ are history-preserving bisimilar, then for every $\mu\mathcal{L}_A$ closed formula $\Phi$, we have:

$$\Upsilon_1 \models \Phi \iff \Upsilon_2 \models \Phi.$$  

Persistence-preserving bisimulation requires each isomorphism to be witnessed by a bijection that extends the bijection used in the previous step, restricted only to the persisting objects.

Theorem

If $\Upsilon_1$ and $\Upsilon_2$ are persistence-preserving bisimilar, then for every $\mu\mathcal{L}_P$ closed formula $\Phi$, we have:

$$\Upsilon_1 \models \Phi \iff \Upsilon_2 \models \Phi.$$
Conditions for DCDSs

We devise two conditions over the transition system $\Upsilon_S$ of a DCDS $S$.

### Run boundedness

Each run of $\Upsilon_S$ accumulates only a **bounded number of objects**.

- No bound on the overall number of objects: $\Upsilon_S$ is still infinite-state, due to infinite branching induced by service calls.
- Unboundedly many deterministic service calls can still be issued with a bounded number of inputs.
- Only boundedly many nondeterministic service calls can be issued.
Conditions for DCDSs

We devise two conditions over the transition system $\Upsilon_S$ of a DCDS $S$.

**Run boundedness**

Each run of $\Upsilon_S$ accumulates only a **bounded number of objects**.

- No bound on the overall number of objects: $\Upsilon_S$ is still infinite-state, due to infinite branching induced by service calls.
- Unboundedly many deterministic service calls can still be issued with a bounded number of inputs.
- Only boundedly many nondeterministic service calls can be issued.

**State boundedness**

Each state of $\Upsilon_S$ contains only a **bounded number of objects**.

- Relaxation of run-boundedness: unboundedly many objects along a run, provided that they are not accumulated in the same state.
- $\Upsilon_S$ can contain infinite branches and infinite runs.
Summary of Results

- Unrestricted DCDSs (Turing complete)
- State-bounded DCDSs
- Run-bounded DCDSs
- Finite-state DCDSs
- GR
- +acyclic DCDSs
- GR-acyclic DCDSs
- Weakly-acyclic DCDSs with det. services
- Finite-range DCDSs

---

Marco Montali
Verification of Relational DCDSs
PODS 2013
### Summary of Results

<table>
<thead>
<tr>
<th></th>
<th>Unrestricted</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu L_{FO}$</td>
<td>$U$</td>
</tr>
<tr>
<td>$\mu L_A$</td>
<td>$U$</td>
</tr>
<tr>
<td>$\mu L_P$</td>
<td>$U$</td>
</tr>
<tr>
<td>$\mu L$</td>
<td>$U$</td>
</tr>
</tbody>
</table>

**D**: decidable; **U**: undecidable; **N**: no finite abstraction.
Summary of Results

<table>
<thead>
<tr>
<th></th>
<th>Unrestricted</th>
<th>Finite-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu \mathcal{L}_{FO}$</td>
<td>U</td>
<td>D</td>
</tr>
<tr>
<td>$\mu \mathcal{L}_A$</td>
<td>U</td>
<td>D</td>
</tr>
<tr>
<td>$\mu \mathcal{L}_P$</td>
<td>U</td>
<td>D</td>
</tr>
<tr>
<td>$\mu \mathcal{L}$</td>
<td>U</td>
<td>D</td>
</tr>
</tbody>
</table>

**D**: decidable; **U**: undecidable; **N**: no finite abstraction.
Summary of Results

<table>
<thead>
<tr>
<th>Unrestricted</th>
<th>Finite-state</th>
</tr>
</thead>
</table>
| $\mu L_{FO}$ | D \n| $\mu L_A$   | D \n| $\mu L_P$   | D \n| $\mu L$     | D \n
D: decidable; U: undecidable; N: no finite abstraction.
Summary of Results

Unrestricted DCDSs (Turing complete)

Finite-state DCDSs

Run-bounded DCDSs

State-bounded DCDSs

Finite-range DCDSs

<table>
<thead>
<tr>
<th></th>
<th>Unrestricted</th>
<th>Run-bounded</th>
<th>Finite-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu L_{FO}$</td>
<td>U</td>
<td>N</td>
<td>D</td>
</tr>
<tr>
<td>$\mu L_A$</td>
<td>U</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>$\mu L_P$</td>
<td>U</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>$\mu L$</td>
<td>U</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

D: decidable; U: undecidable; N: no finite abstraction.
Summary of Results

<table>
<thead>
<tr>
<th></th>
<th>Unrestricted</th>
<th>Run-bounded</th>
<th>Finite-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu\mathcal{L}_{FO}$</td>
<td>U</td>
<td>N</td>
<td>D</td>
</tr>
<tr>
<td>$\mu\mathcal{L}_A$</td>
<td>U</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>$\mu\mathcal{L}_P$</td>
<td>U</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>$\mu\mathcal{L}$</td>
<td>U</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

D: decidable; U: undecidable; N: no finite abstraction.
Summary of Results

<table>
<thead>
<tr>
<th></th>
<th>Unrestricted</th>
<th>State-bounded</th>
<th>Run-bounded</th>
<th>Finite-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu L_{FO}$</td>
<td>U</td>
<td>U</td>
<td>N</td>
<td>D</td>
</tr>
<tr>
<td>$\mu L_A$</td>
<td>U</td>
<td>U</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>$\mu L_P$</td>
<td>U</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>$\mu L$</td>
<td>U</td>
<td></td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

D: decidable; U: undecidable; N: no finite abstraction.
Summary of Results

<table>
<thead>
<tr>
<th></th>
<th>Unrestricted</th>
<th>State-bounded</th>
<th>Run-bounded</th>
<th>Finite-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu \mathcal{L}_{FO}$</td>
<td>$U$</td>
<td>$U$</td>
<td>$N$</td>
<td>$D$</td>
</tr>
<tr>
<td>$\mu \mathcal{L}_A$</td>
<td>$U$</td>
<td>$U$</td>
<td>$D$</td>
<td>$D$</td>
</tr>
<tr>
<td>$\mu \mathcal{L}_P$</td>
<td>$U$</td>
<td>$D$</td>
<td>$D$</td>
<td>$D$</td>
</tr>
<tr>
<td>$\mu \mathcal{L}$</td>
<td>$U$</td>
<td>$D$</td>
<td>$D$</td>
<td>$D$</td>
</tr>
</tbody>
</table>

$D$: decidable; $U$: undecidable; $N$: no finite abstraction.
Summary of Results

<table>
<thead>
<tr>
<th></th>
<th>Unrestricted</th>
<th>State-bounded</th>
<th>Run-bounded</th>
<th>Finite-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu L_{FO}$</td>
<td>U</td>
<td>U</td>
<td>N</td>
<td>D</td>
</tr>
<tr>
<td>$\mu L_A$</td>
<td>U</td>
<td>U</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>$\mu L_P$</td>
<td>U</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>$\mu L$</td>
<td>U</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

D: decidable; U: undecidable; N: no finite abstraction.
Theorem

Verification of $\mu\mathcal{L}_A$ over run-bounded DCDSs is decidable and can be reduced to model checking of propositional $\mu\mathcal{L}$ over a finite TS.

Crux: construct a faithful abstraction $\Theta_S$ for $\Upsilon_S$, collapsing infinite branching.
Run-Bounded Systems: Decidability for $\mu\mathcal{L}_A$

**Theorem**

Verification of $\mu\mathcal{L}_A$ over run-bounded DCDSs is **decidable** and can be reduced to model checking of propositional $\mu\mathcal{L}$ over a finite TS.

**Crux:** construct a **faithful abstraction** $\Theta_S$ for $\Upsilon_S$, collapsing infinite branching.

- We use **isomorphic types** instead of actual service call results.
State-bounded Systems: Undecidability for $\mu \mathcal{L}_A$

Theorem

Verification of $\mu \mathcal{L}_A$ over state-bounded DCDSs is undecidable.

Intuition: $\mu \mathcal{L}_A$ can use quantification to store and compare the unboundedly many values encountered along the runs.

Crux: reduction from satisfiability of LTL with freeze quantifiers.

- $\mu \mathcal{L}_A$ can express LTL with freeze quantifier by making registers explicit.
- There is a state-bounded DCDS that simulates all the possible traces with register assignments (i.e., data words).
- Satisfiability via model checking.
State-bounded Systems: Decidability for $\mu \mathcal{L}_P$

**Theorem**

Verification of $\mu \mathcal{L}_P$ over state-bounded DCDSs is decidable and can be reduced to model checking of propositional $\mu$-calculus over a finite transition system.

Crux: construct a **faithful abstraction** $\Theta_S$ for $\Upsilon_S$, collapsing infinite branching and compacting infinite runs.

1. **Prune** infinite branching (isomorphic types).
State-bounded Systems: Decidability for $\mu\mathcal{L}_P$

**Theorem**

Verification of $\mu\mathcal{L}_P$ over state-bounded DCDSs is decidable and can be reduced to model checking of propositional $\mu$-calculus over a finite transition system.

Crux: construct a **faithful abstraction** $\Theta_S$ for $\Upsilon_S$, collapsing infinite branching and compacting infinite runs.

1. **Prune** infinite branching (isomorphic types).
State-bounded Systems: Decidability for $\mu L_P$

**Theorem**

Verification of $\mu L_P$ over state-bounded DCDSs is decidable and can be reduced to model checking of propositional $\mu$-calculus over a finite transition system.

Crux: construct a **faithful abstraction** $\Theta_S$ for $\Upsilon_S$, collapsing infinite branching and compacting infinite runs.

1. **Prune** infinite branching (isomorphic types).
State-bounded Systems: Decidability for $\mu L_P$

**Theorem**

Verification of $\mu L_P$ over state-bounded DCDSs is decidable and can be reduced to model checking of propositional $\mu$-calculus over a finite transition system.

Crux: construct a **faithful abstraction** $\Theta_S$ for $\Upsilon_S$, collapsing infinite branching and compacting infinite runs.

1. **Prune** infinite branching (isomorphic types).
2. Finite abstraction along the runs:
   - **Recycle** old, non-persisting objects instead of inventing new ones.
Sufficient Syntactic Conditions

State- and run-boundedness are semantic conditions. We show they are undecidable to check.

We then introduce two *incomparable* sufficient syntactic conditions:

- **Weak acyclicity** (cf. data exchange), to check whether a DCDS with deterministic services is run-bounded.

- **Generate-recall acyclicity**, to check whether a DCDS is state-bounded.

Both conditions are checked against a dependency graph that abstracts the data-flow of the DCDS process layer.
Consider a DCDS $S$ with process \{true $\rightarrow\alpha()$\},

\[
\begin{align*}
\text{action } \alpha() : \quad & \left\{ 
\begin{array}{l}
P(x) \triangleright P(x) \\
Q(x) \triangleright Q(x)
\end{array}
\right. \\
& \left\{ 
\begin{array}{l}
P(x) \triangleright Q(f(x)) \\
Q(x) \triangleright Q(x)
\end{array}
\right.
\end{align*}
\]

Consider \textbf{nondeterministic} service calls.

$S$ is \textbf{not} state-bounded.

The problem comes from the interplay between:

- a \textbf{generate cycle} that continuously feeds a path issuing service calls;
- a \textbf{recall cycle} that accumulates the obtained results.
- (+ the fact that both cycles are active at the same time)

\textbf{GR-acyclicity} detects exactly these undesired situations.
Consider a DCDS $S$ with process $\{ \text{true} \xrightarrow{} \alpha(), \text{true} \xrightarrow{} \beta() \}$,

actions $\alpha() : \{ P(x) \rightsquigarrow Q(f(x)) \}$
$\beta() : \{ Q(x) \rightsquigarrow P(x) \}$

Consider deterministic service calls.

$S$ is not run-bounded.

The problem comes from:

- repeated calls to the same service.
- every time using fresh values that are directly (or indirectly) obtained by manipulating previous results produced by the same service.

Weak acyclicity detects these undesired situations.
Conclusions

• Our work is grounded in real-world data-aware processes.
• We study robust conditions for decidability.
  ▶ no conditions on the structure of the database;
  ▶ the database changes over time;
  ▶ suitable restrictions are posed on the process layer.
• Complexity wise, our techniques are exponential in the size of the initial database.
• However, most often processes change only a small (logarithmic \( \Theta \)?) portion of the entire database.
• Next step: formalize this intuition.
History-Preserving Bisimulation

Given $\Upsilon_1$, $\Upsilon_2$ over $\neq$ data domains $\Delta_1$ and $\Delta_2$, with states $\Sigma_1$ and $\Sigma_2$...

- Is a ternary relation $\approx \subseteq \Sigma_1 \times H \times \Sigma_2$, connecting pairs of states under a bijection that tracks the history.

- In particular, $s_1 \approx_h s_2$ implies that:
  1. $h$ is a partial bijection between $\Delta_1$ and $\Delta_2$ that induces an isomorphism between $db_1(s_1)$ and $db_2(s_2)$;
  2. for each $s'_1$, if $s_1 \Rightarrow_1 s'_1$ then there is an $s'_2$ with $s_2 \Rightarrow_2 s'_2$ and a bijection $h'$ that extends $h$, such that $s'_1 \approx_{h'} s'_2$;
  3. for each $s'_2$, if $s_2 \Rightarrow_2 s'_2$ then there is an $s'_1$ with $s_1 \Rightarrow_1 s'_1$ and a bijection $h'$ that extends $h$, such that $s'_1 \approx_{h'} s'_2$.

- $\Upsilon_1 \approx \Upsilon_2$ if there exists a partial bijection $h_0$ such that $s_{01} \approx_{h_0} s_{02}$.
History-Preserving Bisimulation

Given $\Upsilon_1$, $\Upsilon_2$ over $\neq$ data domains $\Delta_1$ and $\Delta_2$, with states $\Sigma_1$ and $\Sigma_2$...

- Is a ternary relation $\approx \subseteq \Sigma_1 \times H \times \Sigma_2$, connecting pairs of states under a bijection that tracks the history.
- In particular, $s_1 \approx_h s_2$ implies that:
  1. $h$ is a partial bijection between $\Delta_1$ and $\Delta_2$ that induces an isomorphism between $db_1(s_1)$ and $db_2(s_2)$;
  2. for each $s'_1$, if $s_1 \Rightarrow_1 s'_1$ then there is an $s'_2$ with $s_2 \Rightarrow_2 s'_2$ and a bijection $h'$ that extends $h$, such that $s'_1 \approx_{h'} s'_2$;
  3. for each $s'_2$, if $s_2 \Rightarrow_2 s'_2$ then there is an $s'_1$ with $s_1 \Rightarrow_1 s'_1$ and a bijection $h'$ that extends $h$, such that $s'_1 \approx_{h'} s'_2$.

- $\Upsilon_1 \approx \Upsilon_2$ if there exists a partial bijection $h_0$ such that $s_{01} \approx_{h_0} s_{02}$.

Theorem

If $\Upsilon_1 \approx \Upsilon_2$, then for every $\mu L_A$ closed formula $\Phi$, we have:

$$\Upsilon_1 \models \Phi \quad \text{if and only if} \quad \Upsilon_2 \models \Phi.$$
Persistence-Preserving Bisimulation

Given $\Upsilon_1$, $\Upsilon_2$ over $\neq$ data domains $\Delta_1$ and $\Delta_2$, with states $\Sigma_1$ and $\Sigma_2$...

- It is a ternary relation $\sim \subseteq \Sigma_1 \times H \times \Sigma_2$, connecting pairs of states under a bijection that tracks the history of persisting objects.

- In particular, $s_1 \sim_h s_2$ implies that:

  1. $h$ is a partial bijection between $\Delta_1$ and $\Delta_2$ that induces an isomorphism between $db_1(s_1)$ and $db_2(s_2)$;

  2. for each $s_1'$, if $s_1 \Rightarrow_1 s_1'$ then there exists an $s_2'$ with $s_2 \Rightarrow_2 s_2'$ and a bijection $h'$ that extends $h$ restricted on $\text{ADOM}(db_1(s_1)) \cap \text{ADOM}(db_1(s_1'))$, such that $s_1' \sim_{h'} s_2'$;

  3. for each $s_2'$, if $s_2 \Rightarrow_2 s_2'$ then there exists an $s_1'$ with $s_1 \Rightarrow_1 s_1'$ and a bijection $h'$ that extends $h$ restricted on $\text{ADOM}(db_1(s_1)) \cap \text{ADOM}(db_1(s_1'))$, such that $s_1' \sim_{h'} s_2'$.

- $\Upsilon_1 \sim \Upsilon_2$ if there exists a partial bijection $h_0$ such that $s_{01} \sim_{h_0} s_{02}$.
 Persistence-Preserving Bisimulation

Given $\Upsilon_1, \Upsilon_2$ over $\neq$ data domains $\Delta_1$ and $\Delta_2$, with states $\Sigma_1$ and $\Sigma_2$...

- It is a ternary relation $\sim \subseteq \Sigma_1 \times H \times \Sigma_2$, connecting pairs of states under a bijection that tracks the history of persisting objects.

- In particular, $s_1 \sim_h s_2$ implies that:
  1. $h$ is a partial bijection between $\Delta_1$ and $\Delta_2$ that induces an isomorphism between $db_1(s_1)$ and $db_2(s_2)$;
  2. for each $s'_1$, if $s_1 \Rightarrow_1 s'_1$ then there exists an $s'_2$ with $s_2 \Rightarrow_2 s'_2$ and a bijection $h'$ that extends $h$ restricted on $ADOM(db_1(s_1)) \cap ADOM(db_1(s'_1))$, such that $s'_1 \sim_{h'} s'_2$;
  3. for each $s'_2$, if $s_2 \Rightarrow_2 s'_2$ then there exists an $s'_1$ with $s_1 \Rightarrow_1 s'_1$ and a bijection $h'$ that extends $h$ restricted on $ADOM(db_1(s_1)) \cap ADOM(db_1(s'_1))$, such that $s'_1 \sim_{h'} s'_2$.

- $\Upsilon_1 \sim \Upsilon_2$ if there exists a partial bijection $h_0$ such that $s_{01} \sim_{h_0} s_{02}$.

**Theorem**

*If $\Upsilon_1 \sim \Upsilon_2$, then for every $\mu L_P$ closed formula $\Phi$, we have:*

$$\Upsilon_1 \models \Phi \quad \text{if and only if} \quad \Upsilon_2 \models \Phi.$$