

Verification of Data-Aware Processes

Boundaries of Decidability: Negative Results

Diego Calvanese, Marco Montali

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



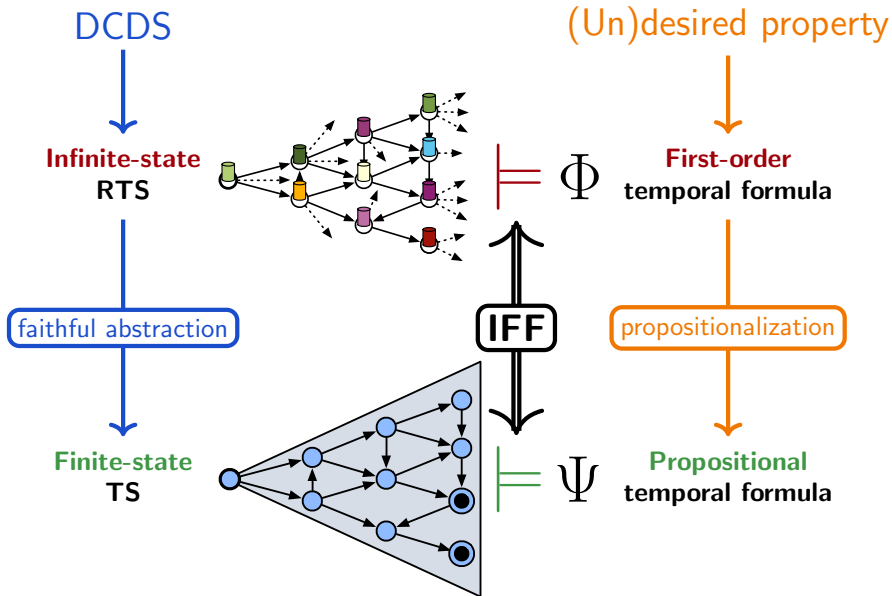
29th European Summer School in Logic, Language, and Information
(ESLLI 2017)

Toulouse, France – 17–28 July 2017

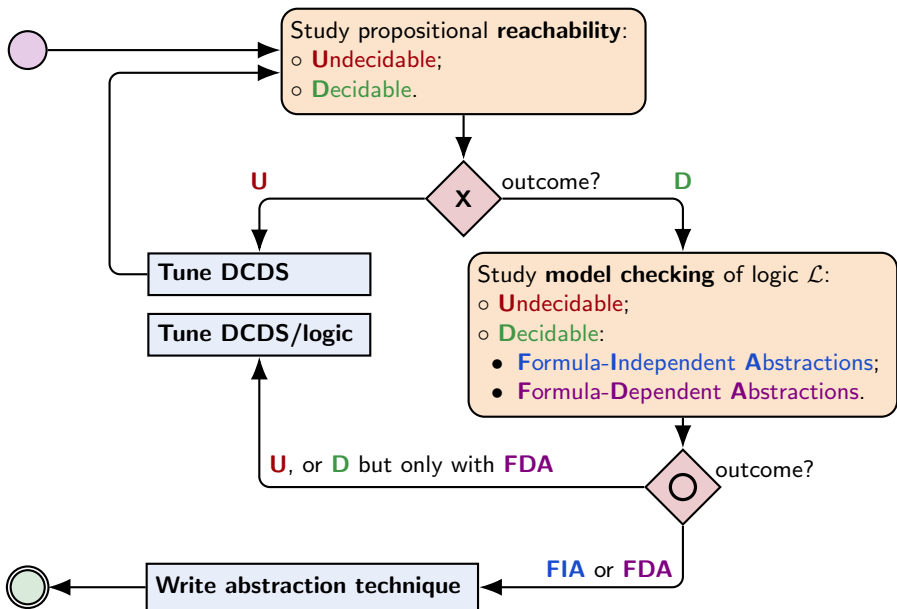
Outline

- 1 The Good, the Bad, and the Ugly
- 2 Counter Automata
- 3 From Counter Automata to DCDSs
- 4 State-Bounded DCDSs
- 5 Key Properties, and Their Impact on State-Boundedness
- 6 Negative Results for State-Bounded DCDSs

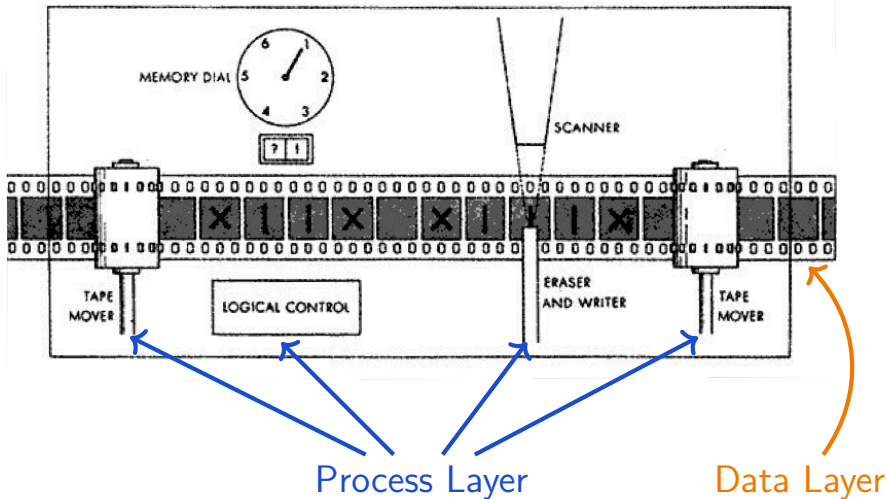
Our goal



The good

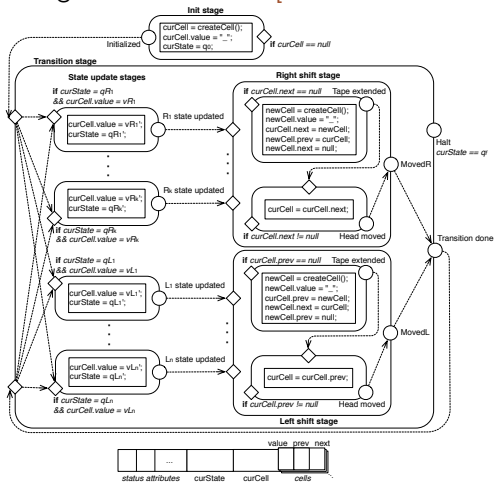


The bad



The bad

A Turing Machine in GSM [Solomakhin et al. 2013]



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**.

To prove these theorems, we need to reduce from a simple, yet Turing-powerful, computation mechanism.

Outline

- 1 The Good, the Bad, and the Ugly
- 2 Counter Automata**
- 3 From Counter Automata to DCDSs
- 4 State-Bounded DCDSs
- 5 Key Properties, and Their Impact on State-Boundedness
- 6 Negative Results for State-Bounded DCDSs

Counter automaton (CA)

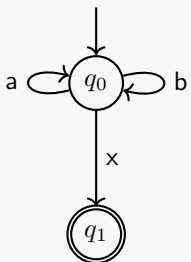
A finite-state automaton, whose transitions:

- recognize symbols from a finite alphabet;
- **manipulate and test** counters: registers storing non-negative values.

Three forms of test/manipulation:

- **increment** a counter (**inc**) - always executable;
- **decrement** a **positive** counter (**dec**) - executable only if counter > 0 ;
- **test** a counter **for zero** (**ifz**) - executable only if counter $== 0$.

Finite-state automaton recognizing finite words over $\Sigma = \{a,b,x\}$



Some recognized words:

x
 ax
 abx
 abbx
 aaabbbabx
 abababaabx
 abababaaaabx

Counter automaton (CA)

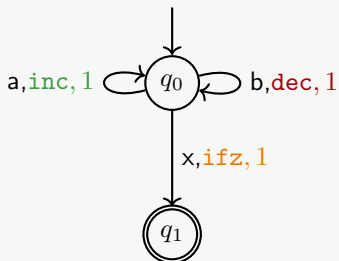
A finite-state automaton, whose transitions:

- recognize symbols from a finite alphabet;
- **manipulate and test** counters: registers storing non-negative values.

Three forms of test/manipulation:

- **increment** a counter (**inc**) - always executable;
- **decrement** a **positive** counter (**dec**) - executable only if counter > 0 ;
- **test** a counter **for zero** (**ifz**) - executable only if counter $== 0$.

CA (with one counter initially 0) recognizing finite words over $\Sigma = \{a,b,x\}$



Some recognized words:

x
ax
abx
abbx
aaabbbabx
abababaabx
abababaaaabx

Counter automaton: formal definition [Demri and Lazic 2009]

A **counter automaton (CA)** \mathcal{M} is a tuple $\langle \Sigma, Q, q_I, n, \delta, F \rangle$, where:

- Σ is a finite alphabet;
- Q is a finite set of *locations*;
- $q_I \in Q$ is the *initial location*;
- $n \in \mathbb{N}$ is the number of *counters*;
- $\delta \subseteq Q \times \Sigma \times L \times Q$ is a labeled transition relation capturing the *control configuration updates*.
 L is the *instruction set* $\{\text{inc}, \text{dec}, \text{ifz}\} \times \{1, \dots, n\}$
- $F \subseteq Q$ is the set of *accepting locations*.

When a CA recognizes a word, we have to remember:

- the **current state**;
- the **current value of each counter**.

A **configuration** of counter automaton $\langle \Sigma, Q, q_I, n, \delta, F \rangle$ is a pair $\langle q, v \rangle$, where:

- $q \in Q$ is a location;
- $v : \{1, \dots, n\} \rightarrow \mathbb{N}$ is a counter valuation.

Minsky counter automata (MCA)

A counter automaton that **correctly** manipulates its counters.

A **transition** of MCA $\langle \Sigma, Q, q_I, n, \delta, F \rangle$ is of the form $\langle q, v \rangle \xrightarrow{w, l} \langle q', v' \rangle$, where:

- $\langle q, v \rangle$ and $\langle q', v' \rangle$ are configurations of the MCA;
- $w \in \Sigma$ is a symbol;
- $l \in L$ is an instruction;
- the transition agrees with the control configuration updates δ , i.e.,
 $\langle q, w, l, q' \rangle \in \delta$;
- the transition correctly tests and manipulates the counters according to l , i.e., given $c \in \{1, \dots, n\}$:
 - if $l = \langle \text{inc}, c \rangle$, then v' is identical to v , except $v'(c) = v(c) + 1$;
 - if $l = \langle \text{dec}, c \rangle$, then $v(c) > 0$, and v' is identical to v , except $v'(c) = v(c) - 1$;
 - if $l = \langle \text{ifz}, c \rangle$, then $v(c) = 0$, and v' is identical to v .

Recognizing words using MCA

A **run** of MCA $\langle \Sigma, Q, q_I, n, \delta, F \rangle$

is a sequence of transitions $\langle q_0, v_0 \rangle \xrightarrow{w_0, l_0} \langle q_1, v_1 \rangle \xrightarrow{w_1, l_1} \dots$, where

- $q_0 = q_I$ (the run starts from the initial location of the MCA);
- v_0 is such that for every $c \in \{1, \dots, n\}$, $v_0(c) = 0$ (counters start from 0).

A run is **accepting** if

- **finite run**: it **ends with** an accepting location;
- **infinite run**: it visits an accepting location **infinitely often**.

The MCA **accepts** word $w_0 w_1 \dots$ if

there exists a run of the MCA of the form

$$\langle q_0, v_0 \rangle \xrightarrow{w_0, l_0} \langle q_1, v_1 \rangle \xrightarrow{w_1, l_1} \dots$$

that is **accepting**.

(Non)emptiness over finite and infinite words immediately follow.

Faulty CA with unreliable counters

Lossy CA [Mayr 2003]

CA with unreliable counters that nondeterministically **leak down**.

- A (seemingly) **positive counter may return true when tested for zero**, because it could have leaked down to 0.

Incrementing CA (ICA) [Demri and Lazic 2009]

CA with unreliable counters that nondeterministically **leak up**.

- A (seemingly) **zero counter may accept a decrement**, because it could have leaked up to whatever positive number.

A **transition** of ICA $\langle \Sigma, Q, q_I, n, \delta, F \rangle$ is of the form $\xrightarrow{w,l}_\dagger$, where:

$\langle q, v \rangle \xrightarrow{w,l}_\dagger \langle q', v' \rangle$ if and only if

- there exists v'' such that $\langle q, v \rangle \xrightarrow{w,l} \langle q', v'' \rangle$ (Minsky semantics);
- v' is such that for every counter $c \in \{1, \dots, n\}$, $v'(c) \geq v''(c)$.

A transition in an ICA coincides with that of the corresponding MCA, possibly experiencing an unpredictable incrementation of one or more counters.

Counter automata are powerful

An i -CA is a CA using i counters.

Theorem ([Minsky 1967])

Emptiness of **2-MCA** over **finite/infinite** words is **undecidable**.

Theorem ([Demri and Lazic 2009])

Emptiness of **2-ICA** over **infinite** words is **undecidable**.

In general, many intriguing results on decidability and complexity of various reasoning tasks over faulty counter machines.

- Depending on the property that one wants to check, they may behave like MCAs, or turn out to be simpler.
- Useful tool to prove decidability and undecidability results for systems that are not able to fully simulate the behaviours exhibited by an MCA.

Outline

- 1 The Good, the Bad, and the Ugly
- 2 Counter Automata
- 3 From Counter Automata to DCDSs**
- 4 State-Bounded DCDSs
- 5 Key Properties, and Their Impact on State-Boundedness
- 6 Negative Results for State-Bounded DCDSs

Representing MCAs with DCDSs

Crux:

- show how to simulate two counters in the DCDS data layer;
- show how to simulate increment, decrement, zero testing in the DCDS process layer.

The control part is straightforward (being propositional).

Representing MCAs with DCDSs

Crux:

- show how to simulate two counters in the DCDS data layer;
- show how to simulate increment, decrement, zero testing in the DCDS process layer.

The control part is straightforward (being propositional).

Unary Relations

Data layer: two unary relations.

$$c_1 = 3 \left\{ \begin{array}{c} \mathbf{C}_1 \\ \boxed{a} \\ \boxed{b} \\ \boxed{c} \end{array} \right. \left. \begin{array}{c} \mathbf{C}_2 \\ \boxed{d} \\ \boxed{a} \end{array} \right\} c_2 = 2$$

- Counter: size of a unary relation C in the DB.
- **Increment**: insert a **fresh** value into C .
- **Decrement**: pick a value from C , and remove it from the extension of C .
- **Zero testing**: check that C does not contain any value.

Representing MCAs with DCDSs

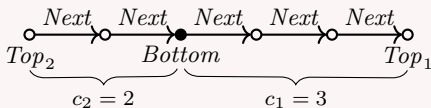
CruX:

- show how to simulate two counters in the DCDS data layer;
- show how to simulate increment, decrement, zero testing in the DCDS process layer.

The control part is straightforward (being propositional).

Binary Relation

Data layer: one binary relation, three unary relations.



- Counter: length of the chain from the bottom value to left/right top.
- **Increment**: extend the chain of one step (**fresh** top).
- **Decrement**: cut the chain of its last step.
- **Zero testing**: check that the bottom and top element coincide.

Warm-up: fresh inputs

Fact

Service calls may return whatever value, possibly being already used in the current DB, possibly not.

Recall the cart example: no guarantee that adding a product to the cart will result in a proper update.

- The obtained barcode could be already in use!

Observation

The DCDS may reject certain returned values through constraints!

This is the basis to simulate **local** and **global** freshness.

- We are going to show this on a simple example where we consider relation-freshness, i.e., freshness w.r.t. the values stored in a unary relation.
- Generalization to relations with multiple arity, or to the entire active domain of the DB, are straightforward.

Simulating freshness

DCDS “adding” elements to a unary relation

Data layer: unary relation R , no constraint.

Process layer:

- one nondeterministic service call $f()$;
- one action α inserting an external value in R :
 - $\alpha \cdot \text{PARS} = \emptyset$;
 - $\alpha \cdot \text{EFF} = \mathbf{add}\{\text{true} \rightsquigarrow R(f())\}$.
- a single condition-action rule $\text{true} \mapsto \alpha()$ (tells that α is always applicable).

Fact: this DCDS models **faulty insertions**

No guarantee that the value injected using $f()$ is **fresh**.

- If it is not, no addition happens (due to set semantics).

Is it possible to guarantee freshness of $f()$?

- **global** freshness: fresh by considering the entire history.
- **local** freshness: fresh by considering the current DB.

Simulating global Freshness

Compile DCDS into new DCDS with additional relations, constraints, effects.

DCDS “adding” elements to a unary relation

Data layer: unary relation R , no constraint.

Process layer:

- one nondeterministic service call $f()$;
- one action α inserting an external value in R :

$$\alpha \cdot \text{PARS} = \emptyset \quad \alpha \cdot \text{EFF} = \{\text{true} \rightsquigarrow \mathbf{add}\{R(f())\}\}$$

Addition of **globally fresh** elements

Data layer: two **additional** unary relations R_{new} and R_{old} .

- Subject to a disjointness constraint $\forall x. \neg (R_{new}(x) \wedge R_{old}(x))$.

Process layer:

- α stores in R_{new} the **newly inserted** values, in R_{old} the **historical** values:

$$\alpha \cdot \text{EFF} = \left\{ \begin{array}{l} \text{true} \rightsquigarrow \mathbf{add}\{R(f()), R_{new}(f())\} \\ R_{new}(x) \rightsquigarrow \mathbf{del}\{R_{new}(x)\} \mathbf{add}\{R_{old}(x)\} \\ R_{old}(x) \rightsquigarrow \mathbf{del}\{R_{old}(x)\} \end{array} \right\}$$

- Due to disjointness: if $f()$ is not globally fresh, the update is rejected!
- **Note:** even if R stays bounded in size, R_{old} may grow unboundedly!

Simulating local Freshness

Compile DCDS into new DCDS with additional relations, constraints, effects.

DCDS “adding” elements to a unary relation

Data layer: unary relation R , no constraint.

Process layer:

- one nondeterministic service call $f()$;
- one action α inserting an external value in R :

$$\alpha \cdot \text{PARS} = \emptyset \quad \alpha \cdot \text{EFF} = \{\text{true} \rightsquigarrow \mathbf{add}\{R(f())\}\}$$

Addition of **locally fresh** elements

Data layer: two **additional** unary relations R_{new} and R_{old} .

- Subject to a disjointness constraint $\forall x. \neg (R_{new}(x) \wedge R_{old}(x))$.

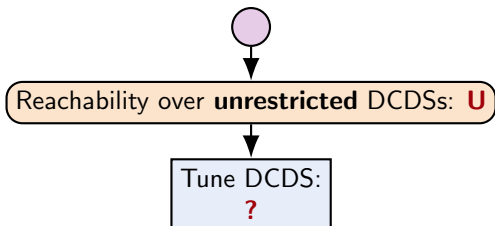
Process layer:

- α stores in R_{new} the **newly inserted** values, in R_{old} the **previous** values:

$$\alpha \cdot \text{EFF} = \left\{ \begin{array}{l} \text{true} \rightsquigarrow \mathbf{add}\{R(f()), R_{new}(f())\} \\ R_{new}(x) \rightsquigarrow \mathbf{del}\{R_{new}(x)\} \mathbf{add}\{R_{old}(x)\} \\ R_{old}(x) \rightsquigarrow \mathbf{del}\{R_{old}(x)\} \end{array} \right\}$$

- Due to disjointness: if $f()$ is not locally fresh, the update is rejected!
- **Note:** If R stays bounded in size, so does R_{old} !

Current overall picture



Outline

- 1 The Good, the Bad, and the Ugly
- 2 Counter Automata
- 3 From Counter Automata to DCDSs
- 4 State-Bounded DCDSs**
- 5 Key Properties, and Their Impact on State-Boundedness
- 6 Negative Results for State-Bounded DCDSs

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.

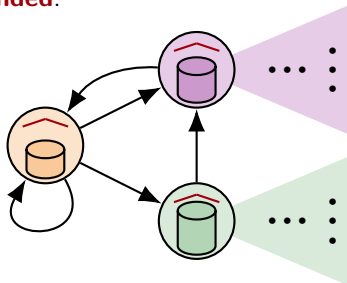
A DCDS \mathcal{X} is **state-bounded**

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** : given $\Upsilon_{\mathcal{X}} = \langle \Delta, \mathcal{R}, S, s_0, db, \Rightarrow \rangle$, for each state $s \in S$, we have $|\text{ADOM}(db(s))| \leq b$.

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. □

Outline

- 1 The Good, the Bad, and the Ugly
- 2 Counter Automata
- 3 From Counter Automata to DCDSs
- 4 State-Bounded DCDSs
- 5 Key Properties, and Their Impact on State-Boundedness**
- 6 Negative Results for State-Bounded DCDSs

Two key properties of DCDSs

DCDS are ...

Markovian

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

Based on generic queries

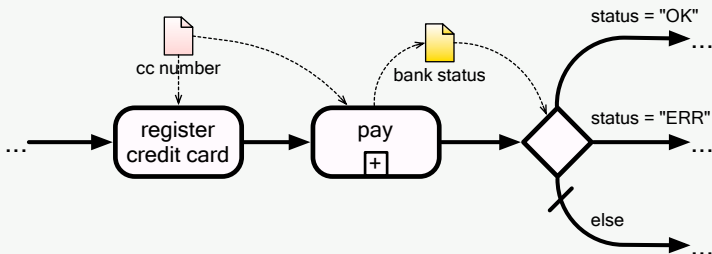
FO/SQL (as virtually all query languages) does not distinguish structures that are identical modulo uniform renaming of data objects.

- 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
 \rightsquigarrow only **how it relates** to the other objects!

Genericity, formally

We want to define a **genericity property** capturing RTSs that do not distinguish values as such, but only how they participate to relations.

We consider **isomorphisms** \sim_h between relational states, where h is a bijection between the domains that preserves relations and constants.

Fix a **finite** set $\Delta_0 \subset \Delta$ of distinguished constant.

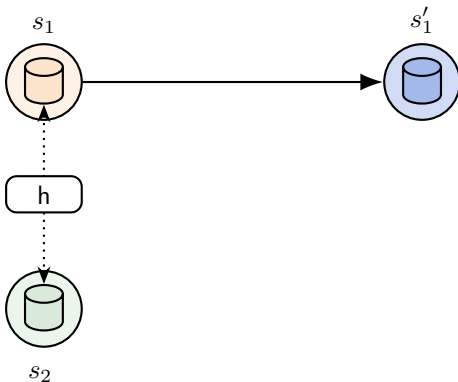
An RTS $\Upsilon = \langle \Delta, \mathcal{R}, S, s_0, db, \Rightarrow \rangle$ is **generic**

if for all states $s_1, s_2 \in S$, and every bijection $h : \Delta \mapsto \Delta$ that is the identity over Δ_0 :

if $db(s_1) \sim_h db(s_2)$ and there exists s'_1 such that $s_1 \Rightarrow s'_1$,
then there exists s'_2 such that $s_2 \Rightarrow s'_2$ and $db(s'_1) \sim_h db(s'_2)$.

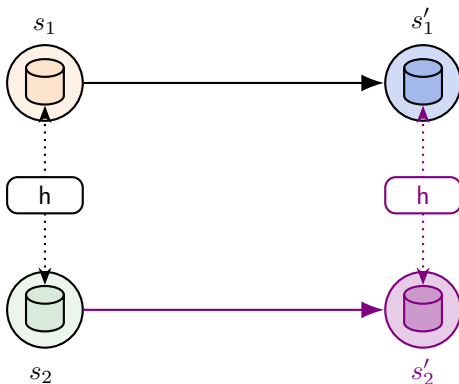
Genericity, graphically

If...



Genericity, graphically

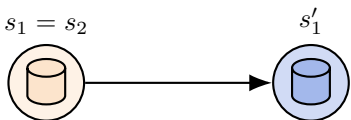
If... Then



Genericity, graphically

Note: s_1 and s_2 can be the same state, hence the existence of a successor state induces the existence of all successor states isomorphic to it.

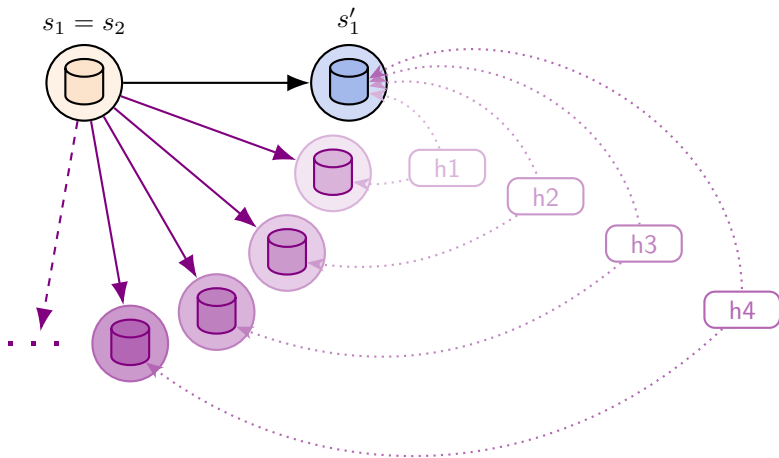
If...



Genericity, graphically

Note: s_1 and s_2 can be the same state, hence the existence of a successor state induces the existence of all successor states isomorphic to it.

If... Then



DCDSs always induce generic RTSs

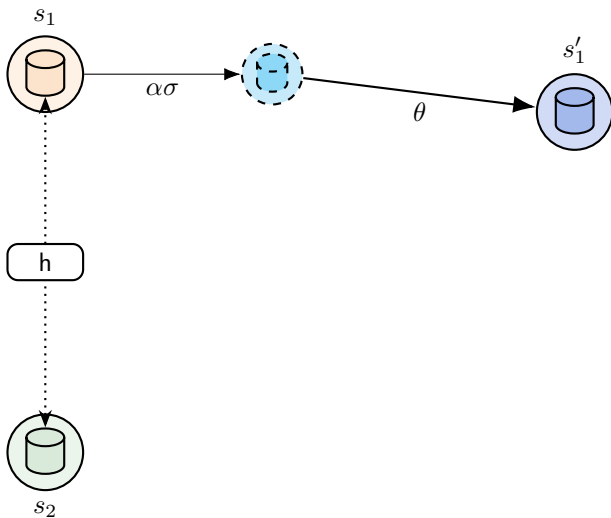
DCDSs induce **generic RTSs** since:

- The progression mechanism is defined through FO, whose queries are generic.
- The progression mechanism is markovian, i.e., next states only depend on the current state.
- The semantics of service calls induces the existence of a successor state for each combination of values returned by the service calls, such that constraints are satisfied.

Successor states generated through the execution of a DCDS action are **“indistinguishable”** modulo isomorphisms on the results of service calls.

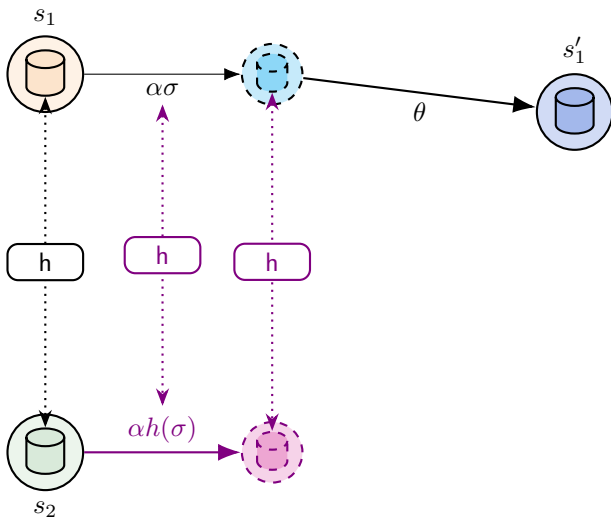
DCDSs are Generic

If...



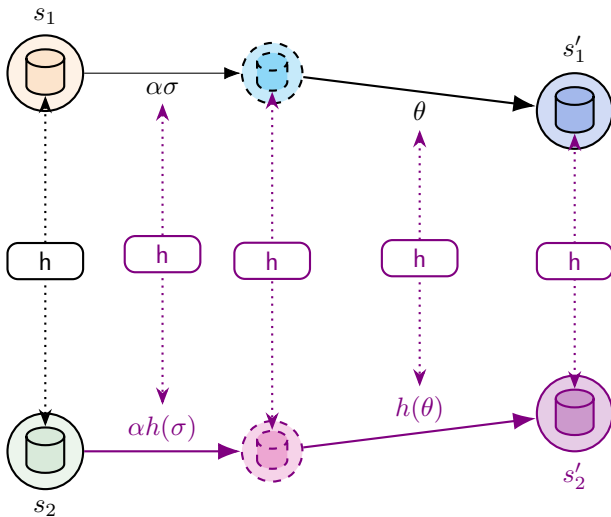
DCDSs are Generic

If... Then



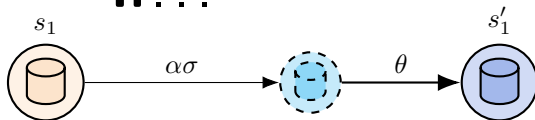
DCDSs are Generic

If... Then



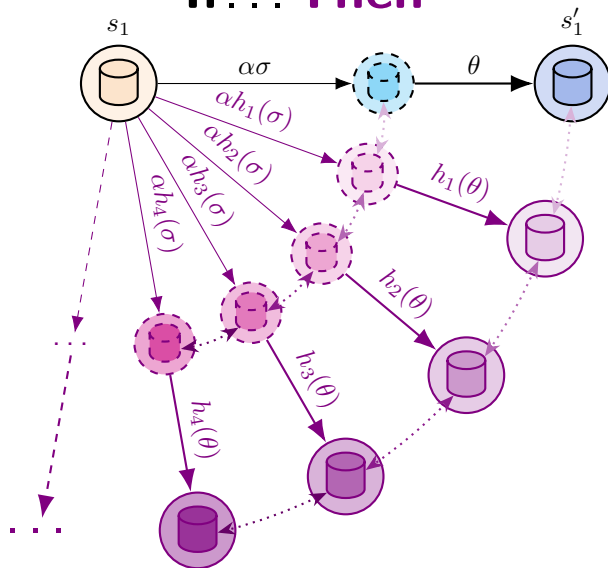
DCDSs are generic

If...



DCDSs are generic

If... Then



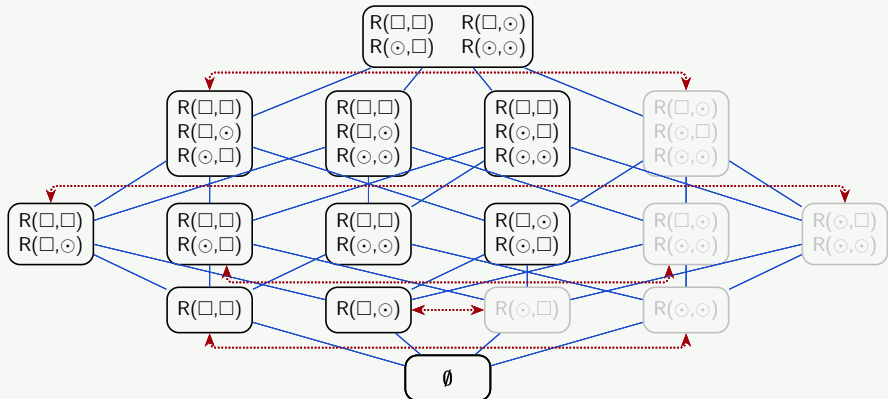
Proof of decidability

Consider a b -bounded DCDS \mathcal{X} .

How many isomorphic DBs exist in a b -bounded system?

Finitely many! Just pick b distinct values and do (wise) combinatorics.

E.g., for a single binary relation R , bound $b = 2$, $\Delta_0 = \emptyset$



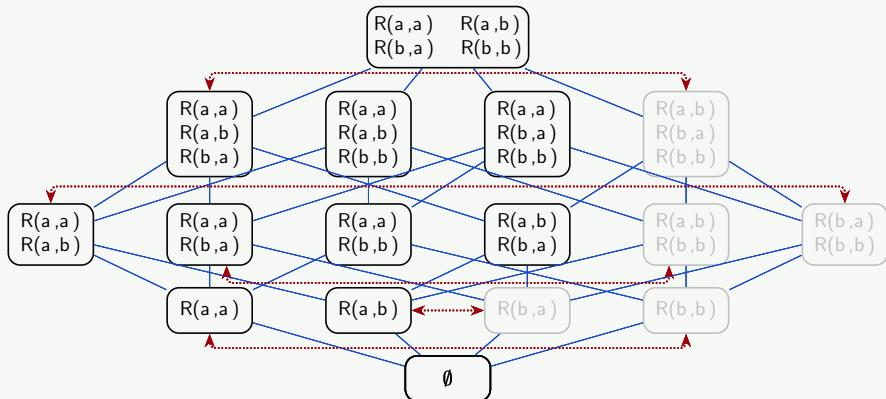
Proof of decidability

Consider a b -bounded DCDS \mathcal{X} .

How many isomorphic DBs exist in a b -bounded system?

Finitely many! Just pick b distinct values and do (wise) combinatorics.

E.g., for a single binary relation R , bound $b = 2$, $\Delta_0 = \emptyset$, $\Delta = \{a, b\}$



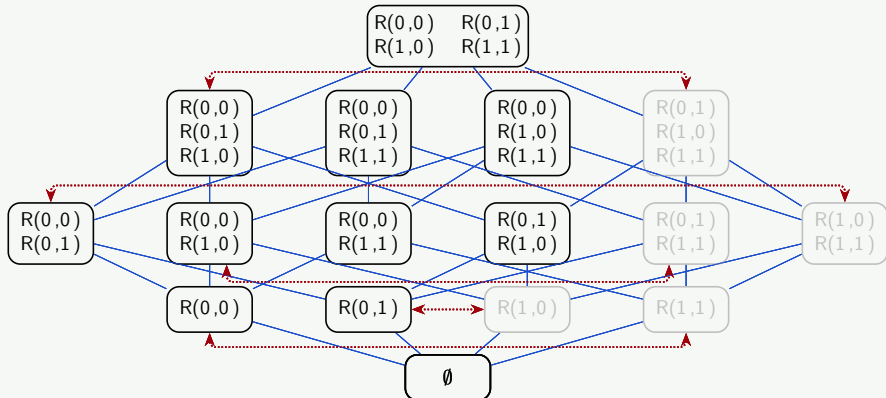
Proof of decidability

Consider a b -bounded DCDS \mathcal{X} .

How many isomorphic DBs exist in a b -bounded system?

Finitely many! Just pick b distinct values and do (wise) combinatorics.

E.g., for a single binary relation R , bound $b = 2$, $\Delta_0 = \emptyset$, $\Delta = \{0, 1\}$



Proof of decidability

Consider a b -bounded DCDS \mathcal{X} , whose schema contains a proposition *Hit*.

Procedure

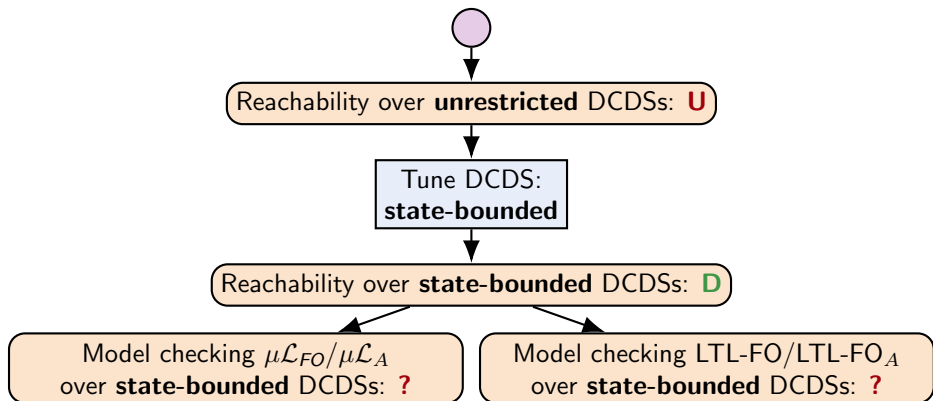
- 1 Fix b values over Δ (in addition to Δ_0).
- 2 Construct all DBs of \mathcal{X} over the schema of \mathcal{X} and values from $\Delta \uplus \Delta_0$.
Note: as seen in previous example, not all are needed, since some are isomorphic to each other.
Note₂: we assume $\text{ADOM}(\mathcal{I}_0) \subseteq \Delta_0$, hence one such DBs is exactly \mathcal{I}_0 .
- 3 For each pair of DBs, check whether the second is a legal successor of the first (cf. DCDS execution semantics).
- 4 We get a **finite-state** RTS $\Theta_{\mathcal{X}}$.
- 5 Thanks to **genericity**:

$$\textit{Hit} \text{ reachable in } \Upsilon_{\mathcal{X}} \quad \text{iff} \quad \textit{Hit} \text{ reachable in } \Theta_{\mathcal{X}}.$$
- 6 Check whether *Hit* is reachable $\Theta_{\mathcal{X}}$ (clearly decidable).

What if I **do not know** b ?

Decidability still holds! See later.

Current overall picture



Outline

- 1 The Good, the Bad, and the Ugly
- 2 Counter Automata
- 3 From Counter Automata to DCDSs
- 4 State-Bounded DCDSs
- 5 Key Properties, and Their Impact on State-Boundedness
- 6 Negative Results for State-Bounded DCDSs**

A mildly negative result: $\mu\mathcal{L}_A$ + state-boundedness

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.

Proof.

- 1 We construct a DCDS \mathcal{X} working over a single unary relation and that induces a very simple, 1-bounded RTS.
- 2 We construct a $\mu\mathcal{L}_A$ formula $\Phi_{\geq i}$ parameterized by number i , asserting that **at least i distinct values are encountered in the initial prefix of a run.**
- 3 We show that $\mathcal{X} \models \Phi_{\geq i}$ for every $i \in \mathbb{N}$.
- 4 We imagine that finite-state, formula-independent abstraction $\Theta_{\mathcal{X}}$ of $\Upsilon_{\mathcal{X}}$, exists.
- 5 Since it is finite-state, $\Theta_{\mathcal{X}}$ contains finitely many values overall, say k .
- 6 But then, it is impossible for $\Theta_{\mathcal{X}}$ to encounter $k + 1$ distinct values, i.e., while $\mathcal{X} \models \Phi_{\geq k+1}$, we have that $\Theta_{\mathcal{X}} \not\models \Phi_{\geq k+1}$.
- 7 **Contradiction!** Hence, $\Theta_{\mathcal{X}}$ cannot exist. □

Note: this **does not imply** undecidability of $\mu\mathcal{L}_A$ model checking!

Proof details: DCDS

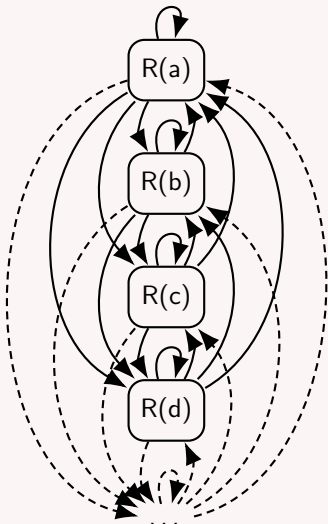
1-bounded DCDS \mathcal{X}

Data layer: unary relation R , no constraint,
initial DB $\{R(a)\}$

Process layer:

- one nondeterministic service call $g()$;
- one action α nondeterministically replacing the single value in R with another one (possibly the same):
 - $\alpha \cdot \text{PARS} = \emptyset$;
 - $\alpha \cdot \text{EFF} = \left\{ \begin{array}{l} R(x) \rightsquigarrow \mathbf{del}\{R(x)\} \\ \mathbf{add}\{R(g())\} \end{array} \right\}$
- a single condition-action rule
 $\text{true} \mapsto \alpha()$
 (tells that α is always applicable).

RTS $\Upsilon_{\mathcal{X}}$



Proof details: formula

The $\mu\mathcal{L}_A$ formula $\Phi_{\geq i}$

$$\exists x_1. \text{LIVE}(x_1) \wedge \langle - \rangle \left(\begin{array}{l} \exists x_2. \text{LIVE}(x_2) \wedge x_2 \neq x_1 \\ \wedge \langle - \rangle \left(\begin{array}{l} \exists x_3. \text{LIVE}(x_3) \wedge x_3 \neq x_1 \wedge x_3 \neq x_2 \\ \wedge \langle - \rangle \left(\begin{array}{l} \dots \\ \wedge \langle - \rangle (\exists x_i. \text{LIVE}(x_i) \wedge x_i \neq x_1 \wedge \dots \wedge x_{i-1} \neq x_i) \end{array} \right) \end{array} \right) \end{array} \right)$$

Note: the formula is fixpoint-free.

For every $i \in \mathbb{N}$, we have $\mathcal{X} \models \Phi_{\geq i}$

Actually, all states of $\Upsilon_{\mathcal{X}}$ satisfy $\Phi_{\geq i}$.

A strongly negative result: $\text{LTL-FO}_A + \text{state-boundedness}$

Here, the situation is even more critical!

Theorem

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

Proof.

Careful reconstruction of the proof of undecidability of satisfiability of LTL with freeze-quantifier [Demri and Lazic 2009]: we reduce emptiness of 2-ICAs to verification of LTL-FO_A properties over a 1-bounded DCDS. Idea:

- 1 We translate the input CA \mathcal{M} into a 1-bounded DCDS $\mathcal{X}_{\mathcal{M}}$ that “simulates” the runs of \mathcal{M} by just focusing the control part of \mathcal{M} . Counter values cannot be retained, since $\mathcal{X}_{\mathcal{M}}$ is 1-bounded.
- 2 Since $\mathcal{X}_{\mathcal{M}}$ does not capture counter values, the simulation of \mathcal{M} is faulty: $\mathcal{X}_{\mathcal{M}}$ captures **all proper runs of \mathcal{M}** , but also many other **spurious runs** where transitions are performed even when the counters would block them.
- 3 We write an LTL-FO_A formula $\Phi_{\mathcal{M}}$ that: (i) **separates the correct runs** from the spurious runs, (ii) encodes the **nonacceptance condition** of \mathcal{M} over infinite words.
- 4 $\Phi_{\mathcal{M}}$ is not powerful enough to capture the Minsky semantics of \mathcal{M} , but is able to capture its **incrementing semantics**: this suffices for **undecidability!** □

Proof details: DCDS

Given ICA $\mathcal{M} = \langle \Sigma, Q, q_I, n, \delta, F \rangle$, we construct a DCDS $\mathcal{X}_{\mathcal{M}} = \langle \mathcal{D}_{\mathcal{M}}, \mathcal{P}_{\mathcal{M}}, \emptyset \rangle$, such that each state of $\mathcal{X}_{\mathcal{M}}$ holds a **control configuration update** $\langle q, w, op, c, q' \rangle$ of \mathcal{M} . This, in turn, *may* correspond to a potential transition of \mathcal{M} , once the counter valuation is projected away. Runs of $\mathcal{X}_{\mathcal{M}}$ are (**proper** or **spurious**) sequences of control config updates.

A sequence $\langle q_0^c, w_0, op_0, c_0, q_0^n \rangle, \langle q_1^c, w_1, op_1, c_1, q_1^n \rangle, \dots$ of control config updates is:

- **Proper**, if it corresponds to an actual run of \mathcal{M} , i.e.:
 - ① $q_0^c = q_I$ (the sequence starts from the initial location of \mathcal{M});
 - ② for every $i > 0$ we have $q_i^n = q_{i+1}^c$ (the sequence of locations forms a chain);
 - ③ there exists a run of \mathcal{M} of the form

$$\langle q_0^c, v_0 \rangle \xrightarrow{w_0, \langle op_0, c_0 \rangle} \dagger \langle q_1^c, v_1 \rangle \xrightarrow{w_1, \langle op_1, c_1 \rangle} \dagger \dots$$

This in particular means that the control config updates correspond to proper counter tests and operations (depending on the chosen semantics).

- **Spurious** otherwise.

Proof details: DCDS

Given ICA $\mathcal{M} = \langle \Sigma, Q, q_I, n, \delta, F \rangle$, we construct a DCDS $\mathcal{X}_{\mathcal{M}} = \langle \mathcal{D}_{\mathcal{M}}, \mathcal{P}_{\mathcal{M}}, \emptyset \rangle$, such that each state of $\mathcal{X}_{\mathcal{M}}$ holds a **control configuration update** $\langle q, w, op, c, q' \rangle$ of \mathcal{M} . This, in turn, *may* correspond to a potential transition of \mathcal{M} , once the counter valuation is projected away. Runs of $\mathcal{X}_{\mathcal{M}}$ are (**proper** or **spurious**) sequences of control config updates.

In addition, $\mathcal{X}_{\mathcal{M}}$ uses a 1-bounded, unary relation *Color*, which assigns a value from Δ to a state. This is interpreted as a way of **coloring** each control configuration, where Δ provides infinitely many colors.

Operationally, at each step $\mathcal{X}_{\mathcal{M}}$ guesses a control configuration and a color:

- there are finitely many control configurations for \mathcal{M} ;
- but infinitely many ways of coloring them.

Proof details: DCDS

Note: We model $\mathcal{X}_{\mathcal{M}}$ by using relations also for control configurations. This is just syntactic sugar for easing the presentation: $\mathcal{X}_{\mathcal{M}}$ can be reformulated as a truly 1-bounded DCDS, at the cost of increasing the number of actions.

Data layer $\mathcal{D}_{\mathcal{M}}$

No constraint, schema:

- 1-bounded unary relations *CurLoc* and *NextLoc* (current/next location).
- 1-bounded unary relation *CurSym* (current symbol).
- 1-bounded binary relation *Op* (counter instruction: operation and counter).
- 1-bounded unary relation *Color* (configuration coloring).

Proof details: DCDS

Note: We model $\mathcal{X}_{\mathcal{M}}$ by using relations also for control configurations. This is just syntactic sugar for easing the presentation: $\mathcal{X}_{\mathcal{M}}$ can be reformulated as a truly 1-bounded DCDS, at the cost of increasing the number of actions.

Process layer $\mathcal{P}_{\mathcal{M}}$

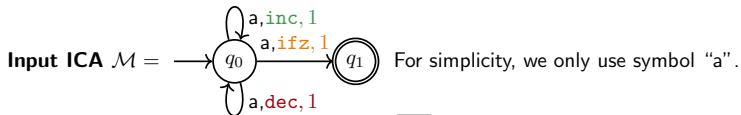
- Nullary nondeterministic service call $inputcol()$ for color guessing.
- Action $setColTCP$ taking as parameters the components of a transition control part ($setColTCP \cdot PARS = \langle q_c, w, op, c, q_n \rangle$), and setting them together with a guessed color: $setColTCP \cdot EFF =$


$$\left\{ \begin{array}{l} CurLoc(x) \rightsquigarrow \mathbf{del} \{ CurLoc(x) \} \\ NextLoc(x) \rightsquigarrow \mathbf{del} \{ NextLoc(x) \} \\ CurSym(x) \rightsquigarrow \mathbf{del} \{ CurSym(x) \} \\ Op(x, y) \rightsquigarrow \mathbf{del} \{ Op(x, y) \} \\ Color(x) \rightsquigarrow \mathbf{del} \{ Color(x) \} \\ \text{true} \rightsquigarrow \mathbf{add} \{ CurLoc(q_c), CurSym(w), Op(op, c), NextLoc(q_n) \} \\ \text{true} \rightsquigarrow \mathbf{add} \{ Color(inputcol()) \} \end{array} \right\}$$

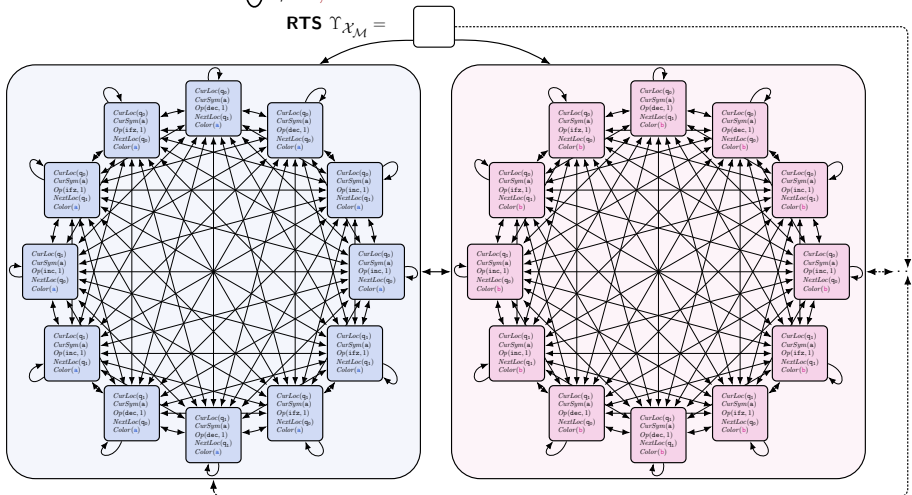
- For each transition control part $\langle q, w, op, c, q' \rangle$ of \mathcal{M} , a condition-action rule setting that control part:

$$q_c = q \wedge w = w \wedge op = op \wedge c = c \wedge q_n = q' \mapsto \mathbf{setColTCP}(q_c, w, op, c, q_n)$$

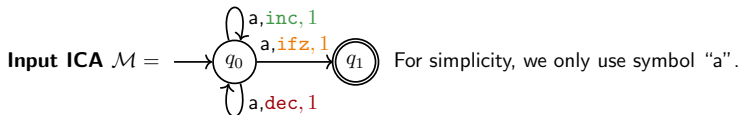
Proof details: DCDS example



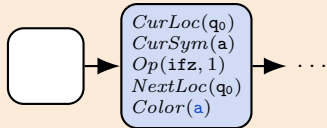
RTS $\Upsilon_{\mathcal{X}_{\mathcal{M}}} =$ 



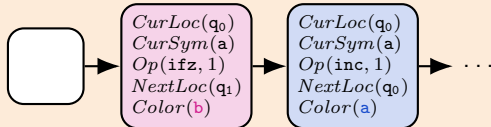
Proof details: examples of correct and spurious run prefixes



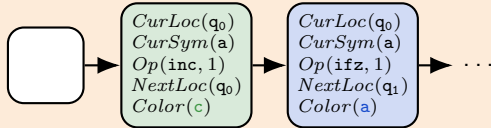
Spurious runs



\mathcal{M} does not have a transition from q_0 to q_0 labeled by instruction $\langle \text{ifz}, 1 \rangle$

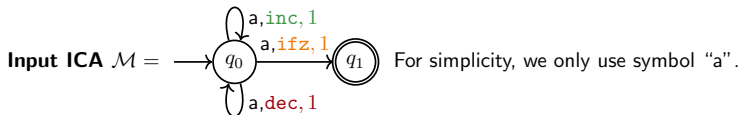


It is impossible to move to q_1 and then do a subsequent transition that starts from q_0

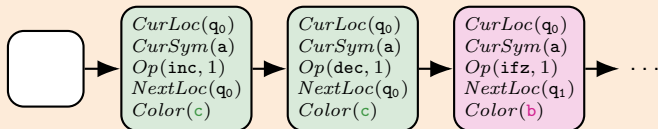


It is impossible to move by incrementing counter 1 and then move testing it for zero

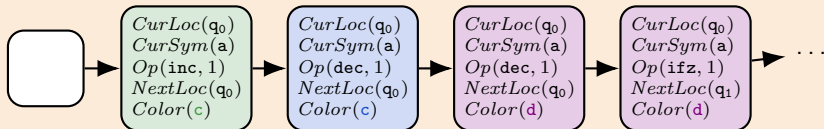
Proof details: examples of correct and spurious run prefixes



Proper runs



The run moves along proper transitions of \mathcal{M} . In particular, the counter (initially 0) is incremented, decremented, and then correctly tested for 0.



The run moves along proper transitions of \mathcal{M} . The counter (initially 0), is incremented and then decremented twice. This is ok: the counter is an unreliable ICA, so it may have leaked up before the second decrement!

Proof details: characterization of proper runs

From now on we focus on **infinite runs** of $\mathcal{X}_{\mathcal{M}}$.

We have already defined the notion of **proper run**. But... how can we **check** that a run is proper?

An infinite run $\tau = s_0 s_1 \dots$ of $\mathcal{X}_{\mathcal{M}}$ is **proper** if

- (**Location correctness**) The initial state s_0 has q_I as current location, and every two consequent states s_i and s_{i+1} agree on their intermediate location, i.e., the next location contained in s_i coincides with the current location contained in s_{i+1} , for every i .
- (**Control correctness**) Each state s_i holds information that reconstructs one control configuration update of \mathcal{M} .
- (**Counter correctness**) Every state s_i contains a counter operation that is actually executable in that state. For an ICA, increment and decrement operations are always possible. The only counter correctness check relates to test-for-zero: **a zero counter can be tested for zero in the future iff, in between, it is subject to at least as many decrements than increments**. If this is not the case, the counter is for sure positive.

Proof details: location and control correctness in $LTL\text{-}FO_A$

We encode location and control correctness as $LTL\text{-}FO_A$ formulae over \mathcal{X}_M .

Location correctness is encoded in $LTL\text{-}FO_A$ as:

$$\Phi_{\text{location}}^M = CurLoc(q_I) \wedge \mathbf{G} \left(\bigwedge_{q \in Q} (NextLoc(q) \rightarrow \mathbf{X} CurLoc(q)) \right)$$

Control correctness is encoded in $LTL\text{-}FO_A$ as:

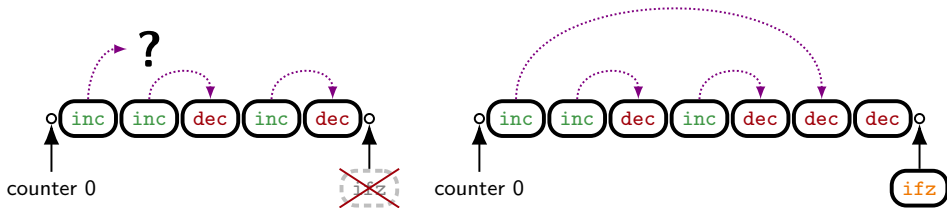
$$\Phi_{\text{control}}^M = \mathbf{G} \left(\bigvee_{\langle q, w, \langle op, i \rangle, q' \rangle \in \delta} (CurLoc(q) \wedge CurSym(w) \wedge Op(op, i) \wedge NextLoc(q')) \right)$$

Actually, Φ_{location}^M and Φ_{control}^M are just propositional LTL formulae.

Proof details: counter correctness mathematically

Counter correctness can be mathematically captured as follows:

- Consider a sequence of operations over the same counter, assuming that the counter is zero at the beginning of the sequence.
- Then, it can be successfully tested for zero at the end of the sequence only if **there exists an injection from the increment to the decrement operations** in the sequence: each increment must have a dedicated corresponding decrement.

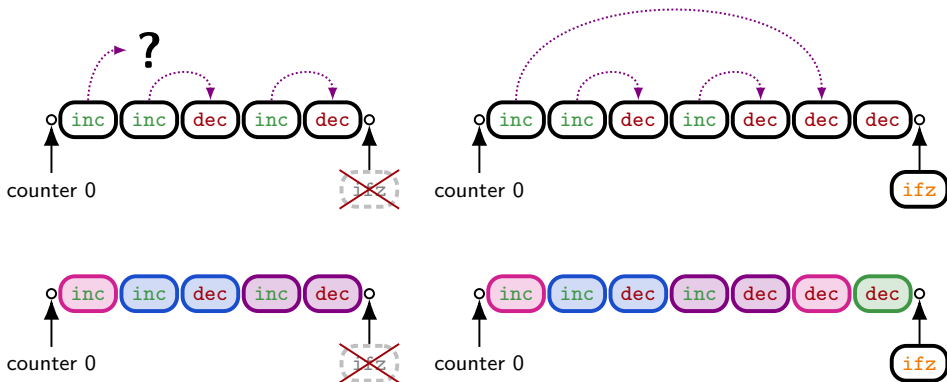


Note: for a MCA, this should be a **bijection**, ensuring that there are exactly as many decrements as increments!

Proof details: counter correctness via coloring

The **injection** can be expressed by suitably **coloring** the operations as follows:

- no two increments of the sequence get the same color;
- no two decrements of the sequence get the same color;
- each increment of the sequence must have a corresponding decrement within the sequence and with matching color.



Proof details: counter correctness in $LTL-FO_A$

We encode location and control correctness as an $LTL-FO_A$ formula over \mathcal{X}_M . We use the intuition of coloring given before, but with an additional difficulty: we have to identify the **sequences** of operations.

- Each is a sequence of operations over the same counter delimited by two zero tests over that counter, or by the initial state and the first zero test.

Counter correctness is encoded as $\Phi_{ifz}^M = \Phi_{inc}^M \wedge \Phi_{dec}^M \wedge \Phi_{match}^M \wedge \Phi_{order}^M$, where:

$$\Phi_{inc}^M =$$

$$\bigwedge_{i \in \{1, \dots, n\}} \mathbf{G} \forall x. \left(Op(\mathbf{inc}, i) \wedge Color(x) \rightarrow \neg \mathbf{X F} (Op(\mathbf{inc}, i) \wedge Color(x)) \right)$$

Different increment instructions for the same counter must be associated to different colors.

Proof details: counter correctness in $LTL-FO_A$

We encode location and control correctness as an $LTL-FO_A$ formula over \mathcal{X}_M . We use the intuition of coloring given before, but with an additional difficulty: we have to identify the **sequences** of operations.

- Each is a sequence of operations over the same counter delimited by two zero tests over that counter, or by the initial state and the first zero test.

Counter correctness is encoded as $\Phi_{ifz}^M = \Phi_{inc}^M \wedge \Phi_{dec}^M \wedge \Phi_{match}^M \wedge \Phi_{order}^M$, where:

$$\Phi_{dec}^M =$$

$$\bigwedge_{i \in \{1, \dots, n\}} \mathbf{G} \forall x. \left(Op(\mathbf{dec}, i) \wedge Color(x) \rightarrow \neg \mathbf{X F} (Op(\mathbf{dec}, i) \wedge Color(x)) \right)$$

Different decrement instructions for the same counter must be associated to different colors.

Proof details: counter correctness in $LTL-FO_A$

We encode location and control correctness as an $LTL-FO_A$ formula over \mathcal{X}_M . We use the intuition of coloring given before, but with an additional difficulty: we have to identify the **sequences** of operations.

- Each is a sequence of operations over the same counter delimited by two zero tests over that counter, or by the initial state and the first zero test.

Counter correctness is encoded as $\Phi_{ifz}^M = \Phi_{inc}^M \wedge \Phi_{dec}^M \wedge \Phi_{match}^M \wedge \Phi_{order}^M$, where:

$$\Phi_{match}^M =$$

$$\bigwedge_{i \in \{1, \dots, n\}} \mathbf{G} \left(\forall x. (Op(\text{inc}, i) \wedge Color(x) \wedge \mathbf{X F} Op(\text{ifz}, i)) \rightarrow \mathbf{X F} (Op(\text{dec}, i) \wedge Color(x)) \right)$$

Whenever there is an x -colored increment instruction for counter i that is eventually followed by a zero test instruction for i , then it must also be eventually followed by a corresponding decrement configuration for i with matching color x .

Proof details: counter correctness in $LTL-FO_A$

We encode location and control correctness as an $LTL-FO_A$ formula over \mathcal{X}_M . We use the intuition of coloring given before, but with an additional difficulty: we have to identify the **sequences** of operations.

- Each is a sequence of operations over the same counter delimited by two zero tests over that counter, or by the initial state and the first zero test.

Counter correctness is encoded as $\Phi_{ifz}^M = \Phi_{inc}^M \wedge \Phi_{dec}^M \wedge \Phi_{match}^M \wedge \Phi_{order}^M$, where:

$$\Phi_{order}^M =$$

$$\bigwedge_{i \in \{1, \dots, n\}} \neg \mathbf{F} \left(\begin{array}{l} Op(\text{inc}, i) \\ \wedge \mathbf{X} \mathbf{F} \left(\exists x. (Op(\text{ifz}, i) \wedge Color(x) \wedge \mathbf{X} \mathbf{F} (Op(\text{dec}, i) \wedge Color(x))) \right) \end{array} \right)$$

Φ_{match}^M does not guarantee that the matching decrement occurs within the same sequence, i.e., *before* the zero test. So, Φ_{order}^M ensures that an increment can match a decrement only if there is no zero test in between. This is expressed in a negative form, by forbidding to match an increment and a decrement of the same counter, if there is a zero test in between.

Proof details: putting everything together

We construct the LTL-FO_A formula $\Phi_{\mathcal{M}}$ stating that:

If the run of interest over $\mathcal{X}_{\mathcal{M}}$ is a **proper** run, **then** it **does not accept**, i.e., it does not recur infinitely often over an accepting location:

$$\Phi_{\mathcal{M}} = \left(\Phi_{\text{location}}^{\mathcal{M}} \wedge \Phi_{\text{control}}^{\mathcal{M}} \wedge \Phi_{\text{ifz}}^{\mathcal{M}} \right) \rightarrow \neg \mathbf{GF} \left(\bigvee_{q \in F} \text{CurLoc}(q) \right)$$

Given an ICA $\mathcal{M} \dots$

- Construct 1-bounded, unary DCDS $\mathcal{X}_{\mathcal{M}}$, and LTL-FO_A formula $\Phi_{\mathcal{M}}$.
- $\mathcal{X}_{\mathcal{M}} \models_{\text{LTL}} \Phi_{\mathcal{M}}$ **if and only if** for every infinite run τ of $\mathcal{X}_{\mathcal{M}}$, whenever τ reconstructs a **proper** sequence of control configuration updates of \mathcal{M} , then the corresponding run of \mathcal{M} is **non-accepting**.
- I.e., $\mathcal{X}_{\mathcal{M}} \models_{\text{LTL}} \Phi_{\mathcal{M}}$ **if and only if** \mathcal{M} over infinite words **is empty**.
- Hence, **LTL-FO_A model checking over 1-bounded, unary DCDSs is undecidable**.

Key observations

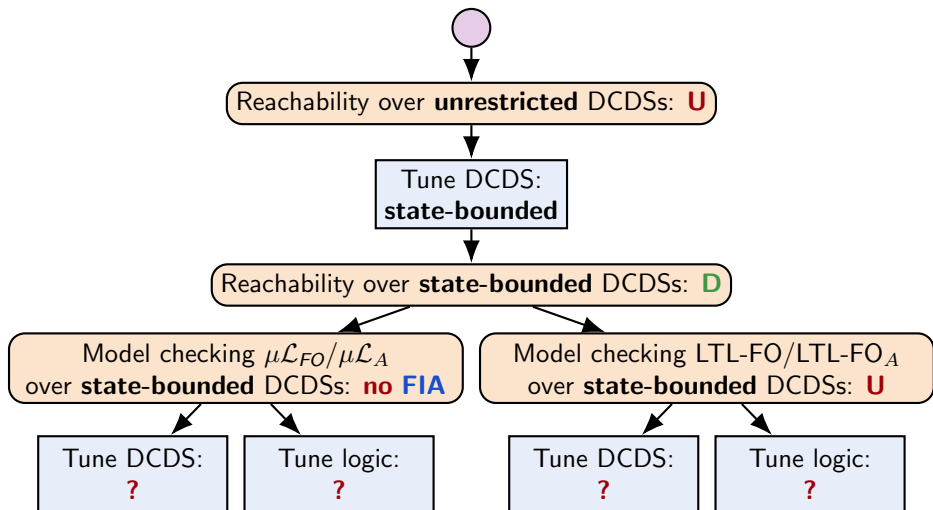
The **undecidability proof** relies on **two crucial points**:

- The only interesting feature of the DCDS \mathcal{X}_M is that it may encounter **unboundedly many different colors along a run**.
- The only interesting feature of the LTL-FO_A formula Φ_M is that it fully exploits **first-order quantification across time**.

Two open questions:

- 1 What happens if we limit the ability of the DCDS to encounter unboundedly many values along its runs? Which subclass of state-bounded DCDSs can capture this condition?
- 2 What happens if we control quantification across in the verification logic? In particular, what happens with LTL-FO_P?

Current overall picture



References

- [1] Dmitry Solomakhin et al. “Verification of Artifact-Centric Systems: Decidability and Modeling Issues”. In: vol. 8274. *Lecture Notes in Computer Science*. Springer, 2013, pp. 252–266.
- [2] Stéphane Demri and Ranko Lazic. “LTL with the Freeze Quantifier and Register Automata”. In: *ACM Trans. on Computational Logic* 10.3 (2009).
- [3] Richard Mayr. “Undecidable problems in unreliable computations”. In: *Theoretical Computer Science* 297.1-3 (2003), pp. 337–354.
- [4] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.