Verification of Data-Centric Dynamic Systems

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

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

Currently on sabbatical leave at Technical University Vienna, Austria

EPCL Basic Training Camp 2012/2013
10–21/12/2012 Dresden, Germany
Outline

1. Combining static and dynamic aspects
2. Data-Centric Dynamic Systems
3. Semantics of DCDS
4. Verification
5. Run-boundedness
6. State-boundedness
7. Conclusions
Outline

1. Combining static and dynamic aspects
2. Data-Centric Dynamic Systems
3. Semantics of DCDS
4. Verification
5. Run-boundedness
6. State-boundedness
7. Conclusions
Artifacts are a sort of middle ground between a conceptual formalization of a dynamic system and an actual implementation of the system itself.

Artifacts systems are characterized by:

- **Information model**: takes into account the structural properties.
- **Process**: takes into account the dynamic properties.
The problem: reasoning on dynamic entities carrying data

We need to decide whether dynamic/temporal properties of interest hold over the life of such systems:

- Verification of temporal formulas.
- Checking dominance/simulation/bisimulation/containment properties.
- Automated composition of artifacts-based systems.
- Automated process synthesis from dynamic/temporal specifications.

*Note*: Currently (i.e., 2010’s), the scientific community is quite good at each of these, but only in a **finite state setting**!
The problem: reasoning on dynamic entities carrying data

- Information model affects the number of different states of the system.
- Presence of data makes the systems potentially infinite-state.
- Usual techniques, e.g., model checking, used for finite-state systems don’t work off-the-shelf.

We aim at exploring suitable representation formalisms:

- that are expressive enough some real life scenarios;
- should admit decidability of reasoning.
A solution for reasoning on dynamic entities carrying data

We make use of contribution coming from different areas:

- work on **data integration** and **data exchange** that advocate a semantic view of the data ← Databases;
- work on **data access and update** through ontologies and description logics ← KR and Databases;
- work in **reasoning about actions** formalize dynamic systems using logics ← KR and AI;

**Key idea**

Work by Fagin & Kolaitis (IBM Almaden) and others on the use of **data dependency theory** for data exchange (Databases) can be seen as talking about actions effects (KR and AI).

**Finite chase** ←→ **Finite state system**

We devise a reduction to **reasoning on finite state systems**.
Outline

1. Combining static and dynamic aspects
2. Data-Centric Dynamic Systems
3. Semantics of DCDS
4. Verification
5. Run-boundedness
6. State-boundedness
7. Conclusions
Data-Centric Dynamic Systems

We consider systems where the **process** controlling the dynamics and the manipulated **data** are **equally central**:

- Provides a general, abstract framework.
- Artifact-centric systems are a special case of DCDSs.

Two key components:

- **Data Layer**: holds the relevant information to be manipulated
- **Process Layer**:
  - **Atomic actions**: access and update data.
  - **Process**: finite state control over conditional action invocation.
  - **External service calls**: to communicate with the external environment (other systems, user choices, ...), possibly acquiring new data objects.
Data Layer

- Represents the **information of interest** in our application.
- We focus on **relational** data.

The data layer is a tuple \( \mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle \) where:

- **\( \mathcal{C} \)** is a countably infinite set of **constants/values**.
- **\( \mathcal{R} = \{ R_1, \ldots, R_n \} \)** is a **database schema**, i.e. a set of relation schemas.
- **\( \mathcal{E} \)** is a finite set of **equality constraints** \( Q_i \rightarrow \bigwedge_{j=1}^{k} z_{ij} = y_{ij} \).
  - \( Q_i \) is a domain independent FO query over \( \mathcal{R} \) using constants from the active domain \( \text{ADOM}(\mathcal{I}_0) \) and whose free variables are \( \vec{x} \).
  - \( z_{ij} \) and \( y_{ij} \) are either variables in \( \vec{x} \) or constants in \( \text{ADOM}(\mathcal{I}_0) \).
  - Note: we could generalize to denials and arbitrary constraints!
- **\( \mathcal{I}_0 \)** is a database instance representing the **initial state** of the data layer:
  - It conforms to the database schema \( \mathcal{R} \).
  - It **satisfies** the constraints \( \mathcal{E} \): for each constraint \( Q_i \rightarrow \bigwedge_{j=1}^{k} z_{ij} = y_{ij} \) and for each tuple \( \theta \in \text{ans}(Q_i, \mathcal{I}_0) \), \( z_{ij} \theta = y_{ij} \theta \).
Process Layer

- Constitutes the **progression mechanism** for the DCDS.
- High-level: rule-based approach that can accommodate any process with a finite state control flow.
- Parallelism represented by interleaving.

A process layer $\mathcal{P}$ over a data layer $\mathcal{D}$ is a tuple $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$ where:

- $\mathcal{F}$ is a finite set of functions representing **external service** interfaces, whose behavior is **unknown** to the DCDS;
- $\mathcal{A}$ is a finite set of atomic **actions**;
- $\varrho$ is a finite set of **condition-action rules** forming the specification of the overall **process**.
Actions

An action is constituted by:

- a **name**;
- a list $\vec{x}$ of **input parameters** (to be substituted by individuals/ constants);
- a set $\{e_1(\vec{x}), \ldots, e_n(\vec{x})\}$ of **effects**, which are assumed to take place simultaneously when the action is executed.

Each effect $e_i(\vec{x})$ has the form $q_i^+(\vec{x}, \vec{y}) \land Q_i^-(\vec{x}, \vec{y}) \rightsquigarrow E_i(\vec{x}, \vec{y})$ where:

- $q_i^+(\vec{x}, \vec{y}) \land Q_i^-(\vec{x}, \vec{y})$ is a query over $\mathcal{R}$ and constants of $\text{ADOM}(\mathcal{I}_0)$:
  - $q_i^+$ is a **UCQ** over $\mathcal{R}$ that acts as a selector of data of interest.
  - $Q_i^-$ is a **FOL** query that acts as a **filter** (i.e., the free variables of $Q_i^-$ are included in those of $q_i^+$).
  - Note: the query may include some of the input parameters $\vec{x}$ as terms.
- $E_i$ is a set of facts over $\mathcal{R}$, which may include as terms:
  - constants in $\text{ADOM}(\mathcal{I}_0)$,
  - parameters $\vec{x}$ and other free variables $\vec{y}$ of $q_i^+$, and
  - **functions calls** that formalize calls to (atomic) **external services**.
  - These calls may introduce new values in the data maintained by the DCDS!
Outline

1. Combining static and dynamic aspects
2. Data-Centric Dynamic Systems
3. Semantics of DCDS
4. Verification
5. Run-boundedness
6. State-boundedness
7. Conclusions
Process and action execution

The **process** is a finite set of **condition-action rules** $Q(\bar{x}) \rightarrow \alpha(\bar{x})$, where:

- $\alpha(\bar{x})$ is an **action** in $\mathcal{A}$ with parameters $(\bar{x})$;
- $Q(\bar{x})$ is a **FO query** over $\mathcal{R}$ with free variables $\bar{x}$, whose other terms can be either quantified variables or constants in $\text{ADOM}(\mathcal{I}_0)$.

To execute an action $\alpha(\bar{x})$ in state $s$ according to $Q(\bar{x}) \rightarrow \alpha(\bar{x})$:

1. evaluate $Q(\bar{x})$ over $db(s)$, and if the result is non-empty, then $\alpha(\bar{x})$ is **executable**;
2. among the returned tuples choose a parameters assignment $\sigma$ for $\bar{x}$;
3. $\alpha\sigma$ is executed: for each effect $q_i^+ \land Q_i^- \leadsto E_i$
   1. $(q_i^+ \land Q_i^-)\sigma$ is evaluated over $db(s)$, getting variables assignments $\theta_1, \ldots, \theta_n$;
   2. for each $\theta_i$, the grounded facts $E_i \theta_i$ are obtained;
   3. all **service calls** contained in $E_i \theta_i$ are **issued**;
   4. the next state is obtained by asserting each $E_i \theta_i$ after the inclusion of all service call results.
DCDS: Example

Data Layer

Schema

peer

Customer

In Debt Customer

owes

closed

Gold Customer

Loan

Instance

Cust(ann)
peer(mark, john)
Gold(john)
owes(mark, @25)

Process Layer

Conditions

peer(x, y) ∧ Gold(y)

GetLoan(x)

Service Calls

UInput(x)

Actions

GetLoan(x):

∃y.peer(x, y) ↦ {owes(x, UInput(x))},
Cust(z) ↦ {Cust(z)},
Loan(z) ↦ {Loan(z)},
InDebt(z) ↦ {InDebt(z)},
Gold(z) ↦ {Gold(z)}
Semantics of a DCDS is given in terms of a **transition system** $\Upsilon = \langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$:

- $\Delta$ is a countably infinite set of values;
- $\mathcal{R}$ is a database schema;
- $\Sigma$ is a set of states;
- $s_0 \in \Sigma$ is the initial state;
- $db$ is a function that, given a state $s \in \Sigma$, returns the database associated to $s$, which is made up of values in $\Delta$ and conforms to $\mathcal{R}$;
- $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation between pairs of states.

**Note:** $\Upsilon$ is in general **infinite state**.
Construction of $\Upsilon_S$

1. Start from $s_0 = \langle I_0, \emptyset \rangle \in \Sigma$.
2. Repeat forever:
   - Pick a state $s \in \Sigma$.
   - For every action executable in $s$, every “legal” parameters assignment, every possible service calls results’ configuration . . .
     - generate the successor state $s'$ and put it in $\Sigma$;
     - insert $\langle s, s' \rangle$ in $\Rightarrow$.

Note: three sources of non-determinism in each step:
- actions non-determinism;
- parameters non-determinism;
- service call results non-determinism: leads to potentially infinite branching.
Deterministic vs. non-deterministic services

We distinguish between two different semantics for service-execution:

**Deterministic services semantics**

Along the same run, when the same service is invoked again with the same arguments, it returns the same result as in the previous call.

Are used to model an environment whose behavior is completely determined by the parameters.

**Example:** temperature, given the location and the date and time

**Non-deterministic services semantics**

Along the same run, when the same service is invoked again with the same arguments, it may return a different value than in the previous call.

Are used to model:

- an environment whose behavior is determined by parameters that are outside the control of the system;
- input of external users, whose choices depend on external factors.

**Example:** current temperature, given the location
Transition system with deterministic services

- Semantics in terms of **concrete transition system**
  \( \Upsilon_S = \langle C, R, \Sigma, s_0, db, \Rightarrow \rangle \).
- Each state \( s \in \Sigma \) remembers all previous service call results: \( s = \langle I, M \rangle \), where \( I \) is a database and \( M \) is a map from service calls to results in \( C \).
- \( db(\langle I, M \rangle) = I \).
- Action execution:
  - before issuing a service call, it is checked whether the (deterministic) result is **already contained** in \( M \);
  - if it is, then the result is **already known**;
  - if not, the call is **issued** and the obtained **result is stored** in \( M \).
Static and dynamic aspects DCDSs
Semantics of DCDS
Verification
Run-boundedness
State-boundedness
Conclusions

Deterministic services semantics – Via transition systems

\[
\begin{align*}
\{ & P(x) \leadsto P(x) \land Q(f(x), g(x)) \\
& Q(a, a) \land P(x) \leadsto R(x),
\}
\end{align*}
\]

\[\mathcal{I} = \{ P(a), Q(a, a) \}\]

\[
\begin{align*}
P(a) & \quad Q(a, a) \\
\hline
\end{align*}
\]

\[
\begin{align*}
f(a) & \mapsto a \\
g(a) & \mapsto a \\
\hline
P(a) & \quad R(a) & \quad Q(a, a)
\end{align*}
\]

\[
\begin{align*}
f(a) & \mapsto a \\
g(a) & \mapsto b \\
\hline
P(a) & \quad R(a) & \quad Q(a, b)
\end{align*}
\]

\[
\begin{align*}
f(a) & \mapsto b \\
g(a) & \mapsto a \\
\hline
P(a) & \quad R(a) & \quad Q(b, a)
\end{align*}
\]

\[
\begin{align*}
f(a) & \mapsto b \\
g(a) & \mapsto b \\
\hline
P(a) & \quad R(a) & \quad Q(b, b)
\end{align*}
\]

\[
\begin{align*}
f(a) & \mapsto a \\
g(a) & \mapsto b \\
\hline
P(a) & \quad Q(a, b)
\end{align*}
\]

\[
\begin{align*}
f(a) & \mapsto b \\
g(a) & \mapsto a \\
\hline
P(a) & \quad Q(b, a)
\end{align*}
\]

\[
\begin{align*}
f(a) & \mapsto b \\
g(a) & \mapsto b \\
\hline
P(a) & \quad Q(b, b)
\end{align*}
\]

\[
\begin{align*}
f(a) & \mapsto a \\
g(a) & \mapsto b \\
\hline
P(a) & \quad Q(a, b)
\end{align*}
\]

\[
\begin{align*}
f(a) & \mapsto b \\
g(a) & \mapsto b \\
\hline
P(a) & \quad Q(b, b)
\end{align*}
\]
Outline

1. Combining static and dynamic aspects
2. Data-Centric Dynamic Systems
3. Semantics of DCDS
4. Verification
5. Run-boundedness
6. State-boundedness
7. Conclusions
Verification

- For a DCDS $S$, action execution starting from the initial state according to the process gives rise to a **transition system** $\Upsilon_S$.

- We are interested in the **verification** of temporal properties over $\Upsilon_S$.

**Problem:** $\Upsilon_S$ is in general **infinite state**

... since the calls to external services inject **new data values** into the system:

- Unbounded number of new values.
- Size of the database in each state is not bounded a priori.

**Idea:**

1. Devise a **finite-state** transition system $\Theta_S$ that is a **faithful abstraction** of $\Upsilon_S$ **independent of the formula** to verify.

2. Reduce the verification problem $\Upsilon_S \models \Phi$ to the verification of $\Theta_S \models \Phi$. 
To verify temporal properties over a DCDS $S$ we proceed as follows:

1. Do a **syntactic check** over $S$ testing whether $\gamma_S$ admits a finite-state abstraction $\Theta_S$

2. If so, construct $\Theta_S$

3. Model check $\Phi$ over $\Theta_S$ with **standard model checking** techniques
Verification formalism

- We adopt **FO** variants of the **μ-calculus**
  - **µL** is a very expressive logic!

- **µL_{FO}** formulas over a DCDS \( S \) have the form:
  \[
  \Phi ::= Q \mid \neg\Phi \mid \Phi_1 \land \Phi_2 \mid \exists x.\Phi \mid \langle-\rangle\Phi \mid Z \mid \mu Z.\Phi
  \]

  where \( Q \) is an FO query over the database schema \( R \) of \( S \), and \( Z \) is a predicate variable.

**Example**

An example of **µL** formula is:

\[
\exists x_1, \ldots, x_n \cdot \bigwedge_{i \neq j} x_i \neq x_j \land \bigwedge_{i \in \{1,\ldots,n\}} \mu Z. [\text{Stud}(x_i) \lor \langle-\rangle Z]
\]

This **defeats** any kind of finite-state abstraction.
<table>
<thead>
<tr>
<th>Static and dynamic aspects</th>
<th>DCDSs</th>
<th>Semantics of DCDS</th>
<th>Verification</th>
<th>Run-boundedness</th>
<th>State-boundedness</th>
<th>Conclusions</th>
</tr>
</thead>
</table>

**Outline**

1. Combining static and dynamic aspects
2. Data-Centric Dynamic Systems
3. Semantics of DCDS
4. Verification
5. Run-boundedness
6. State-boundedness
7. Conclusions
History preserving mu-calculus ($\mu L_A$)

- Restricts quantification over individuals to those present in the current database (denoted by $\text{LIVE}(x)$).
- Syntax:

$$\Phi ::= Q \mid \neg \Phi \mid \Phi_1 \land \Phi_2 \mid \exists x.\text{LIVE}(x) \land \Phi \mid \langle - \rangle \Phi \mid Z \mid \mu Z.\Phi$$

- We abbreviate $\neg(\exists x.\text{LIVE}(x) \land \neg \Phi)$ as $\forall x.\text{LIVE}(x) \rightarrow \Phi$.

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 that eventually leads to a graduation of the student (with some final mark $y$).
Consider two transition systems \( \Upsilon_1 = \langle \Delta_1, R, \Sigma_1, s_{01}, db_1, \Rightarrow_1 \rangle \) and \( \Upsilon_2 = \langle \Delta_2, R, \Sigma_2, s_{02}, db_2, \Rightarrow_2 \rangle \).

**Problem:** \( \Upsilon_1 \) and \( \Upsilon_2 \) are over **different data domains** \( \Delta_1 \) and \( \Delta_2 \), and a correspondence between elements must be preserved over time.

Is a relation \( B \subseteq \Sigma_1 \times H \times \Sigma_2 \) such that \( \langle s_1, h, s_2 \rangle \in B \) 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 \( \langle s'_1, h', s'_2 \rangle \in B \);
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 \( \langle s'_1, h', s'_2 \rangle \in B \).

We have \( \Upsilon_1 \approx \Upsilon_2 \) if there exists a partial bijection \( h_0 \) and a history preserving bisimulation \( B \) between \( \Upsilon_1 \) and \( \Upsilon_2 \) such that \( \langle s_{01}, h_0, s_{02} \rangle \in B \).

**Theorem**

If \( \Upsilon_1 \approx \Upsilon_2 \), then for every \( \mu \mathcal{L}_A \) closed formula \( \Phi \), we have:

\[ \Upsilon_1 \models \Phi \ \text{if and only if} \ \Upsilon_2 \models \Phi. \]
(Un)decidability results

**Theorem**

There exists a DCDS $S$ with deterministic services, and a propositional LTL safety property $\Phi$, such that checking $\Upsilon_S \models \Phi$ is undecidable.

To gain decidability, we need restrictions on the DCDS: run-boundedness

- For every run $\tau$ in $\Upsilon_S$ we have $|\bigcup_{s \text{ state of } \tau} \text{ADOM}(db(s))| < b$
- I.e., there exists a bound $b$ such that every run in $\Upsilon_S$ encounters at most $b$ different values.
- A (data) unbounded run represents an execution in which infinitely many different service calls are issued.

**Theorem**

Verification of $\mu\mathcal{L}_A$ properties on run-bounded DCDSs with deterministic services is decidable.

1. We devise a finite-state abstraction $\Theta_S$ for a run-bounded DCDS $S$.
2. We prove that $\Theta_S \approx \Upsilon_S$, hence they satisfy the same $\mu\mathcal{L}_A$ formulae.
Ensuring run-boundedness

Theorem
Checking run-boundedness of DCDSs with deterministic services is undecidable.

We have devised a sufficient syntactic condition that guarantees run-boundedness: **weak acyclicity**
- Depends only on action specifications, and not on data.
- Is polynomially checkable.

Theorem
Verification of $\mu L_A$ properties for weakly acyclic DCDSs with deterministic services is decidable, and can be reduced to model checking of propositional $\mu$-calculus over a finite transition system.
Outline

1. Combining static and dynamic aspects
2. Data-Centric Dynamic Systems
3. Semantics of DCDS
4. Verification
5. Run-boundedness
6. State-boundedness
7. Conclusions
Non-deterministic services

- The same service call issued later may give a different result.
- Can be used to model:
  - user input;
  - unpredictable external environment.

**Theorem**

There exists a DCDS $S$ with nondeterministic services, and a propositional LTL safety property $\Phi$, such that checking $\gamma_S \models \Phi$ is undecidable.

To gain decidability, we need again restrictions on the DCDS:

- Run-boundedness is too strong: it bounds the overall number of service calls.
- We consider instead **state boundedness**: there is a finite bound $b$ such that for each state $I$ of $\gamma_S$, $|\text{ADOM}(I)| < b$.

However . . .

**Theorem**

Verification of $\mu L_A$ properties on state-bounded DCDSs with nondeterministic services is undecidable.
Persistence preserving mu-calculus ($\mu L_P$)

- Restricts quantification over **individuals that continuously persist** along the system evolution, i.e., that continue to be $\text{LIVE}(x)$.

- Syntax:

  $$\Phi ::= Q \mid \neg \Phi \mid \Phi_1 \land \Phi_2 \mid \exists x.\text{LIVE}(x) \land \Phi \mid \langle \neg \rangle(\text{LIVE}(\vec{x}) \land \Phi) \mid [\neg](\text{LIVE}(\vec{x}) \land \Phi) \mid Z \mid \mu Z.\Phi$$

  where in $\text{LIVE}(\vec{x}) \land \langle \neg \rangle \Phi$ and $\text{LIVE}(\vec{x}) \land [\neg] \Phi$, the free variables of $\Phi$ are $\vec{x}$.

- We abbreviate $\neg [\neg](\text{LIVE}(\vec{x}) \land \neg \Phi)$ as $\langle \neg \rangle(\text{LIVE}(\vec{x}) \rightarrow \Phi)$

  and $\neg \langle \neg \rangle(\text{LIVE}(\vec{x}) \land \neg \Phi)$ as $[\neg](\text{LIVE}(\vec{x}) \rightarrow \Phi)$.

---

**Example**

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

Along every path, it is always true, for each student $x$, that there exists an evolution in which $x$ **persists in the database until either $x$ does not persist, or** she eventually graduates.
Consider two transition systems $\Upsilon_1 = \langle \Delta_1, R, \Sigma_1, s_{01}, db_1, \Rightarrow_1 \rangle$ and $\Upsilon_2 = \langle \Delta_2, R, \Sigma_2, s_{02}, db_2, \Rightarrow_2 \rangle$.

W.r.t. $\mu L_A$, it is now sufficient to preserve the correspondence between elements of $\Delta_1$ and $\Delta_2$ that persist over time.

Is a relation $B \subseteq \Sigma_1 \times H \times \Sigma_2$ such that $\langle s_1, h, s_2 \rangle \in B$ implies that:

1. $h$ is 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 $h|_{\text{ADOM}(db_1(s_1)) \cap \text{ADOM}(db_1(s'_1))}$, such that $\langle s'_1, h', s'_2 \rangle \in B$;
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 $h|_{\text{ADOM}(db_1(s_1)) \cap \text{ADOM}(db_1(s'_1))}$, such that $\langle s'_1, h', s'_2 \rangle \in B$.

We have $\Upsilon_1 \sim \Upsilon_2$ if there exists a partial bijection $h_0$ and a persistence preserving bisimulation $B$ between $\Upsilon_1$ and $\Upsilon_2$ such that $\langle s_{01}, h_0, s_{02} \rangle \in B$.

**Theorem**

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

$$\Upsilon_1 \models \Phi \quad \text{if and only if} \quad \Upsilon_2 \models \Phi.$$
Verification of state-bounded systems

Theorem

Verification of $\mu \mathcal{L}_P$ properties on state-bounded DCDSs with non-deterministic services is decidable.

1. We devise a finite-state abstraction $\Theta_S$ for a state-bounded DCDS $S$.
2. We prove that $\Theta_S \sim \Upsilon_S$, hence they satisfy the same $\mu \mathcal{L}_P$ formulae.

However . . .

Theorem

Checking state-boundedness of DCDSs with non-deterministic services is undecidable.
Ensuring state-boundedness

We have devised a sufficient syntactic condition that guarantees state-boundedness: **generate-recall acyclicity**

- Depends only on action specifications, and not on data.
- Is polynomially checkable.

**Theorem**

Verification of $\mu\mathcal{L}_P$ properties for generate-recall acyclic DCDSs with non-deterministic services is decidable, and can be reduced to model checking of propositional $\mu$-calculus over a finite transition system.
Consider $\mathcal{I}_0 = \{R(a)\}$, process \{true $\mapsto \alpha$\}, and action $\alpha = \left\{ \begin{array}{l} R(x) \leadsto R(x) \\ R(x) \leadsto Q(f(x)) \end{array} \right\}$.

- The resulting process layer is **generate-recall acyclic**.
- Service call $f(a)$ is continuously issued, leading to possibly **generate** infinitely many distinct values, but such values are **not recalled**.
- Since $\mu\mathcal{L}_P$ formulae only focus on persisting values, the possibly infinitely many distinct results obtained by issuing $f(a)$ are irrelevant.
Example

Consider $I_0 = \{R(a)\}$, process $\{\text{true} \xrightarrow{} \alpha\}$, and

$$\alpha = \begin{cases} 
R(x) \xrightarrow{} R(x) \\
R(x) \xrightarrow{} Q(f(x)) \\
Q(x) \xrightarrow{} Q(x) 
\end{cases}$$

The resulting process layer is not generate-recall acyclic.

Service call $f(a)$ is continuously issued, leading to possibly generate infinitely many distinct values, which are recalled in $Q$.

It is not possible to find a faithful finite abstraction, because there exist runs accumulating unboundedly many distinct values inside their states.
Generate-recall acyclicity

Example

Consider $I_0 = \{R(a)\}$, process $\{\text{true} \mapsto \alpha, \text{true} \mapsto \beta\}$, and actions $\alpha = \{R(x) \leadsto R(x), R(x) \leadsto Q(f(x))\}$ and $\beta = \{Q(x) \leadsto Q(x)\}$.

- The resulting process layer is generate-recall acyclic.
- It resembles the previous case, but now effects belong to two distinct actions.
- While $\alpha$ is getting a new value for $f(a)$, $Q$ tuples are lost.
- While $\beta$ is copying $Q$ tuples, no new value is inserted.
Outline

1. Combining static and dynamic aspects
2. Data-Centric Dynamic Systems
3. Semantics of DCDS
4. Verification
5. Run-boundedness
6. State-boundedness
7. Conclusions
### Summary of results on verification of DCDSs

#### Deterministic Services vs. Non-deterministic Services

<table>
<thead>
<tr>
<th></th>
<th>Deterministic Services</th>
<th>Non-Deterministic Services</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>μL_{FO}</strong></td>
<td>U ← U ← U</td>
<td>U ← U ← U</td>
</tr>
<tr>
<td><strong>μL_{A}</strong></td>
<td>D → D</td>
<td>↑ ← ↑</td>
</tr>
<tr>
<td><strong>μL_{P}</strong></td>
<td></td>
<td>U ← U</td>
</tr>
</tbody>
</table>

**D**: Verification is decidable  
**U**: Verification is undecidable

### Main Contribution

A way to mix data and process that is robust:

- Expressive data representation formalism: relational databases;
- Full fledged verification logic: variants of FO μ-calculus;
- Not too restrictive conditions for decidability: weak acyclicity / generate-recall acyclicity.
Ongoing and future work

- Consider various settings with **incomplete information**:
  - the data is directly represented through an ontology;
  - an ontology-layer is built on top of the data layer, and connected to it through mappings (OBDA).
  - Major challenge: propagation of updates done at the ontology-level to the relational level;
  - the system deals with inconsistency with respect to the ontological constraints;

- Generalize syntactic conditions for run-boundedness and state-boundedness.

- Establish relationship to other popular models for representing processes and data, and carry over decidability results.

These topics are currently being explored in the EU FP7 Project ACSI (Artifact Centric Service Interoperation).