Integrated Modeling and Verification of Processes and Data

Verification

Diego Calvanese, Marco Montali

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

15th International Conference on Business Process Management
Barcelona, Spain – 12 September 2017

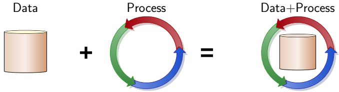
Data-Centric Dynamic Systems (DCDS)

- Abstract model underlying variants of artifact-centric systems.
- Semantically equivalent to the most expressive models for business process systems (e.g., GSM).

- Data Layer: Relational databases / ontologies
  - Data schema, specifying constraints on the allowed states
  - Data instance: state of the DCDS
- Process Layer: key elements are
  - Atomic actions
  - Condition-action-rules for application of actions
  - **Service calls:** communication with external environment, **new data!**

Data-Centric Dynamic Systems (DCDS)

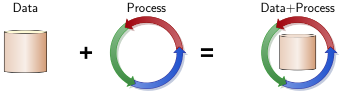
- Abstract model underlying variants of artifact-centric systems.
- Semantically equivalent to the most expressive models for business process systems (e.g., GSM).



- Data Layer: Relational databases / ontologies
  - Data schema, specifying constraints on the allowed states
  - Data instance: state of the DCDS
- Process Layer: key elements are
  - Atomic actions
  - Condition-action-rules for application of actions
  - **Service calls:** communication with external environment, **new data!**

Data-Centric Dynamic Systems (DCDS)

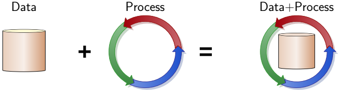
- Abstract model underlying variants of artifact-centric systems.
- Semantically equivalent to the most expressive models for business process systems (e.g., GSM).



- Data Layer: Relational databases / ontologies
  - Data schema, specifying constraints on the allowed states
  - Data instance: state of the DCDS
- Process Layer: key elements are
  - Atomic actions
  - Condition-action-rules for application of actions
  - **Service calls:** communication with external environment, **new data!**

Data-Centric Dynamic Systems (DCDS)

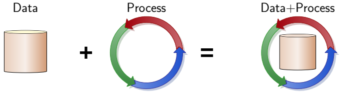
- Abstract model underlying variants of artifact-centric systems.
- Semantically equivalent to the most expressive models for business process systems (e.g., GSM).



- Data Layer: Relational databases / ontologies
  - Data schema, specifying constraints on the allowed states
  - Data instance: state of the DCDS
- Process Layer: key elements are
  - Atomic actions
  - Condition-action-rules for application of actions
  - **Service calls:** communication with external environment, **new data!**

Data-Centric Dynamic Systems (DCDS)

- Abstract model underlying variants of artifact-centric systems.
- Semantically equivalent to the most expressive models for business process systems (e.g., GSM).



- Data Layer: Relational databases / ontologies
  - Data schema, specifying constraints on the allowed states
  - Data instance: state of the DCDS
- Process Layer: key elements are
  - Atomic actions
  - Condition-action-rules for application of actions
  - **Service calls:** communication with external environment, **new data!**

Deterministic vs. non-deterministic services

DCDSs admit two different semantics for service-execution:

Deterministic services semantics

Along a run, when the **same service** is called 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 a run, when the **same service** is called again with the **same arguments**, it may return a **different result** 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

Deterministic vs. non-deterministic services

DCDSs admit two different semantics for service-execution:

Deterministic services semantics

Along a run, when the **same service** is called 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 a run, when the **same service** is called again with the **same arguments**, it may return a **different result** 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

An example: Hotels and price conversion

Data Layer: Info about room prices for hotels and their currency

$$\mathsf{Cur} = \langle \mathit{UserCurrency} \rangle \qquad \mathsf{CH} = \langle \underline{\mathit{Hotel}}, \mathit{Currency} \rangle$$
$$\mathsf{PEntry} = \langle \underline{\mathit{Hotel}}, \mathit{Price}, \underline{\mathit{Date}} \rangle$$

Process Layer/1: User selection of a currency

- Process: $\mathit{true} \longmapsto \mathsf{ChooseCur}()$
- Service call for currency selection: $\textsc{uInputCurr}()$
  - Models user input with **non-deterministic** behavior.
- $\mathsf{ChooseCur}() : \left\{ \begin{array}{r} \mathsf{Cur}(c) \rightsquigarrow \textbf{del}\{\mathsf{Cur}(c)\} \\ \mathit{true} \rightsquigarrow \textbf{add}\{\mathsf{Cur}(\textsc{uInputCurr}())\} \end{array} \right\}$

An example: Hotels and price conversion

Data Layer: Info about room prices for hotels and their currency

$$\mathsf{Cur} = \langle \mathit{UserCurrency} \rangle \qquad \mathsf{CH} = \langle \underline{\mathit{Hotel}}, \mathit{Currency} \rangle$$
$$\mathsf{PEntry} = \langle \underline{\mathit{Hotel}}, \mathit{Price}, \underline{\mathit{Date}} \rangle$$

Process Layer/2: Price conversion for a hotel

- Process: $\mathsf{Cur}(c) \land \mathsf{CH}(h, c_h) \land c_h \neq c \longmapsto \mathsf{ApplyConv}(h, c)$
- Service call for currency selection: $\textsc{conv}(\mathit{price}, \mathit{from}, \mathit{to}, \mathit{date})$
  - Models historical conversion with deterministic behavior.
- $\mathsf{ApplyConv}(h, c)$:

$$\left\{ \begin{array}{r} \mathsf{PEntry}(h, p, d) \rightsquigarrow \mathbf{del}\{\mathsf{PEntry}(h, p, d)\} \\ \mathsf{PEntry}(h, p, d) \land \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \\ \mathsf{CH}(h, c_{old}) \rightsquigarrow \mathbf{add}\{\mathsf{PEntry}(h, \textsc{conv}(p, c_{old}, c, d), d)\} \\ \mathsf{CH}(h, c_{old}) \rightsquigarrow \mathbf{del}\{\mathsf{CH}(h, c_{old})\},\ \mathbf{add}\{\mathsf{CH}(h, c)\} \end{array} \right\}$$

Run of the system

HC	
h_1	eur
h_2	eur

PEntry		
h_1	95	apr-25
h_1	80	sep-18
h_2	80	sep-18

Run of the system

HC	
h_1	eur
h_2	eur

PEntry		
h_1	95	apr-25
h_1	80	sep-18
h_2	80	sep-18

ChooseCur(): $\textsc{uInputCurr}() = ?$

Run of the system

HC	
h_1	eur
h_2	eur

PEntry		
h_1	95	apr-25
h_1	80	sep-18
h_2	80	sep-18

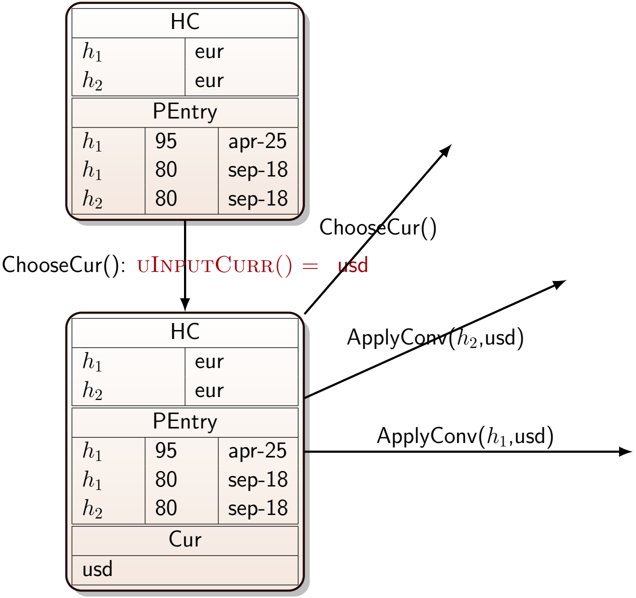
ChooseCur(): $\textsc{uInputCurr}() =$ usd

HC	
h_1	eur
h_2	eur

PEntry		
h_1	95	apr-25
h_1	80	sep-18
h_2	80	sep-18

Cur
usd

Run of the system



Run of the system

HC	
h_1	eur
h_2	eur

PEntry		
h_1	95	apr-25
h_1	80	sep-18
h_2	80	sep-18

ChooseCur(): $\textsc{uInputCurr}() =$ usd

HC	
h_1	eur
h_2	eur

PEntry		
h_1	95	apr-25
h_1	80	sep-18
h_2	80	sep-18

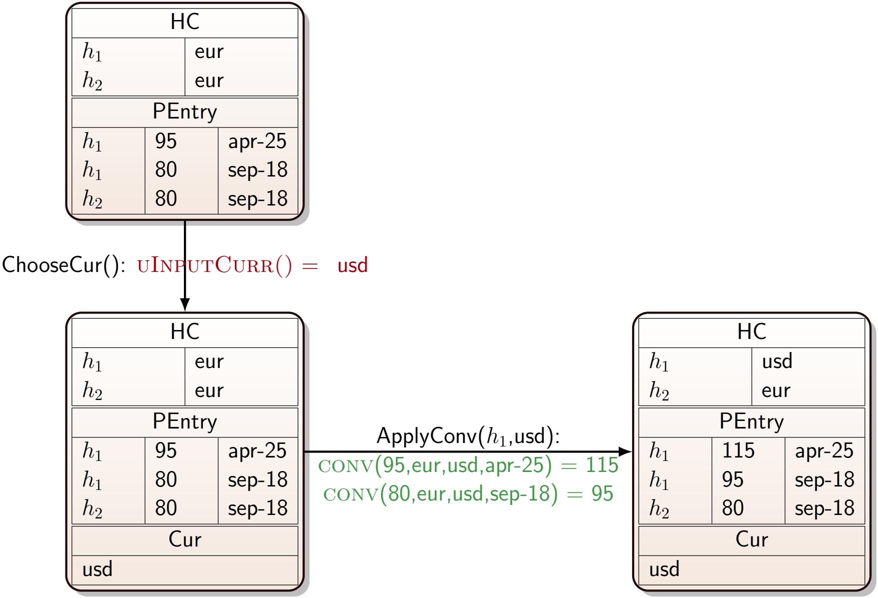
Cur
usd

ApplyConv(h_1,usd):

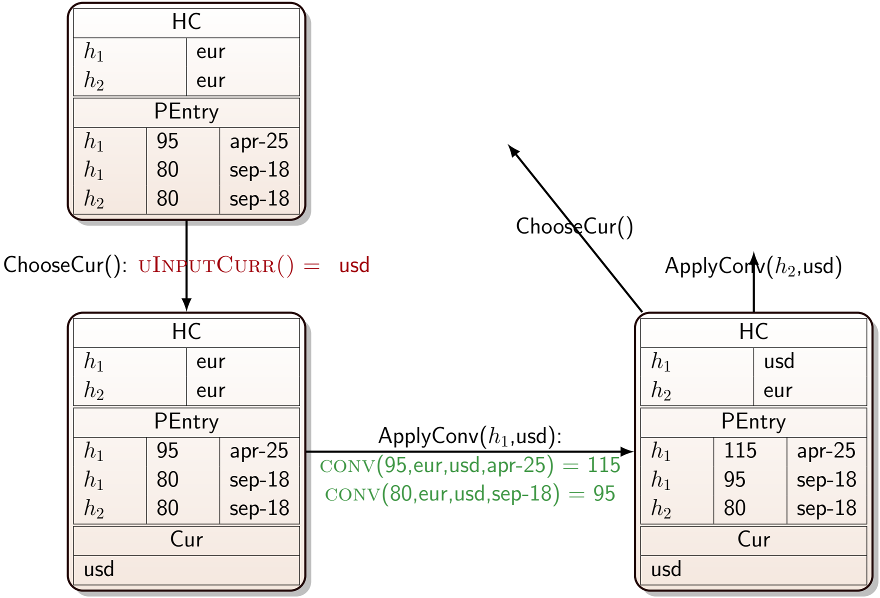
$\textsc{conv}$(95,eur,usd,apr-25) $=$?

$\textsc{conv}$(80,eur,usd,sep-18) $=$?

Run of the system



Run of the system



Run of the system

HC	
h_1	eur
h_2	eur

PEntry		
h_1	95	apr-25
h_1	80	sep-18
h_2	80	sep-18

ChooseCur(): $\textsc{uInputCurr}() =$ usd

HC	
h_1	eur
h_2	eur

PEntry		
h_1	95	apr-25
h_1	80	sep-18
h_2	80	sep-18

Cur
usd

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

HC	
h_1	usd
h_2	eur

PEntry		
h_1	115	apr-25
h_1	95	sep-18
h_2	80	sep-18

Cur
usd

ApplyConv(h_2,usd)
$\textsc{conv}$(80,eur,usd,sep-18) $=$ **95**

HC	
h_1	usd
h_2	usd

PEntry		
h_1	115	apr-25
h_1	95	sep-18
h_2	95	sep-18

Cur
usd

Execution semantics of dynamic systems

Typically given in the form of a **transition system**.

(Propositional) transition system

Given a set Σ of state propositions, a (propositional) transition system is a tuple $\langle S, s_0, prop, \Rightarrow \rangle$, where:

- S is a **finite** set of states;
- $s_0 \in S$ is the initial state;
- $prop : S \to 2^{\Sigma}$ is an assignment, mapping each state in S to the set of propositions from Σ holding in that state;
- $\Rightarrow \subseteq S \times S$ is the transition relation between states.

Usually, the transitions are labeled with corresponding **actions**.

Impact of data on verification

The presence of data complicates verification significantly:

- **States** must be **modeled relationally** rather than propositionally.
- The resulting transition system is typically **infinite state**.
- Query languages for analysis need to combine two dimensions:
  - a temporal dimension to query the process execution flow, and
  - a first-order dimension to query the data present in the relational structures.

$\leadsto$ We need **first-order variants of temporal logics**.

What if the system evolves a database?

We get a transition system in which each state is a relational database.

We can assume to have an infinite domain Δ of data items (also called values).

Relational transition system (RTS)

Is a tuple $\langle \Delta, \mathcal{R}, S, s_0, db, \Rightarrow \rangle$, where:

- $\mathcal{R}$ is a database schema;
- S is a **possibly infinite** set of states;
- $s_0 \in S$ is the initial state;
- db is a function associating to each state s in S a database instance $db(s)$ over $\mathcal{R}$ and Δ;
- $\Rightarrow \subseteq S \times S$ is the transition relation between states.

Execution semantics of a DCDS

Is determined by the **relational transition system** that accounts for all possible runs of the DCDS:

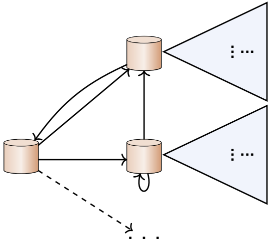
- States **are** database instances (i.e., db is the identity function).
- Transitions: correspond to *legal* applications of an action with parameter instantiation + service call evaluations.
  - Action with param. instantiation: executable according to the process rules.
  - Satisfaction of constraints ensured by each DB instance.

We obtain a possibly **infinite-state** (relational) transition system $\Upsilon_{\mathcal{X}}$, intuitively constructed as follows:

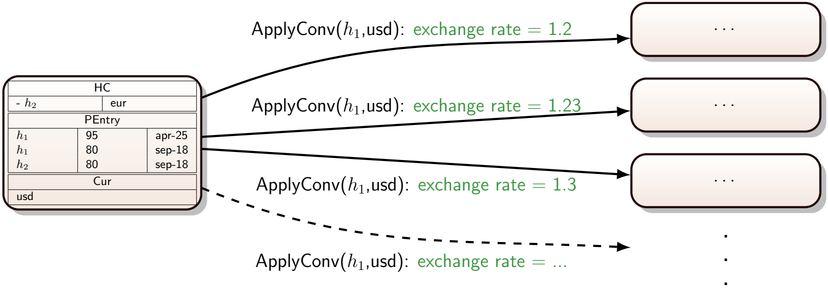
- start from the initial DB;
- apply transitions in all possible ways;
- continue (ad infinitum) on the newly obtained states.

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).

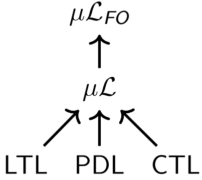


First-order μ-calculi for DCDSs

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

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

- Extends the propositional μ-calculus $\mu\mathcal{L}$ with first-order quantification.
- The first-order quantifiers range over all objects in the transition system (and not only over those in the current state or in the current run).



We also adopt the standard abbreviations, including:

- $[-]\Phi$ for $\neg\langle-\rangle\neg\Phi$
- $\nu Z.\Phi$ for $\neg\mu Z.\Phi_{[Z/\neg Z]}$

Example

$\forall x.\mathsf{Student}(x) \rightarrow \mu Z.((\exists y.\mathsf{Graduate}(x,y)) \vee \langle-\rangle Z)$

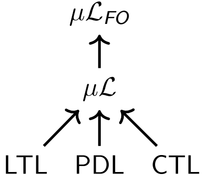
For each student x (in the current state), there exists an evolution that eventually leads to the graduation of x (with some final mark y).

First-order μ-calculi for DCDSs

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

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

- Extends the propositional μ-calculus $\mu\mathcal{L}$ with first-order quantification.
- The first-order quantifiers range over all objects in the transition system (and not only over those in the current state or in the current run).



We also adopt the standard abbreviations, including:

- $[-]\Phi$ for $\neg\langle - \rangle\neg\Phi$
- $\nu Z.\Phi$ for $\neg\mu Z.\Phi_{[Z/\neg Z]}$

Example

$\forall x.\mathsf{Student}(x) \rightarrow \mu Z.((\exists y.\mathsf{Graduate}(x, y)) \vee \langle - \rangle Z)$

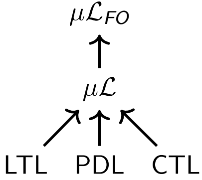
For each student x (in the current state), there exists an evolution that eventually leads to the graduation of x (with some final mark y).

First-order μ-calculi for DCDSs

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

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

- Extends the propositional μ-calculus $\mu\mathcal{L}$ with first-order quantification.
- The first-order quantifiers range over all objects in the transition system (and not only over those in the current state or in the current run).



We also adopt the standard abbreviations, including:

- $[-]\Phi$ for $\neg\langle - \rangle\neg\Phi$
- $\nu Z.\Phi$ for $\neg\mu Z.\Phi_{[Z/\neg Z]}$

Example

$\forall x.\mathsf{Student}(x) \rightarrow \mu Z.((\exists y.\mathsf{Graduate}(x, y)) \vee \langle - \rangle Z)$

For each student x (in the current state), there exists an evolution that eventually leads to the graduation of x (with some final mark y).

Model checking $\mu\mathcal{L}_{FO}$

Model checking a RTS

Input:

- a RTS $\Upsilon = \langle \Delta, \mathcal{R}, S, s_0, db, \Rightarrow \rangle$
- a $\mu\mathcal{L}_{FO}$ formula Φ that is closed (i.e., without free variables)

Output: yes, iff Φ holds in the initial state s_0 of Υ

In this case, we write $\Upsilon \models \Phi$.

Model checking a DCDS

Input:

- a DCDS $\mathcal{X}$ (generating a RTS $\Upsilon_{\mathcal{X}}$)
- a closed $\mu\mathcal{L}_{FO}$ formula Φ

Output: yes, iff $\Upsilon_{\mathcal{X}} \models \Phi$.

In this case, we write $\mathcal{X} \models \Phi$.

Model checking $\mu\mathcal{L}_{FO}$

Model checking a RTS

Input:

- a RTS $\Upsilon = \langle \Delta, \mathcal{R}, S, s_0, db, \Rightarrow \rangle$
- a $\mu\mathcal{L}_{FO}$ formula Φ that is closed (i.e., without free variables)

Output: yes, iff Φ holds in the initial state s_0 of Υ

In this case, we write $\Upsilon \models \Phi$.

Model checking a DCDS

Input:

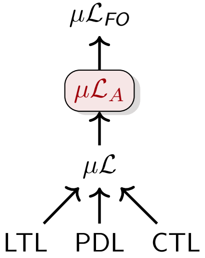
- a DCDS $\mathcal{X}$ (generating a RTS $\Upsilon_{\mathcal{X}}$)
- a closed $\mu\mathcal{L}_{FO}$ formula Φ

Output: yes, iff $\Upsilon_{\mathcal{X}} \models \Phi$.

In this case, we write $\mathcal{X} \models \Phi$.

History-preserving μ-calculus ($\mu\mathcal{L}_A$)

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



$$\exists x.\Phi \quad \leadsto \quad \exists x.\textsc{live}(x) \land \Phi$$
$$\forall x.\Phi \quad \leadsto \quad \forall x.\textsc{live}(x) \rightarrow \Phi$$

where $\textsc{live}(x)$ states that x is present in the current active domain (easily expressible in FO).

Note: $\mu\mathcal{L}_A$ is a syntactic restriction of $\mu\mathcal{L}_{FO}$.

Example

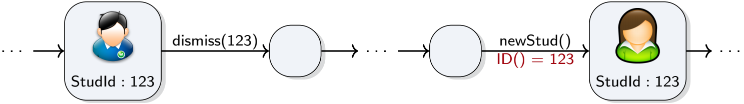
$$\nu W.(\forall x.\textsc{live}(x) \land \mathsf{Student}(x) \rightarrow$$
$$\mu Z.(\exists y.\textsc{live}(y) \land \mathsf{Graduate}(x,y) \lor \langle - \rangle Z) \land [-]W)$$

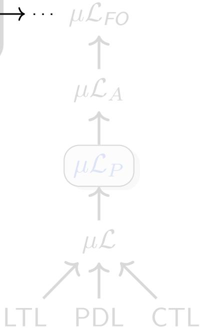
Along every path, it is always true, for each student x, that there exists an evolution eventually leading to a graduation of the student (with some final mark y).

Note: No guarantee that all such students graduate within the same run.

Persistence-preserving μ-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).





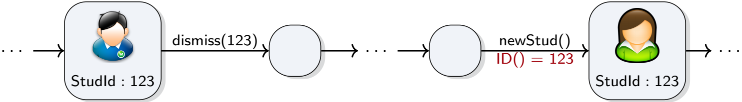
$\mu\mathcal{L}_P$ restricts $\mu\mathcal{L}_A$ to **quantification over persisting objects only**, i.e., objects that *continue* to be LIVE.

$$\exists x.\Phi \quad \rightsquigarrow \quad \exists x.\text{LIVE}(x) \wedge \Phi$$
$$\forall x.\Phi \quad \rightsquigarrow \quad \forall x.\text{LIVE}(x) \rightarrow \Phi$$
$$\langle - \rangle \Phi(\vec{x}) \quad \rightsquigarrow \quad \begin{cases} \text{LIVE}(\vec{x}) \wedge \langle - \rangle \Phi(\vec{x}) & \text{(strong persistence)} \\ \text{LIVE}(\vec{x}) \rightarrow \langle - \rangle \Phi(\vec{x}) & \text{(weak persistence)} \end{cases}$$
$$[-] \Phi(\vec{x}) \quad \rightsquigarrow \quad \begin{cases} \text{LIVE}(\vec{x}) \wedge [-] \Phi(\vec{x}) & \text{(strong persistence)} \\ \text{LIVE}(\vec{x}) \rightarrow [-] \Phi(\vec{x}) & \text{(weak persistence)} \end{cases}$$

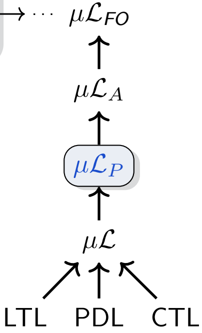
Note: $\mu\mathcal{L}_P$ is a syntactic restriction of $\mu\mathcal{L}_A$.

Persistence-preserving μ-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.



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

$$\forall x.\Phi \quad \rightsquigarrow \quad \forall x.\text{LIVE}(x) \rightarrow \Phi$$

$$\langle - \rangle \Phi(\vec{x}) \quad \rightsquigarrow \quad \begin{cases} \text{LIVE}(\vec{x}) \wedge \langle - \rangle \Phi(\vec{x}) & \text{(strong persistence)} \\ \text{LIVE}(\vec{x}) \rightarrow \langle - \rangle \Phi(\vec{x}) & \text{(weak persistence)} \end{cases}$$

$$[-]\Phi(\vec{x}) \quad \rightsquigarrow \quad \begin{cases} \text{LIVE}(\vec{x}) \wedge [-]\Phi(\vec{x}) & \text{(strong persistence)} \\ \text{LIVE}(\vec{x}) \rightarrow [-]\Phi(\vec{x}) & \text{(weak persistence)} \end{cases}$$

Note: $\mu\mathcal{L}_P$ is a syntactic restriction of $\mu\mathcal{L}_A$.

Strong vs. weak persistence

Strong persistence: property falsified by an object that disappears

$$\nu W.(\forall x.\text{LIVE}(x) \wedge \mathsf{Student}(x) \rightarrow$$
$$\mu Z.(\exists y.\text{LIVE}(y) \wedge \mathsf{Graduate}(x,y) \vee (\text{LIVE}(x) \wedge \langle - \rangle Z)) \wedge [-]W)$$

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.

Weak persistence: property verified by an object that disappears

$$\nu W.(\forall x.\text{LIVE}(x) \wedge \mathsf{Student}(x) \rightarrow$$
$$\mu Z.(\exists y.\text{LIVE}(y) \wedge \mathsf{Graduate}(x,y) \vee (\text{LIVE}(x) \rightarrow \langle - \rangle Z)) \wedge [-]W)$$

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.

Strong vs. weak persistence

Strong persistence: property falsified by an object that disappears

$$\nu W.(\forall x.\textsc{live}(x) \wedge \mathsf{Student}(x) \rightarrow$$
$$\mu Z.(\exists y.\textsc{live}(y) \wedge \mathsf{Graduate}(x, y) \vee (\textsc{live}(x) \wedge \langle - \rangle Z)) \wedge [-]W)$$

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.

Weak persistence: property verified by an object that disappears

$$\nu W.(\forall x.\textsc{live}(x) \wedge \mathsf{Student}(x) \rightarrow$$
$$\mu Z.(\exists y.\textsc{live}(y) \wedge \mathsf{Graduate}(x, y) \vee (\textsc{live}(x) \rightarrow \langle - \rangle Z)) \wedge [-]W)$$

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.

First-order linear temporal logics for DCDSs

LTL-FO extends propositional LTL with the possibility of querying the system states using first-order formulas with quantification across:

$$\Phi ::= \varphi \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.\Phi \mid \mathbf{X}\,\Phi \mid \Phi_1\,\mathbf{U}\,\Phi_2$$

We also adopt the standard abbreviations, including:

- $\mathbf{F}\,\Phi$ for $\mathsf{true}\,\mathbf{U}\,\Phi$ (Φ holds in the future)
- $\mathbf{G}\,\Phi$ for $\neg\,\mathbf{F}\,\neg\Phi$ (Φ holds globally)

Example

$\forall x.\mathsf{Student}(x) \rightarrow \mathbf{F}\,\exists y.\mathsf{Graduate}(x,y)$

For each student x (in the current state), x will graduate sometimes in the future (with some final mark y).

Note: all encountered students graduate within the same run.

First-order linear temporal logics for DCDSs

LTL-FO extends propositional LTL with the possibility of querying the system states using first-order formulas with quantification across:

$$\Phi ::= \varphi \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.\Phi \mid \mathbf{X}\,\Phi \mid \Phi_1\,\mathbf{U}\,\Phi_2$$

We also adopt the standard abbreviations, including:

- $\mathbf{F}\,\Phi$ for $\text{true}\,\mathbf{U}\,\Phi$ (Φ holds in the future)
- $\mathbf{G}\,\Phi$ for $\neg\,\mathbf{F}\,\neg\Phi$ (Φ holds globally)

Example

$\forall x.\text{Student}(x) \rightarrow \mathbf{F}\,\exists y.\text{Graduate}(x,y)$

For each student x (in the current state), x will graduate sometimes in the future (with some final mark y).

Note: all encountered students graduate within the same run.

Model checking LTL-FO

LTL model checking an RTS

Input:

- an RTS $\Upsilon = \langle \Delta, \mathcal{R}, S, s_0, db, \Rightarrow \rangle$
- a closed LTL-FO formula Φ

Output: yes, iff for every run τ over Υ, Φ holds in the initial state of τ.

In this case, we write $\Upsilon \models_{\text{LTL}} \Phi$.

LTL Model checking a DCDS

Input:

- a DCDS $\mathcal{X}$ (generating a RTS $\Upsilon_{\mathcal{X}}$)
- a closed LTL-FO formula Φ

Output: yes, iff $\Upsilon_{\mathcal{X}} \models_{\text{LTL}} \Phi$.

In this case, we write $\mathcal{X} \models_{\text{LTL}} \Phi$.

Model checking LTL-FO

LTL model checking an RTS

Input:

- an RTS $\Upsilon = \langle \Delta, \mathcal{R}, S, s_0, db, \Rightarrow \rangle$
- a closed LTL-FO formula Φ

Output: yes, iff for every run τ over Υ, Φ holds in the initial state of τ.

In this case, we write $\Upsilon \models_{\text{LTL}} \Phi$.

LTL Model checking a DCDS

Input:

- a DCDS $\mathcal{X}$ (generating a RTS $\Upsilon_{\mathcal{X}}$)
- a closed LTL-FO formula Φ

Output: yes, iff $\Upsilon_{\mathcal{X}} \models_{\text{LTL}} \Phi$.

In this case, we write $\mathcal{X} \models_{\text{LTL}} \Phi$.

First-order LTL with restricted quantification

History-preserving quantification: LTL-FO$_A$

FO quantification ranges over current active domain only:

$$\begin{aligned} \exists x.\Phi &\rightsquigarrow \exists x.\textsc{live}(x) \land \Phi \\ \forall x.\Phi &\rightsquigarrow \forall x.\textsc{live}(x) \rightarrow \Phi \end{aligned}$$

Example: $\forall x.\textsc{live}(x) \land \mathsf{Customer}(x) \rightarrow \mathbf{F}\,\mathsf{Gold}(x)$

Persistence-preserving quantification: LTL-FO$_P$

FO quantification ranges over persisting individuals only.

$$\begin{aligned} \exists x.\Phi &\rightsquigarrow \exists x.\textsc{live}(x) \land \Phi \\ \forall x.\Phi &\rightsquigarrow \forall x.\textsc{live}(x) \rightarrow \Phi \\ \mathbf{X}\,\Phi(\vec{x}) &\rightsquigarrow \begin{cases} \textsc{live}(\vec{x}) \land \mathbf{X}\,\Phi(\vec{x}) & \text{(strong persistence)} \\ \textsc{live}(\vec{x}) \rightarrow \mathbf{X}\,\Phi(\vec{x}) & \text{(weak persistence)} \end{cases} \\ \Phi_1\,\mathbf{U}\,\Phi_2(\vec{x}) &\rightsquigarrow \begin{cases} (\textsc{live}(\vec{x}) \land \Phi_1)\,\mathbf{U}\,\Phi_2(\vec{x}) & \text{(s.p.)} \\ (\textsc{live}(\vec{x}) \land \Phi_1)\,\mathbf{U}(\textsc{live}(\vec{x}) \rightarrow \Phi_2(\vec{x})) & \text{(w.p.)} \end{cases} \end{aligned}$$

Example: $\forall x.(\textsc{live}(x) \land \mathsf{Gold}(x)) \rightarrow \neg(\textsc{live}(x)\,\mathbf{U}\,\neg\mathsf{Gold}(x))$

$$\text{LTL} \longrightarrow \text{LTL-FO}$$

First-order LTL with restricted quantification

History-preserving quantification: LTL-FO$_A$

FO quantification ranges over current active domain only:

$$\exists x.\Phi \quad \rightsquigarrow \quad \exists x.\text{LIVE}(x) \wedge \Phi$$
$$\forall x.\Phi \quad \rightsquigarrow \quad \forall x.\text{LIVE}(x) \rightarrow \Phi$$

Example: $\forall x.\text{LIVE}(x) \wedge \mathsf{Customer}(x) \rightarrow \mathbf{F}\,\mathsf{Gold}(x)$

Persistence-preserving quantification: LTL-FO$_P$

FO quantification ranges over persisting individuals only.

$$\exists x.\Phi \quad \rightsquigarrow \quad \exists x.\text{LIVE}(x) \wedge \Phi$$
$$\forall x.\Phi \quad \rightsquigarrow \quad \forall x.\text{LIVE}(x) \rightarrow \Phi$$

$$\mathbf{X}\,\Phi(\vec{x}) \quad \rightsquigarrow \quad \begin{cases} \text{LIVE}(\vec{x}) \wedge \mathbf{X}\,\Phi(\vec{x}) & \text{(strong persistence)} \\ \text{LIVE}(\vec{x}) \rightarrow \mathbf{X}\,\Phi(\vec{x}) & \text{(weak persistence)} \end{cases}$$

$$\Phi_1\,\mathbf{U}\,\Phi_2(\vec{x}) \quad \rightsquigarrow \quad \begin{cases} (\text{LIVE}(\vec{x}) \wedge \Phi_1)\,\mathbf{U}\,\Phi_2(\vec{x}) & \text{(s.p.)} \\ (\text{LIVE}(\vec{x}) \wedge \Phi_1)\,\mathbf{U}(\text{LIVE}(\vec{x}) \rightarrow \Phi_2(\vec{x})) & \text{(w.p.)} \end{cases}$$

Example: $\forall x.(\text{LIVE}(x) \wedge \mathsf{Gold}(x)) \rightarrow \neg(\text{LIVE}(x)\,\mathbf{U}\,\neg\mathsf{Gold}(x))$

LTL-FO

$\uparrow$

LTL-FO$_A$

$\uparrow$

LTL

First-order LTL with restricted quantification

History-preserving quantification: LTL-FO$_A$

FO quantification ranges over current active domain only:

$$\exists x.\Phi \quad \rightsquigarrow \quad \exists x.\textsc{live}(x) \wedge \Phi$$
$$\forall x.\Phi \quad \rightsquigarrow \quad \forall x.\textsc{live}(x) \rightarrow \Phi$$

Example: $\forall x.\textsc{live}(x) \wedge \mathsf{Customer}(x) \rightarrow \mathbf{F}\, \mathsf{Gold}(x)$

Persistence-preserving quantification: LTL-FO$_P$

FO quantification ranges over persisting individuals only.

$$\exists x.\Phi \quad \rightsquigarrow \quad \exists x.\textsc{live}(x) \wedge \Phi$$
$$\forall x.\Phi \quad \rightsquigarrow \quad \forall x.\textsc{live}(x) \rightarrow \Phi$$
$$\mathbf{X}\,\Phi(\vec{x}) \quad \rightsquigarrow \quad \begin{cases} \textsc{live}(\vec{x}) \wedge \mathbf{X}\,\Phi(\vec{x}) & \text{(strong persistence)} \\ \textsc{live}(\vec{x}) \rightarrow \mathbf{X}\,\Phi(\vec{x}) & \text{(weak persistence)} \end{cases}$$
$$\Phi_1\,\mathbf{U}\,\Phi_2(\vec{x}) \quad \rightsquigarrow \quad \begin{cases} (\textsc{live}(\vec{x}) \wedge \Phi_1)\,\mathbf{U}\,\Phi_2(\vec{x}) & \text{(s.p.)} \\ (\textsc{live}(\vec{x}) \wedge \Phi_1)\,\mathbf{U}(\textsc{live}(\vec{x}) \rightarrow \Phi_2(\vec{x})) & \text{(w.p.)} \end{cases}$$

Example: $\forall x.(\textsc{live}(x) \wedge \mathsf{Gold}(x)) \rightarrow \neg(\textsc{live}(x)\,\mathbf{U}\,\neg\mathsf{Gold}(x))$

$$\text{LTL} \rightarrow \text{LTL-FO}_P \rightarrow \text{LTL-FO}_A \rightarrow \text{LTL-FO}$$

Delineating the boundaries of verifiability

Understand the boundaries of verifiability for DCDSs:

- Considering propositional **reachability** as the bottom line, then moving towards model checking branching and linear time FO temporal logics.
- Striving for **robust** conditions that lend themselves to be enforced in practice.
- Aiming at reducing the problem to **conventional model checking**.

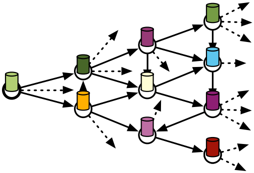
Our goal

DCDS

(Un)desired property

Our goal

DCDS ↓ **Infinite-state RTS**



$\models \Phi$

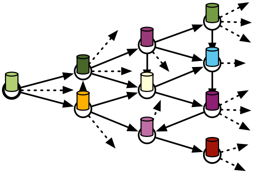
(Un)desired property ↓ **First-order temporal formula**

Our goal

DCDS

(Un)desired property

Infinite-state RTS

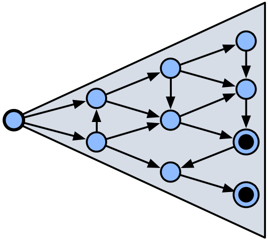


$\models \Phi$

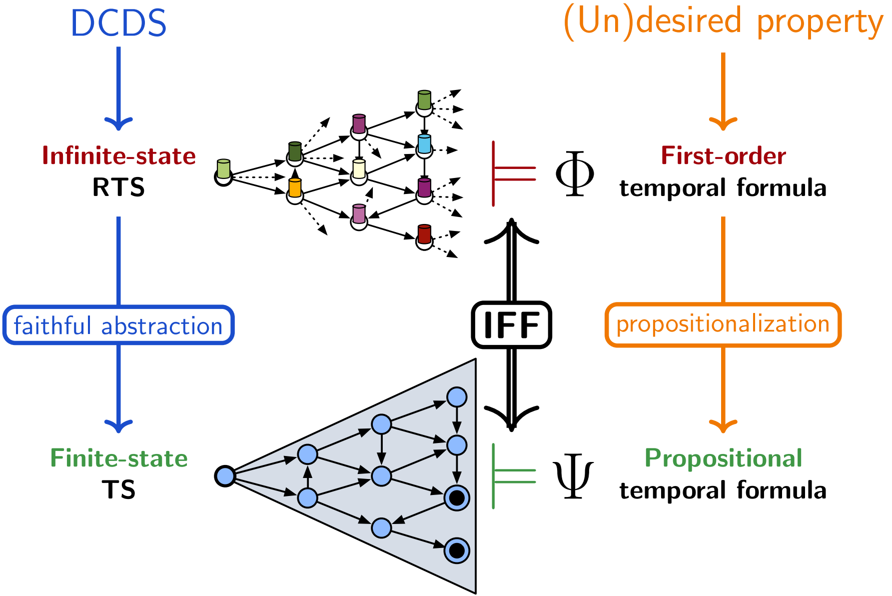
First-order temporal formula

faithful abstraction

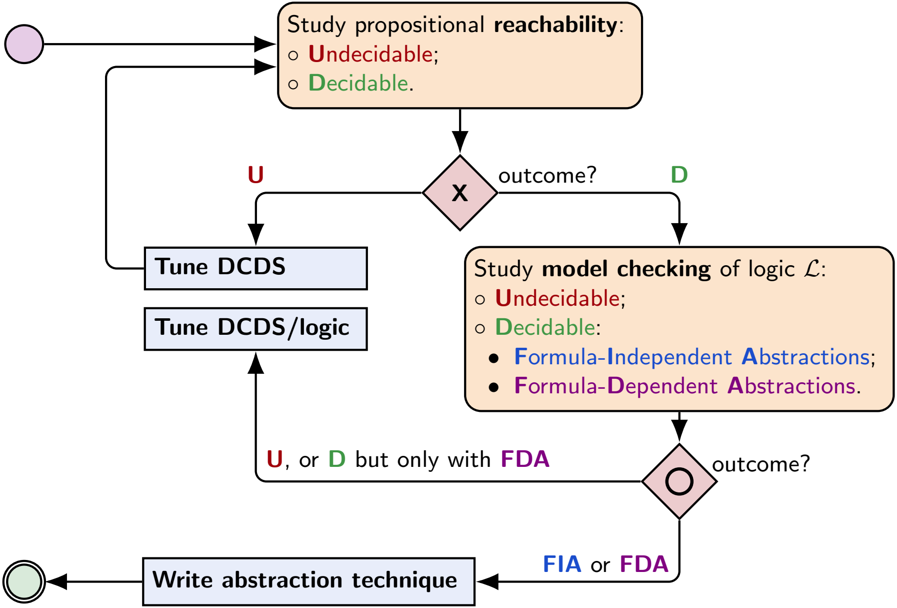
Finite-state TS



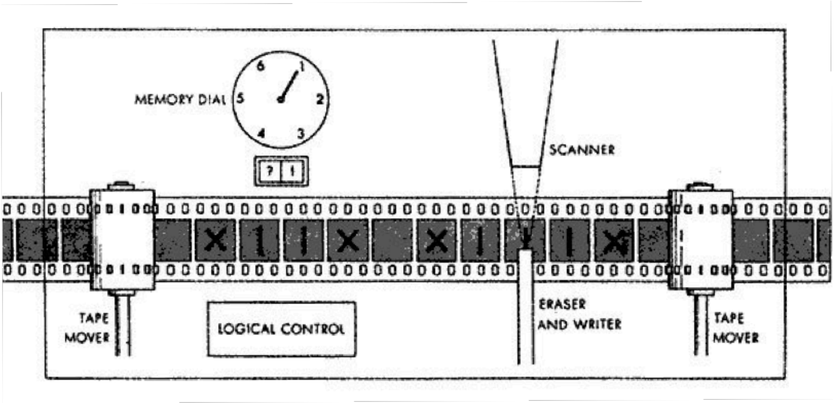
Our goal


DCDS
(Un)desired property
Infinite-state RTS
$\models \Phi$
First-order temporal formula
faithful abstraction
IFF
propositionalization
Finite-state TS
$\models \Psi$
Propositional temporal formula

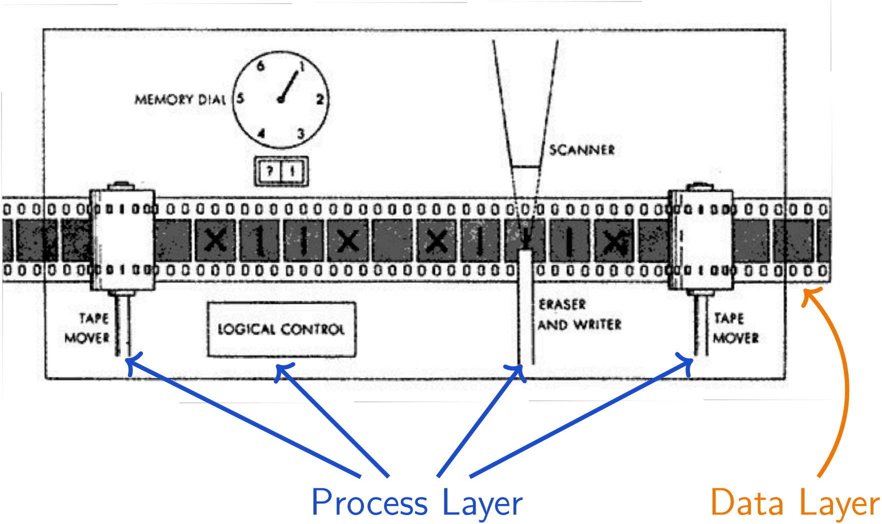
The good


Study propositional **reachability**:
○ Undecidable;
○ Decidable.
outcome?
U
D
X
Tune DCDS
Tune DCDS/logic
Study **model checking** of logic $\mathcal{L}$:
○ Undecidable;
○ Decidable:
• Formula-Independent Abstractions;
• Formula-Dependent Abstractions.
U, or D but only with FDA
outcome?
FIA or FDA
Write abstraction technique

The bad

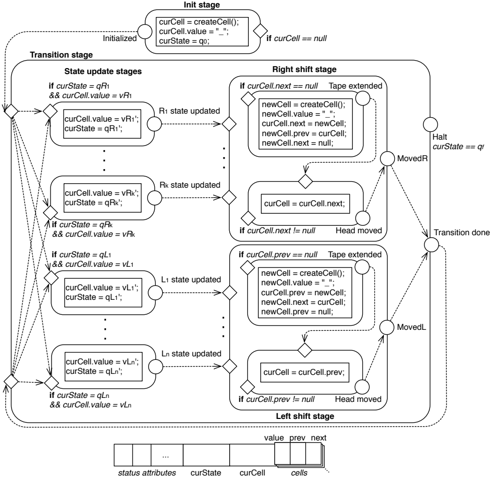

MEMORY DIAL
6
1
2
3
4
5
? 1
SCANNER
TAPE
MOVER
LOGICAL CONTROL
ERASER
AND WRITER
TAPE
MOVER

The bad


MEMORY DIAL
SCANNER
TAPE MOVER
LOGICAL CONTROL
ERASER AND WRITER
TAPE MOVER
Process Layer
Data Layer

The bad

A Turing Machine in GSM

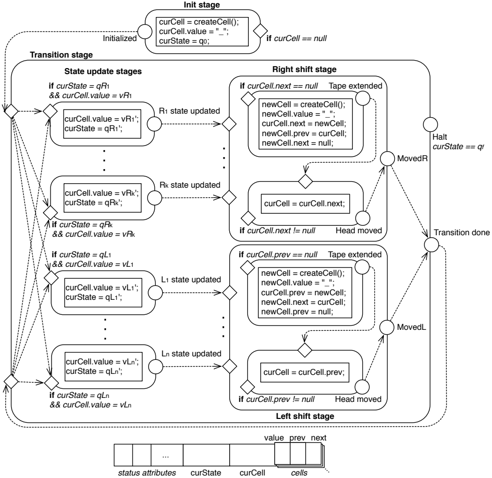


Question

Do we need all such complications to encode Turing-powerful computations?

The bad

A Turing Machine in GSM



Question

Do we need all such complications to encode Turing-powerful computations?

The ugly

To encode Turing-powerful computations, we just need...

- unary relations and queries with **negation**;
- a single binary relation and no negation.

Negation and binary relations are essential features!

Theorem

Verification of **propositional reachability** over DCDSs employing only unary relations, is **undecidable**.

Theorem

Verification of **propositional reachability** over DCDSs employing unary relations, a single binary relation, and only positive queries, is **undecidable**.

The ugly

To encode Turing-powerful computations, we just need...

- unary relations and queries with **negation**;
- a single binary relation and no negation.

Negation and binary relations are essential features!

Theorem

Verification of **propositional reachability** over DCDSs employing only unary relations, is **undecidable**.

Theorem

Verification of **propositional reachability** over DCDSs employing unary relations, a single binary relation, and only positive queries, is **undecidable**.

State boundedness

Main reason for undecidability

The DCDS database may accumulate unbounded information.

Idea: we **control** the way the process layer can use the data layer.

A DCDS $\mathcal{X}$ is state-bounded

if there exists a fixed number b such that the number of values used **in each single state** of $\mathcal{X}$, is **bounded by** b.

If we know b, we say that the DCDS is b-**bounded**.

Note:

- Even a 1-bounded DCDS may still induce an infinite RTS.
- However, the unboundedly many encountered values cannot be accumulated in a single DB.
- State-boundedness is a semantic condition.

State boundedness

Main reason for undecidability

The DCDS database may accumulate unbounded information.

Idea: we **control** the way the process layer can use the data layer.

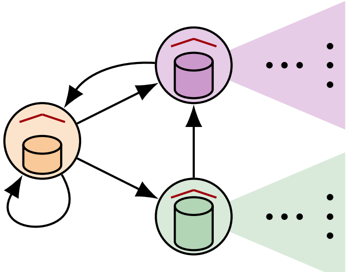
A DCDS $\mathcal{X}$ is state-bounded

if there exists a fixed number b such that the number of values used **in each single state** of $\mathcal{X}$, is **bounded by** b.

If we know b, we say that the DCDS is b-**bounded**.

Note:

- Even a 1-bounded DCDS may still induce an infinite RTS.
- However, the unboundedly many encountered values cannot be accumulated in a single DB.
- State-boundedness is a semantic condition.



State-boundedness to the rescue

Theorem

Reachability over state-bounded DCDS is **decidable**.

Proof.

State-boundedness combines well with two key formal properties of DCDSs and the RTSs they induce. □

Two key properties of DCDSs

DCDS are ...

Markovian

Next state only depends on the current state and the input.

Based on **generic queries**

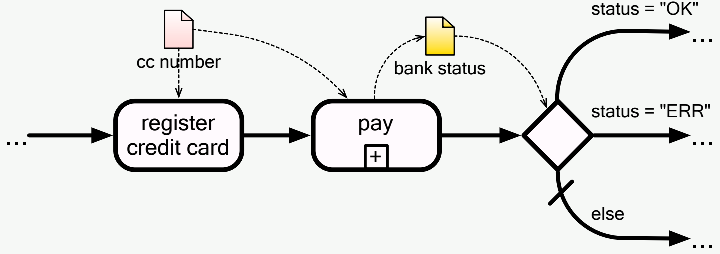
FO/SQL (as virtually all query languages) does not distinguish structures that are identical modulo uniform renaming of data objects.

- Consider two isomorphic databases D_1 and D_2.
- Let h be a bijection between the active domains of D_1 and D_2, witnessing their isomorphisms (i.e., preserving relations).
- For every query Q, by applying h on the answers obtained by issuing Q over D_1, we exactly get the answers obtained by issuing Q over D_2.

These two properties, together, lead to a crucial **genericity property** of the dynamics induced by DCDSs.

Genericity, intuitively

Travel payment



For analyzing the system (considering all possible executions):

- The actual credit card number does not matter.
- What matters is the outcome of the payment.

The process behavior:

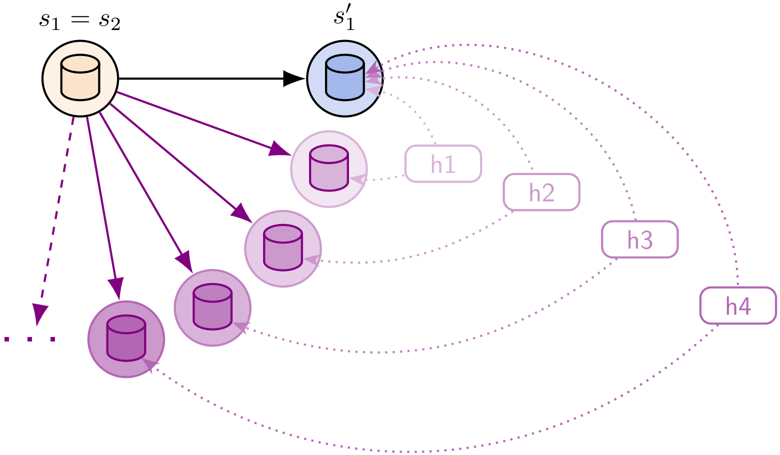
- Distinguishes the bank status.
- Does not really “see” the actual cc number
  $\leadsto$ only **how it relates** to the other objects!

Genericity, graphically

If...

Genericity, graphically

If... Then



Model checking state-bounded DCDSs: negative results

Theorem

There exists a 1-bounded DCDS that **does not admit any formula-independent, finite-state abstraction** preserving exactly the same $\mu\mathcal{L}_A$ (and, hence, $\mu\mathcal{L}_{FO}$) properties.

N.B.: this does not imply undecidability!

Theorem

There exists a 1-bounded DCDS over which verifying LTL-FO$_A$ properties is **undecidable**.

Reason for this negative results:

Unrestricted interplay between temporal modalities and FO quantification across states.

Model checking state-bounded DCDSs: negative results

Theorem

There exists a 1-bounded DCDS that **does not admit any formula-independent, finite-state abstraction** preserving exactly the same $\mu\mathcal{L}_A$ (and, hence, $\mu\mathcal{L}_{FO}$) properties.

N.B.: this does not imply undecidability!

Theorem

There exists a 1-bounded DCDS over which verifying LTL-FO$_A$ properties is **undecidable**.

Reason for this negative results:

Unrestricted interplay between temporal modalities and FO quantification across states.

Model checking state-bounded DCDSs: negative results

Theorem

There exists a 1-bounded DCDS that **does not admit any formula-independent, finite-state abstraction** preserving exactly the same $\mu\mathcal{L}_A$ (and, hence, $\mu\mathcal{L}_{FO}$) properties.

N.B.: this does not imply undecidability!

Theorem

There exists a 1-bounded DCDS over which verifying LTL-FO$_A$ properties is **undecidable**.

Reason for this negative results:

Unrestricted interplay between temporal modalities and FO quantification across states.

Summary of negative results so far

We have seen the following results:

- Without restrictions on the form of the DCDS, even the simplest properties (reachability) is undecidable.
  $\rightsquigarrow$ Towards decidability, we deal only with **state bounded DCDSs** and with logics with **active domain quantification** ($\mu\mathcal{L}_A$, LTL-FO$_A$).
- Even for state bounded DCDS, we have that:
  - Model checking $\mu\mathcal{L}_A$ does not admit formula-independent abstractions.
  - Model checking LTL-FO$_A$ (and hence LTL-FO) is undecidable.

To overcome these problems, we can follow different approaches:

- We consider a further restriction on DCDSs: run-boundedness
- We consider a further restriction on the logics: $\mu\mathcal{L}_P$ and LTL-FO$_P$.
- We study formula-dependent abstractions.

Summary of negative results so far

We have see the following results:

- Without restrictions on the form of the DCDS, even the simplest properties (reachability) is undecidable.
  $\rightsquigarrow$ Towards decidability, we deal only with **state bounded DCDSs** and with logics with **active domain quantification** ($\mu\mathcal{L}_A$, LTL-FO$_A$).
- Even for state bounded DCDS, we have that:
  - Model checking $\mu\mathcal{L}_A$ does not admit formula-independent abstractions.
  - Model checking LTL-FO$_A$ (and hence LTL-FO) is undecidable.

To overcome these problems, we can follow different approaches:

- We consider a further restriction on DCDSs: run-boundedness
- We consider a further restriction on the logics: $\mu\mathcal{L}_P$ and LTL-FO$_P$.
- We study formula-dependent abstractions.

Summary of negative results so far

We have seen the following results:

- Without restrictions on the form of the DCDS, even the simplest properties (reachability) is undecidable.
  $\leadsto$ Towards decidability, we deal only with **state bounded DCDSs** and with logics with **active domain quantification** ($\mu\mathcal{L}_A$, LTL-FO$_A$).
- Even for state bounded DCDS, we have that:
  - Model checking $\mu\mathcal{L}_A$ does not admit formula-independent abstractions.
  - Model checking LTL-FO$_A$ (and hence LTL-FO) is undecidable.

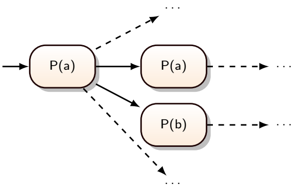
To overcome these problems, we can follow different approaches:

- We consider a further restriction on DCDSs: run-boundedness
- We consider a further restriction on the logics: $\mu\mathcal{L}_P$ and LTL-FO$_P$.
- We study formula-dependent abstractions.

Towards decidability

We need to tame the two sources of infinity in the RTS $\Upsilon_{\mathcal{X}}$ generated by a DCDS $\mathcal{X}$:

- infinite branching, due to external input;
- infinite runs, i.e., runs visiting infinitely many DBs.



To prove decidability of model checking for restricted DCDSs and a specific verification logic $\mathcal{L}$:

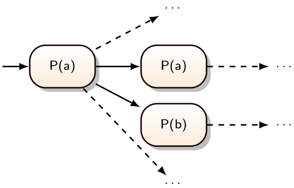
- We use as a tool bisimulations for the logic $\mathcal{L}$.
- We show that we can construct a finite-state RTS $\Theta_{\mathcal{X}}$ that provides a faithful abstraction of $\Upsilon_{\mathcal{X}}$ for formulas of $\mathcal{L}$.

  In other words, $\Theta_{\mathcal{X}}$ and $\Upsilon_{\mathcal{X}}$ are bisimilar, under the bisimulation for $\mathcal{L}$.

Towards decidability

We need to tame the two sources of infinity in the RTS $\Upsilon_{\mathcal{X}}$ generated by a DCDS $\mathcal{X}$:

- infinite branching, due to external input;
- infinite runs, i.e., runs visiting infinitely many DBs.



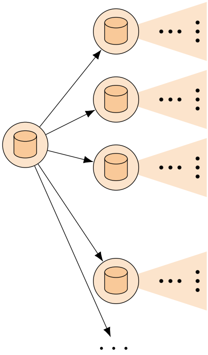
To prove decidability of model checking for restricted DCDSs and a specific verification logic $\mathcal{L}$:

- We use as a tool bisimulations for the logic $\mathcal{L}$.
- We show that we can construct a finite-state RTS $\Theta_{\mathcal{X}}$ that provides a faithful abstraction of $\Upsilon_{\mathcal{X}}$ for formulas of $\mathcal{L}$.

  In other words, $\Theta_{\mathcal{X}}$ and $\Upsilon_{\mathcal{X}}$ are bisimilar, under the bisimulation for $\mathcal{L}$.

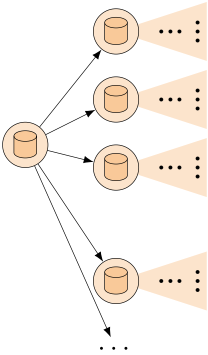
Dealing with infinite branching

- Infinite branching is caused by the infinite number of possible combinations of values returned by the service calls.



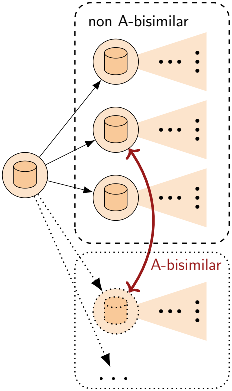
Dealing with infinite branching

- Infinite branching is caused by the infinite number of possible combinations of values returned by the service calls.
- Notice, however, that for each state along a run:
  - only a finite number of values have been encountered so far, and
  - only a finite number of service calls are issued when an action is executed.
- Hence, due to genericity, we need only to take into account:
  - whether a new value is equal to or differs from a value encountered so far;
  - whether new values obtained from different service calls are equal to or differ from each other.



Dealing with infinite branching

- Infinite branching is caused by the infinite number of possible combinations of values returned by the service calls.
- Notice, however, that for each state along a run:
  - only a finite number of values have been encountered so far, and
  - only a finite number of service calls are issued when an action is executed.
- Hence, due to genericity, we need only to take into account:
  - whether a new value is equal to or differs from a value encountered so far;
  - whether new values obtained from different service calls are equal to or differ from each other.



Dealing with infinite runs

We still need to address infiniteness of the RTS coming from possibly **infinite runs**, which may accumulate infinitely many new values along the run.

Two approaches to deal with this:

1. Restrict the DCDS, by ruling out a priori the accumulation of infinitely many values along a run.
   $\leadsto$ run-bounded DCDSs
2. Restrict the logics, making them "insensitive" to the infinitely many values.
   $\leadsto$ persistence-preserving variants of $\mu\mathcal{L}_{FO}$ and LTL-FO

Recall: the DCDSs we consider are state-bounded!

Dealing with infinite runs

We still need to address infiniteness of the RTS coming from possibly **infinite runs**, which may accumulate infinitely many new values along the run.

Two approaches to deal with this:

1. Restrict the DCDS, by ruling out a priori the accumulation of infinitely many values along a run.
   $\rightsquigarrow$ run-bounded DCDSs
2. Restrict the logics, making them "insensitive" to the infinitely many values.
   $\rightsquigarrow$ persistence-preserving variants of $\mu\mathcal{L}_{FO}$ and LTL-FO

Recall: the DCDSs we consider are state-bounded!

Run-boundedness

A DCDS $\mathcal{X}$ is run-bounded

if there exists a fixed number b such that the number of values used **in each (infinite) run** of $\mathcal{X}$, is **bounded by b**.

Note:

- In general, even when $\mathcal{X}$ is run-bounded, $\Upsilon_{\mathcal{X}}$ is still infinite-state due to infinite branching (but we have seen how to cope with this).
- Run-boundedness is a semantic condition.

Theorem

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

Run-boundedness

A DCDS $\mathcal{X}$ is run-bounded

if there exists a fixed number b such that the number of values used **in each (infinite) run** of $\mathcal{X}$, is **bounded by b**.

Note:

- In general, even when $\mathcal{X}$ is run-bounded, $\Upsilon_{\mathcal{X}}$ is still infinite-state due to infinite branching (but we have seen how to cope with this).
- Run-boundedness is a semantic condition.

Theorem

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

Run-boundedness

A DCDS $\mathcal{X}$ is run-bounded

if there exists a fixed number b such that the number of values used **in each (infinite) run** of $\mathcal{X}$, is **bounded by b**.

Note:

- In general, even when $\mathcal{X}$ is run-bounded, $\Upsilon_{\mathcal{X}}$ is still infinite-state due to infinite branching (but we have seen how to cope with this).
- Run-boundedness is a semantic condition.

Theorem

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

Avoiding run-boundedness

Run-boundedness is a **rather restrictive condition** for DCDSs

- With non-deterministic services: only a finite number of service calls . . .
- With deterministic services: only a finite number of distinct service calls . . .

. . . may be issued along a run.

Instead of requiring run-boundedness, we:

- restrict the form of quantification, and
- show how to construct a finite faithful abstraction in which we reuse values along runs.

Avoiding run-boundedness

Run-boundedness is a **rather restrictive condition** for DCDSs

- With non-deterministic services: only a finite number of service calls ...
- With deterministic services: only a finite number of distinct service calls ...

... may be issued along a run.

Instead of requiring run-boundedness, we:

- restrict the form of quantification, and
- show how to construct a finite faithful abstraction in which we reuse values along runs.

Eventually recycling pruning

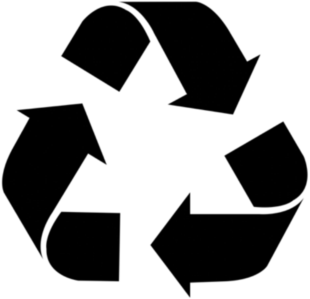
Intuition:

- We consider logics with persistence-preserving quantification, which cannot quantify over values, once they have left the active domain.
- When we need to return new values from service calls, we "recycle" those values that previously disappeared.
- We incorporate the recycling into the construction of the RTS for the DCDS, effectively pruning the set of generated states.
- If the DCDS is b-bounded, the recycling algorithm will introduce at most $2 \cdot b$ new values overall. Namely, for each state s:
  - at most b values that constitute $\text{ADOM}(db(s))$;
  - at most b new values that are introduced by the service calls, and that possibly replace some of the values in $\text{ADOM}(db(s))$.

Decidability for persistence-preserving logics

Prune

Recycle



Theorem

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

Decidability for persistence-preserving logics

- Given as input a state-bounded DCDS $\mathcal{X}$, algorithm RECYCLE constructs a **finite** RTS $\Theta_{\mathcal{X}}$.
- Moreover, $\Theta_{\mathcal{X}}$ and $\Upsilon_{\mathcal{X}}$ are persistence-preserving bisimilar.
- *Note:* the algorithm does not require to know the bound b for the state.

From this, and the fact that $\mu\mathcal{L}_P$ / LTL-FO$_A$ are invariant under persistence-reserving bisimulations, we obtain decidability of verification.

Theorem

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

Decidability for persistence-preserving logics

- Given as input a state-bounded DCDS $\mathcal{X}$, algorithm $\textsc{Recycle}$ constructs a **finite** RTS $\Theta_{\mathcal{X}}$.
- Moreover, $\Theta_{\mathcal{X}}$ and $\Upsilon_{\mathcal{X}}$ are persistence-preserving bisimilar.
- *Note:* the algorithm does not require to know the bound b for the state.

From this, and the fact that $\mu\mathcal{L}_P$ / LTL-FO$_A$ are invariant under persistence-reserving bisimulations, we obtain decidability of verification.

Theorem

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

$\mu\mathcal{L}_A$ and $\mu\mathcal{L}_{FO}$ over state-bounded DCDSs

We have seen that $\mu\mathcal{L}_A$ (and hence $\mu\mathcal{L}_{FO}$) over state-bounded DCDSs does not admit formula-independent abstractions.

But is verification decidable?

- $\mu\mathcal{L}_{FO}$ is not able to single out properties about a run.
- Combined with genericity of the RTS generated by a DCDS $\mathcal{X}$, this limits the ability to express first-order temporal properties over $\Upsilon_{\mathcal{X}}$.
- Hence, given a $\mu\mathcal{L}_{FO}$ formula Φ with n variables, we can introduce n data slots that keep track of their assignments.

Theorem

Given a state-bounded DCDS $\mathcal{X}$ and an integer n, we can construct a finite state abstraction $\Theta_{\mathcal{X}}$ of $\Upsilon_{\mathcal{X}}$ (that depends on n) such that, for every $\mu\mathcal{L}_{FO}$ formula Φ with n variables,

$$\Theta_{\mathcal{X}} \models \Phi \quad \text{if and only if} \quad \Upsilon_{\mathcal{X}} \models \Phi.$$

Final overall picture