Verification of Parameterized Data-Aware Dynamic Systems

Marco Montali

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

Credits to
Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Alin Deutsch, Andrey Rivkin

PV 2014
Data and Processes

The information assets of an organization are constituted by:

- **data**, and
- **processes**, that determine how data changes and evolves over time.
Data and Processes

The information assets of an organization are constituted by:

- data, and
- processes, that determine how data changes and evolves over time.

Conceptual Modeling

Both aspects are modelled conceptually, but:

- Using different modeling tools
- By different teams with different competences
- Connection between the two is NOT modelled conceptually
Data and Processes

The information assets of an organization are constituted by:

- data, and
- processes, that determine how data changes and evolves over time.

Conceptual Modeling

Both aspects are modelled conceptually, but:

- Using different modeling tools
- By different teams with different competences
- Connection between the two is NOT modelled conceptually

Consequence

Full reasoning support, e.g., for verification taking into account both process and data, is not possible!
A Classical Data Model

How are data of this schema evolved over time?
A Classical Process Model

Where are the data?
Reality: A Deployed Process

- **Current data**: fields, text, highlights.
  - Include **status attributes**.
- **Actions**: buttons, links.
- **Input params**: writable fields.
Our goals

Formalization of Data-Aware Dynamic Systems

Lay the foundations for processes dealing with a full-fledged data layer:

- **relational database with constraints** (complete information);
- conceptual model/ontology (incomplete information);
- both (cf. ontology-based data access).

Reasoning support along the entire process lifecycle

Develop techniques for:

- *(design-time)* verification and synthesis;
- *(run-time)* operational support (monitoring);
- *(a-posteriori)* analysis and mining.

Grounding of the approach

Focus on concrete languages/settings like: business artifacts + adaptive case management (GSM, CMMN, ...), healthcare processes, dynamic web apps, ...
Data-Aware Dynamic System

A dynamic system that manipulates data over time.

- **Data layer**: maintains data of interest.
  - Relational database.
  - (Description logic) ontology.
- **Process layer**: evolves the extensional part of the data layer.
  - Control-flow component: determines when actions can be executed.
  - Actions: atomic units of work that update the data.
    - Interact with the external world to inject *fresh data* into the system.
Data-Centric Dynamic Systems (DCDSs) [PODS13]

- Data layer: relational database with FO constraints.
- Actions: specified by (parallel) effects that query the current state and determine the next state.
  - Service calls can be invoked to incorporate new data.

Example

- Data layer: schema \{R/2, Q/1, S/1\}, no constraint.
- Process: \( \exists y. R(x, y) \mapsto t(x) \).
- Action: \( t(p) : \left\{ \begin{array}{c}
R(x, y) \land x \neq p \mapsto R(x, y) \\
R(p, y) \mapsto R(p, f(y)) \\
R(p, y) \mapsto Q(p)
\end{array} \right\} \)
Data-Centric Dynamic Systems (DCDSs) [PODS13]

- Data layer: relational database with FO constraints.
- Actions: specified by (parallel) effects that query the current state and determine the next state.
  - Service calls can be invoked to incorporate new data.

Example

- Data layer: schema \{R/2, Q/1, S/1\}, no constraint.
- Process: \( \exists y. R(x, y) \mapsto t(x) \).
- Action: \( t(p) : \begin{cases} 
  R(x, y) \land x \neq p & \mapsto R(x, y) \\
  R(p, y) & \mapsto R(p, f(y)) \\
  R(p, y) & \mapsto Q(p) 
\end{cases} \)
Data-Centric Dynamic Systems (DCDSs) [PODS13]

- **Data layer**: relational database with FO constraints.
- **Actions**: specified by (parallel) effects that query the current state and determine the next state.
  - Service calls can be invoked to incorporate new data.

### Example

- **Data layer**: schema \( \{ R/2, Q/1, S/1 \} \), no constraint.
- **Process**: \( \exists y. R(x, y) \mapsto t(x) \).
- **Action**: \( t(p) : \)
  \[
  \begin{cases}
  R(x, y) \land x \neq p & \mapsto R(x, y) \\
  R(p, y) & \mapsto R(p, f(y)) \\
  R(p, y) & \mapsto Q(p)
  \end{cases}
  \]

\( R(a,b), R(a,c), R(c,d), Q(d), S(b) \)

\( t(a) \)

\( t(c) \)
Data-Centric Dynamic Systems (DCDSs) [PODS13]

- Data layer: relational database with FO constraints.
- Actions: specified by (parallel) effects that query the current state and determine the next state.
  - Service calls can be invoked to incorporate new data.

Example

- Data layer: schema \( \{ R/2, Q/1, S/1 \} \), no constraint.
- Process: \( \exists y. R(x, y) \mapsto t(x) \).
- Action: \( t(a) : \)
  \[
  \begin{cases}
  R(x, y) \land x \neq a \leadsto R(x, y) \\
  R(a, y) \leadsto R(a, f(y)) \\
  R(a, y) \leadsto Q(a)
  \end{cases}
  \]

\[
R(a, b), R(a, c), \\
R(c, d), \\
Q(d), \\
S(b)
\]

\[
R(c, d), Q(a), \\
R(a, f(b)), R(a, f(c))
\]
Data-Centric Dynamic Systems (DCDSs) [PODS13]

- Data layer: relational database with FO constraints.
- Actions: specified by (parallel) effects that query the current state and determine the next state.
  - Service calls can be invoked to incorporate new data.

**Example**

- Data layer: schema \( \{ R/2, Q/1, S/1 \} \), no constraint.
- Process: \( \exists y. R(x, y) \rightarrow t(x) \).
- Action: \( t(a) : \)
  \[
  \begin{align*}
  R(x, y) \land x \neq a & \leadsto R(x, y) \\
  R(a, y) & \leadsto R(a, f(y)) \\
  R(a, y) & \leadsto Q(a)
  \end{align*}
  \]

\[
\begin{align*}
R(a,b), R(a,c), \\
R(c,d), \\
Q(d), \\
S(b)
\end{align*}
\]

\[
\begin{align*}
R(c,d), Q(a), \\
R(a,f(b)) R(a,f(c))
\end{align*}
\]

\[
\begin{align*}
R(c,d), Q(a), \\
R(a,a)
\end{align*}
\]
**Data-Centric Dynamic Systems (DCDSs)** [PODS13]

- **Data layer:** relational database with FO constraints.
- **Process (control-flow):** condition-action rules.
- **Actions:** specified by (parallel) effects that query the current state and determine the next state.
  - Service calls can be invoked to incorporate new data.

**Example**

- **Data layer:** schema \( \{ R/2, Q/1, S/1 \} \), no constraint.
- **Process:** \( \exists y. R(x, y) \mapsto t(x) \).
  - \( t(a) : \begin{cases} 
  R(x, y) \land x \neq a & \mapsto R(x, y) \\
  R(a, y) & \mapsto R(a, f(y)) \\
  R(a, y) & \mapsto Q(a) 
  \end{cases} \)

\[ \begin{align*}
R(a, b), R(a, c), \\
R(c, d), \\
Q(d), \\
S(b)
\end{align*} \]

\[ \begin{align*}
 t(a) & \quad \rightarrow \quad \begin{align*}
R(c, d), Q(a), \\
R(a, f(b)), R(a, f(c))
\end{align*} \\
\text{f(b) = c, f(c) = u} & \quad \rightarrow \quad \begin{align*}
R(c, d), Q(a), \\
R(a, c), R(a, u)
\end{align*}
\]
Data-Centric Dynamic Systems (DCDSs) [PODS13]

- Data layer: relational database with FO constraints.
- Actions: specified by (parallel) effects that query the current state and determine the next state.
  - Service calls can be invoked to incorporate new data.

Example

- Data layer: schema \{R/2, Q/1, S/1\}, no constraint.
- Process: \( \exists y. R(x, y) \mapsto t(x) \).
- Action: \( t(a) : \)
  \[
  \begin{align*}
  R(x, y) \land x \neq a & \mapsto R(x, y) \\
  R(a, y) & \mapsto R(a, f(y)) \\
  R(a, y) & \mapsto Q(a)
  \end{align*}
  \]

\[\begin{align*}
R(a,b), R(a,c), \\
R(c,d), \\
Q(d), \\
S(b)
\end{align*}\]

\[\begin{align*}
R(c,d), Q(a), \\
R(a,f(b)) R(a,f(c))
\end{align*}\]
Verification

Execution semantics: a relational, infinite-state transition system.

- **Infinite branching**
  (possible results of service calls).

- **Infinite runs**
  (usage of values obtained from unboundedly many service calls).

- **Unbounded DBs**
  (accumulation of such values).

**Verification Problem**
Check whether the dynamic system guarantees a desired property, expressed in some variant of a **first-order temporal** logic.
Verification Logics

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.

\( \mu \mathcal{L}_A \)

First-order \( \mu \)-calculus with active domain quantification.

Example: In every state of the system, each order stored in that state is eventually shipped.

\( \mu \mathcal{L}_P \)

Syntactically limits first-order quantification across states: it applies only to those objects that persist.

- Internal primary keys, student ids, pure names, …

Example: In every state of the system, each order stored in that state persists in the system until it is shipped.
Verification of propositional reachability properties is **undecidable** even for DCDSs with unary relations only.

Counter increment $k1: c++; \text{ goto } k2$

$PC(k1) \rightarrow \text{ inc-C}(k2)$ where \text{ inc-C}(k_n):

\[
\begin{cases}
C(x) \sim \sim C(x) \\
C(x) \sim \sim C_{old}(x) \\
\text{true} \sim \sim C_{new}(f()), C(f()) \\
\text{true} \sim \sim PC(k_{next})
\end{cases}
\]

with fresh name database constraint: $\exists x. C_{new}(x) \land C_{old}(x) \rightarrow false$

Counter conditional decrement $k1: \text{ if}(c=0) \text{ goto } k2; \text{ else} \{c--; \text{ goto } k3;\}$

$PC(k1) \land C(e) \rightarrow \text{ dec-C}(e, k2)$ where \text{ dec-C}(e, k_n):

\[
\begin{cases}
C(x) \land x \neq e \sim \sim C(x) \\
\text{true} \sim \sim PC(k_{next})
\end{cases}
\]

$PC(k1) \land \neg \exists x. C(x) \rightarrow \text{ jmp}(k3)$ where \text{ jmp}(k_n):

\[
\begin{cases}
\text{true} \sim \sim PC(k_{next})
\end{cases}
\]
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.
Bisimulations

Corresponding notions of bisimulation are defined. They 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.
  - For \( \mu \mathcal{L}_P \): forgetting about old, non-persisting objects.

\[
\begin{align*}
\mu \mathcal{L}_{FO} \\
\mu \mathcal{L}_A \\
\mu \mathcal{L}_P \\
\mu \mathcal{L}
\end{align*}
\]

History-Preserving Bisimulation Invariant Languages
Persistence-Preserving Bisimulation Invariant Languages
Propositional Bisimulation Invariant Languages

LTL, PDL, CTL
### Summary of Results [PODS13]

<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.
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.
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 TS.

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

1. **Prune** infinite branching (isomorphic types).

\[ \ldots \]

\[ \ldots \]

\[ \ldots \]

\[ \ldots \]
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 TS.

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 TS.

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 TS.

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 when possible.
   ▶ At some point, no fresh value needs to be introduced anymore.

No need to know the actual bound!
Parameterization with Names

In [AAMAS14]: extension towards data-aware multiagent systems.

- Agents use **names** to exchange data.
- An **institutional agent** manages name creation and deletion.

State-boundedness implies that **unboundedly** many agents/names can be seen in a run, provided that they **do not coexist** in the same state.
State- (and run-) boundedness are semantic properties, **undecidable** to check.

**Big question 1**

Do there exist interesting classes for which state-boundedness is decidable?
On State-Boundedness

State- (and run-) boundedness are semantic properties, **undecidable** to check.

Big question 1

Do there exist interesting classes for which state-boundedness is decidable?

+: Petri nets with name management

By Rosa-Velardo et al.

[WSFM14]: Translation to DCDSs and $\mu\mathcal{L}_P$
On State-Boundedness

State- (and run-) boundedness are semantic properties, **undecidable** to check.

**Big question 1**

Do there exist interesting classes for which state-boundedness is decidable?

**+**: Petri nets with name management

By Rosa-Velardo et al.

**−**: Reset-Transfer Nets

**⊂** Reset Post G-nets by Dufour et al.

[WSFM14]: Translation to DCDSs and $\mu L_P$

[KR14]: “Lossy” correspondence with DCDSs.
Attacking State-Boundedness

These variants of Petri nets correspond to restricted classes of DCDSs with unary relations, limited use of negation, no or limited joins, . . .

Big question 2

How to check/guarantee that a system is state-bounded?
Attacking State-Boundedness

These variants of Petri nets correspond to restricted classes of DCDSs with unary relations, limited use of negation, no or limited joins, ... 

Big question 2

How to check/guarantee that a system is state-bounded?

Strategy 1

Sufficient, syntactic conditions.

Taking inspiration from conditions on chase termination for TGDs:

• Extract a data-flow graph from the DCDS action specification.
• Check how data flow through this graph.
• See [PODS13] and [KR14].
Attacking State-Boundedness

These variants of Petri nets correspond to restricted classes of DCDSs with unary relations, limited use of negation, no or limited joins, ... 

Big question 2

How to check/guarantee that a system is state-bounded?

Strategy 1

Sufficient, syntactic conditions.

Taking inspiration from conditions on chase termination for TGDs:

- Extract a data-flow graph from the DCDS action specification.
- Check how data flow through this graph.
- See [PODS13] and [KR14].

Strategy 2

State-boundedness by design.

Design methodologies so as to guarantee state-boundedness. E.g., in [ICSOC13]:

- Processes are bound to evolving business objects (artifacts).
- Each business object manipulate boundedly many data.
- (New) business objects pick their names from a fixed pool of ids.
Towards Controlled Unboundedness

Observation: Parameterization in data-aware dynamic systems

- In classical process management . . .
  ▶ Central notion of case representing a process instance.
- In artifact-centric process management:
  ▶ Central notion of business object gluing data and behaviour together.
- New cases/objects often created by external stakeholders!
Towards Controlled Unboundedness

Observation: Parameterization in data-aware dynamic systems

- In classical process management ...
  - Central notion of case representing a process instance.
- In artifact-centric process management:
  - Central notion of business object gluing data and behaviour together.
- New cases/objects often created by external stakeholders!

Hence...

No bound on the number of simultaneously active cases/objects.
Towards Controlled Unboundedness

Observation: Parameterization in data-aware dynamic systems

- In classical process management . . .
  ▶ Central notion of case representing a process instance.
- In artifact-centric process management:
  ▶ Central notion of business object gluing data and behaviour together.
- New cases/objects often created by external stakeholders!

Hence . . .
No bound on the number of simultaneously active cases/objects.

But . . .

<table>
<thead>
<tr>
<th>State</th>
<th>Group</th>
<th>MarryM</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>state</td>
<td>id</td>
</tr>
<tr>
<td>12</td>
<td>out</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>in</td>
<td>4</td>
</tr>
<tr>
<td>431</td>
<td>running</td>
<td>431</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Towards Controlled Unboundedness

Observation: Parameterization in data-aware dynamic systems

- In classical process management . . .
  - Central notion of case representing a process instance.
- In artifact-centric process management:
  - Central notion of business object gluing data and behaviour together.
- New cases/objects often created by external stakeholders!

Hence . . .

No bound on the number of simultaneously active cases/objects.

But . . .

<table>
<thead>
<tr>
<th>State</th>
<th>Group</th>
<th>MarryM</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>state</td>
<td>id</td>
</tr>
<tr>
<td>12</td>
<td>out</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>in</td>
<td>4</td>
</tr>
<tr>
<td>431</td>
<td>running</td>
<td>431</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Marco Montali (unibz)
Towards Controlled Unboundedness

In [CIKM14,Subm14] we formally capture the two notions of “data isolation” and “relative boundedness”:

- Modelling guidelines for the data component and how the process component accesses to it.

### Restrictions

1. Queries must be navigational: no arbitrary access to relations.
2. 1-to-many relations require a number restriction on the “many” side.
3. Each case cannot create a chain of tuples of unbounded length.
4. Cases can share tuples only in a controlled way.
   - So as to avoid the construction of chains of cases.

### Key Decidability Result

Verification of navigational $\mu L_P$ properties on the unbounded system can be reduced to verifying a bounded number of cases/business objects!

- A sort of data-aware cutoff technique.
Towards Concrete Domains

Ongoing work with Diego Calvanese and Giorgio Delzanno.

What about concrete domains with specific relations?
E.g., ordered domains with $\geq$.
  - Think about classical ticket-based coordination algorithms.

How does this interact with state-boundedness?

Some key insights:
  - No hope to include the successor relation.
    - 2 data slots are sufficient to encode two counters.
  - OK to include dense linear orders.
    - Thanks to state-boundedness:

| Rigid $\geq$ relation over the entire domain | Non-rigid GreaterThan relation over active domain elements. |
Conclusion

• Our work is grounded in real-world data-aware processes.
• State-boundedness provides the basis for robust conditions for decidability.
• Key idea: only finitely many data are important per sè, the others act as placeholders (cf. pure values).
• Complexity-wise: we are studying how the exponentiality in the data inherent in data-aware systems can be tamed, through a suitable modularization of the read-write data slots into independent portions.
• Parameterization currently works for boundedly many components simultaneously coexisting in the system, or by restricting their interaction through the data component.


Other works consider a description-logic knowledge base or an ontology-based data access system as the data component.

- See [RR12, ECAI12, JAIR13, IJCAI13, RR13, ICSOC13b].