Freie Universität Bozen
Libera Università di Bolzano
Free University of Bozen · Bolzano

Faculty of Computer Science, Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy
Tel: +39 04710 16000, fax: +39 04710 16009

KRDB Research Centre Technical Report:

Query Stability in Data-aware Business Processes

Ognjen Savković[1], Elisa Marengo[1], Werner Nutt[1]

Affiliation	1: KRDB Research Centre for Knowledge and Data, Free University of Bozen-Bolzano
Corresponding author	Ognjen Savković: ognjen.savkovic@unibz.it
Keywords	Data Quality, Business Processes, Query Stability, Automated Verification
Number	KRDB15-1
Date	June 1, 2015
URL	http://www.inf.unibz.it/krdb/pub/tech-rep.php

Contents

1 Introduction

Data quality focuses on understanding how much data is fit for its intended use. This problem has been investigated in database research, considering aspects such as consistency, currency, and completeness [6, 11, 19]. One of the questions that these approaches do not consider or consider marginally, is where and how data originates and how it evolves. Even though in general a database may be updated in arbitrary ways, often data are manipulated according to some business process, implemented in an information system that accesses the DB. We believe that analyzing how business processes generate data allows one to gather additional information on their fitness for use.

In this work, we focus on a particular aspect of data quality, that is the problem whether a business process that reads from and writes into a database can affect the answer of a query or whether the answer will not change as a result of the process. We refer to this problem as *query stability*. The study of this problem has originally been motivated by the student enrollment at our university. We provide a running example inspired by this scenario.

Example. In November the student office distributed a report on the numbers of students enrolled in the offered courses. When comparing the numbers with those of the previous years, the Master in Computer Science (*mscCS*) showed a decrease, in contrast with other courses, like the Master in Economics (*mscEco*), that registered a substantial increase. The rector immediately called the head of the department and asked for an explanation. Apparently, there was no reason for such a decrease. After a long investigation, a secretary of the department discovered that the reason was a complication in the registration process, which foresees two routes to registration: a regular one and a second one via international federated study programs to which some courses, like the *mscCS*, are affiliated. Due to different deadlines, regular registration was concluded in November while registration for students from federated programs was not. Since the *mscCS* is affiliated to some federated programs, but the *mscEco* not, the query asking for all *mscEco* students was stable in November, while the query for all *mscCS* students was not and returned too low a number.

In situations like the above, it would have been possible to reason on query stability if not only the data were available, but also information on the processes and the way it manipulates the data. Having this information would make it possible to automate reasoning on stability and potentially integrate it into some information system. Organizations, for instance, often specify their business processes using standardised languages, such as BPMN, and rely on engines that can run those business processes (e.g., Bonita, Bizagi). However, in these systems how the data is manipulated by the process is hidden in the code, making difficult any reasoning on data quality.

Contributions Assessing query stability by leveraging on processes gives rise to several research questions. (1) What is a good model to represent processes, data and the interplay among the two? (2) What is the overhead of reasoning on query stability? (3) What are the characteristics of the model that may complicate the reasoning? (4) Are there existing mechanisms that can be leveraged to perform the reasoning?
(1) **Data-Aware Business Process Model.** Current approaches either focus on *process* modeling, representing the data in a limited way (like in Petri Nets [15]), or adopt a *data* perspective, leaving the representation of the process implicit [3, 5, 9]. We introduce a formalism called Data-Aware Business Processes (DABPs). In DABPs the process is represented as a graph. The interactions with an underlying database are expressed by annotating the graph with information on which data is read from the database and

which is written into it. DABPs model the possibility to have several process instances executing the process. New information can be brought into the process by starting a fresh process instance (Section 2).
(2) **Query Stability.** In the existing approaches the authors aim at the verification of general properties (e.g. temporal properties), for which reasoning is typically intractable [3, 8, 9]. In contrast, we identify a particular property, i.e. checking query stability for conjunctive queries (Section 3), and we identify interesting cases where the reasoning is tractable.
(3) **DABP Variants.** To understand the sources of complexity of our reasoning problem, we identify the following facets of DABPs: *(i)* a process can (or cannot) read from relations that it can write; *(ii)* negation is (is not) allowed in process conditions; *(iii)* a process can (cannot) have cycles; *(iv)* the process can (cannot) start with pending instances; *(v)* new instances can (cannot) start at any moment. Combinations of these facets define different variants of DABPs, for which we investigate the stability problem (Sections 3–5).
(4) **Datalog Encoding.** For each DABP variant we provide an encoding into a suitable variant of Datalog and we map the problem of query stability into the problem of query answering in Datalog. The encoding generates all the facts that can be produced in the process executions that are relevant w.r.t. stability. Over them it checks if any new query answer is produced. We prove that our approach is optimal w.r.t. worst case complexity in the size of the data, query and in the size of the entire input. Additionally, we introduce as measures the size of the process and of the running process instances. Our analysis identifies tractable cases of the stability problem and provides a possible way for implementation, using engines for SQL or for Datalog.

Related work and conclusion end the paper (Section 6).

A preliminary version of this paper was presented at the AMW workshop [18].

2 Data-aware Business Processes

In this section we introduce our formalism, named *Data-aware Business Processes* (DABPs), which allows us to represent business processes and the way they manipulate data. We rely on this formalism to perform reasoning on query stability.

Notation We adopt standard notation from databases. In particular, we assume an infinite set of relational symbols and an infinite set of constants dom as the *domain of values*, and the positive rationals $\mathbb{Q}^+$ as the *domain of timestamps*. A schema is a finite set of relation symbols. A *database instance* is a finite set of ground atoms, *facts*, over a schema and the *domain* $dom_{\mathbb{Q}^+} = dom \cup \mathbb{Q}^+$. We use upper-case letters for variables, lower-case for constants, and overline for tuples, e.g., $\bar{c}$.

A DABP is a pair $\mathcal{B} = \langle \mathcal{P}, \mathcal{C} \rangle$, consisting of a *process part* $\mathcal{P}$ and a *configuration part* $\mathcal{C}$. Intuitively, the process part is fixed. It defines how and under which conditions actions can change data stored in the configuration part. The configuration is a dynamic part consisting of *(i)* a database, and *(ii)* the process instances that traverse the process.

Process Part: Net The skeleton of the process part is a directed multigraph $N = \langle P, T \rangle$, the *process net*, consisting of a set of vertices P, the *places*, and a set of edges T, the *transitions*. There is one distinguished place in P, the start place *start*. A process instance traverses the graph, starting from *start*. The different transitions emanating from a place represent alternative developments of a process instance.

We use the distinguished relation symbol *In* to specify the input data associated

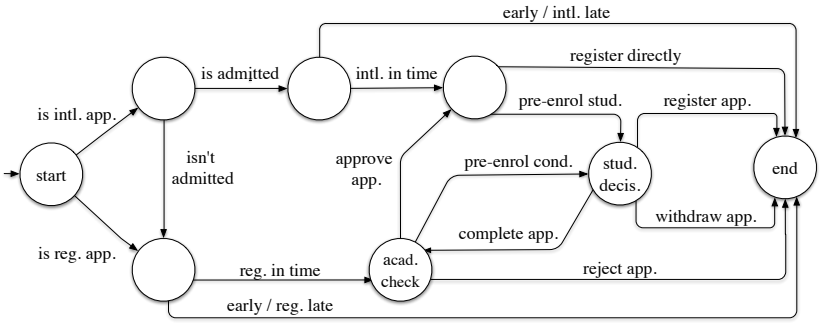


Figure 1: Student Registration Process Net $\mathcal{P}_{reg}$

with a process instance. The last argument of *In* is a timestamp, called the *start time*, to record the time when the process instance was started (e.g., $In(\bar{c}, \tau)$, where $\bar{c}$ is a tuple of constants from $dom_{\mathbb{Q}^+}$ and $\tau \in \mathbb{Q}^+$). We denote with $\Sigma_{\mathcal{B},In}$ and $\Sigma_{\mathcal{B}}$ the schemas of $\mathcal{B}$ with and without *In* resp.

Example. Let us consider our student registration process, denoted $\mathcal{B}_{reg} = \langle \mathcal{P}_{reg}, \mathcal{C}_{reg} \rangle$. Figure 1 reports the process net $\mathcal{P}_{reg}$. A process instance starts when a student submits a request via the on-line system, providing information such as the student name and the name of the course. Automatically, the system attaches a time stamp to the request. The application is then represented as an *In*-record $In(S, C, \mathcal{T})$, where S, C and $\mathcal{T}$ are variables for the student name, the course name and the timestamp resp.

The registration procedure distinguishes among two kinds of applications: regular and international. *International* programs are part of international federated programs involving different universities. In this case, an international commission decides about whom to admit. For *regular* programs, instead, it is the university itself that evaluates the applications and decides whether a candidate satisfies the necessary requirements to be admitted.

Accordingly, the process first checks the kind of application. If it is for an international course ('is intl. app.') then it checks if the student has already been admitted ('is admitted'). If not, it checks if the course is also open to regular applications ('isn't admitted'). If the application is for a regular course ('is reg. app.'), then the bottom branch of the process is followed.

The process is regulated by different deadlines, establishing the valid registration period. Registrations received early ('early') or late ('intl. late', 'reg. late') are not evaluated. The process also foresees a *pre-enrolment* phase. Applications received in this period can benefit of some flexibility in the evaluation procedure, like *(i)* the possibility to complete an application not fully complete w.r.t. the requirements and *(ii)* the possibility to confirm or withdraw the registration before being formally enrolled.

Accordingly, an application to an international course received in time ('intl. in time') is then immediately registered ('register directly') if not in the pre-enrolment phase. Otherwise, the student is pre-enrolled ('pre-enrol stud.') having the possibility ('stud. decis.') to later confirm ('register app.') or withdraw ('withdraw') the registration. In this case, the application fee the student paid will be reimbursed. This kind of interaction of the system with the external (e.g. human intervention or interaction with other systems) is modelled as a non-deterministic choice.

Transition	*Execution Condition* (E)
is intl. app.	$\mathit{In}(S, C, \mathcal{T}), \mathit{StudyPlan}(C, \mathit{intl}, P)$
is admitted	$\mathit{In}(S, C, \mathcal{T}), \mathit{AdmittedIntl}(S, C)$
isn't admitted	$\mathit{In}(S, C, \mathcal{T}), \neg\mathit{AdmittedIntl}(S, C), \mathit{StudyPlan}(C, \mathit{reg}, P)$
early	$\mathit{In}(S, C, \mathcal{T}), \mathit{Deadline}(\mathit{start}, \mathcal{T}_{\mathit{start}}), \mathcal{T} < \mathcal{T}_{\mathit{start}}$
intl. late	$\mathit{In}(S, C, \mathcal{T}), \mathit{Deadline}(\mathit{intl}, \mathcal{T}_{\mathit{intl}}), \mathcal{T} > \mathcal{T}_{\mathit{intl}}$
intl. in time	$\mathit{In}(S, C, \mathcal{T}), \mathit{Deadline}(\mathit{start}, \mathcal{T}_{\mathit{start}}), \mathit{Deadline}(\mathit{intl}, \mathcal{T}_{\mathit{intl}}), \mathcal{T}_{\mathit{start}} \leq \mathcal{T} \leq \mathcal{T}_{\mathit{intl}}$
register directly	$\mathit{In}(S, C, \mathcal{T}), \mathit{Deadline}(\mathit{pre}, \mathcal{T}_{\mathit{pre}}), \mathcal{T} > \mathcal{T}_{\mathit{pre}}$
pre-enrol stud.	$\mathit{In}(S, C, \mathcal{T}), \mathit{Deadline}(\mathit{pre}, \mathcal{T}_{\mathit{pre}}), \mathcal{T} \leq \mathcal{T}_{\mathit{pre}}$
register app.	$\mathit{In}(S, C, \mathcal{T}), \mathit{Pre\text{-}enrolled}(S, C)$
is reg app.	$\mathit{In}(S, C, \mathcal{T}), \mathit{StudyPlan}(C, \mathit{reg}, P), \neg\mathit{StudyPlan}(C, \mathit{intl}, P)$
reg. in time	$\mathit{In}(S, C, \mathcal{T}), \mathit{Deadline}(\mathit{start}, \mathcal{T}_{\mathit{start}}), \mathit{Deadline}(\mathit{reg}, \mathcal{T}_{\mathit{reg}}), \mathcal{T}_{\mathit{start}} \leq \mathcal{T} \leq \mathcal{T}_{\mathit{reg}}$
reg. late	$\mathit{In}(S, C, \mathcal{T}), \mathit{Deadline}(\mathit{reg}, \mathcal{T}_{\mathit{reg}}), \mathcal{T} > \mathcal{T}_{\mathit{reg}}$
pre-enrol cond.	$\mathit{In}(S, C, \mathcal{T}), \mathit{Deadline}(\mathit{pre}, \mathcal{T}_{\mathit{pre}}), \mathcal{T} \leq \mathcal{T}_{\mathit{pre}}$
complete app.	$\mathit{In}(S, C, \mathcal{T}), \mathit{Conditional}(S, C)$
withdraw app. = approve app. = reject app. : *true*	

Transition	*Writing Rule* (W)
register directly	$\mathit{Registered}(S, C) \leftarrow \mathit{In}(S, C, \mathcal{T})$
pre-enrol	$\mathit{Pre\text{-}enrolled}(S, C) \leftarrow \mathit{In}(S, C, \mathcal{T})$
pre-enrol cond.	$\mathit{Conditional}(S, C) \leftarrow \mathit{In}(S, C, \mathcal{T})$
register app.	$\mathit{Registered}(S, C) \leftarrow \mathit{In}(S, C, \mathcal{T})$

Table 1: Execution Condition and Writing rules in $\mathcal{P}_{reg}$

A regular application received in time ('reg. in time') is then evaluated by the academic staff ('acad. check'), the outcome of which can be that the application is: *(i)* approved ('approve app.'); *(ii)* rejected ('reject app.'); or *(iii)* the student can be conditionally pre-enrolled ('pre-enrol cond.'), which is only possible during the pre-enrolment phase. In the latter case, the student can submit the missing document later ('complete app.'). △

Process Part: Labeling Function The whole process part is a pair $\mathcal{P} = \langle N, L \rangle$, which in addition to a network N comprises a *labeling function* L that assigns to every transition $t \in T$ a pair $L(t) = (E_t, W_t)$. Here, E_t, the *execution condition,* is a Boolean query over $\Sigma_{\mathcal{B},\mathit{In}}$ and W_t, the *writing rule,* is a rule $R(\bar{x}) \leftarrow B_t(\bar{x})$ whose head is a relation of $\Sigma_{\mathcal{B}}$ and whose body is a $\Sigma_{\mathcal{B},\mathit{In}}$-query that has the same arity as the head relation. Evaluating W_t over a $\Sigma_{\mathcal{B},\mathit{In}}$-instance $\mathcal{D}$ results in the set of facts $W_t(\mathcal{D}) = \{R(\bar{c}) \mid \bar{c} \in B_t(\mathcal{D})\}$. Intuitively, E_t specifies in which state of the database which process instance can perform the transition t and W_t specifies which new information is (or can be) written into the database when performing t. In this paper we assume that E_t and B_t are conjunctive queries possibly with negated atoms and comparisons involving timestamps.

Example. Table 1 reports the execution conditions and writing rules for the registration process $\mathcal{P}_{reg}$. Consider, for instance, the transition 'is intl. app.' which should be enabled only for applications to courses associated to international programs. This is captured by the condition $\mathit{In}(S, C, \mathcal{T}), \mathit{StudyPlan}(C, \mathit{intl}, P)$, that checks if the course C of the application ($\mathit{In}(S, C, \mathcal{T})$), is stored in the relation *StudyPlan* and associated to a program P whose type is *intl*. The transitions 'isn't admitted', 'is admitted', 'register app.', 'is reg. app.' and 'complete app.' have conditions of a similar kind. The other

transitions are labeled with conditions checking that the application satisfies the foreseen deadlines, stored in the database. For instance, 'pre-enrol cond.' should be enabled for applications received in the pre-enrolment phase. Thus, the condition reads the deadline from the database ($\mathit{Deadline}(\mathit{pre}, \mathcal{T}_{\mathit{pre}})$) and checks that the application has been submitted before it ($\mathcal{T} \leq \mathcal{T}_{\mathit{pre}}$).

The transitions 'register directly', 'pre-enrol', 'pre-enrol cond.' and 'register app.' are also labelled with a writing rule. For instance, when 'pre-enrol cond.' is traversed, then the process records that the student application is accepted conditionally, by storing it in relation *Conditional* ($\mathit{Conditional}(S, C) \leftarrow \mathit{In}(S, C, \mathcal{T})$). △

Configuration Part This part models the dynamic part of the process: process instances and the database. Formally, a configuration is a triple $\langle \mathcal{I}, \mathcal{D}, \tau \rangle$, where $\mathcal{I}$ defines the process instances; $\mathcal{D}$ is a database instance over $\Sigma_{\mathcal{B}}$; τ is a timestamp, the current time.

Each instance in $\mathcal{I}$ is associated with a single *In*-record storing the data carried by the instance. New constants from the infinite domain can be brought into the database by starting new instances. An *In*-record is created when the instance starts and cannot be changed later on. Also, the input record of an instance is only visible for that instance while the database is shared among all instances.

Formally, instance part is a triple $\mathcal{I} = \langle O, M_{\mathit{In}}, M_P \rangle$ where $O = \{o_1, ..., o_k\}$ is a set of process instances (objects); M_{In} is a mapping that maps each $o \in O$ to an *In*-record $\mathit{In}(\bar{c}, \tau) = M_{\mathit{In}}(o)$; M_P is a mapping that maps each $o \in O$ to a current place $M_P(o) \in P$.

For convenience, we also use the notation $\mathcal{B} = \langle \mathcal{P}, \mathcal{I}, \mathcal{D}, \tau \rangle$, $\mathcal{B} = \langle \mathcal{P}, \mathcal{I}, \mathcal{D} \rangle$ (when τ is not relevant) and $\mathcal{B} = \langle \mathcal{P}, \mathcal{D} \rangle$ (for a process that is initially without running instances).

Example. Table 2*(a)* show a database instance $\mathcal{D}_{\mathit{reg}}$ for our running example. Courses offered by the university are stored in the relation *StudyPlan*, together with information on their type (*intl* or *reg*) and the program they are associated. Relation *Deadline* stores the date for the starting of the registration period (*start*), and the end for the pre-enrolment (*pre*), regular (*reg*) and international (*intl*) enrolments. The remaining tables store information about the students. Relation *AdmittedIntl* stores the students admitted to an international course. Students that already completed the registration and that are successfully pre-enrolled or registered are stored in relations *Pre-enrolled* and *Registered* resp, while those that are accepted conditionally are stored in *Conditional*.

Table 2*(b)* reports the running process instances $\mathcal{I}_{\mathit{reg}}$ in the form of a relation. Currently four student applications are in the process. The instance with id o_4, for instance, is associated to John's request for the "db" program, which has been received on 4th Nov. The instance is currently at place *start*. △

Timestamps stored in the database are foreseen to be a rather static part that is created manually at the design phase of the process. At this point we assume that DABPs do not allow process instances to write timestamps into database relations that are joined with comparisons in the process rules. In Section 4.1 we discuss how this restriction allows us to simplify processes and how, in principle, they can be simplified without this restriction.

Execution of a DABP Let $\mathcal{B} = \langle \mathcal{P}, \mathcal{C} \rangle$ be a DABP, with current configuration $\mathcal{C} = \langle \mathcal{I}, \mathcal{D}, \tau \rangle$. There are two kinds of *atomic execution* steps of a DABP: *(i)* the *traversal* of a transition in the net by an instance and *(ii)* the *introduction* of a new instance.

***(i)* Traversal** of an instance. Consider an instance $o \in O$ with $M_{\mathit{In}}(o) = \mathit{In}(\bar{c}, \tau')$ and

StudyPlan

course	*type*	*program*
compLogic	intl	mscCS
compLogic	reg	mscCS
db	reg	mscCS
econ	reg	mscEco

Deadline

label	*date*
start	1st Sep
pre	30th Sep
reg	31st Oct
intl	31st Dec

AdmittedIntl

student	*course*
bob	compLogic
mary	compLogic

Pre-enrolled

student	*course*
bob	compLogic
alice	econ

Conditional

student	*course*
paul	econ

Registered

student	*course*
bob	compLogic
alice	econ

***(a)** Database Instance $\mathcal{D}_{reg}$*

Instances and mapping

id	*In*-record	place
o_4	(john, db, 4th Nov)	*start*
o_3	(paul, econ, 24th Sep)	*end*
o_2	(alice, econ, 20th Sep)	*end*
o_1	(bob, compLogic, 5th Sep)	*end*

***(b)** Process Instances $\mathcal{I}_{reg}$*

Table 2: Database and Instance Representation of the Student Registration Process.

$M_P(o) = q$. That is, o is at place q and $\mathit{In}(\bar{c}, \tau')$ is the input data of o. Let t be a transition from q to p, with execution condition E_t. Then t is *enabled* for o, i.e., it can *traverse* it, if E_t evaluates to *true* over the database $\mathcal{D} \cup \{\mathit{In}(\bar{c}, \tau')\}$. Let $W_t\colon R(\bar{x}) \leftarrow B_t(\bar{x})$ be the writing rule of t. Then the effect of o traversing t is the transition from $\mathcal{C} = \langle \mathcal{I}, \mathcal{D}, \tau \rangle$ to a new configuration $\mathcal{C}' = \langle \mathcal{I}', \mathcal{D}', \tau \rangle$, such that *(i)* the set of instances O and the current time τ is the same; *(ii)* $\mathcal{D}' = \mathcal{D} \cup W_t(\mathcal{D} \cup \{\mathit{In}(\bar{c}, \tau')\})$ is the new database, and *(iii)* $\mathcal{I} = \langle O, M_{\mathit{In}}, M_P \rangle$ is updated to $\mathcal{I}' = \langle O, M_{\mathit{In}}, M'_P \rangle$ reflecting the change of place for the instance o, that is $M'_P(o) = p$ and $M'_P(o') = M_P(o')$ for all other instances o'.

(ii) **Introduction** of an arbitrary instance at the *start* place. Let o' be a fresh instance and let $\mathit{In}(\bar{c}', \tau')$ be an *In*-record where the timestamp τ' is greater or equal than τ, the current time of $\mathcal{C}$. Note that the constants in $\bar{c}'$ need not appear in the database or in the process. The result of introducing o' with info $\bar{c}'$ at time τ' is the configuration $\mathcal{C}' = \langle \mathcal{I}', \mathcal{D}, \tau' \rangle$ such that $\mathcal{I}' = \langle O', M'_{\mathit{In}}, M'_P \rangle$ where: *(i)* the database instance is the same as in $\mathcal{C}$; *(ii)* the set of instances $O' = O \cup \{o'\}$ is augmented by o'; and *(iii)* the mappings M'_{In} and M'_P are an extension of M_{In} and M_P resp., obtained by defining $M'_{\mathit{In}}(o') = \mathit{In}(\bar{c}', \tau')$; $M'_P(o') = \mathit{start}$; $M'_{\mathit{In}}(o) = M_{\mathit{In}}(o)$ and $M'_P(o) = M_P(o)$ for all $o \in O$.

An *execution* Υ in $\mathcal{B} = \langle \mathcal{P}, \mathcal{C} \rangle$ is a finite sequence of configurations $\mathcal{C}_1, \ldots, \mathcal{C}_n$ *(i)* starting with $\mathcal{C}$ $(= \mathcal{C}_1)$, and *(ii)* where each next configuration $\mathcal{C}_{i+1}$ is obtained from $\mathcal{C}_i$ by making an atomic execution step. We denote Υ also with $\mathcal{C}_1 \rightsquigarrow \cdots \rightsquigarrow \mathcal{C}_n$. We say that the execution Υ *produces* facts $A_1, \ldots, A_n$ if the last configuration $\mathcal{C}_n$ in Υ contains $A_1, \ldots, A_n$.

Since at each step fresh instances can start and write new data *(i)* there are infinitely many possible executions and *(ii)* database may grow in an unbounded way over time.

3 The Query Stability Problem

In this section we introduce the main problem of our investigation, that is the problem of query stability in DABPs.

Definition 1 (Query Stability)**.** *For a given DABP $\mathcal{B} = \langle \mathcal{P}, \mathcal{C} \rangle$ with database instance $\mathcal{D}$, a given query Q and timestamp τ, we say that Q is* stable *in $\mathcal{B}$ till the time-point τ, if for any execution $\Upsilon = \mathcal{C} \rightsquigarrow \cdots \rightsquigarrow \mathcal{C}'$ in $\mathcal{B}$ where $\mathcal{C}'$ has database $\mathcal{D}'$ and timestamp τ' such that $\tau' < \tau$, it holds:*

$$Q(\mathcal{D}) = Q(\mathcal{D}').$$

In the above definition, we allow τ to be ∞, the supremum of the domain $\mathbb{Q}^+$. If a query is stable till time point ∞, we say it is *globally stable*, or simply stable. The decision problem that we investigate is: *given a DABP $\mathcal{B}$, a query Q, and a timestamp τ, can we decide if Q is stable in $\mathcal{B}$ till time-point τ?* Given a set of intervals we would like to analyze the stability of a query for each interval. We imagine that having such analysis can be useful for decision makers to have a better understanding of how reliable, in particular how stable, the data is in each interval.

Example. Consider again the student registration example and the queries

$$Q_{\text{cs}}(S) \leftarrow \textit{Registered}(S, C), \textit{StudyPlan}(C, \textit{mscCS}, P), \text{ and}$$
$$Q_{\text{eco}}(S) \leftarrow \textit{Registered}(S, C), \textit{StudyPlan}(C, \textit{mscEco}, P)$$

that ask for the students registered at the master in CS, and the master in Economics, respectively. We analyze the stability of the two queries in different periods, for each of them assuming that the current time of the DABP is within the period. Table 3 shows the results. If the current time is before the 1st Sept., both programs do not

Interval	*< Sept*	*Sept - Oct*	*Nov - Dec*	*> Dec*
Q_{eco}	stable	instable	stable	stable
Q_{cs}	stable	instable	instable	stable

Table 3: Stability of the queries for different intervals.

allow registrations and thus they are stable in this interval, even though they are instable globally since both programs allow for arbitrary new registrations in the next interval (Sept.-Oct.). If the current time is within the interval (Nov.-Dec.) the query Q_{eco} is stable because the program *mscEco* is not affiliated to any international (*intl*) course and the deadline for the regular programs has passed. On the contrary, *mscCS* has an affiliated course to which student Mary is admitted. However, she is not registered yet and potentially she could submit an application that, if before the 31st Dec., would be accepted. Thus Q_{cs} is not stable in this interval. If the admitted students were all registered beforehand, then the query would be stable since no new registration would be possible. The query would be stable also in the case the process is closed for new instances to start (e.g., because the limit on registered students has been reached). In this case, only running instances are allowed to finish their execution. Thus, candidate *Mary* would not be able to register even though she is admitted. If the current time is

after the 31^{st} Dec., both queries are stable regardless if the process is open or closed because all the registration deadlines have expired. $\triangle$

3.1 Facets of DABPs

To investigate the sources of complexity and provide suitable encodings into Datalog, we introduce several facets of DABPs.

General vs. Rowo This facet indicates whether the model allows a process instance to read the facts that itself or another instance has written into the database. In the *general* variant this is allowed, while it is not in the restricted variant, called *rowo* (Read-Only Write-Only). Formally, rowo DABPs split the schema Σ into two disjoint schemas: the reading schema Σ_r and the writing schema Σ_w, such that the execution conditions and the queries in the writing rules range over Σ_r while the heads range over Σ_w.

The DABP in our example is general, since the relations *Pre-enrolled* and *Conditional* are both read and written by the process.

Normal vs. Positive In principle, we allow queries in execution conditions and writing rules to include safe negation. We identify these processes as *normal*, to distinguish them from *positive* processes where negation is not allowed.

Our example is normal, though only with negation on database relations that are not updated by the process.

Cyclic vs. Acyclic In business process management, repetitive actions are often modeled using cycles. For *cyclic* DABPs, it is possible that cycles occur in the process net, as in our running example, while cycles are not allowed in *acyclic* DABPs.

Arbitrary vs. Fresh A DABP is *fresh* if its initial configuration does not contain any running instance. If it may contain running instances, we say it is *arbitrary*.

In our scenario, we can imagine that at the beginning of the registration period the process starts with a fresh configuration (i.e. no running applications). Starting from an arbitrary configuration may be needed to handle exceptions in the registration process. For instance, a regular application received after the deadline for a valid reason, may be placed by a secretary at a certain place in the process that it would not be able to reach starting form the *start* place.

Open vs. Closed Semantics A process under *open semantics* allows new instances to start at any moment, while under *closed semantics* only transition traversals are possible (no new instances can be started). In the closed variant, stability of a query depends only on the unfinished instances, while in open processes, it depends also on the new instances that may start. As a consequence, under closed semantics the only interesting case is the arbitrary one, since a query is trivially stable under fresh configuration.

Our example runs under open semantics. One can imagine that after the last deadline (31^{st} Dec.) the web form for submitting new applications will be no more available. Then the process will run under closed semantics.

Checking Stability in DABP Variants We note that the problem of stability till a time-point τ can be reduced to the problem of checking whether the query is *globally* stable. To achieve this, one can adapt a given DABP by adding a new *start* place and connecting it to the old *start* place via a transition that is enabled only for instances with timestamp smaller than τ. In this way, any query Q would be globally stable in the resulting DABP iff it is stable in the original DABP till time-point τ.

The decision problem of query stability considers as an input a process model $\mathcal{P}$, a configuration $\mathcal{C}$, a query Q and a semantics s under which the DABP $\langle \mathcal{P}, \mathcal{C} \rangle$ executes. The question is:

Is Q globally stable in $\langle \mathcal{P}, \mathcal{C} \rangle$ under semantics s?

For convenience, hereafter we will omit the semantics when it is clear from the context, or we abbreviate it into open (closed) DABPs.

Singleton DABPs A singleton DABP is a closed DABP with a single instance at the *start* place in the initial configuration.

3.2 Summary of Complexity Results

Table 4 summarizes the DABP variants and the corresponding complexities.

Model	Rules	Net	Semantics	Config.	Data	Proc. Instance	Process	Query	Combined	Sect.
general	*normal*	*(a)cyclic*	*open*	*arbitrary*	UNDEC.	UNDEC.	UNDEC.	UNDEC.	UNDEC.	4.3
		(a)cyclic	*open*	*fresh*	UNDEC.	—	UNDEC.	UNDEC.	UNDEC.	4.3
		cyclic	*closed*	*arbitrary*	CO-NP	CO-NP	CO-NEXPTIME	Π_2^P	CO-NEXPTIME	4.8
		acyclic	*closed*	*arbitrary*	in AC^0	CO-NP	PSPACE	Π_2^P	PSPACE	4.4
	positive	*(a)cyclic*	*open*	*arbitrary*	PTIME	CO-NP	EXPTIME	Π_2^P	EXPTIME	4.5, 4.7
		(a)cyclic	*open*	*fresh*	PTIME	—	EXPTIME	Π_2^P	EXPTIME	4.2
		cyclic	*closed*	*arbitrary*	PTIME	CO-NP	EXPTIME	Π_2^P	EXPTIME	4.6
		acyclic	*closed*	*arbitrary*	in AC^0	CO-NP	PSPACE	Π_2^P	PSPACE	4.4
rowo	*	*	*	*	in AC^0	in AC^0	CO-NP	Π_2^P	Π_2^P	5

Table 4: Computational complexities of checking stability in DABPs variants. All decidable variants have matching complexities except for AC^0. All decidable results in the size of data, process, query and combined hold already for singleton DABPs. The symbol * indicates that the results for rowo hold for all non-trivial DABP variants.

4 Stability in General Data-aware Business Process Model

4.1 Abstraction Principles

Under open semantics fresh instances can bring an arbitrary number of new constants into the database with their input. Thus, processes can produce arbitrarily many new facts.

Here we introduce two abstraction principles. The first one explains how to faithfully abstract an unbounded number of timestamps into a finite number of representative ones, and based on that how to rewrite timestamp comparisons in the conditions into equivalent ones, however on database relations. The second principle explains how to faithfully abstract infinitely many executions into finitely many. The second principle holds only for positive and rowo DABPs.

Abstraction Principle for Timestamps. Let $\mathcal{B} = \langle \mathcal{P}, \mathcal{I}, \mathcal{D}, \tau_{\mathcal{B}} \rangle$ be a DABP. Based on $\mathcal{B}$ we construct a DABP $\mathcal{B}' = \langle \mathcal{P}', \mathcal{I}', \mathcal{D}' \rangle$ that produces the same facts as $\mathcal{B}$ but abstracts away timestamps. In particular, $\mathcal{B}'$ *(i)* replaces all comparisons on the timestamps with conditions on database relations, and *(ii)* adapts DABP executions so that fresh instances do not update the current time of configuration. In this way, we obtain a DABP that is easier to analyze.

Let $\tau_1, \ldots, \tau_n$ be the timestamps from $\mathcal{D}$ including $\tau_{\mathcal{B}}$ such that $\tau_i < \tau_{i+1}$. We introduce new timestamps $\tau'_0, \ldots, \tau'_n$ such that

$$\tau'_0 < \tau_1 < \tau'_1 < \cdots < \tau_n < \tau'_n.$$

To make the information about comparisons $<$ and $\leq$ available in the database $\mathcal{D}'$, we introduce binary relations $R_<$ and $R_\leq$, resp. Then we populate those relations with pairs of timestamps among $\tau'_0, \tau_1, \ldots, \tau'_n$ such that for every two such timestamps $\tau < \tilde{\tau}$ the fact $R_<(\tau, \tilde{\tau})$ is added to $\mathcal{D}'$ (and similarly for $\leq$). The rest of $\mathcal{D}'$ is the same as $\mathcal{D}$. Then, we adapt the process part $\mathcal{P}$ such that each $<$ and $\leq$ is replaced with $R_<$ and $R_\leq$ resp. To ensure that only instances with timestamps from $\mathcal{D}'$ that are greater or equal than $\tau_{\mathcal{B}}$ execute the process, we add a condition checking this on each outgoing transition from *start*.

Then, we introduce the *discretization* function $\delta_{\mathcal{D}'} : dom_{\mathbb{Q}^+} \to dom_{\mathbb{Q}^+}$ that based on the timestamps from $\mathcal{D}'$ "discretizes" $\mathbb{Q}^+$ as follows: *(i)* for $\tau \in dom_{\mathbb{Q}^+}$ we define $\delta_{\mathcal{D}'}(\tau) = \tau$ if $\tau = \tau_i$ for some i; *(ii)* $\delta_{\mathcal{D}'}(\tau) = \tau'_i$ if $\tau_i < \tau < \tau_{i+1}$ for some i; *(iii)* $\delta_{\mathcal{D}'}(\tau) = \tau'_0$ if $\tau < \tau_1$; *(iv)* $\delta_{\mathcal{D}'}(\tau) = \tau'_n$ if $\tau_n < \tau$; *(v)* for $a \in dom$ we define $\delta_{\mathcal{D}'}(a) = a$. We extend $\delta_{\mathcal{D}'}$ to all syntactic objects containing constants, including executions. We set $\mathcal{I}' = \delta_{\mathcal{D}'}\mathcal{I}$.

Proposition 1 (Abstraction of Timestamps)**.** *Let $\Upsilon = \mathcal{C} \rightsquigarrow \mathcal{C}_1 \rightsquigarrow \cdots \rightsquigarrow \mathcal{C}_m$ be an execution in $\mathcal{B}$ that produces a set of facts W, and let $\Upsilon' = \delta_{\mathcal{D}'}\Upsilon = \delta_{\mathcal{D}'}\mathcal{C} \rightsquigarrow \delta_{\mathcal{D}'}\mathcal{C}_1 \rightsquigarrow \cdots \rightsquigarrow \delta_{\mathcal{D}'}\mathcal{C}_m$. Further, let Υ'' be an execution in $\mathcal{B}'$ that produces a set of facts W'. It holds:*

a) Υ' is an execution in $\mathcal{B}'$ that produces W;

b) There exists an execution Υ''' in $\mathcal{B}$ that produces W'.

In other words, original and abstracted DABPs produce the same facts, and thus have the same impact on stability.

The claim *a)* is a consequence of disallowing timestamps to be written. Intuitively, it holds because any two timestamps τ and τ' (introduced by fresh instances) that

are between two consecutive timestamps from $\mathcal{D}$, say $\tau_i < \tau, \tau' < \tau_{i+1}$, cannot be distinguished by the process rules. Hence, it is enough to consider only one fresh timestamp for each interval (τ_i, τ_{i+1}).

The executions in $\mathcal{B}$ must introduce instances with increasing timestamps, while $\mathcal{B}'$ need not. Then, the execution Υ''' in claim *b)* can be obtained from Υ'' by first executing the introduction steps in the timestamps order, and then executing the traversal steps.

The above property gives a direct procedure of how to rewrite comparisons from rules as database relations. In principle, the above abstraction procedure can be extended for the case when writing of timestamps is allowed. However, the abstraction procedure would be much more complex as one would need at least exponentially many new timestamps to faithfully simulate all cases.

From now on, we focus on DABPs without comparisons, and we assume that timestamps are part of the domain dom. The obtained results propagate to variants with comparisons.

Abstraction Principle for Fresh Constants. Let $\mathcal{B} = \langle \mathcal{P}, \mathcal{I}, \mathcal{D} \rangle$ be a positive DABP that executes under open semantics. Let Q be a query that we want to check for stability. Based on $\mathcal{B}$ and Q we define the active domain *adom* as the set of all constants that appear in them. We observe that a positive execution condition evaluates to true for some fresh instance already if the input record of the instance takes constants from the active domain. Still, to produce a new query answer, the constants from the active domain may not be enough (in principle, a query may return all possible tuples formed with constants from the active domain). We show that it is enough to consider at most one new constant, in addition to the active domain. We can reason similarly about rowo DABPs.

Let b be a new constant such that $b \notin$ *adom*, and let *adom*$^* =$ *adom* $\cup \{b\}$ be the *extended active domain*. Based on the extended active domain, we define the *abstraction function* $\alpha^b_{adom} \colon dom \to dom$ such that $\alpha^b_{adom}(c) = c$ iff $c \in$ *adom*; otherwise $\alpha^b_{adom}(c) = b$. In other words, the abstraction function maps all constants from the active domain to themselves, and all the rest to b. The abstraction function is straightforwardly extended to all instances that contain constants. Now we are ready to introduce the abstraction principle that holds for all positive and rowo DABPs.

Proposition 2 (Abstraction of Fresh Constants)**.** *Let* $\Upsilon = \mathcal{C} \rightsquigarrow \mathcal{C}_1 \rightsquigarrow \cdots \rightsquigarrow \mathcal{C}_m$ *be an execution in* $\mathcal{B}$ *that produces a set facts* W*. Let* $\Upsilon' = \alpha^b_{adom}\Upsilon = \alpha^b_{adom}\mathcal{C} \rightsquigarrow \alpha^b_{adom}\mathcal{C}_1 \rightsquigarrow \cdots \rightsquigarrow \alpha^b_{adom}\mathcal{C}_m$*, then:*

a) Υ' *is an execution in* $\mathcal{B}$ *that produces* $\alpha^b_{adom}W$*;*

b) $Q(\mathcal{D}) \neq Q(\mathcal{D} \cup W)$ *iff* $Q(\mathcal{D}) \neq Q(\mathcal{D} \cup \alpha^b_{adom}W)$.

In other words, each execution in $\mathcal{B}$ can be α^b_{adom}-abstracted and it will still be an execution, and more importantly, such execution produces a new query answer if and only if the α^b_{adom}-abstracted version produces a new query answer.

Based on the abstraction principle for fresh constants we show how to use Datalog reasoning to check query stability.

4.2 Positive (A)cyclic Fresh Open

In this section we describe how to construct a Datalog program from a given positive, possibly cyclic, fresh DABP under open semantics and a query that checks stability of the query.

Assume we are given a positive fresh DABP $\mathcal{B} = \langle \mathcal{P}, \mathcal{D} \rangle$ and query Q. The encoding program consists of two parts.

i) First part is a program $\Pi^{po,fr}_{\mathcal{P},Q}$ that starting from the initial database computes the maximal extended database that the process $\mathcal{B}$ can produce over the extended active domain. In order to distinguish between facts produced by the process and the facts from the initial database, we introduce a new signature $\Sigma'_{\mathcal{B}}$ which is the same as $\Sigma_{\mathcal{B}}$ except that each relation $R \in \Sigma_{\mathcal{B}}$ is renamed into primed version R' of the same arity. Program $\Pi^{po,fr}_{\mathcal{P},Q}$ uses primed signature.

ii) Then, we construct a program Π^{test}_{Q} that checks if the query Q has a new answer over the maximal extended database.

Encoding into Datalog

To record which fresh instances can reach a place p in $\mathcal{P}$, we introduce relation In_p of size In. That is, $\mathit{In}_p(\bar{s})$ evaluates to true in the program iff an instance with the same $\mathit{In}(\bar{s})$ record can reach p.

Introduction Rule. Initially, all relevant fresh instances (those taking constants from adom^*) reach *start* place. To encode this we introduce the following *introduction rule*:

$$\mathit{In}_{\mathit{start}}(X_1, \ldots, X_n) \leftarrow \mathit{adom}^*(X_1), \ldots, \mathit{adom}^*(X_n).$$

Here, with slight abuse of notation, adom^* represents a unary relation that we initially instantiate with the constants from adom^*.

Copy Rules. Also initially we copy all database facts into the primed signature of $\mathcal{P}$, that is for each relation $R \in \Sigma_{\mathcal{P}}$ we introduce the *copy rule*:

$$R'(\overline{X}) \leftarrow R(\overline{X}).$$

Traversal Rule. Now we want to encode instance traversals. For every transition t that goes from a place q to p we introduce a *traversal rule* that copies all instances that reached q into those that reached p provided that the instances satisfy the execution condition for t.

Let $E_t \colon R_1(\bar{s}_1), \ldots, R_m(\bar{s}_m), \mathit{In}(\bar{s})$ be the execution condition for t, then we use $E'_t(\bar{s})$ as a short hand for the primed version of E_t defined as $E'_t(\bar{s}) \colon R'_1(\bar{s}_1), \ldots, R'_m(\bar{s}_m), \mathit{In}(\bar{s})$. The traversal rule for t is:

$$\mathit{In}_p(\overline{X}) \leftarrow \mathit{In}_q(\overline{X}), E'_t(\overline{X})$$

Here, we use the primed version $E'_t(\overline{X})$ since new transitions may become traversable as new facts are produced.

Generation Rule. To capture which facts are produced by traversing t, we introduce a *generation rule* defined as follows.

Let $W_t\colon R(\bar{u}) \leftarrow B_t(\bar{s})$ be the writing rule for t, with $B_t(\bar{s})$ denoting $R_1(\bar{s}_1), \ldots, R_m(\bar{s}_m), \mathit{In}(\bar{s})$. Then,

$$R'(\bar{u}) \leftarrow \mathit{In}_q(\overline{X}), E'_t(\overline{X}), B'_t(\overline{X})$$

Here, similarly to E'_t, $B'_t(\overline{X})$ denotes the primed version of $B_t(\overline{X})$.

Summary. Then, we have that the maximal extended database of $\mathcal{B}$ exactly corresponds to the set of facts that evaluates to true in $\Pi^{\mathit{po,fr}}_{\mathcal{P},Q} \cup \mathcal{D}$.

Lemma 1. *Let $R'(\bar{u})$ be a fact defined over adom*, then the following is equivalent:*

- *there is an execution in $\mathcal{B}$ that produces $R(\bar{u})$*
- $\Pi^{\mathit{po,fr}}_{\mathcal{P},Q} \cup \mathcal{D} \models R'(\bar{u})$

Testing Program

Now we define Π^{test}_Q that checks if $\Pi^{\mathit{po,fr}}_{\mathcal{P},Q} \cup \mathcal{D}$ produces a new query answer for Q.

Copy Rules. For every relation $R \in \Sigma_Q$ we introduce the primed version R' and, similarly as before, the copy rule

$$R'(\overline{X}) \leftarrow R(\overline{X})$$

in order to copy facts that are not copied by $\Pi^{\mathit{po,fr}}_{\mathcal{P},Q}$.

Q'-rule. Given a query $Q(\bar{x}) \leftarrow R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, we introduce a *$Q'$-rule* as the primed version of Q, as follows

$$Q'(\bar{x}) \leftarrow R'_1(\bar{u}_1), \ldots, R'_m(\bar{u}_m)$$

Test Rule. Then, if there is a new query answer, the *test rule* fires fact *Instable*:

$$\mathit{Instable} \leftarrow Q'(\overline{X}), \neg Q(\overline{X})$$

Summary. Let Π^{test}_Q be the program that contains Q, Q', the copy and test rules.

Theorem 1. *The following two are equivalent:*

- *Q is instable in $\mathcal{B}$ under open semantics;*
- $\Pi^{\mathit{po,fr}}_{\mathcal{P},Q} \cup \mathcal{D} \cup \Pi^{\mathit{test}}_Q \models \mathit{Instable}$.

Complexity results

4.2.1 EXPTIME- and PTIME-hardness in Process and Data Complexity

Program $\Pi^{\mathit{po,fr}}_{\mathcal{P},Q} \cup \mathcal{D} \cup \Pi^{\mathit{test}}_Q$ is a Datalog program with stratified negation for which the reasoning is as complex as for positive Datalog, that is EXPTIME for process and combined complexities and PTIME for data complexity.

Theorem 1 shows that the recursive power of Datalog (and some limited negation) is sufficient to reason on stability for positive fresh DABPs. In the following we show that the use of Datalog is an optimal approach as stability is as hard as Datalog query answering. In fact, we show that it is already "Datalog"-hard for *(a)* positive acyclic

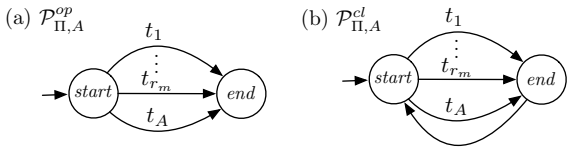


Figure 2: Datalog query answering encoded into positive DABPs under (a) open semantics and (b) closed semantics.

fresh DABPs under open semantics, and *(b)* positive cyclic singleton DABPs under closed semantics.

Let $\Pi \cup \mathcal{D}$ be a Datalog program, and let A be a fact. We define a positive acyclic fresh DABP $\langle \mathcal{P}^{op}_{\Pi,A}, \mathcal{D} \rangle$ that produces the LFP of $\Pi \cup \mathcal{D}$. As depicted in Figure 2(a), for every rule r in Π we introduce a transition t_r that connects *start* with *end* and we set *true* as execution condition and r as writing rule. Additionally, we add transition t_A with the writing rule *dummy* $\leftarrow A$. As initial database we take $\mathcal{D}$. Let $Q_{test} \leftarrow$ *dummy* be the test query.

Lemma 2. *$\Pi \cup \mathcal{D} \not\models A$ iff Q_{test} is stable in $\langle \mathcal{P}^{op}_{\Pi,A}, \mathcal{D} \rangle$.*

Lemma 2 can be adapted for positive cyclic arbitrary DABPs under closed semantics by extending $\mathcal{P}^{op}_{\Pi,A}$ with a transition from *end* to *start* that creates a cycle (see Figure 2(b)). We denote this process with $\mathcal{P}^{cl}_{\Pi,A}$. Now, let $\mathcal{C}_{\mathcal{D}}$ be a singleton configuration with $\mathcal{D}$ as initial database.

Corollary 1. *$\Pi \cup \mathcal{D} \not\models A$ iff Q_{test} is stable in $\langle \mathcal{P}^{cl}_{\Pi,A}, \mathcal{C}_{\mathcal{D}} \rangle$.*

The above two claims give lower-bounds for process (EXPTIME) and data complexity (PTIME). At this point, we can conclude that there are two sources of complexity in positive DABPs that elevate them to the complexity of Datalog: open semantics and presence of cycles. In Section 4.4 we show that dropping both of them lowers the combined complexity for stability checking.

4.2.2 Π^{P}_2-hardness in Query Complexity

We remind the reader that checking whether a Boolean CQ evaluates to true over a fixed database is already NP-hard in the size of the query. In this paragraph we show that checking query instability can be done in NP time using an NP oracle, which leads to Π^{P}_2 complexity for checking query stability.

An example of a Π^{P}_2 problem is the graph *3-coloring extension* that is the following problem: *Can any 3-coloring of the leaves of a graph be extended to a 3-coloring of all of the graph?*

Lemma 3 (Query Complexity)**.** *The following holds:*

a) There exist two databases $\mathcal{D}$, $\mathcal{D}'$ where $\mathcal{D} \subseteq \mathcal{D}'$ s.t. for a CQ Q checking if $Q(\mathcal{D}) = Q(\mathcal{D}')$ is Π^{P}_2-hard in the query size;

b) For any two databases $\mathcal{D}$, $\mathcal{D}'$ s.t. $\mathcal{D} \subseteq \mathcal{D}'$ checking for a CQ Q if $Q(\mathcal{D}) = Q(\mathcal{D}')$ is in Π^{P}_2 in the query size.

Proof. *a)* Assume we are given a graph $G = \langle V, E\rangle$ with vertices $V = \{v_1, ..., v_n\}$ and where w.l.o.g. leaves are $v_1, ..., v_m$, for $m \leq n$. Based on G we construct a query

$$Q_G(X_1, ..., X_m) \leftarrow \bigwedge_{(v_i,v_j)\in E} Eg(X_i, X_j).$$

We set $\mathcal{D}$ to be the six correct colorings $\{Eg(red, blue), Eg(red, green), \dots\}$, and we set D' to be all nine possible colorings $\{Eg(red, red), Eg(red, blue), \dots\}$. We observe the following. *(i)* Each tuple returned by Q_G when evaluated over $\mathcal{D}$ corresponds one coloring of the leaves variables where the rest of the variables are also colored according to $\mathcal{D}$. *(ii)* Q_G returns at least one tuple over $\mathcal{D}$ iff G is 3-colorable. This is because variables from Q_G can take only one color when evaluated over $\mathcal{D}$, and in addition any two adjacent vertices must have different colors. *(iii)* Then from the two above points we have that the tuples from $\mathcal{D}(Q_G)$ represent all colorings of the leaves for which there are correct 3-coloring of the rest of G. Thus, if the tuples from $\mathcal{D}(Q_G)$ are all possible colorings of the leaves then G is 3-coloring extendable. *(iv)* Moreover, Q_G evaluated over $\mathcal{D}'$ returns all possible colorings of the leaves. This is because one can assign any colors to the leaves variables from Q_G, where the rest of variables take any color, and this is still a satisfying assignment over $\mathcal{D}'$. Altogether, we have that $Q_G(\mathcal{D}) = Q_G(\mathcal{D}')$ iff G is 3-colorable extendable.

b) To show that checking $Q(\mathcal{D}) = Q(\mathcal{D}')$ is in Π_2^{P} it is sufficient to show that $Q(\mathcal{D}) \neq Q(\mathcal{D}')$ is in Σ_2^P. To show this, one can guess in NP time an answer $\bar{a} \in Q(\mathcal{D}')$ and then check using an NP oracle that $\bar{a} \notin Q(\mathcal{D})$. □

Following Lemma 3 *a)*, we can define a DABP under open semantics and a DABP under closed semantics that starting from $\mathcal{D}$ produces $\mathcal{D}'$. In fact, for both DABPs it is enough to consider the simplest variant of rowo.

Proposition 3. *Checking stability is Π_2^{P}-hard in query complexity for positive open DABPs. It is Π_2^{P}-hard already for*

a) positive acyclic fresh rowo DABPs under open semantics, and

b) positive acyclic arbitrary rowo DABPs under closed semantics.

Proof. *a)* Following the proof of Lemma 3 we construct a rowo DABP $\mathcal{B}_0 = \langle \mathcal{P}_0, \mathcal{D}_0\rangle$ such that $\mathcal{D}_0 = \{Eg(red, blue), Eg(red, green), \dots\}$ contains six correct colorings, and $\mathcal{P}_0$ is a process depicted on Figure 3 where edges t_{red}, t_{blue} and t_{green} insert the re-

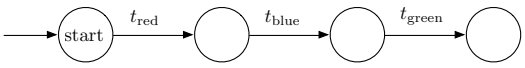


Figure 3: Process net of rowo $\mathcal{P}_0$

maining three colorings for edges: $Eg(red, red)$, $Eg(blue, blue)$ and $Eg(green, green)$ resp. The maximal extended database that $\mathcal{B}$ can produce is $\mathcal{D}'_0$ that contains all nine possible colorings. Then for a given graph $G = \langle V, E\rangle$ we construct the test query $Q_G(X_1, ..., X_m) \leftarrow \bigwedge_{(v_i,v_j)\in E} Eg(X_i, X_j)$ as defined in Lemma 3 for which it holds: Q_G is stable in $\mathcal{B}$ under open semantics iff G is 3-colorable extendable. The claim follows from there.

b) For closed semantics it is enough to take singleton DABP $\mathcal{B}'_0 = \langle \mathcal{P}_0, \mathcal{I}_0, \mathcal{D}_0\rangle$ where $\mathcal{P}_0$ and $\mathcal{D}_0$ are the same as in *a)*. The claim follows from there as in *a)*. □

Query Complexity Upper-Bound. From Theorem 1 we have that to check stability it is sufficient to compare the answers of Q over $\mathcal{D}$ and Q' over the maximal extended database entailed by $\Pi^{po,fr}_{\mathcal{P},Q} \cup \mathcal{D}$. According to Lemma 3 *b)* this can be done in Π^{P}_2 time in the size of Q.

Complexity Summary

Corollary 2 (Complexity Summary)**.** *For positive acyclic and cyclic fresh DABPs under open semantics checking query stability is*

1. EXPTIME*-complete in process and in combined complexity;*
2. Π^{P}_2*-complete in query complexity;*
3. PTIME*-complete in data complexity for fresh configurations.*

4.3 Normal (A)cyclic Fresh/Arbitrary Open

In this part we investigate stability for DABPs that execute under open semantics and that allow negation in the rules. Negation in the rules may impose that fresh instances need to introduce more than one new constant in order to execute the process. In fact, we show that it is not possible to compute the number of new constants that a process needs to introduce, as we show that normal acyclic DABPs under open semantics can simulate Turing machines (TMs). From this we have that checking stability is undecidable, and it is already undecidable in data and query complexity.

4.3.1 Undecidability in Data, Instance and Process Complexity

To show undecidability in data complexity we construct a DABP such that the process part simulates the executions of a given TM which is stored in the database. To simulate the TM executions, the process part has to deal with two features of TMs: *(i)* potentially unbounded number of updates on the TM configurations (= number of execution steps in the TM), and *(ii)* potentially infinite tape of the TM. The encoding process is organized in three subprocesses: the first two deal with *(i)* and *(ii)* and the third one simulates the execution of the TM. In the following we provide an intuition for these three subprocesses

DABPs cannot update facts in the database, however, we are able to simulate an update of a fact by augmenting each fact with a constant that represents a version of this fact. Then, an update of a fact is simulated with an inseration of the augmented fact where a new version is specified. Since we have arbitrary number of constants on the input we can do arbitrary number of updates. Using rules with negation, we can ensure that *(i)* a new version differs from all previous versions, and *(ii)* the process orders versions in a linear order by inserting a new version as successor of the last version (that is the only version without successor).

To encode a potentially infinite tape, the process again uses new constants from the input to index tape positions. To order tape indexes, the process uses a mechanism similar to the one used for ordering the versions.

The third subprocess uses versions and tape indexes produced by the other two to simulate the TM executions. Since it is acyclic, the subprocess needs a fresh instance for each execution step of the TM. It produces *halt* predicate iff the TM reaches a final state.

Let $\mathcal{P}_0$ be the process that comprises the three subprocesses described above and let $Q_{halt} \leftarrow halt$ be a test query. Let T be an arbitrary TM, s be an input for T, and let $D_{T,s}$ be a database that encodes T and s. Then the following holds.

Lemma 4 (Reduction of Turing Machines I)**.**

$$T \text{ halts on input } s \quad \textit{iff} \quad Q_{halt} \text{ is instable in } \langle \mathcal{P}_0, D_{T,s} \rangle \text{ under open semantics.}$$

Proof. See Subsection below. □

This gives us undecidability for data complexity. It also gives us undecidability for process complexity, since for a TM T and an input s we can augment $\mathcal{P}_0$ so that it first inserts all facts of $D_{T,s}$ into the database that is initially empty. Similarly, we obtain undecidability for instance complexity by introducing an arbitrary instance for each fact in $D_{T,s}$ that inserts that fact at the beginning.

4.3.2 Undecidability in Query Complexity

To show undecidability in query complexity the idea is to encode the given input s into a test query $Q_{halt,s}$ as a set of facts, that is $Q_{halt,s} \leftarrow halt, D_s$ where D_s encodes s. The difficulty now, w.r.t the previous cases, is that D_s from the query cannot be written into the database. To deal with this aspect we define a process part $\mathcal{P}'_0$ that extends $\mathcal{P}_0$ with a subprocess that can generate any input for a TM, that is for each input s' there is an execution where this subprocess generates $D_{s'}$. This subprocess executes first. Let T be a TM, we define a DABP $\langle \mathcal{P}'_0, D_T \rangle$ where D_T is the same as $D_{T,s}$ from Lemma 4 but without the encoding for the input s.

Then we have that $Q_{halt,s}$ is instable in $\langle \mathcal{P}'_0, D_T \rangle$ iff *(i)* the process first generates the encoding D_s of the input s and then *(ii)* the process produces *halt* predicate, i.e., the TM T halts for s. Since the process can generate any input, there exists an execution that generates the input that is encoded in the query; thus we have that the query is instable iff *(ii)* holds.

Corollary 3 (Reduction of Turing Machines II)**.**

$$T \textit{ halts on input } s \qquad \textit{iff} \qquad Q_{halt,s} \textit{ is instable in } \langle \mathcal{P}'_0, D_T \rangle \textit{ under open semantics.}$$

Proof. See Subsection below. □

If as TM we take the Universal TM (UTM), we have that the query is instable iff UTM halts for the encoded input. Checking the latter for an arbitrary input is undecidable.

Theorem 2 (Undecidable Cases)**.** *Checking stability is undecidable for normal DABPs under open semantics. Undecidability already holds for normal acyclic fresh DABPs under open semantics and*

1. *data complexity;*
2. *instance complexity;*
3. *process complexity;*
4. *query complexity.*

Proof of Lemma 4

To prove Lemma 4 we want to define a process such that executions of the process correspond to runs of a TM for any TM. We encode the TM into the data, so that in this way we can

- show undecidability of data complexity; and
- show undecidability also for acyclic processes (where cycles are encoded introducing as many new process instances as needed).

Encoding Principles. We now provide the intuition for the encoding.

In a Turing machine the content of a cell can be updated by an execution step. Since DABPs cannot update relations we have to simulate updates. To this end we use

constants, called versions $v_1, v_2, \ldots$, to store different versions of the TM configuration (i.e. to encode the steps of a run of a TM).

Additionally, we need to index cells on a tape which can be arbitrarily long. Positions on the tape are encoded by indexing each position of the tape using indexes $i_1, i_2, \ldots$ (see Figure 4). Since we do not know the length of a run of a TM, neither how much space on the tape it uses, the process needs to generate a sufficient number of indexes and versions.

Generated versions and indexes need to be ordered. As we will show, this this is possible using the negation in the rules.

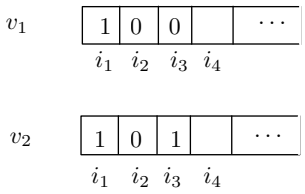


Figure 4: Use of versions and indexes in the encoding of TM into DABP

In the following we describe how to construct a DABP for a given TM and a given input. The encoding is divided into three subprocesses (see Figure 5) connected to the *start*, such that:

1. **Execution Simulation.** Simulates the execution of the Turing Machine (TM)
2. **Version Generation.** Generates constants that are used for the versioning of TM configurations
3. **Index Generation.** Generates constants that are used for indexing the cells of the tape

For convenience, we will use notation $t_1^{TM}, t_2^{TM}, \ldots$, for the transitions in the subprocess for the **Execution Simulation** Similarly, we will use $t_1^{vs}, t_2^{vs}, \ldots$ and $t_1^{idx}, t_2^{idx}, \ldots$ for the transitions in the subprocesses for **Version Generation** and **Index Generation** resp.

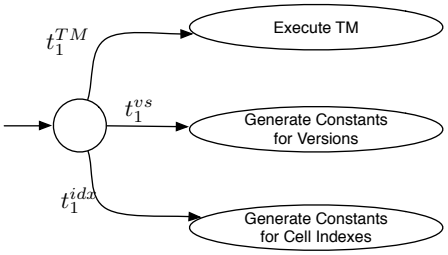


Figure 5: Encoding of a Turing Machine into DABPs

Encoding of Subprocess for the Execution of the TM

Before describing the subprocess that simulates the execution of a TM, we report the definition of a TM.

Turing Machine. A Turing Machine is defined as a 7-tuple

$$\mathcal{M} = \langle Q, F, q_0, \Sigma, \sqcup, \Gamma, \delta \rangle$$

where:

- $Q = \{q_0, q_1, \ldots, q_n\}$ – set of states;
- $F = \{f_1, \ldots, f_m\} \subseteq Q$ – set of accepting (final) states s.t. $m \leq n$;
- $q_0 \in Q$ – starting state;
- $\Sigma = \{0, 1\}$ – input symbols;
- $\sqcup$ – blank symbol that occurs infinitely often on the tape;
- $\Gamma = \Sigma \cup \{\sqcup\}$ – tape symbols;
- $\delta : (Q \backslash F) \times \Gamma \to Q \times \Gamma \times \{\textit{left}, \textit{right}\}$ – transition function.
  Given a state and a read symbol the machine transits into a new state, writes a symbol and moves left or right on the tape. E.g. $\delta(q, 1) \to (q', 0, \textit{left})$.

Vocabulary. Later on we will use the following:

- $v_1, v_2, v_3, \ldots$ as symbols to enumerate TM configurations;
- $i_1, i_2, i_3, \ldots$ to index the cells of the tape. We consider a tape unbounded in one direction only;
- *halt* as unary predicate that is true iff TM halts, that is if it reaches a final state.

Version and Index Relations. To encode versions we introduce relation *Version* to store symbols that are used for versions. We also introduce relation *NextVersion* to store the immediate successor of a version in the order, and *VersionUsed* to store the versions symbols that have already been used in the subprocess. The encoding is such that it always hold: *VersionUsed* $\subseteq$ *Version*. More precisely:

- *Version*(v) is true iff constant v is a version;
- *NextVersion*(v_1, v_2) represents that the next version after version v_1 is version v_2;
- *VersionUsed*(v) holds iff version v has been used in the subprocess;

Similarly to versions, we introduce relation *Index* to store the symbols used to index positions on the tape. We use *NextIndex* to store the order among the index symbols. That is,

- *Index*(i) is true iff constant i is an index;
- *NextIndex*(i_1, i_2) holds iff index i_2 points to the cell that is on the right of the cell pointed to by index i_1. For convenience, in the following we will say "cell i" instead of "pointed to by index i".

TM relations. To encode a TM we will use relation δ' of arity 5, to encode the transition function δ of the TM. We use relation *Final* as a unary relation containing the final states of the TM.

- $\delta'(q_1, s_1; q_2, s_2, d)$: for each argument (q_1, s_1) and the corresponding result (q_2, s_2, d) in δ we have a record $\delta'(q_1, s_1; q_2, s_2, d)$ in the database;
- *Final*(q) iff q is a final state.

Relations for TM execution. Finally, to encode the execution of the TM we use relation *Cell* to store the content of a cell in a version. Then, we introduce relation *Head* to store the cell to which the TM points in a certain version, and *State* to store the state in which the TM is in a certain version. We also introduce an auxiliary unary relation *LockTM* to ensure that at most one process instance at a time executes the subprocess.

- *Cell*(v, i, s) holds iff in version v the content of cell i is symbol s;
- *Head*(v, i) iff in version v the head of the TM is over cell i;
- *State*(v, q) is true iff in version v the TM is in state q;
- *LockTM*(v) stores that the current version v is being executed by a process instance. Thus, other process instances cannot execute.

Initialization. We now describe how relations are initialized.

Assume we are given an input $s = s_1, \ldots, s_k$. We introduce indexes $i_0, i_1, \ldots, i_k$ for indexing the cells, where i_0 is an auxiliary symbol that we use for technical reason to properly generate new indexes in the subprocess that generates indexs (that we will discuss later). Index i_1 is the index of the first cell.

We populate the relations *Index* and *NextIndex* as follows:

Index
i_0
i_1
$\vdots$
i_k

NextIndex	
i_0	i_1
i_1	i_2
$\vdots$	$\vdots$
i_{k-1}	i_k

As initial version we take v_1. We initialize the relation *Cell* for the given input as follows:

Cell		
v_1	i_1	s_1
$\vdots$	$\vdots$	$\vdots$
v_1	i_k	s_k

In the case we have an empty input, *Index* and *Cell* relations are initialized as follows:

Index
i_0
i_1

Cell		
v_1	i_1	$\sqcup$

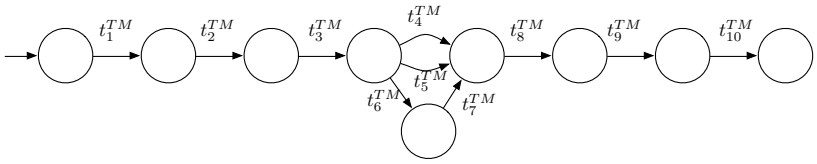


Figure 6: Subprocess that simulates the execution of the TM

Then, we initialize the relations $Head$ and *State* for the execution of the TM. Specifically, we have that at version v_1 the head points to the first cell. This is encoded with $\mathit{Head}(v_1, i_1)$. The starting state q_0 is encoded with $\mathit{State}(v_1, q_0)$.

For technical reasons, we introduce version v_0, which is an auxiliary version used to correctly initialize the versioning of the tape. We initialize the versioning relations as follows:

Version	*VersionUsed*	*NextVersion*		*LockTM*
v_0	v_0	v_0	v_1	v_0
v_1				

Process Part. In the following we define the subprocess that simulates the execution of a TM. The subprocess is depicted in Figure 6.

Intuitively, each of the transitions $t_2^{TM}, \ldots, t_{10}^{TM}$ of the subprocess in Figure 6 updates one relation that encodes the execution of the TM. Since in DABP it is not possible to write more than one relation at a time, we need to make sure that only one process instance at a time can traverse transitions $t_2^{TM}, \ldots, t_{10}^{TM}$. If we would allow more than one instance to execute the subprocess, we may have interleaving executions of instances in the process that do not correspond to any execution of the TM. Therefore, we define transition t_1^{TM} in such a way that, once traversed by an instance, it will be disabled for other instances to enter the subprocess until the running instance reaches the end of the subprocess. This ensures that only after all updates for the current step of the TM are made, the next step of TM can be executed.

Note that, if we could write into several relations in one transition, then two transitions would be enough: one for storing the new configuration of the tape and one to check whether a final state has been reached (currently done with transition t_9).

For convenience we define $\mathit{CurrentVersion}(V)$ as abbreviation for the following condition:

$$\mathit{CurrentVersion}(V) \colon \mathit{NextVersion}(V_1, V), \mathit{VersionUsed}(V_1), \neg \mathit{VersionUsed}(V).$$

Intuitively, the current version is a version that has not been used yet and that is the successor of a version that has been used. The subprocess generating version symbols makes sure that there is always exactly one constant that is the current version.

We now describe execution conditions and writing rules for the transitions. We omit execution conditions when they are *true*.

We use transition t_1^{TM} to ensure that the subprocess is executed by one instance at a time. To achieve this we enforce that the transition can be traversed only if the current version is not locked:

$$E_{t_1^{TM}} \colon \mathit{CurrentVersion}(V), \neg \mathit{LockTM}(V).$$

Initially, t_1^{TM} is not locked for version v_1.

Immediately after an instance traverses t_1^{TM}, this transition becomes disabled for other instances to traverse. This is achieved by locking the current version with the writing rule:

$$W_{t_1^{TM}}: \mathit{LockTM}(V) \leftarrow \mathit{CurrentVersion}(V).$$

The transition t_1^{TM} is unlocked again (for a new current version) once a new current version is introduced by traversing transition t_{10}^{TM} (as we show later).

By means of transition t_2^{TM} we model the update of the *Cell* pointed by the *Head* by inserting a symbol for the cell in the next version. Conditions for an instance to progress traversing t_2^{TM} are that:

i) there is a next version for the current version, i.e. the subprocess 2 has generated enough version constants to proceed (If this is not the case, the execution of this subprocess is blocked until new constants are generated by the subprocess generating new versions.);

ii) given the current state of the TM and the read input, according to the transition function δ' there is a next state.

These conditions are encoded in the following rule:

$$\begin{aligned} E_{t_2^{TM}}: \ & \mathit{CurrentVersion}(V), \mathit{NextVersion}(V, V_1), \\ & \mathit{Head}(V, I), \mathit{State}(V, Q), \\ & \mathit{Cell}(V, I, S), \delta'(Q, S; \_, \_, \_). \end{aligned}$$

While traversing, it is necessary to update the symbol in the cell pointed by the head. This is done by inserting in the cell pointed by the head, the symbol according to transition function δ'. This is done for the new version the symbol. The following rule does this:

$$\begin{aligned} W_{t_2^{TM}}: \mathit{Cell}(V_1, I, S_2) \leftarrow & \mathit{CurrentVersion}(V), \mathit{NextVersion}(V, V_1), \\ & \mathit{Head}(V, I), \mathit{State}(V, Q), \\ & \mathit{Cell}(V, I, S_1), \delta'(Q, S_1; \_, S_2, \_). \end{aligned}$$

Transition t_3^{TM} updates the version of the cells that are not pointed by the *Head*, by copying in the next version the content of the cells not pointed by the head:

$$\begin{aligned} W_{t_3^{TM}}: \mathit{Cell}(V_1, I_1, S) \leftarrow & \mathit{CurrentVersion}(V), \mathit{NextVersion}(V, V_1), \\ & \mathit{Head}(V, I), \mathit{Cell}(V, I_1, S), I_1 \neq I. \end{aligned}$$

Transitions t_4^{TM}, t_5^{TM}, t_6^{TM} and t_7^{TM} update the TM head by updating the index of the cell it is pointing at. These transitions distinguishes the cases in which the head is moving *left* (t_4^{TM}) or *right*. If the head moves to the right, we also distinguish the cases in which a cell on the right exists in relation *Cell* (t_5^{TM}) or not. The latter case happens if the new cell was never visited before, and thus we have no *Cell*-record with the index on the right. In this case we write $\sqcup$ symbol to instantiate newly visited cells (t_6^{TM}, t_7^{TM}).

- Transition t_4^{TM} updates the *Head* in case the move is to the *left*. Condition for moving *left* is that the head does not currently point to the first position (i_1), which

represents the left-hand border of the tape. If this is the case the subprocess will be blocked. This is because we assume the tape to be infinite only on the right:

$$\begin{aligned} E_{t_4^{TM}}\colon\ & \mathit{CurrentVersion}(V), \mathit{Head}(V, I), \\ & \mathit{NextIndex}(I_1, I), I_1 \neq i_0, \\ & \mathit{State}(V, Q), \mathit{Cell}(V, I, S), \delta'(Q, S; \_, \_, \mathit{left}). \end{aligned}$$

By traversing t_4^{TM}, the head is updated storing the index of the cell it is pointing at in the new version:

$$\begin{aligned} W_{t_4^{TM}}\colon \mathit{Head}(V_1, I_1) \leftarrow & \mathit{CurrentVersion}(V), \mathit{NextVersion}(V, V_1) \\ & \mathit{Head}(V, I), \mathit{NextIndex}(I_1, I). \end{aligned}$$

- Transition t_5^{TM} updates the *Head* in case the move is to the *right*. Similarly to transition t_4^{TM}, the transition is executable if there exists an index for the next cell the head should point at, i.e. the index has been generated by the subprocess generating the tape indexes. If this is not the case subprocess will be blocked until new constants are generated. Additionally, t_5^{TM} can be traversed if there exists a *Cell*-record for the cell on the right:

$$\begin{aligned} E_{t_5^{TM}}\colon\ & \mathit{CurrentVersion}(V), \mathit{Head}(V, I), \\ & \mathit{NextIndex}(I, I_1), \mathit{Cell}(V, I_1, \_)\mathit{State}(V, Q), \mathit{Cell}(V, I, S), \\ & \delta'(Q, S; \_, \_, \mathit{right}). \end{aligned}$$

If t_5^{TM} is traversed, the head is updated for the next version:

$$\begin{aligned} W_{t_5^{TM}}\colon \mathit{Head}(V_1, I_1) \leftarrow & \mathit{CurrentVersion}(V), \mathit{Head}(V, I), \\ & \mathit{NextIndex}(I, I_1), \mathit{NextVersion}(V, V_1). \end{aligned}$$

- Transition t_6^{TM} is executable when the TM moves *right* and there is no *Cell*-record for the index on the right:

$$\begin{aligned} E_{t_6^{TM}}\colon\ & \mathit{CurrentVersion}(V), \mathit{Head}(V, I), \\ & \mathit{NextIndex}(I, I_1), \mathit{State}(V, Q), \mathit{Cell}(V, I, S), \\ & \neg\mathit{Cell}(V, I_1, S_1), \Gamma(S_1), \delta'(Q, S; \_, \_, \mathit{right}). \end{aligned}$$

As explained, we write $\sqcup$ symbol to instantiate newly visited cells:

$$\begin{aligned} W_{t_6^{TM}}\colon \mathit{Cell}(V_1, I_1, \sqcup) \leftarrow & \mathit{CurrentVersion}(V), \mathit{Head}(V, I), \\ & \mathit{NextIndex}(I, I_1), \mathit{NextVersion}(V, V_1). \end{aligned}$$

- Once the $\sqcup$ symbol has been stored for the new cell, transition t_7^{TM} updates the head. At this point it is not necessary to check if the other subprocesses have generated the next values for the version and the index resp. This condition is guaranteed by previous transitions (t_2^{TM} for version and t_6^{TM} for the index). Similarly to transition t_5^{TM}, t_7^{TM} updates the head as follows:

$$\begin{aligned} W_{t_7^{TM}}\colon \mathit{Head}(V_1, I_1) \leftarrow & \mathit{CurrentVersion}(V), \mathit{Head}(V, I), \\ & \mathit{NextIndex}(I, I_1), \mathit{NextVersion}(V, V_1). \end{aligned}$$

Transition t_8^{TM} updates the *State* of the machine according to δ':

$$W_{t_8^{TM}}: \textit{State}(V_1, Q_1) \leftarrow \textit{CurrentVersion}(V), \textit{Head}(V, I), \\ \textit{NextVersion}(V, V_1), \textit{State}(V, Q), \\ \textit{Cell}(V, I, S), \delta'(Q, S; Q_1, \_, \_).$$

Transition t_9^{TM} checks whether the machine is in a final state. In this case it produces *halt*:

$$W_{t_9^{TM}}: \textit{halt} \leftarrow \textit{CurrentVersion}(V), \textit{NextVersion}(V, V_1), \textit{State}(V_1, Q), Final(Q).$$

Finally, transition t_{10}^{TM} updates *CurrentVersion* if TM did not reach a final state:

$$E_{t_{10}^{TM}}: \neg \textit{halt}.$$

Updating the current version will make transition t_1^{TM} again executable for a new process instance:

$$W_{t_{10}^{TM}}: \textit{VersionUsed}(V) \leftarrow \textit{CurrentVersion}(V).$$

Encoding of Subprocess for the Generation of New Versions

In this suprocess we encode the generation of version symbols. In particular, we want to populate relation *Version* described before, to store symbols used to enumerate different versions. We also want to define a linear order among them and store it in relation *NextVersion*, such that it holds:

$$\textit{NextVersion}(v_0, v_1), \textit{NextVersion}(v_1, v_2), \textit{NextVersion}(v_2, v_3), \ldots$$

Relations. In addition to the relations already defined, we introduce relation *PredVersion* to store that a certain version is the predecessor of another version. This will allow us to identify the last version in the order, as the one that is not predecessor of any other version. We also introduce relation *LockVersion* that is used similarly to *LockTM* to ensure that one instance at a time executes the subprocess. More preciesely:

- *PredVersion*(v) holds iff version v is the predecessor of some version, i.e. there exist v' s.t. *NextVersion*(v, v'). *PredVersion* is the projection of *NextVersion* on the first component. This is needed to encode negation with "not exists", since our rules allow only safe negation.
- *LockVersion*(v) iff the subprocess is locked for version v.

Initialization. To correctly initialize the process we use v_0 as an auxiliary version and we initialize relations *Version*, *VersionUsed* and *NextVersion* as shown in the initialization of the previous subprocess.

Additionally, we initialize relations *LockVersion* and *PredVersion* as follows:

LockVersion	*PredVersion*
v_0	v_0

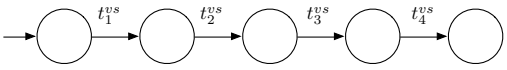


Figure 7: Subprocess that generates of new version indexes

Process Part. To generate the symbols for the versions and order them we define the process depicted in Figure 7.

As for the subprocess described previously, to correctly generate the versions and the ordering among them we need to make sure that at most one process instance at a time executes the subprocess. We enforce this as in the previous case.

For convenience we define *LastVersion*(V) as follows:

$$\begin{aligned} \textit{LastVersion}(V) : \textit{NextVersion}(V_1, V) \\ \textit{PredVersion}(V_1), \neg \textit{PredVersion}(V). \end{aligned}$$

This condition selects a version V that is the successor of another version V_1 and for which it does not exist a next version.

We now describe in detail each transition of the subprocess.

Transition t_1^{vs} collects new version candidates. Input relation *In* is needed to bring new constants, therefore, *In* is unary. New constants are used for counting versions:

$$\begin{aligned} E_{t_1^{vs}} : \textit{In}(V), \neg \textit{Version}(V); \\ W_{t_1^{vs}} : \textit{Version}(V) \leftarrow \textit{In}(V). \end{aligned}$$

Transition t_2^{vs} ensures that one instance at a time is executing the subprocess 2. Similarly to subprocess 1, this is achieved by expressing the condition that an instance can traverse if the last version is not locked:

$$E_{t_2^{vs}} : \textit{LastVersion}(V), \neg \textit{LockVersion}(V).$$

By traversing, the instance locks the last version, preventing other instances to execute the subprocess. This ensures that only one new version is created:

$$W_{t_2^{vs}} : \textit{LockVersion}(V) \leftarrow \textit{LastVersion}(V).$$

Transition t_3^{vs} updates the *NextVersion*, by inserting the value that comes with the instance *In* as next version of the current last version:

$$W_{t_3^{vs}} : \textit{NextVersion}(V_1, V_2) \leftarrow \textit{In}(V_2), \textit{LastVersion}(V_1);$$

Transition t_4^{vs} updates the *PredVersion* relation. The consequence of this is that the *LastVersion* is now updated, to the new version inserted by the instance. This enables transition t_2^{vs} for a new instance to traverse.

$$W_{t_3^{vs}} : \textit{PredVersion}(V) \leftarrow \textit{LastVersion}(V).$$

Encoding of Subprocess for the Generation of New Indexes

The generation of new tape indexes is done similarly to version generation.

Relations. As before, we introduce relation *Index* to store the symbols used to enumerate cells of the tape. We define a linear order among these indexes and we define the relation

PredIndex to store indexes that are the predecessor of another indexes. This relation is used to identify the last index in the order as the one that is not the predecessor of any index.

- *PredIndex*(i) is true iff index i is the predecessor of some index, i.e. it exists an index i_1 s.t. *NextIndex*(i, i_1) holds. In other words, *PredIndex* is the projection of *NextIndex* on the first component, used to encode the negation as "not exists".
- *LockIndex*(i) holds iff the subprocess is locked for index i.

Initialization. For the correct generation of indexes, we introduce indexes i_0, i_1 and we introduce *NextIndex*(i_0, i_1).

Additionally, for a given input $s = s_1, \ldots, s_k$ we introduce indexes $i_1, \ldots, i_k$ and we populate relations *Index*, *NextIndex*, *PredIndex* and *LockIndex* as follows:

Index
i_0
i_1
$\vdots$
i_k

NextIndex	
i_0	i_1
i_1	i_2
$\vdots$	$\vdots$
i_{k-1}	i_k

PredIndex
i_0
$\vdots$
i_{k-1}

LockIndex
i_0
$\vdots$
i_{k-1}

Note that if the input is empty the initialization will be:

Index
i_0
i_1

NextIndex	
i_0	i_1

PredIndex
i_0

LockIndex
i_0

Process Part. Figure 8 report the subprocess for the generation of the indexes, which is similar to the subprocess for the generation of the versions.

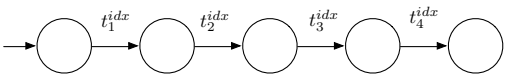


Figure 8: Subprocess that generates new indexes

Similarly to versions generation we define *LastIndex* as follows:

$$\begin{aligned} \textit{LastIndex}(I) : \textit{NextIndex}(I_1, I),& \\ \textit{PredIndex}(I_1),& \\ \neg\textit{PredIndex}(I).& \end{aligned}$$

Transition t_1^{idx} collects new index candidates:

$$\begin{aligned} E_{t_1^{idx}} &: \textit{In}(I), \neg\textit{Index}(I); \\ W_{t_1^{idx}} &: \textit{Index}(I) \leftarrow \textit{In}(I). \end{aligned}$$

Similarly to subprocess for version generation, *In* is a unary relation that is used to bring new constants used for counting tape positions.

Transition t_2^{idx} is traversable by an instance if *LastIndex* is not locked. By traversing, the last index becomes locked and this ensures that one instance at a time is executing the subprocess:

$$E_{t_2^{idx}}: \mathit{LastIndex}(I), \neg \mathit{LockIndex}(I);$$
$$W_{t_2^{idx}}: \mathit{LockIndex}(I) \leftarrow \mathit{LastIndex}(I).$$

Transition t_3^{idx} inserts a new tuple in relation *NextIndex* such that the value that comes with the input *In* is the immediate successor of the last index:

$$W_{t_3^{idx}}: \mathit{NextIndex}(I_1, I_2) \leftarrow \mathit{In}(I_2), \mathit{LastIndex}(I_1).$$

Finally, transition t_4^{idx} adds to relation *PredIndex* the last index. This will make transition t_2^{idx} again executable by a new instance:

$$W_{t_4^{idx}}: \mathit{PredIndex}(I) \leftarrow \mathit{LastIndex}(I).$$

Proof of Corollary 3

Now we want to show undecidability in query complexity. To do this we encode a given input s into a test query $Q_{halt,s}$ as a set of facts, that is $Q_{halt,s} \leftarrow \mathit{halt}, D_s$ where D_s encodes s. Since D_s cannot be written into the database, we extend the process described in the previous section with an additional suprocess, as reported in Figure 9, which can generate any input for a TM. Then, we enforce this subprocess to execute first.

Encoding of subprocess for Input Generation. For a given input $s = s_1, \ldots, s_n$ we create the following query:

$$Q \leftarrow \mathit{halt}, \mathit{Cell}(v_1, i_1, s_1), \ldots, \mathit{Cell}(v_1, i_n, s_n), \mathit{Cell}(v_1, i_{n+1}, \sqcup),$$

which is not stable iff the guessed values for the initial configuration of the TM correspond to the input s and then if the TM halts for this input.

Then, we define the process that guesses the input. Intuitively, starting from the first position on the tape, the process non-deterministically decides whether to guess a symbol from the input alphabet and store it in the cell, or whether to guess the end of the input and store the corresponding symbol $\sqcup$ in the cess. In the former case, the process continues by making another guess for the next position on the tape. In the latter case, the execution of the current subprocess ends, and the execution of the subprocess simulating the execution of the TM will be enabled.

To achieve this, we will use unary predicate *runTM* to denote that the end of the initial configuration has been guessed. Then, to allow other the subprocess for TM execution to execute only when this predicate holds we modify the execution condition of t_1^{TM} as follows:

$$E_{t_1^{TM}}: \mathit{runTM}, \mathit{CurrentVersion}(V), \neg \mathit{LockTM}(V).$$

Relations. To guess the input we will use relation $Sigma'$ to store the input alphabet. Then, we use relation *Guessed* to store the indexes for which a symbol has been guessed, and *LockGuess* to store that the execution of the process is currently locked for the guessing of a symbol at a certain position. More precisely,

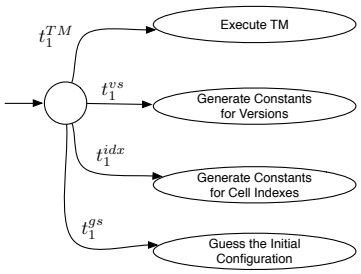


Figure 9: Encoding of TM extended with the guessing of the initial configuration.

- $\Sigma'(s)$ is true iff s is a symbol of the input alphabet, i.e 0 or 1.
- *Guessed*(i) holds iff the symbol pointed by index i has already been guessed.
- *LockGuess*(i) is an auxiliary unary relation that we use to ensure that at most one instance at a time executes the subprocess.
- *runTM* is a nullary predicate that is true iff the subprocess has finished with guessing.

Initialization. For a correct execution of the subprocess we initialize relations *Guessed* and *LockGuess* as follows:

$$\frac{\textit{Guessed}}{i_0} \qquad \frac{\textit{LockGuess}}{i_0}$$

Process Part. The process net is reported in Figure 10. To determine the first index for which no guess has been made yet, we define *CurrentGuess* as follows:

$$\begin{aligned}\textit{CurrentGuess}(I) : \textit{NextIndex}(I_1, I),\\ \textit{Guessed}(I_1),\\ \neg\textit{Guessed}(I).\end{aligned}$$

Similarly to previous cases, we use t_1^{gs} to prevent new instances to execute the subprocess when an instance is executing it. Note that, in case it does not exist an index

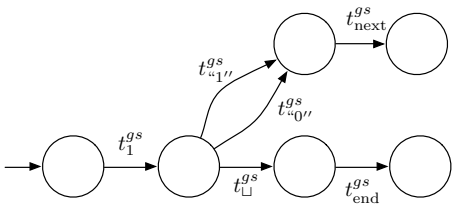


Figure 10: Subprocess that guesses the initial configuration

that satisfies condition *CurrentGuess* the process is blocked until the subprocess for index generation generates new indexes:

$$E_{t_1^{gs}} : \mathit{CurrentGuess}(I), \neg \mathit{LockGuess}(I);$$

$$W_{t_1^{gs}} : \mathit{LockGuess}(I) \leftarrow \mathit{CurrentGuess}(I).$$

Transitions $t^{gs}_{\text{“}1\text{”}}$ and $t^{gs}_{\text{“}0\text{”}}$ are used to respectively guess symbol 1 or 0 for the *CurrentGuess* index. One among the two transitions is executed non-deterministically. While traversing, the relation *Cell* will be updated for the current guess, storing for the initial version v_1 the value 1 if traversing $t^{gs}_{\text{“}1\text{”}}$ and the value 0 if traversing $t^{gs}_{\text{“}0\text{”}}$:

$$W_{t^{gs}_{\text{“}1\text{”}}} : \mathit{Cell}(v_1, I, 1) \leftarrow \mathit{CurrentGuess}(I);$$
$$W_{t^{gs}_{\text{“}0\text{”}}} : \mathit{Cell}(v_1, I, 0) \leftarrow \mathit{CurrentGuess}(I).$$

Transition t^{gs}_{next} updates relation *Guessed* with the index that has been guessed. As a consequence, transition t_1^{gs} will be enabled for a new instance to traverse and make a guess for the next index:

$$W_{\mathit{next}} : \mathit{Guessed}(I) \leftarrow \mathit{CurrentGuess}(I).$$

Transition $t^{gs}_{\sqcup}$ is traversed when the end of the input is guessed. While traversing, the relation *Cell* is updated storing that the cell corresponding to the current guess contains symbol $\sqcup$ in the initial version v_1:

$$W_{t^{gs}_{\sqcup}} : \mathit{Cell}(v_1, I, \sqcup) \leftarrow \mathit{CurrentGuess}(I).$$

Transition t^{gs}_{end} is executed after transition $t^{gs}_{\sqcup}$ and therefore after the end of the initial configuration is guessed. The traversal by an instance produces the unary predicate *runTM*. This will enable the subprocess for the exeuction of the TM to execute:

$$W_{t^{gs}_{end}} : \mathit{runTM} \leftarrow \mathit{true}.$$

Note that, since *CurrentGuess* is not updated when the end of the input is guessed, no new instances will be able to execute transition t_1^{gs}.

Universal Turing Machine. If we encode a Universal Turing Machine (UTM) in our process then we have that

$$Q \leftarrow \mathit{halt}, \mathit{Cell}(v_1, i_1, s_1), \ldots, \mathit{Cell}(v_1, i_n, s_n), \mathit{Cell}(v_1, i_{n+1}, \sqcup)$$

is not stable iff

i) there is an execution where the process first guesses the input

$$\mathit{Cell}(v_1, i_1, s_1), \ldots, \mathit{Cell}(v_1, i_n, s_n), \mathit{Cell}(v_1, i_{n+1}, \sqcup),$$

and it is obvious that such an execution exists;

ii) and then the UTM halts for this input, which is undecidable.

4.4 Normal Acyclic Arbitrary Closed

In this section we consider DABPs under closed semantics where initially process instances are scattered over process net. This also includes the cases when an instance is initially at a place that is not reachable from the *start*.

For this variant of DABPs we observe the following.

i) The number of new facts that a process can produce is finite. In fact, there are exponentially many different maximal extended databases that can be produced, due to the fact that arbitrary instances may need to choose between several alternative paths that they traverse (and the facts they produce). This is a difference w.r.t. DABPs in Section 4.2, where a fresh DABP can produce infinitely many new facts, but according to the abstraction principle to check query stability it is enough to consider one maximal extended database.

ii) The maximal length of an execution is finite since an instance cannot traverse a transition more than once. Consider a DABP $\mathcal{B} = \langle \mathcal{P}, \mathcal{I}, \mathcal{D} \rangle$ with $o_1, \ldots, o_k$ instances and $t_1, \ldots, t_m$ transitions in $\mathcal{P}$ then the maximal length of an execution is mk.

iii) Since the length of each execution is bounded, we can use relations to store possible paths and what hold after their execution. Let Υ be a closed execution of size $i \leq mk$ where at the first execution step an instance o_{l_1} traverses a transition t_{h_1}, then o_{l_2} traverses t_{h_2} and so on. Then an execution Υ of size i can be encoded as a sequence $o_{l_1}, t_{h_1}, \ldots, o_{l_i}, t_{h_i}$ and stored as a tuple:

$$\langle o_{l_1}, t_{h_1}, \ldots, o_{l_i}, t_{h_i} \rangle.$$

Below, we define a Datalog program that produces the facts that hold after an execution of size i and store them in relations R^i's.

Encoding into Non-Recursive Datalog

For each relation R in $\mathcal{P}$ and Q we introduce relations R^i (for i up to mk) to store all R-facts produced by an execution of size i. Let Υ be the closed execution from above and let $\langle o_{l_1}, t_{h_1}, \ldots, o_{l_i}, t_{h_i} \rangle$ be the tuple representing it. Then, a relation R^i has the form

$$R^i(o_{l_1}, t_{h_1}, \ldots, o_{l_i}, t_{h_i}; \bar{s})$$

and it holds iff Υ produces fact $R(\bar{s})$. Later on we use $\bar{\omega}$ to represent the tuple $\langle o_{l_1}, t_{h_1}, \ldots \rangle$ for an execution Υ. R^i-facts are then represented as $R^i(\bar{\omega}; \bar{s})$. For convenience, we use semicolon (;) instead of comma (,) to separate the encodings of the arguments of different types.

Further, we want to record positions of instances after the execution of Υ. For this we introduce relation State^i such that $\mathit{State}^i(\bar{\omega}; p_1, \ldots, p_k)$ is true iff after Υ is executed the instances $o_1, \ldots, o_k$ are resp. at places $p_1, \ldots, p_k$.

Additionally, we introduce auxiliary relation In^0 that associates instances with their *In*-records, that is $\mathit{In}^0(o; \bar{s})$ is true iff the instance o has input record $\mathit{In}(\bar{s})$. With slight abuse of notation, we use $\bar{\omega}$ to denote also the corresponding closed execution Υ.

We now define a program $\Pi^{\mathsf{ac,cl}}_{\mathcal{P},\mathcal{I}}$ that computes R^i's and State^i's for all possible executions.

Initialization Rules. We need initialization rules *(i)* to store the facts holding at the beginning of the execution and *(ii)* to store the initial position of the instances. Accordingly, we introduce the following rules.

- For each relation $R \in \Sigma_{\mathcal{P}}$ we introduce a rule to store what holds at step 0:

$$R^0(\overline{Y}) \leftarrow R(\overline{Y}).$$

- To represent that in the initial configuration o_1 is at place p_1, o_2 at p_2, and so on, we introduce the following fact rule:

$$\mathit{State}^0(p_1, \ldots, p_k) \leftarrow \mathit{true}$$

Traversal Rules. We now want to record how an instance position changes after it traverses a transition. An instance o_j at step $i+1$ can traverse a transition t from place q to p if o_j is at place q at step i and it satisfies the execution condition E_t. The following *traversal rule* captures this:

$$\begin{aligned}\mathit{State}^{i+1}(\overline{W}, o_j, t; P_1, \ldots, P_{j-1}, p, P_{j+1}, \ldots, P_k) \leftarrow \\ \mathit{State}^{i}(\overline{W}; P_1, \ldots, P_{j-1}, q, P_{j+1}, \ldots, P_k), E^i_t(\overline{W}; o_j)\end{aligned}$$

Here, $E^i_t(\overline{W}; o_j)$ is a shorthand for the condition obtained from E_t by replacing $\mathit{In}(\bar{s})$ with $\mathit{In}^0(o_j; \bar{s})$ and by replacing each atom $R(\bar{s})$ with $R^i(\overline{W}; \bar{s})$. The tuple $\overline{W}$ is defined as $2i$ many distinct variables to match every possible execution of size i. It ensures that only facts produced by $\overline{W}$ are considered.

Formally, let E_t denote the condition $\mathit{In}(\bar{s}), R_1(\bar{s}_1), \ldots, R_n(\bar{s}_n), \neg R_{n+1}(\bar{s}_{n+1}), \ldots, \neg R_m(\bar{s}_m)$, then with $E^i_t(\overline{W}; o)$ we denote the condition $\mathit{In}^0(o; \bar{s}), R^i_1(\overline{W}; \bar{s}_1), \ldots, R^i_n(\overline{W}; \bar{s}_n), \neg R^i_{n+1}(\overline{W}; \bar{s}_{n+1}), \ldots, \neg R^i_m(\overline{W}; \bar{s}_m)$

Copy Rules. A fact in R^{i+1} may hold because *(i)* it was produced at some prior step in the execution or *(ii)* it is produced by traversing transition t at step $i+1$. Facts produced at previous steps are propagated with the *copy rule*:

$$R^{i+1}(\overline{W}, O, T; \overline{Y}) \leftarrow \mathit{State}^{i+1}(\overline{W}, O, T; \_), R^i(\overline{W}; \overline{Y})$$

copying facts $R(\overline{Y})$ holding at step i to step $i+1$.

Generation Rules. Facts that are produced by traversing a transition t with the writing rule $W_t\colon\ R(\bar{u}) \leftarrow B_t(\bar{s})$ are generated with the *generation rule*:

$$R^{i+1}(\overline{W}, O, t; \bar{u}) \leftarrow \mathit{State}^{i+1}(\overline{W}, O, t; \_), B^i_t(\overline{W}; O)$$

where condition $B^i_t(\overline{W}; O)$ is obtained in the same way as $E^i_t(\overline{W}; O)$. In particular, let $B_t(\bar{s})$ denote the condition $\mathit{In}(\bar{s}), R_1(\bar{s}_1), \ldots, R_n(\bar{s}_n), \neg R_{n+1}(\bar{s}_{n+1}), \ldots, \neg R_m(\bar{s}_m)$, then, with $B^i_t(\overline{W}; O)$ we denote the condition $\mathit{In}^0(O; \bar{s}), R^i_1(\overline{W}; \bar{s}_1), \ldots, R^i_n(\overline{W}; \bar{s}_n), \neg R^i_{n+1}(\overline{W}; \bar{s}_{n+1}), \ldots, \neg R^i_m(\overline{W}; \bar{s}_m)$.

Summary. The above rules define the program $\Pi^{\mathit{ac,cl}}_{\mathcal{P},\mathcal{I}}$ for acyclic closed DABPs that is polynomial in the size of $\mathcal{P}$ and $\mathcal{I}$. We have that, the program $\Pi^{\mathit{ac,cl}}_{\mathcal{P},\mathcal{I}} \cup \mathcal{D}$ generates all facts produced by an execution.

Lemma 5. *Let $\bar{\omega}$ be an execution in $\mathcal{B}$ of size i, and let $R_1(\bar{s}_1), \ldots, R_n(\bar{s}_n)$ be a set of facts. The following is equivalent:*

- *Facts $R_1(\bar{s}_1), \ldots, R_n(\bar{s}_n)$ are produced by $\bar{\omega}$;*
- $\Pi^{\mathit{ac,cl}}_{\mathcal{P},\mathcal{I}} \cup \mathcal{D} \models R^i_1(\bar{\omega}; \bar{s}_1), \ldots, R^i_n(\bar{\omega}; \bar{s}_n)$.

Testing Program

Now we want to test stability of a query $Q(\overline{X}) \leftarrow R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$.

Initialization and Copy Rules. Similarly to relations from $\Sigma_{\mathcal{P}}$, for each relation $R \in \Sigma_Q$ we introduce the *initialization rules* of the following kind:

$$R^0(\overline{Y}) \leftarrow R(\overline{Y})$$

For each relation $R \in \Sigma_Q$ we also introduce a *copy rule* as for relations in $\Sigma_{\mathcal{P}}$.

Q'-rule. Then, as for open variants, we collect all potential Q-answers in query Q'. A new query answer may be produced by an execution of any size i up to mk. Thus for each execution of a size i up to mk we introduce the *Q'-rule*:

$$Q'(\overline{X}) \leftarrow R_1^i(\overline{W}; \bar{u}_1), \ldots, R_n^i(\overline{W}; \bar{u}_n)$$

As a difference from the previous case, Q' is now defined on an execution $\overline{W}$.

Test Rule. Finally, we introduce the *test rule* as before.

$$\mathit{Instable} \leftarrow Q'(\overline{X}), \neg Q(\overline{X}).$$

Summary. Let $\Pi^{\mathit{test}}_{\mathcal{P},\mathcal{I},Q}$ be the program that contains the above rules depending on the executions of the process $\mathcal{P}$ by the running instances $\mathcal{I}$ and on the query Q. Then, the following Theorem holds.

Theorem 3. *The following is equivalent:*

- *Q is instable in $\mathcal{B}$ under closed semantics;*
- $\Pi^{\mathit{ac,cl}}_{\mathcal{P},\mathcal{I}} \cup \mathcal{D} \cup \Pi^{\mathit{test}}_{\mathcal{P},\mathcal{I},Q} \models \mathit{Instable}$.

Complexity Results

From Theorem 4 we obtain combined complexity and upper-bound complexities for process, data and instances. Note that, $\Pi^{\mathit{ac,cl}}_{\mathcal{P},\mathcal{I}} \cup \mathcal{D} \cup \Pi^{\mathit{test}}_{\mathcal{P},\mathcal{I},Q}$ is a non-recursive Datalog program with negation. Non-recursive Datalog is enough since in closed acyclic DABPs the maximal length of an execution is bounded by the number of running instances and transitions in the process. Checking query answering for such programs is as complex as for non-recursive positive ones which is in PSPACE (which is better than query answering for recursive Datalog).

4.4.1 PSPACE-hardness in Process Complexity

To show PSPACE-hardness, we encode query answering for non-recursive Datalog program into stability checking. Let $\Pi = \{r_1, \ldots, r_n\}$ be a non-recursive Datalog program and let A be a fact. Since Π is non-recursive we can order the rules in Π in such a way that applying them in that order produces the LFP of Π. Wlog, let an order be $r_{i_1}, r_{i_2}, \ldots, r_{i_n}$ where $i_1, \ldots, i_n$ is a permutation of $1, \ldots, n$. Then, we define an acyclic positive DABP $\mathcal{P}_{\Pi,A}$ as depicted in Figure 11, defined as follows. For each rule r_{i_j} we introduce a transition t_{i_j} having r_{i_j} as the writing rule. Additionally, we introduce transition t_A with writing rule W_{t_A} : *dummy* $\leftarrow A$. We initialize singleton configuration $\mathcal{C}_0$ with an empty database, and we take Q_{test} as the test query.

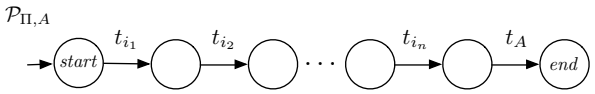


Figure 11: Datalog query answering using acyclic closed DABPs

Lemma 6. *$\Pi \not\models A$ iff Q_{test} is stable in $\langle \mathcal{P}_{\Pi,A}, \mathcal{C}_0 \rangle$.*

From Theorem 4, we have that data complexity is the same as data complexity for non-recursive Datalog programs, that is in AC^0.

Notice that in acyclic closed DABPs having negation in the rules does not increase the complexities w.r.t. the positive variant. This is because the execution length is bounded. If we allow cycles, this does not hold, and we later see that stability becomes more complex in the variants with negation than in the positive variants.

Instance Complexity For the instance complexity we show that the complexity is significantly higher than data complexity, that is CO-NP-hard already for positive acyclic closed DABPs.

Lemma 7. *There exists a positive acyclic $\mathcal{P}_0$, and a database $\mathcal{D}_0$ s.t. for every graph G one can construct an instance part $\mathcal{I}_G$ s.t.*

$$G \text{ is not 3-colorable} \quad \text{iff} \quad Q_{test} \text{ is stable in } \langle \mathcal{P}_0, \mathcal{I}_G, \mathcal{D}_0 \rangle.$$

Proof. See Subsection 4.4.2 □

Clearly, from Lemma 7 we also have that checking stability for normal arbitrary DABPs is CO-NP-hard in instance complexity. It is not hard to see that CO-NP is also the upper-bound: one can first guess a sequence $\bar{\omega}$, which is polynomial in the size of the instances, and then check in polynomial time if $\bar{\omega}$ is indeed an execution in the process and if such $\bar{\omega}$ yields any new query answer. The check can be done in polynomial time since process, data and query are assumed to be fixed.

Query Complexity When measuring query complexity, the DABP is fixed, thus the number of maximal extended databases that can be produced is constant. Then to check stability of a query Q it is sufficient to compare finitely many times the answers of Q over the maximal extended databases and the initial one. From Lemma 3 *(b)* we have that each such check is in Π_2^{P}.

Complexity Summary

Corollary 4 (Complexity Summary)**.** *For normal and positive acyclic arbitrary DABPs for under closed semantics checking stability is*

1. PSPACE*-complete in process and combined complexity;*
2. Π_2^{P}*-complete in query complexity;*
3. CO-NP*-complete in instance complexity;*
4. *in* AC^0 *in data complexity.*

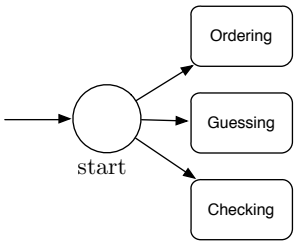


Figure 12: DABP Process Net encoding three colorability of a graph

4.4.2 CO-NP-hardness in Instance Complexity

In this part we prove Lemma 7 defined above.

In particular, we want to encode the problem of 3-colorability for a graph G into positive acyclic arbitrary DABPs $\mathcal{B}_G = \langle \mathcal{P}_0, \mathcal{I}_G, \mathcal{D}_0 \rangle$ under closed semantics and a query Q_{test} such that the query is not stable in $\mathcal{B}_G$ iff G is 3-colorable.

Let the graph $G = \langle V, E \rangle$ be with the vertices $V = \{v_1, \ldots, v_n\}$ and the edges $E = \{e_1, \ldots, e_m\}$.

We define a process $\mathcal{P}_0$ composed of three parts.

i) The first part stores an order among the edges, that we later use to check that all vertexes are correctly colored.

ii) The second part guesses a coloring for each vertex.

iii) The last part checks that the guessed coloring is correct.

First we establish a linear order $<$ on E such that:

$$e_1 < e_2 < \cdots < e_m.$$

Later we use the linear order to check whether the edges are correctly colored by checking each edge individually following the order. To store the order in database relations we introduce relations *Next*, *First* and *Last* to store resp. the successors in the order, the first and the last element of the order. In addition, we introduce relations *Color* and *Colored* to record the color assigned to each vertex and to store edges that are properly colored. Finally, we use relation *OkColor* to store the correct colorings for two adjacent vertexes.

The relations are populated by the process such that: *(i)* *Next*$(v'_i, v''_i, v'_{i+1}, v''_{i+1})$ is true iff edge (v'_{i+1}, v''_{i+1}) is immediate successor of edge (v'_i, v''_i) in the order; *(ii)* *First*(v'_1, v''_1) is true if the edge (v'_1, v''_1) is the first in the order; *(iii)* *Last*(v'_m, v''_m) is true if the edge (v'_m, v''_m) is the last in the order; *(iv)* *Color*(v, c) is true if vertex v is colored with color c; *(v)* *Colored*(v_i, v_j) is true if the edge (v_i, v_j) is properly colored.

We define initial database $\mathcal{D}_0$ to contain the six correct colorings for two adjacent vertexes, that is $\{$*OkColor*(*red*, *blue*), *OkColor*(*red*, *green*), $\ldots\}$.

In the following we define the process part $\mathcal{P}_0$. The process net of the process depicted in Figure 12. We assume input relation *In* of arity four. In the following we define the process rules.

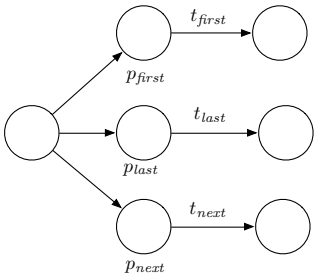


Figure 13: Subrocess to store the ordering among the edges

Writing the linear ordering. The subprocess reported in Figure 13 populates relations *First*, *Last* and *Next* according to an order among the edges. From now on, where not otherwise defined, we assume the transitions have execution conditions set to *true*.

Writing rules for transitions t_{first}, t_{last}, and t_{next} are defined as follows:

$$W_{t_{first}} : \mathit{First}(X, Y) \leftarrow \mathit{In}(X, Y, \_, \_);$$
$$W_{t_{last}} : \mathit{Last}(X, Y) \leftarrow \mathit{In}(X, Y, \_, \_);$$
$$W_{t_{next}} : \mathit{Next}(X_1, Y_1, X, Y) \leftarrow \mathit{In}(X_1, Y_1, X, Y).$$

For each fact that encodes the order, we introduce one process instance that by traversing a transition writes this fact. Later we use the linear order to check whether the edges are correctly colored by checking each edge individually following the order.

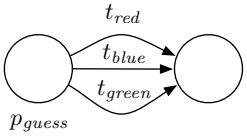


Figure 14: Subprocess to guess a coloring.

Guessing of a color. The subprocess reported in Figure 14 is used to guess a coloring for the vertexes. For each vertex we introduce one process instance that by nondeterministic choice of a transition among t_{red}, t_{blue} and t_{green} writes the color of that vertex into database.

Writing rules for these transitions is defined as follows:

$$W_{t_{red}} : \mathit{Color}(X, \mathit{red}) \leftarrow \mathit{In}(X, \_, \_, \_);$$
$$W_{t_{blue}} : \mathit{Color}(X, \mathit{blue}) \leftarrow \mathit{In}(X, \_, \_, \_);$$
$$W_{t_{green}} : \mathit{Color}(X, \mathit{green}) \leftarrow \mathit{In}(X, \_, \_, \_).$$

Checking of a color. Finally, the last subprocess in Figure 15 checks that the guessed coloring is indeed correct.

For each we introduce a process instance that writes a fact if the coloring is correct. At the last stage we check if the facts are written for all edges.

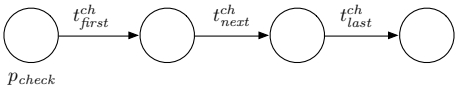


Figure 15: Subrocess to check the coloring

Transition t^{ch}_{first} checks the coloring for the first edge in the graph. In particular, if the first edge in the order is correctly colored, it will produce fact $\mathit{Colored}(v_1, v'_1)$. This is captured by the following writing rule:

$$W_{t_{first}} : \mathit{Colored}(X,Y) \leftarrow \mathit{In}(X,Y,\_,\_), \mathit{First}(X,Y), \\ \mathit{Color}(X,C_X), \mathit{Color}(Y,C_Y), \\ \mathit{OkColor}(C_X,C_Y).$$

Similarly, transition t^{ch}_{next} will check the coloring for the edges in the ordering: for an edge $\mathit{In}(v'_j, v''_j, b, b)$ (where b is a constant used for techincal reasons, since relation *In* must be of size 4) it will produce fact $\mathit{Colored}(v'_j, v''_j)$ iff v'_j and v''_j are correctly colored and if the previous edge is also correctly colored. This is captured by the following writing rule:

$$W_{t_6} : \mathit{Colored}(X,Y) \leftarrow \mathit{In}(X,Y,\_,\_), \\ \mathit{Color}(X,C_X), \mathit{Color}(Y,C_Y), \mathit{OkColor}(C_X,C_Y), \\ \mathit{Next}(X_1,Y_1,X,Y), \mathit{Colored}(X_1,Y_1).$$

Finally, transition t_{last} will produce fact *dummy* iff the last edge is correctly colored and all previous edges are also correctly colored:

$$W_{t_{last}} : \mathit{colorable} \leftarrow \mathit{In}(X,Y,\_,\_), \mathit{Last}(X,Y) \\ \mathit{Color}(X,C_X), \mathit{Color}(Y,C_Y), \mathit{OkColor}(C_X,C_Y), \\ \mathit{Next}(X_1,Y_1,X,Y), \mathit{Colored}(X_1,Y_1).$$

Instance Part. We then introduce $(m+n+m)$ instances $\mathcal{I}_G$ in the following way.

- In the subprocess in Figure 13 we introduce m instances, one for each edge in the following way:
  - for the first edge $e_1 = (v'_1, v''_1)$ in the order we introduce an instance o_{e_1} with *In*-record $\mathit{In}(v'_1, v''_1, b, b)$ at place p_{first};
  - similarly, for the last edge $e_{m+1} = (v'_m, v''_m)$ in the order we introduce instance $o_{e_{m+1}}$ with *In*-record $\mathit{In}(v'_m, v''_m, b, b)$ at place p_{last};
  - for any two immediate successor $e_i < e_{i+1}$ in the order we introduce instance $o_{e_{i+1}}$ with *In*-record $\mathit{In}(v'_i, v''_i, v'_{i+1}, v''_{i+1})$ at place p_{next}.
    Here, fresh constants b is needed for technical reasons since we use the input relation of size four.
- In the subprocess in Figure 14 we introduce n instances, that is, one instance o_{v_i} with *In*-record $\mathit{In}(v_i, b, b, b)$ for each vertex v_i at place p_{guess}.
- In the subprocess in Figure 15 we introduce m instances, that is, for each edge $e_j = (v'_j, v''_j)$ in the graph we introduce an instance o_{e_j} with *In*-record $\mathit{In}(v'_j, v''_j, b, b)$ and place it at p_{check}.

Conclusion. As the test query we use $Q_{test} \leftarrow dummy$.

Then it is not hard to check that the query is stable iff G is not colorable. Process and database parts are fixed and the size of instance part is linear w.r.t. the size of the instance part. This concludes the proof.

4.5 Positive Acyclic Arbitrary Open

In this variant we have to combine what can be produced by the running instances (from the initial configuration), and what can be produced by fresh instances that can be introduced at the *start* place.

Assume we are given a DABP $\mathcal{B} = \langle \mathcal{P}, \mathcal{I}, \mathcal{D} \rangle$ that executes under open semantics with instances $o_1, \ldots, o_k$ in the initial configuration. We make the following observations:

i) Similar to the closed variant in Section 4.4, at each execution the running instances can make at most mk steps, where m is the number of transitions in $\mathcal{P}$. Thus, as shown there, executions of running instances can be encoded as a tuple.

ii) Considering fresh instances, in principle, there is no bound on the length of an execution, since arbitrary many fresh instances can start at any moment. Still, to analyze stability, as in the fresh open case in Section 4.2, it is enough to consider the *maximal execution* that produces the largest set of facts possible taking the constants from the extended active domain.

iii) The idea is to combine the maximal executions for fresh instances with bounded executions for running instances. In particular, we extend the maximal execution for fresh instances by making mk maximal executions, each after one execution step of a running instance. In this way we obtain the largest set of facts possible taking the constants from the extended active domain. According to the abstraction principles this one is enough to check stability.

Following the above observations, we define *maximal executions* that are constructed by alternating the following two steps:

1. first, fresh instances that take constants from the extended active domain traverse the process in all possible ways producing the maximum that they can produce;
2. then, one running instance makes a single transition, and then we again execute the first step.

In the execution above, one can record the steps by the fresh instances in the same way as for the fresh variant, except that we augment it with the executions of running instances.

In fact, execution steps of running instances uniquely characterizes such maximal executions, since they are the only choices that maximal executions can have (whereas steps by fresh instances are deterministic as they traverse in all possible ways).

Then we again use a tuple of size mk

$$\langle o_{l_1}, t_{h_1}, o_{l_2}, t_{h_2}, \ldots \rangle$$

to record such maximal executions where the recorded steps are the traversals of running instances.

Similarly, to record that a fact $R(\bar{s})$ is produced by such maximal execution $\bar{\omega} = \langle o_{l_1}, t_{h_1}, o_{l_2}, t_{h_2}, \ldots \rangle$ we use

$$R^i(\bar{\omega}; \bar{s}).$$

Encoding into Datalog

To encode what is produced by the maximal executions we adopt the encoding relations as for the closed variant (Section 4.4).

We adapt the relations from the fresh variant (Section 4.2), as follows:

- $\mathit{In}_{p,i}$ for $i = 1, \ldots, mk$ of arity $\mathit{arity}(\mathit{In}) + 2i$ such that $\mathit{In}_{p,i}(\bar{\omega}; \bar{s})$ is true iff a fresh instance with *In*-record can reach place p provided that running instances traverse according to $\bar{\omega} = \langle o_{l_1}, t_{h_1}, \ldots, o_{l_i}, t_{h_i} \rangle$.
- R^i of arity $2i + \mathit{arity}(R)$ such that $R^i(\bar{\omega}, \bar{s})$ is true iff a maximal execution where running instances traverse according to $\bar{\omega}$ produces $R(\bar{s})$.

To define a Datalog program, we take all the rules from the closed variant and we adapt the rules from the fresh variant as follows.

Traversal Rule. For every execution step i up to mk and for every transition t from a place q to a place p, we introduce a *traversal rule* that copies all fresh instances that were able to reach q and that satisfy the execution condition E_t:

$$\mathit{In}_{p,i}(\overline{W}; \overline{X}) \leftarrow \mathit{In}_{q,i}(\overline{W}; \overline{X}), E^i_t(\overline{W}; \overline{X})$$

where $E^i_t(\overline{W}; \overline{X})$ denotes the condition obtained from E_t as in the closed variant.

Generation Rule. We adapt the generation rule to store the facts that are produced by a fresh instance in relations R^i's. For each transition t with writing rule W_t: $R(\bar{u}) \leftarrow B_t(\bar{s})$ we introduce the following *generation rule* for each i up to mk:

$$R^i(\overline{W}; \bar{u}) \leftarrow \mathit{In}_{q,i}(\overline{W}; \overline{X}), E^i_t(\overline{W}; \overline{X}), B^i_t(\overline{W}; \overline{X}).$$

Here $B^i_t(\overline{W}; \overline{X})$ denotes the condition obtained from B_t as in the closed variant.

Summary. The above rules define the program $\Pi^{\mathit{po,ac}}_{\mathcal{P},\mathcal{I},\mathcal{Q}}$.

Lemma 8. *Let $\bar{\omega}$ be a maximal execution in $\mathcal{B}$ of size i, and let $R_1(\bar{s}_1), \ldots, R_n(\bar{s}_n)$ be a set of facts. The following is equivalent:*

- *Facts $R_1(\bar{s}_1), \ldots, R_n(\bar{s}_n)$ are produced by $\bar{\omega}$;*
- $\Pi^{\mathit{po,ac}}_{\mathcal{P},\mathcal{I},\mathcal{Q}} \cup \mathcal{D} \models R^i_1(\bar{\omega}; \bar{s}_1), \ldots, R^i_n(\bar{\omega}; \bar{s}_n)$.

Testing Program

As testing program we adopt the same program from the closed variant.

Summary. Let $\Pi^{\mathit{test}}_{\mathcal{P},\mathcal{I},Q}$ be the testing program. Then, the following Theorem holds.

Theorem 4. *The following is equivalent:*

- *Q is instable in $\mathcal{B}$ under closed semantics;*
- $\Pi^{\mathit{po,ac}}_{\mathcal{P},\mathcal{I},\mathcal{Q}} \cup \mathcal{D} \cup \Pi^{\mathit{test}}_{\mathcal{P},\mathcal{I},Q} \models \mathit{Instable}$.

Complexity Results

Complexities are inherited from the fresh variant, except for the instance complexity which is inherited from the arbitrary variant.

Corollary 5 (Complexity Summary)**.** *For positive acyclic and cyclic fresh DABPs under open semantics checking query stability is*

1. EXPTIME*-complete in process and in combined complexity;*
2. Π_2^{P}*-complete in query complexity;*
3. CO-NP*-complete in instance complexity;*
4. PTIME*-complete in data complexity.*

4.6 Positive Cyclic Arbitrary Closed

In contrast to acyclic cases, in cyclic DABPs, there is no bound on the length of an execution. Still to check stability in DABPs (even with negation) that executes under closed semantics it is enough to consider executions up to a certain length.

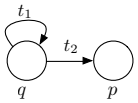
Naive Approach Consider a DABP $\mathcal{B} = \langle \mathcal{P}, \mathcal{C} \rangle$, possibly with cycles, which has c different constants, r different relations, and the maximal arity of a relation in $\mathcal{P}$ is a. We make the following observations.

1. For each relation in $\mathcal{B}$ of arity up to a there are up to $c^{arity(R)}$ new facts that $\mathcal{B}$ can produce. Thus, $\mathcal{B}$ can produce up to rc^a new facts.
2. Further, we observe that it is sufficient to consider executions that produce at least one new fact each mk steps. An execution that produces no facts in mk steps has at least one instance in t hose mk steps that visits the same place two times without producing any new fact; thus those execution steps can be cancelled from the execution without affecting the facts that are produced.
3. From the first two points we can conclude that instead of analyzing all executions it sufficient to consider only executions of maximal length

$$mkrc^a.$$

In principle, we can encode $mkrc^a$ long executions as in the acyclic variant, and obtain a program that checks for stability. Still such approach is not optimal as it applies a decision procedure that runs in exponential time (query answering in Datalog) to an exponentially big input (the encoding of executions in the program), thus yielding double-exponential running time. Instead, we establish a more optimal approach that considers only a special subset of all possible executions that are sufficient to check stability. In particular, we look into executions that produce the largest number of new facts possible, called *greedy executions*.

Greedy Executions In the following we define *greedy executions*, that are intuitively executions that produce the largest number of new facts. To illustrate this consider the positive DABP depicted below.



$E_{t_1} = E_{t_2} = \mathit{true}$
$W_{t_1} : R(X,Y) \leftarrow R(X,Z), R(Z,Y)$
$W_{t_2} : S(X,Y) \leftarrow R(X,Y)$

R	
a	b
b	c
c	d

Assume the initial configuration has one instance o at place q. Then, possible executions of o are of the form $t_1, \ldots, t_1, t_2$ where t_1 repeats an arbitrary number of times. An execution produces the largest number of facts if it traverses t_1 sufficiently number of times to produce the transitive closure of R. Then it traverses t_2.

In greedy executions we distinguish between two kinds of execution steps: *safe step* and *critical step*.

- A safe step is an execution step of an instance after which the instance can return to the original place given the current state of the database. E.g., in the above example, o traversing t_1 is a safe step.
- A critical step is an execution step that is not safe. E.g., o traversing t_2 is a critical step.

Based on that we define *greedy sequence* and *greedy execution*.

- A greedy sequence is a sequence of safe steps that produces the largest number of new facts possible. E.g., a sequence where o traverses t_1 at least two times, thus generating the transitive closure of R, is a greedy sequence.
- A greedy execution is an execution where first a greedy sequence is executed, then a critical step is executed, then a greedy sequence, then a critical step, and so on.

Later we show that each execution can be transformed into a greedy one that produces at least the same facts as the original one.

Strongly Connected Components in Greedy Executions Let Υ be a greedy execution with i alternations of greedy sequences and critical steps. In the following, we characterize what are the transitions that instances traverse in the $i+1$-th greedy sequences and then in the $i+1$-th critical step. For process instance o and database D_Υ produced after Υ we define the *enabled graph* $N_{\Upsilon,o}$ as the graph whose vertexes are the places from N (i.e., the process net of $\mathcal{B}$) and edges are those transitions from N that are enabled for o given database D_Υ. Let $SCC(N_{\Upsilon,o})$ denotes the set of strongly connected components (SCCs) of $N_{\Upsilon,o}$. We note that two different instances may have different enabled graphs and thus different SCCs. Further, let o be at place p after Υ is executed, and let $N^p_{\Upsilon,o}$ be the SCC in $SCC(N_{\Upsilon,o}$ that contains p. Then in the next greedy sequence, each instance o traverses the component $N^p_{\Upsilon,o}$ in all possible ways until no new facts can be produced, meaning that all instances traverse simultaneously. Conversely, the next critical step is an execution step where an instance o traverses a transition that is not part of $N^p_{\Upsilon,o}$, and thus it moves to another SCC.

Properties of Greedy Executions In order to motivate the encoding of greedy executions we make the following observations:

(i) How can one characterize greedy executions? There may be several ways how a greedy sequence orders safe steps. In all of them the same facts are produced: the largest number of new facts. Thus, how safe steps are ordered is not important as they all have the same impact on stability. In contrast, which critical step is executed can be important for stability as an instance may need to choose among several possible transitions and possibly the instance may never be able to reach the previous position in order to execute the remaining transitions. Therefore, a greedy execution is uniquely characterized with its critical steps, since how the greedy sequences are composed in not relevant for stability.

(ii) How long can greedy executions can be? After each critical step, the number of SCCs of an instance o that the instance has not reached yet decreases or stays the same. If o is preforming the critical step then this number decreases since the instance changes the current SCC. If another instance is performing the critical step then new facts may be inserted and thus some of the SCCs in $SCC(N_{\Upsilon,o})$ may merge. Hence, the number of unvisited SCCs from $SCC(N_{\Upsilon,o})$ may only decrease or stay the same. And similarly after a greedy sequence. Thus, if there are m transitions then there can be at most m critical steps for o, and overall, there can be at most mk critical steps where k is the number of instances.

Based on the observations we define a Datalog program that for a given DABP computes facts produced by the greedy executions.

Encoding into Datalog with Stratified Negation

Let $\mathcal{B}$ be a positive (possibly with cycles) DABP with m transitions and k instances. Since, critical steps uniquely characterize a greedy execution, we use a tuple of size up to mk to encode greedy executions similarly to the acyclic variant from Section 4.4. For example, if in a greedy execution at the first critical step instance o_{l_1} traverses transition t_{h_1}, in the second o_{l_2} traverses t_{h_2}, and so on up to step i, we encode this with the tuple

$$\bar{\omega} = \langle o_{l_1}, t_{h_1}, \ldots, o_{l_i}, t_{h_i} \rangle.$$

With slight abuse of notation, we use $\bar{\omega}$ to denote also the greedy execution Υ.

- Again similarly, we use $R^i(\bar{\omega}; \bar{s})$ to encode that the greedy execution $\bar{\omega}$ with i critical steps produces fact $R(\bar{s})$.
- To record positions of instances after each critical step we adapt State^i from acyclic variant such that now $\mathit{State}^i(\bar{\omega}; p_1, \ldots, p_k)$ encodes that after $\bar{\omega}$ is executed instance o_1 is at $N^{p_1}_{\bar{\omega},o_1}$, o_2 is at $N^{p_2}_{\bar{\omega},o_2}$, and so on.
- To store SCCs of the enabled graph we introduce relations SCC^i such that for a process instance o and a place p we have that transition t belongs to $N^p_{\bar{\omega},o}$ iff $SCC^i(\bar{\omega}; o, p, t)$ is true.
- To compute SCC^i relations we first need to compute what are the places reachable by an instance o from a place p. For that we introduce auxiliary relations Reach^i such that in the enabled graph $N_{\bar{\omega},o}$ instance o can reach place p' from p iff $\mathit{Reach}^i(\bar{\omega}; o, p, p')$ is true.

The rest of relations are the same as in the acyclic variant. Now we are ready to define rules that compute those relations.

Greedy Sequence: Traversal Rules. To compute facts produced by a greedy sequence we first define rules to compute reachability. Let $\bar{\omega}$ be a greedy execution of size i. Relation *Reach* are meant to contain the transitive closure of the enabled graph $N_{\bar{\omega},o}$ for each $\bar{\omega}$ and o. First, a transition t from q to be p gives rise to an edge in the enabled graph $N_{\bar{\omega},o}$ if instance o can traverse that t:

$$\mathit{Reach}^i(\overline{W}; O, q, p) \leftarrow E^i_t(\overline{W}; O).$$

Then the transitive closure is computed with the following rule:

$$\begin{aligned}\mathit{Reach}^i(\overline{W}; O, P_1, P_3) \leftarrow \\ \mathit{Reach}^i(\overline{W}; O, P_1, P_2), \mathit{Reach}^i(\overline{W}; O, P_2, P_3).\end{aligned}$$

Based on Reach^i, SCC^i is computed by taking every transition t that an instance can reach, traverse, and return to the current place:

$$\begin{aligned}SCC^i(\overline{W}; O, P, t) \leftarrow \mathit{Reach}^i(\overline{W}; O, P, q), E^i_t(\overline{W}; O), \\ \mathit{Reach}^i(\overline{W}; O, p, P).\end{aligned}$$

Here t goes from q to p.

Greedy Generation Rule. Now we are ready to compute facts produced by the next greedy sequence. For each instance o_j at place p_j after the last critical step in $\bar{\omega}$, and

for each transition t that is in $N^{p_j}_{\bar{\omega},o_j}$ with writing rule $R(\bar{u}) \leftarrow B_t(\bar{s})$, we introduce the following *greedy generation rule*:

$$R^i(\overline{W};\bar{u}) \leftarrow \textit{State}^i(\overline{W};\_,\ldots,\_,P_j,\_,\ldots,\_),\\ SCC^i(\overline{W};o_j,P_j,t), B^i_t(\overline{W};o_j).$$

In other words, all transitions t that are in $N^{p_j}_{\bar{\omega},o_j}$ of o_j are fired simultaneously, and this is done for all instances.

Critical Steps: Traversal Rules. To encode critical steps we adapt the traversal rules from the acyclic variant by: *(i)* ensuring that at each critical step the traversing instance changes current SCC, and *(ii)* allowing that critical step can be taken from any place in the current SCC (not only current place), encoded with:

$$\begin{aligned}&\textit{State}^{i+1}(\overline{W},O,t;P_1,\ldots,P_{j-1},p,P_{j+1},\ldots,P_k) \leftarrow\\ &\quad \textit{State}^i(\overline{W};P_1,\ldots,P_{j-1},P,P_{j+1},\ldots,P_k), E^i_t(\overline{W};o_j),\\ &\quad \neg SCC^i(\overline{W};o_j,P,t),\\ &\quad \textit{Reach}^i(\overline{W};o_j,P,q), \textit{Reach}^i(\overline{W};o_j,q,P).\end{aligned}$$

Here, condition *(i)* above is ensured by $\neg SCC^i(\overline{W};o_j,P,t)$; condition *(ii)* is ensured by $\textit{Reach}^i(\overline{W};o_j,P,q), \textit{Reach}^i(\overline{W};o_j,q,P)$.

Critical Generation Rule. A critical step is an execution step, thus we define *critical generation rule* to be the same generation rule as in the acyclic variant:

$$R^{i+1}(\overline{W},O,t;\bar{u}) \leftarrow \textit{State}^{i+1}(\overline{W},O,t;\_), B^i_t(\overline{W};O).$$

The rest of the program is the same as in the acyclic variant.

Summary. Let us denote the above program with $\Pi^{\textit{po,cl}}_{\mathcal{P},\mathcal{I}}$.

Lemma 9. *Let $\bar{\omega}$ be a greedy execution in $\langle \mathcal{P},\mathcal{I},D\rangle$ of length i, and $R_1(\bar{s}_1),\ldots,R_n(\bar{s}_n)$ be a set of facts. The following is equivalent:*

- *Facts $R_1(\bar{s}_1),\ldots,R_n(\bar{s}_n)$ are produced by $\bar{\omega}$;*
- $\Pi^{\textit{po,cl}}_{\mathcal{P},\mathcal{I}} \cup D \models R^i_1(\bar{\omega};\bar{s}_1),\ldots,R^i_n(\bar{\omega};\bar{s}_n)$.

Reduction to Greedy Executions

So far we seen that greedy executions and what they produce can be encoded in a compact way using Datalog. Now we show the second important property of greedy executions: every execution can be transformed into a greedy one such that the greedy execution produces the same facts (and possibly more) as the original execution. The idea is that for each instance in some execution Υ one can identify at most m execution steps that are *critical*. Here m is the number of transitions in the process. We observe that for an instance o and transition t there can be at most one critical step where o traverses t. For example, the first occurrence of o traversing t in Υ. Thus, for k instances there are in total at most mk critical steps. Based on those critical step in Υ we create a greedy execution $\bar{\omega}_\Upsilon$ that contains all execution steps of Υ and that produces all facts produced by Υ. By considering one by one the critical steps in Υ we decide to put a critical step in $\bar{\omega}_\Upsilon$ iff it is not a safe step till that moment. If the step is safe then it becomes part of a greedy sequence in $\bar{\omega}_\Upsilon$.

Lemma 10. *Let Υ be a closed execution in positive DABP $\mathcal{B}$ that produces ground atoms $R_1(\bar{s}_1), \ldots, R_n(\bar{s}_n)$ then there exists a greedy execution $\bar{\omega}_\Upsilon$ in $\mathcal{B}$ that also produces $R_1(\bar{s}_1), \ldots, R_n(\bar{s}_n)$.*

Proof idea. By induction on the steps in Υ by identifying each step as either a critical or a safe step in $\bar{\omega}_\Upsilon$. □

Testing Program

Now assume we want to check a query Q for stability. From the given process and query we construct test program $\Pi^{test}_{\mathcal{P},\mathcal{I},Q}$ in the same way as in the acyclic variant. Then, from Lemmas 9 and 10 the following holds.

Theorem 5. *The following two are equivalent:*

- *Q is instable in $\mathcal{B}$ under closed semantics;*
- *$\Pi^{po,cl}_{\mathcal{P},\mathcal{I}} \cup \mathcal{D} \cup \Pi^{test}_{\mathcal{P},\mathcal{I},Q} \models$ Instable.*

Complexity Results

From Theorem 5 it follows that process and combined complexity are the same as program complexity for Datalog, that is ExpTime. From Corollary 1 we have that it is also ExpTime-hard, thus it is ExpTime-complete. In the same way, from Theorem 5 and Corollary 1 we have that data complexity is PTime-complete. For query and instance complexity, we can apply the same complexity analysis of the acyclic variant, and thus complexities do not change.

Corollary 6 (Complexity Summary)**.** *For positive cyclic arbitrary DABPs under closed semantics checking stability is*

1. ExpTime*-complete in process and combined complexity;*
2. Π^{P}_2*-complete in query complexity;*
3. co-NP*-complete in instance complexity;*
4. PTime*-complete in data complexity.*

4.7 Positive Cyclic Arbitrary Open

Similarly to the acyclic case, this variant is obtained as a combination of the fresh variant under open semantics (Section 4.2) and the arbitrary variant under closed semantics (Section 4.6).

The idea is to extend greedy executions introduced for the closed variant. In particular, we extend greedy sequences to include traversals of fresh instances, meaning that now a greedy sequence includes traversals of running instances to produce the maximum that they can produce by cycling the process net as before, and in addition introductions and traversals of fresh instances till they produce all that they can produce. Following abstraction principle, for fresh instances it is sufficient to consider constants from the extended active domain, thus there is an upper-bound on what a greedy sequence can produce. A critical step is again a non safe step by a running instance. Similar to the closed case, to characterize greedy execution under open semantics it is sufficient to consider its critical steps only.

We take the encoding for positive cyclic arbitrary closed variant and we adapt the encoding for the positive fresh variant (Section 4.6).

Enconding into Datalog

For the encoding we consider the following relations

- $\mathit{In}_{p,i}$ for $i = 1, \ldots, mk$ of arity $\mathit{arity}(\mathit{In}) + 2i$ such that $\mathit{In}_{p,i}(\bar{\omega}; \bar{s})$ is true iff a fresh instance with *In*-record can reach place p after the greedy execution $\bar{\omega} = \langle o_{l_1}, t_{h_1}, \ldots, o_{l_i}, t_{h_i} \rangle$.
- R^i of arity $2i + \mathit{arity}(R)$ such that $R^i(\bar{\omega}, \bar{s})$ is true iff the greedy execution $\bar{\omega}$ produces $R(\bar{s})$.

To define a Datalog program, we take all the rules from the closed variant and we adapt the rules from the fresh variant as follows.

Traversal Rule. First we adapt the traversal rule from the fresh case in order to consider what is produced in a greedy execution of length i. Thus, for each execution step i up to mk and for every transition t from a place q to a place p, we introduce a *traversal rule* that copies all fresh instances that were able to reach q and that satisfy the execution condition E_t.

$$\mathit{In}_{p,i}(\overline{W}; \overline{X}) \leftarrow \mathit{In}_{q,i}(\overline{W}; \overline{X}), E^i_t(\overline{W}; \overline{X})$$

where $E^i_t(\overline{W}; \overline{X})$ denotes the condition $R^i_1(\overline{W}, \bar{s}_1), \ldots, R^i_m(\overline{W}, \bar{s}_m), \mathit{In}(\overline{X})$

Generation Rule. Similarly, we adapt the generation rule to store the facts that are produced in relations R^i's. For each transition t with writing rule $W_t\colon\ R(\bar{u}) \leftarrow B_t(\bar{s})$ we introduce the following *generation rule* for each i up to mk:

$$R^i(\overline{W}; \bar{u}) \leftarrow \mathit{In}_{q,i}(\overline{W}; \overline{X}), E^i_t(\overline{W}; \overline{X}), B^i_t(\overline{W}; \overline{X})$$

Here $B^i_t(\overline{W}; \overline{X})$ denotes a condition similar to $E^i_t(\overline{W}; \overline{X})$.

The rest of the encoding is as in the positive cyclic closed variant (Section 4.6).

Summary. Let us denote the above program with $\Pi^{po}_{\mathcal{P},\mathcal{I},\mathcal{Q}}$.

Lemma 11. *Let $\bar{\omega}$ be a greedy execution in $\langle \mathcal{P}, \mathcal{I}, D \rangle$ of length i, and $R_1(\bar{s}_1), \ldots, R_n(\bar{s}_n)$ be a set of facts. The following is equivalent:*

- *Facts* $R_1(\bar{s}_1), \ldots, R_n(\bar{s}_n)$ *are produced by* $\bar{\omega}$*;*
- $\Pi^{po}_{\mathcal{P},\mathcal{I},\mathcal{Q}} \cup D \models R^i_1(\bar{\omega}; \bar{s}_1), \ldots, R^i_n(\bar{\omega}; \bar{s}_n)$.

Testing Program

As testing program we adopt the program $\Pi^{test}_{\mathcal{P},\mathcal{I},Q}$ from the positive arbitrary closed variant.

Theorem 6. *The following two are equivalent:*

- Q *is instable in* $\mathcal{B}$ *under open semantics;*
- $\Pi^{po}_{\mathcal{P},\mathcal{I},\mathcal{Q}} \cup \mathcal{D} \cup \Pi^{test}_{\mathcal{P},\mathcal{I},Q} \models$ *Instable.*

Complexity Results

Complexities are inherited from the arbitrary variant.

Corollary 7 (Complexity Summary)**.** *For positive cyclic arbitrary DABPs under open semantics checking stability is*

1. EXPTIME*-complete in process and combined complexity;*
2. Π^{P}_2*-complete in query complexity;*
3. CO-NP*-complete in instance complexity;*
4. PTIME*-complete in data complexity.*

4.8 Normal Cyclic Arbitrary Closed

As shown, to check stability in the positive cyclic DABPs, it is sufficient to consider a special kind of executions, called greedy executions, that produce the maximal extended databases. For such executions the number of critical execution steps is polynomial in the size of instances and transitions. In the presence of negation, inserting new facts may also disable transitions. In principle, a transition may switch from being enabled to being disabled many times. Thus, the greedy executions in DABPs with negation cannot be efficiently applied as enabled graphs may also shrink, and then the number of critical steps that one has to consider is not polynomial anymore (it is exponential). Hence, if we directly encode the greedy executions as tuples we obtain a decision procedure that checks stability in double exponential time (as discussed in Section 4.6).

In the following we establish a more optimal approach. We establish correspondence between stability and brave query answering for Datalog with (unstratified) negation. For a Datalog program Π with negation we consider *Stable Model Semantics* (SMS [7]). Intuitively, under SMS, the program Π may have more than one set of facts that satisfy all rules (a model) that is minimal, called *stable model* (SM). A query Q has tuple $\bar{a}$ as a *brave answer* over Π if there exists a SM of Π such that $\bar{a}$ is an answer of Q over that model.

In the following we establish an encoding into Datalog with negation such that for a given DABP $\mathcal{B} = \langle \mathcal{P}, \mathcal{I}, \mathcal{D} \rangle$ and a query Q,

i) each execution in $\mathcal{B}$ is encoded as a SM of the encoding program, and

ii) checking stability of the query is done using brave query answering over the program.

That is, a query Q is instable iff Q has a new query answer in one of the SMs of the program.

Generating Exponentially Big Linear Order

Assume we are given a DABP $\mathcal{B} = \langle \mathcal{P}, \mathcal{I}, \mathcal{D} \rangle$ possible with cycles and negation in the rules. As we discussed (in Section 4.6), to check stability in $\mathcal{B}$ cyclic DABPs it is sufficient to consider executions that have up to

$$mkrc^a$$

executions steps, where m, k, r, c and a are parameters of $\mathcal{B}$ as defined in Section 4.6. The idea is to define a program that for each execution step non-deterministically selects one process instance and one transition, meaning that in that step the selected instance traverses the selected transition. In that way, each SM of the program is a combination of possible selections of instances and transitions, that represents an execution of length $mkrc^a$. Since, the selection is done for every combination of instances and transitions, each execution will have a corresponding SM. On the other hand, some SMs may not correspond to any execution. We do not consider these cases when checking stability.

In order to enumerate $mkrc^a$ execution steps we can, in principle, introduce a linear order of that size. However, such an approach will not be optimal as it requires exponential space in the size of the process to store the order. Again, as a more optimal approach we introduce a Datalog program that generates a liner order of size $mkrc^a$ starting from a much smaller order (exponentially smaller). To define a small order we

introduce a set of constants, called *digits*, $\mathit{Dig}_{\mathcal{B}} = \{d_1, \ldots, d_l\}$ of size l an we establish one linear order $<$ on $\mathit{Dig}_{\mathcal{B}}$:

$$d_1 < d_2 < \cdots < d_l$$

Assume that $\mathit{Dig}^g_{\mathcal{B}}$ is the Cartesian power of $\mathit{Dig}_{\mathcal{B}}$ of size g. That is, each tuple $\bar{k}$ from $\mathit{Dig}^g_{\mathcal{B}}$ is of the form:

$$\bar{k} = \langle d_{i_1}, \ldots, d_{i_g} \rangle,$$

for $d_{i_1}, \ldots, d_{i_g} \in \mathit{Dig}_{\mathcal{B}}$. We define $<^g$ as the lexicographical order on the tuples from $\mathit{Dig}^g_{\mathcal{B}}$. Here l and g are selected such that *(i)* $l = kc$ and thus depends on $\mathcal{P}, \mathcal{I}$ and D, and *(ii)* $g > m + r + a$ where m, r, and a are the parameters that depend only on $\mathcal{P}$. Then, it is not hard to check that it holds

$$l^g \geq mkrc^a.$$

In other words, linear order on $<^g$ is sufficient to enumerate all executions steps.

Adopting the idea in [7], we define a positive Datalog program that generates $<^g$. In particular, we want generate a relation *Succ* that stores the immediate successor in the order $<^g$.

We introduce the following relations that we use to encode the counting of the execution steps.

- *First*$(\bar{k})$ – auxiliary relation of arity g that is true iff $\bar{k}$ is the first element of the linear order $<^g$. We use the first element to enumerate the initialization step, thus it does not correspond to an execution step.
- *Step*$(\bar{k})$ – g-ary relation that is true iff $\bar{k}$ is a tuple from $\mathit{Dig}^g_{\mathcal{B}}$ that corresponds to some execution step, that is any tuple from $\mathit{Dig}^g_{\mathcal{B}}$ except for the first one in the order $<^g$
- *Succ*$(\bar{k}_1; \bar{k}_2)$ – relation of arity $2g$ that is true iff $\bar{k}_2$ is the immediate successor of $\bar{k}_1$ in the order $<^g$

To achieve this we need to generate successor relations Succ^i of size $2i$ (for $1 \leq i \leq g$). For that we use the following relations:

- *Digit*(d) – is true iff $d \in \mathit{Dig}_{\mathcal{B}}$
- $\mathit{First}^i(\bar{d})$ – is true iff $\bar{d}$ is the first element of the linear order $<^i$
- $\mathit{Last}^i(\bar{y})$ – is true iff $\bar{y}$ is the last element of the linear order $<^i$
- $\mathit{Succ}^i(\bar{x}, \bar{y})$ – is true iff $\bar{x}$ is the successor of $\bar{y}$ in the linear order $<^i$

Initialization. We initialize relations *Digit*, First^1, Last^1 and Succ^1 as follows:

Digit
d_1
$\vdots$
d_l

First^1
d_1

Last^1
d_l

Succ^1	
d_2	d_1
$\vdots$	
d_l	d_{l-1}

Successor Rules. To generate Succ^i, First^i, and Last^i we introduce the following rules:

- $\mathit{Succ}^{i+1}(Z, \overline{X}; Z, \overline{Y}) \leftarrow \mathit{Digit}(Z), \mathit{Succ}^{i}(\overline{X}; \overline{Y})$
- $\mathit{Succ}^{i+1}(Z_1, \overline{X}; Z_2, \overline{Y}) \leftarrow \mathit{Succ}^{1}(Z_1; Z_2), \mathit{First}^{i}(\overline{X}), \mathit{Last}^{i}(\overline{Y})$
- $\mathit{First}^{i+1}(X_1, \overline{X}) \leftarrow \mathit{First}^{1}(X_1), \mathit{First}^{i}(\overline{X})$
- $\mathit{Last}^{i+1}(Y_1, \overline{Y}) \leftarrow \mathit{Last}^{1}(Y_1), \mathit{Last}^{i}(\overline{Y})$

Then, we populate relation *Step* with the following rule:

$$\mathit{Step}(K_1, \ldots, K_g) \leftarrow \mathit{Digit}(K_1), \ldots, \mathit{Digit}(K_g), \neg \mathit{First}(K_1, \ldots, K_g)$$

In the following we use *Succ* for Succ^g, *First* for First^g, and *Last* for Last^g.

We denote the above program as $\Pi^{succ}_{\mathcal{P}} \cup D_{\mathcal{B}}$. Here, $\Pi^{succ}_{\mathcal{P}}$ is a set of Datalog rules that is polynomial in the size of g and thus depends only on $\mathcal{P}$, and $D_{\mathcal{B}}$ is a database instance that contains digits $D_{\mathcal{B}} = \{\mathit{Digit}(d) \mid d \in \mathit{Dig}_{\mathcal{B}}\}$ and thus depends on $\mathcal{B}$. The program generates $<^g$ by generating immediate successors and storing them in relation *Succ* of size $2g$. In addition, the program generates all tuples from $\mathit{Dig}^g_{\mathcal{B}}$ and stores them in relation *Step* of size g. In particular,

Lemma 12. *Let $\bar{k}$, $\bar{k}_1$ and $\bar{k}_2$ be tuples of size g, then:*

(i) $\Pi^{succ}_{\mathcal{P}} \cup D_{\mathcal{B}} \models \mathit{Succ}(\bar{k}_1, \bar{k}_2)$ *iff* $\bar{k}_2$ *is the successor of* $\bar{k}_1$.

(i) $\Pi^{succ}_{\mathcal{P}} \cup D_{\mathcal{B}} \models \mathit{Step}(\bar{k})$ *iff* $\bar{k} \in \mathit{Dig}^g$*;*

In other words, the program $\Pi^{succ}_{\mathcal{P}} \cup D_{\mathcal{B}}$ generates the linear order in PTIME in the size of data and instances, and in EXPTIME in the size of process. It is believed that EXPTIME is a strict subset of EXPSPACE.

Encoding Stability into Datalog with Negation

In the following we define a Datalog program with negation that, based on the linear order from above, produces all maximal extended databases. Each maximal extended database is going to be encoded as one of the SMs of the program. The program adapts *guess and check* approach from answer-set programming [16] that organizes rules in guessing rules that generate SM candidates, and checking rules that discard bad candidates. Let us consider the following example to provide the intuition.

Example. Consider unary relations *Select*, *NonSelect*, and *Const*. Assume that relation *Const* is populated with a set constants $\{c_1, \ldots, c_n\}$. We want to define a program whose stable models are such that for each model at most one constant c_j is selected to be in *Select* while the others are in *NonSelect*. I.e., we want to get the following SMs:

$$\begin{aligned}
&\{\mathit{Select}(c_1), \mathit{NonSelect}(c_2), \ldots, \mathit{NonSelect}(c_n)\},\\
&\{\mathit{NonSelect}(c_1), \mathit{Select}(c_2), \ldots, \mathit{NonSelect}(c_n)\},\\
&\quad \ldots\\
&\{\mathit{NonSelect}(c1), \mathit{NonSelect}(c_2), \ldots, \mathit{Select}(c_n)\}.
\end{aligned}$$

We start by the following two rules that partition the set of constants into *Select* and *NonSelect*:

$$\begin{aligned}
\mathit{Select}(X) &\leftarrow \mathit{Const}(X), \neg \mathit{NonSelect}(X)\\
\mathit{NonSelect}(X) &\leftarrow \mathit{Const}(X), \neg \mathit{Select}(X).
\end{aligned}$$

Intuitively, they "enforce" each constant from *Const* to be either in *Select* or in *NonSelect*.

To have only SMs such that at most one constant is in *Select*, we introduce the following integrity constraint:

$$\bot \leftarrow \textit{Select}(X_1), \textit{Select}(X_2), X_1 \neq X_2$$

If special symbol $\bot$ is implied in a SM, then such SM is discarded. Note that, this behaviour can be achieved even without the special symbol $\bot$ but in a less convenient way. $\triangle$

Encoding. Now we are ready to provide the encoding for our program. In particular, our program

i) has guessing rules that generate each possible execution of the processes up to the size of the linear order;

ii) has checking rules that discard badly guessed executions (e.g., two instances traverse at the same time), and

iii) has rules that computes all facts produced at each step of the guessed executions.

To encode guessing of executions we introduce the following relations of size $g + 1$:

- *Moved*$(\bar{k}, o)$ means that instance o traverses at step $\bar{k}$;
- *NotMoved*$(\bar{k}, o)$ means the opposite;
- *Trans*$(\bar{k}, t)$ means that at step $\bar{k}$ transition t is traversed; and
- *NotTrans*$(\bar{k}, t)$ means the opposite.

Then, we introduce relations to store what is produced by a guessed execution.

- *Completed*$(\bar{k})$ – relation of size g that we use to keep track of the steps that are completed. That is, *Completed*$(\bar{k})$ is true if step $\bar{k}$ is completed and steps that precede $\bar{k}$ are also completed.
- *Place*$(\bar{k}, o, p)$ – is true iff after $\bar{k}$-th step instance o is at place p.

Then, similarly to fresh DABPs, to store facts that are produced up to a certain step we introduce prime version relation R' for each R in $\Sigma_{\mathcal{B}}$.

- $R'(\bar{k}, \bar{s})$ is true iff $R(\bar{s})$ is produced up to step $\bar{k}$.

Finally, we assume transitions of the process to be stored in the relation *Transition*.

Initialization Rules. The first step *First*$(\overline{K})$ corresponds to the initial state of the process.

We first initialize the relation *Place* to store the initial position of the running instances. That is, for every running instance o initially at place p we initialize the starting position as follows:

$$\textit{Place}(\overline{K}; o, p) \leftarrow \textit{First}(\overline{K})$$

Then, we say that the initialization step is conclude in the following way:

$$\textit{Completed}(\overline{K}) \leftarrow \textit{First}(\overline{K})$$

Transition guessing. Then, for each execution step we introduce rules to guess a transition to be traversed.

At step $\overline{K}$ a transition t is either guessed to be executed (*Trans*$(\overline{K};T)$) or it is guessed not to be executed (*NotTrans*$(\overline{K};T)$), using the following rules:

$$\textit{Trans}(\overline{K};T) \leftarrow \textit{Step}(\overline{K}), \textit{Transition}(T), \neg\textit{NotTrans}(\overline{K};T)$$
$$\textit{NotTrans}(\overline{K};T) \leftarrow \textit{Step}(\overline{K}), \textit{Transition}(T), \neg\textit{Trans}(\overline{K};T)$$

Transition Checking. To ensure that only one transition is in relation *Trans* for step $\overline{K}$ we add the following functional integrity constraint on *Trans*.

$$\bot \leftarrow \textit{Step}(\overline{K}), \textit{Trans}(\overline{K};T_1), \textit{Trans}(\overline{K};T_2), T_1 \neq T_2$$

Intuitively, symbol $\bot$ means: disallow each candidate for a SM that fires this rule. Similarly, we introduce another functional integrity constraint to ensure that at least one instance is selected at each execution step.

Instance guessing. Similarly to transition guessing, we encode the guessing of a single instance with the following rules

$$\textit{Moved}(\overline{K};O) \leftarrow \textit{Step}(\overline{K}), \textit{In}^0(O,\_), \neg\textit{NotMoved}(\overline{K};O)$$
$$\textit{NotMoved}(\overline{K};O) \leftarrow \textit{Step}(\overline{K}), \textit{In}^0(O,\_), \neg\textit{Moved}(\overline{K};O)$$

Instance Checking. And the functional integrity constraint on *Moved* is:

$$\bot \leftarrow \textit{Step}(\overline{K}), \textit{Moved}(\overline{K};O_1), \textit{Moved}(\overline{K};O_2), O_1 \neq O_2$$

Copy Rules. At the first step all facts from the database are produced. Thus we introduce the following rule

$$R'(\overline{K};\overline{Y}) \leftarrow R(\overline{Y}), \textit{First}(\overline{K})$$

Then, what is produced in a step is copied to the subsequent step with the following rule

$$R'(\overline{K}_2;\overline{Y}) \leftarrow R'(\overline{K}_1;\overline{Y}), \textit{Succ}(\overline{K}_2,\overline{K}_1)$$

Execution rules. The above rules selects for each execution step, an instance that executes and a transition to be traversed. However, in order for the step to be a legal execution step we need to ensure that:

i) the selected instance o satisfies the execution condition of the selected transition t;

ii) that the instance o is at place q where t originates; and

iii) all previous execution steps were already completed.

Thus, for every transition t that issues from q we introduce the following rule:

$$\begin{aligned}\textit{Completed}(\overline{K}_2) \leftarrow &\textit{Moved}(\overline{K}_2;O), \textit{Trans}(\overline{K}_2;t), \textit{Succ}(\overline{K}_2,\overline{K}_1),\\ &\textit{Completed}(\overline{K}_1), E_t(\overline{K}_1;O), \textit{Place}(\overline{K}_1;O;q).\end{aligned}$$

Condition $E_t(\overline{K}_1; O)$ is defined similarly to the positive acyclic variant, where the execution $\bar{\omega}$ is replaced with the execution step $\overline{K}_1$.

Generation rule. For every transition t with writing rule $R(\bar{u}) \leftarrow \mathit{In}(\bar{s}), B_t(\bar{s})$ we introduce the following rule:

$$R'(\overline{K}_2; \bar{u}) \leftarrow \mathit{Moved}(\overline{K}_2; O), \mathit{Trans}(\overline{K}_2; t), \mathit{Succ}(\overline{K}_2, \overline{K}_1), \\ \mathit{Completed}(\overline{K}_2), \mathit{In}^0(O; \bar{s}), B_t(\overline{K}_1; \bar{s}).$$

Condition $B_t(\overline{K}_1; O)$ is defined similarly to the positive acyclic variant, where the execution $\bar{\omega}$ is replaced with the execution step $\overline{K}_1$.

Updating position. To update the position of the object that executes transition t pointing at place p, we introduce the following rule

$$\mathit{Place}(\overline{K}; O; p) \leftarrow \mathit{Moved}(\overline{K}; O), \mathit{Trans}(\overline{K}; t), \mathit{Completed}(\overline{K})$$

All other instances will not change their places

$$\mathit{Place}(\overline{K}_2; O; P) \leftarrow \mathit{Place}(\overline{K}_1; O; P), \mathit{Succ}(\overline{K}_2, \overline{K}_1), \\ \mathit{Completed}(\overline{K}_2), \neg \mathit{Moved}(\overline{K}_2; O)$$

Summary. Let the above rules define the program $\Pi^{cl}_{\mathcal{P}}$ for closed (*cl*) DABPs, and let $D_{\mathcal{I}}$ be a database instance that contains In^0 facts.

Lemma 13. *Let $\bar{k}$ be an execution step in $\mathcal{B}$, and let $R(\bar{s})$ be a fact. The following is equivalent:*

- *There is an execution of length $\bar{k}$ in $\mathcal{B}$ that produces $R(\bar{s})$;*
- $\Pi^{\mathit{succ}}_{\mathcal{P}} \cup D_{\mathcal{B}} \cup \Pi^{\mathit{cl}}_{\mathcal{P}} \cup D_{\mathcal{I}} \cup D \models_{\mathit{brave}} R'(\bar{k}; \bar{s})$.

Note that in this case the encoding rules have simpler format than in the positive case. This is because the complexity of the problem is now transferred to the semantics of the encoding program, that is SMS.

Testing Program

Now we want to test query Q for stability. Similarly as before, we collect new query answers in relation Q' with the rule:

$$Q'(\overline{X}) \leftarrow R'_1(\overline{K}; \bar{u}_1), \ldots, R'_n(\overline{K}; \bar{u}_n).$$

Let Π^{test}_Q be the test program containing Q, Q' and the test rule as in previous cases. Then the following holds:

Theorem 7. *The following is equivalent:*

- *Q is instable in $\mathcal{B} = \langle \mathcal{P}, \mathcal{I}, \mathcal{D} \rangle$ under closed semantics;*
- $\Pi^{\mathit{succ}}_{\mathcal{P}} \cup D_{\mathcal{B}} \cup \Pi^{\mathit{cl}}_{\mathcal{P}} \cup D_{\mathcal{I}} \cup D \cup \Pi^{\mathit{test}}_Q \models_{\mathit{brave}} \mathit{Instable}$.

Complexity Results

The above Theorem provides an upper bound for process, data and combined complexity.

Process complexity The program from Theorem 7 is a Datalog program with negation that is interpreted under SMS. Query answering in such programs is in CO-NEXPTIME. We can show also the opposite, that the reasoning is CO-NEXPTIME-hard. Thus, our encoding of the program is optimal (as hard as the stability problem).

To show the hardness we show that instability is NEXPTIME-hard by simulating brave query answering over a Datalog program. For a program $\Pi \cup \mathcal{D}$ and a fact query A, we construct a singleton DABP $\mathcal{B}_{\Pi,A} = \langle \mathcal{P}_{\Pi,A}, \mathcal{I}_0, \mathcal{D} \rangle$ that produces *dummy* iff there exists a stable model of $\Pi \cup \mathcal{D}$ that contains A. Here, program Π is encoded in the process part and data part of the program $\mathcal{D}$ is encoded in the database of the process. As usual, the test query is Q_{test}.

Proposition 4. *$\Pi \cup \mathcal{D} \models_{brave} A$ iff Q_{test} is stable in $\langle \mathcal{P}_{\Pi,A}, \mathcal{I}_0, \mathcal{D} \rangle$.*

Proof. See Subsection 4.8.1. □

Instance, Data and Query Complexity From Theorem 7 we have that instance complexity is in coNP since data complexity of brave query answering is NP-hard. Also this is the lower-bound as from Proposition 4 we have that data complexity of brave query answering is at least as hard as checking instability. Similarly, from Theorem 7 we have that instance complexity is in CO-NP. It is also CO-NP-complete since it is CO-NP-hard already for acyclic case. Finally, query complexity is Π_2^{P}-complete for the same reasons as in other cases.

Corollary 8 (Complexity Summary)**.** *For normal cyclic arbitrary DABPs under closed semantics checking stability is*

1. CO-NEXPTIME*-complete in process and combined complexity;*
2. Π_2^{P}*-complete in query complexity;*
3. CO-NP*-complete in instance and data complexity.*

4.8.1 CO-NEXPTIME- and CO-NP-hardness in Process and Data

In the following we prove Proposition 4 defined above.

In particular, we show how to encode the brave reasoning under Stable Model Semantics (SMS) for a given Datalog program $\Pi \cup \mathcal{D}$ with negation into stability problem for normal cyclic singleton DABP $\langle \mathcal{P}_{\Pi,A}, \mathcal{I}_0, \mathcal{D} \rangle$ under closed semantics, where program Π is encoded in the process part and data part of the program $\mathcal{D}$ is encoded in the database of the process. As usual, the test query is Q_{test}.

Standard notation for Datalog program with negation. For Datalog programs under stable model semantics (SMS) we use the following notation. A normal Datalog rule is a rule of the form

$$R(\bar{u}) \leftarrow R_1(\bar{u}_1), \ldots, R_l(\bar{u}_l), \neg R_{l+1}(\bar{u}_{l+1}), \ldots, \neg R_h(\bar{u}_h).$$

We use H to denote the head of the rule $R(\bar{u})$, and $A_1, \ldots, A_l, \neg A_{l+1}, \ldots, \neg A_h$ to denote body atoms $R_1(\bar{u}_1), \ldots, R_l(\bar{u}_l), \neg R_{l+1}(\bar{u}_{l+1}), \ldots, \neg R_h(\bar{u}_h)$ Then we can write the rules r as:

$$H \leftarrow A_1, \ldots, A_l, \neg A_{l+1}, \ldots, \neg A_h.$$

We represent a fact $R(\bar{u})$ as a Datalog *fact rule* $R(\bar{u}) \leftarrow$.

A Datalog program with negation Π is a finite set of normal Datalog rules $\{r_1, \ldots, r_k\}$.

Grounding of a Datalog program:

- Let r be a normal Datalog rule and C a set of constants. The *grounding* $gnd_C(r)$ of r is a set of rules without variables obtained by substituting the variables in r with constants from C in all possible ways. In this way we can obtain several grounded rules from a non-grounded rule.
- The grounding *gnd*(Π) for a program Π is a program obtained by grounding rules in Π using the constants from Π.
- We note that program *gnd*(Π) and Π have the same semantic properties (they have the same SM, see later). Program *gnd*(Π) is just an expanded version of Π (it can be exponentially bigger than Π).

Stable model semantics. Concerning stable model semantics we use the following notation. An *interpretation* of a program represented as a set of facts. Let M be an interpretation. We define the *reduct* of Π for M as the ground positive program

$$\Pi^M = \{A \leftarrow A_1, \ldots, A_l \mid A \leftarrow A_1, \ldots, A_l, \neg A_{l+1}, \ldots, \neg A_h \in gnd(\Pi), \\ M \cap \{A_{l+1}, \ldots, A_h\} = \emptyset\}$$

Since Π^M is a positive ground program it has a unique Minimal Model (MM), in the inclusion sense Then,

$$M \text{ is a } \textit{stable model} \text{ (SM) of } \Pi \quad \text{iff} \quad M \text{ is the minimal model of } \Pi^M.$$

Given a program Π and a fact A we say that

$$\Pi \models_{brave} A$$

if there exists a SM M of Π such that $A \in M$.

For a given Π and a fact A, deciding whether $\Pi \models_{brave} A$ is NEXPTIME-hard.

Encoding of Brave Entailment into Stability Problem

Given a program $\Pi \cup \mathcal{D}$ and a fact A we construct a DABP $\mathcal{B}_{\mathcal{P},\mathcal{D},A}\langle \mathcal{P}_{\Pi,A}, \mathcal{I}_0, \mathcal{D}\rangle$ such that for a test query $Q_{test} \leftarrow$ *dummy* the following holds:

$$\Pi \cup \mathcal{D} \models_{brave} A \text{ iff } Q_{test} \text{ is stable in } \langle \mathcal{P}_{\Pi,A}, \mathcal{I}_0, \mathcal{D}\rangle.$$

For convenience, in the following we use Π to denote $\Pi \cup \mathcal{D}$, unless otherwise is stated. Intuitively, process $\mathcal{B}_{\mathcal{P},\mathcal{D},A}$ is constructed such that the following holds.

- The process generates all possible interpretations for Π using the variables and constants from Π. That is, it generates all possible candidates for SMs of Π.
- For every such SM candidate M, the process checks if M is a SM of Π by:

  i) computing the MM of Π^M denoted with M';

  ii) checking if $M' = M$.
- If M is a SM of Π then the process checks for the given fact A whether it holds that $A \in M$. If so, the process produces *dummy*.

We organize $\mathcal{B}_{\Pi,A}$ in 6 subprocesses represented in Figure 16.

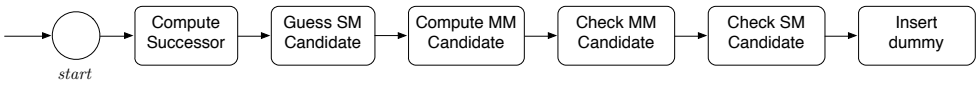


Figure 16: Subprocesses composing the process net of $\mathcal{P}_{\Pi,A}$.

The subprocesses are intuitively defined as follows:

Subp 1. *(Compute successor relations)* First we compute the successor relations $Succ^i$ of sufficient size i, that we need in the next steps. This we need for technical reasons.

Subp 2. *(Guess a SM candidate)* At this step, the process produces a SM candidate by non-deterministically producing facts obtained from relations and constants that appear in the program. Let R be a relation in Π. Then, for each R-fact that can be obtained by taking the constants from Π, a process does an execution step at *the choice place* from which if an instance traverse one way the process produces this R-facts, and if it traverses the other way then it does not.

We denote with M the guessed SM candidate.

Subp 3. *(Compute a MM candidate of the reduct)* We want compute the MM of the reduct Π^M. To do so, we first compute a candidate M' for the MM by by non-deterministically applying the rules of Π^M. Computing a candidate and the testing if the candidate is the MM is our approach to find the MM.

Subp 4. *(Check if M' is the MM of the reduct)* At this step we check if M' is indeed the minimal model of the reduct Π^M. If this is not the case, the process is not going to progress further.

Subp 5. *(Check if SM candidate is a SM)* If we are at this step then M' is the MM of Π^M. Now we check if $M' = M$. If this is the case, then M is a stable model of Π.

Subp 6. *(Insert dummy)* Finally, we check if $A \in M$. If this is the case then the process produces *dummy*.

Instance and data part.

We initialize the instance part $\mathcal{I}_0$ by placing a single instance at the start place, we set database to be the data part of the program $\mathcal{D}$.

Process part.

In the following we construct the process part $\mathcal{P}_{\Pi,A}$.

Subp 1: Computing successor relations.

In order to nondeterministically select which R-facts to produce for a relation R in Π, we introduce sufficiently big linear order that index all R-facts. Since there are exponentially many R-facts we define the process rules that compute the order starting from an order of a polynomial size. The rules that compute the exponentially big order uses the same rules define as in Lemma 12. Here, the difference is that we use constants from Π as digits.

Let $C = \{b_1, \ldots, b_c\}$ be the constants from Π. We define a linear order $<$ on C such that

$$b_1 < b_2 < \cdots < b_c.$$

Let $<^j$ be the lexicographical order linear order on C^j, defined from $<$ for some $j > 0$.

Further, let n be the maximum between

- the maximal arity of a relation in Π; and
- the largest number of variables in a rule in Π.

We want to compute the successor relation *Succ*^{j} that contains immediate successors in the order $<^j$ for $j = 1, \ldots, n$

Vocabulary and Symbols. To encode the order as database relations we introduce relations: *Const* of size 1 to store constants from Π; *Succ*^{j} of size $2j$ to store immediate successors in the order $<^j$; *First*^{j} and *Last*^{j} to store the first and the last element of the order $<^j$. That is,

- *Const*(b) – is true iff b is a constant from Π.
- *Succ*$^j(\bar{b}, \bar{b}')$ – is true iff $\bar{b}$ is the immediate successor of $\bar{b}'$ in the order $<^j$;
- *First*$^j(\bar{b})$ – is true iff $\bar{b}$ is the first element in the order $<^j$;
- *Last*$^j(\bar{b})$ – is true iff $\bar{b}$ is the last element in the order $<^j$.

Initialization. We initialize relations for the ordering as follows:

- *Const*(b) – we intialize relation *Const* with all the constants from Π;

- $Succ^1(b, b')$ – we initialize relation $Succ^1$ saying that b' is the successor of b;
- $First^j(b, \ldots, b)$ – is the initialization for relation $First^j$ such that b is the first element in the order $<$;
- $Last^j(b, \ldots, b)$ – is the initialization for relation $Last^j$ such that b is the last element in the order $<$.

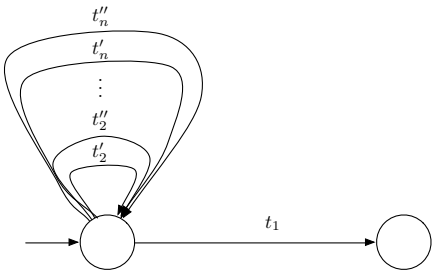


Figure 17: Subprocess 1 computes successor relations $Succ^i$ for $i = 1, \ldots, n$

Encoding into the process. We introduce $2n - 1$ transitions $t_1, t'_2, t''_2, \ldots, t'_n, t''_n$ (see Figure 17) such that t'_j and t''_j are used to generate $Succ^j$. Then, we set the execution condition for these transitions to be always executable:

$$E_{t'_j} = E_{t''_j} = true.$$

We use the writing rules to populate the relations $Succ^j$ for $1 < j \leq n$ as follows:

$$W_{t'_{j+1}} : Succ^{j+1}(Z, \overline{X}, Z, \overline{Y}) \leftarrow Const(Z), Succ^j(\overline{X}, \overline{Y});$$
$$W_{t''_{j+1}} : Succ^{j+1}(Z_1, \overline{X}, Z_2, \overline{Y}) \leftarrow Succ^1(Z_1, Z_2), First^j(\overline{X}), Last^j(\overline{Y}).$$

Once all successor relations are generated transition t_1 can be executed:

$$E_{t_1} : Succ^n(\overline{X}, \_), Last^n(\overline{X}).$$

Subp 2: Guessing a SM candidate.

Let $R_1, \ldots, R_m$ be the relations in Π. For every relation R in Π we create a subprocess *Guess-*R that non-deterministically guesses R-facts that belong to a SM candidate M.

Subprocess 2 is composed by connecting subprocess *Guess-*R for each relation R as depicted in Figure 18.

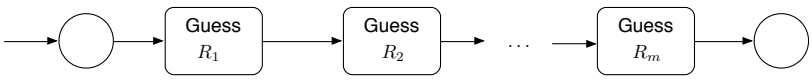


Figure 18: Subprocess 2

Notation. We assume the following notations: a arity of a relation R in Π; m is the number of relations in Π; Done_R is a relation of arity a such that $\mathit{Done}_R(\bar{u})$ is true after the subprocess *Guess-*R has guessed whether to include $R(\bar{u})$-fact in the SM candidate or not

Encoding into DABPs. The subprocess *Guess-*R is defined as in Figure 19

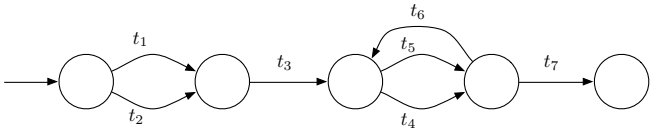


Figure 19: Subprocess *Guess-*R

For convenience introduce condition $\mathit{Current}_R(\overline{X})$ that is true if the next $R(\overline{X})$-fact for which the process has to decide whether to include it in the SM candidate or not. The condition is defined with:

$$\mathit{Current}_R(\overline{X}) : \mathit{Succ}^a(\overline{X}, \overline{Y}), \mathit{Done}_R(\overline{Y}), \neg \mathit{Done}_R(\overline{X}).$$

Transitions t_1 and t_2 are executed non-deterministically. Intuitively, they non-deterministically decide whether the R-fact, obtained by grounding R with constants from First^a, belongs to the SM candidate (t_1) or not (t_2):

$$E_{t_1} = E_{t_2} : \mathit{true};$$
$$W_{t_1} : R(\overline{X}) \leftarrow \mathit{First}^a(\overline{X});$$
$$W_{t_2} : \mathit{true} \leftarrow \mathit{true}.$$

Then, transition t_3 inserts that the guess for the first R-fact has been made by inserting $\mathit{Done}_R(\bar{x})$:

$$E_{t_3} : \mathit{true};$$
$$W_{t_3} : \mathit{Done}_R(\overline{X}) \leftarrow \mathit{First}^a(\overline{X}).$$

Transitions t_4 and t_5, similarly to transitions t_1 and t_2, non-deterministically guess whether the next $R(\overline{X})$-fact belongs to the SM candidate or not:

$$E_{t_4} = E_{t_5} : \mathit{true};$$
$$W_{t_4} : R(\overline{X}) \leftarrow \mathit{Current}_R(\overline{X});$$
$$W_{t_5} : \mathit{true} \leftarrow \mathit{true}.$$

Transition t_6, similarly to transition t_3, inserts fact $\mathit{Done}_R(\overline{X})$ after decision for $R(\overline{X})$-fact has been made:

$$E_{t_6} : \mathit{true};$$
$$W_{t_6} : \mathit{Done}_R(\overline{X}) \leftarrow \mathit{Current}_R(\overline{X}).$$

When all guesses have been made, transition t_7 can be executed and the next subprocess will be executed:

$$E_{t_7} : \mathit{Done}_R(\overline{X}), \mathit{Last}^a(\overline{X});$$
$$W_{t_7} : \mathit{true} \leftarrow \mathit{true}.$$

Subp 3: Compute the minimal model of the reduct.

The subprocesses 3 and 4 compute the MM M' of Π^M. Intuitively, this done in the following way:

- Since Π^M is a positive ground program the MM of Π^M is unique and it can be computed as the Least Fixed Point (LFP) on the rules in Π^M.
- In subprocess 3, depicted in Figure 20, the process produces facts that are in the LFP of Π^M. For every relation R we introduce a relation R' that stores facts produced by the LFP computation.
- In principle, subprocess 3 can produce all facts from the LFP if it executes a sufficient number of times. However, it can produce also only a part of the LFP if it decides to traverse t_{k+1}.
- In other words, subprocess 3 non-deterministically decides how many facts from the LFP to produce.
- In subprocess 4 we check if all facts from the LFP of Π^M are indeed produced at subprocess 3.

Vocabulary and Symbols.

- $R'(\bar{u})$ – holds iff $R(\bar{u})$ is in the LFP of Π^M (i.e. it is in the MM of Π^M) and it is computed by subprocess 3.

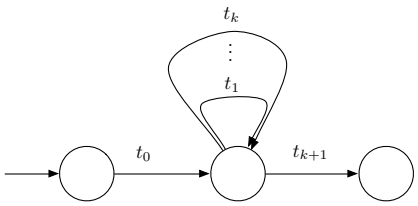


Figure 20: Subprocess 3 computes the MM candidate of the reduct

Encoding into DABP. Let $\{r_1, \ldots, r_k\}$ be the rules in Π. For every rule r_i of the form $H \leftarrow A_1, \ldots, A_l, \neg A_{l+1}, \ldots, \neg A_h$ we introduce transition t_i as depicted in Figure 20 with execution condition:

$$E_{t_i} : \textit{true}$$

and writing rule as follows:

$$W_{t_i} : H' \leftarrow A'_1, \ldots, A'_l, A_1, \ldots, A_l, \neg A_{l+1}, \ldots, \neg A_h.$$

Here, atoms $H', A'_1, \ldots, A'_l$ are the same as $H, A_1, \ldots, A_l$, except that each relation name R is renamed with R'. Atoms $A_1, \ldots, A_l, \neg A_{l+1}, \ldots, \neg A_h$ evaluates over M and they are true iff there exists a grounding substitution θ (a substitution that replaces variables with constants) such that the ground rule $\theta A \leftarrow \theta A_1, \ldots, \theta A_l$ is in the reduct Π^M. For $l = 0$, the fact $\theta H'$ is produced by the process since the rule $\theta H \leftarrow$ is in Π^M as thus H is in the LFP of Π^M. For $l > 0$, assume that $\theta A'_1, \ldots, \theta A'_l$ are already produced by the process such that $\theta A_1, \ldots, \theta A_l$ are in the LFP of Π^M. Then we have that $\theta H'$ is produced by the process iff θH is in the LFP of Π^M.

Subp 4: Check if the computed model is a minimal model of the reduct. After the execution of subprocess 3 we obtain a MM candidate M' as a set of R'-facts produced by the process.

In this step we check if M' is indeed a MM of Π^M, because in the preceding step it may be that the process has generated a part of the LFP of Π^M.

For the check we define the process as in Figure 21, where each transition t_{r_i} checks if M' contains all facts in the LFP that can be produced by the rule r_i.

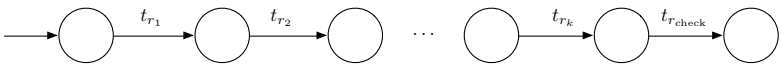


Figure 21: Subprocess 4 checks if the MM candidate is the MM of the reduct

Notation.

We introduce unary predicate fail_{MM} that is true iff M' is not a MM.

Encoding into DABPs.

For every rule r we introduce a transition t_r with execution condition

$$E_{t_r} : \mathit{true},$$

and with writing rule as follows:

$$W_{t_r} : \mathit{fail}_{MM} \leftarrow A'_1, \ldots, A'_l, \neg H', A_1, \ldots, A_l, \neg A_{l+1}, \ldots, \neg A_h$$

Fact fail_{MM} is produced by the process iff facts $\theta A'_1, \ldots, \theta A'_l$ are produced by the subprocess 3 while $\theta H'$ is not, for some substitution θ. Obviously, this is true iff M' is not the MM of the reduct.

Last transition t_{check} is executable if none of the previous steps has generated the fail_{MM} predicate:

$$E_{t_{check}} : \neg \mathit{fail}_{MM}.$$

Subp 5: Checking if SM candidate is a SM. If the process execution can reach subprocess 5 it means that M' is indeed the MM of reduct Π^M. It remains to check if M is a SM of Π, that is if $M' = M$.

For this check we define the subprocess as in Figure 22.

Transition t'_i checks if there is a R'_i-fact for which there is no R_i-fact and transition t''_i checks if there is a R_i-fact for which there is no R'_i-fact.

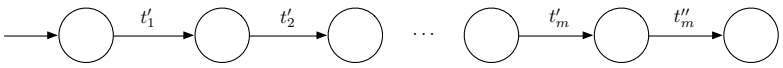


Figure 22: Subprocess 5 checks if SM candidate is a SM

Notation. We introduce unary predicate fail_i that holds if $M' \neq M$.

Encoding into DABPs. Transition t'_i is encoded as follows:

$$W_{t'_i} : \mathit{fail}_i \leftarrow R'_i(X), \neg R_i(X).$$

Transition t''_i is encoded as follows:

$$W_{t''_i} : \mathit{fail}_i \leftarrow R_i(X), \neg R'_i(X).$$

Subp 6: Insert *dummy*. After the execution of subprocess 5 if no $fail_i$ facts were produced, then $M' = M$

Subprocess 6 checks whether this is the case. If $M' = M$ and $A \in M$ the process inserts *dummy*

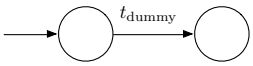


Figure 23: Subprocess 6 inserts *dummy*

Encoding into DABPs. The subprocess is depicted in Figure 23 Transition t_{dummy} checks if $M' = M$ with the execution condition:

$$E_{t_{dummy}} : \neg fail_1, \ldots, \neg fail_m.$$

By traversing t_{dummy} if condition $A \in M$ then *dummy* is inserted with the following writing rule:

$$W_{t_{dummy}} : dummy \leftarrow A.$$

All together, we have that fact A is produced by the process iff there exists a SM of the program that contains A. This concludes the proof.

5 Stability in Rowo Data-aware Business Process Model

In general, reasoning about stability in DABPs can be very complex in the worst case. The main reason is that processes can read relations that can also be written, and in that way create recursive inferences. Rowo (read-only write-only) DABPs forbid this. We will see that this affects complexities to drop significantly.

5.1 Introduction

Checking whether an arbitrary DABP is rowo can be done in linear time in the size of the process. In some cases, when a DABP is not rowo, it may behave like rowo since some of non-rowo transitions may not be executable. For instance, for a processes in a certain time interval a transition may be not executable because the deadline to execute it has already expired or it is early to execute it.

Example. Continuing our running example, imagine that the process is in the period after the pre-enrollment (after 30th Sept.). In this period, students do not have the opportunity to complete their applications, neither to confirm or withdraw their applications. Then, transitions 'pre-enrol cond.', 'complete app.', 'pre-enrol stud.', 'register app.', and 'withdraw app.' cannot be traversed since their execution conditions cannot be satisfied or they cannot be reached. The only part of the process that can be executed is shown in Figure 24. We observe that this remaining part is actually a rowo process. Thus reasoning in this case may become less complex. △

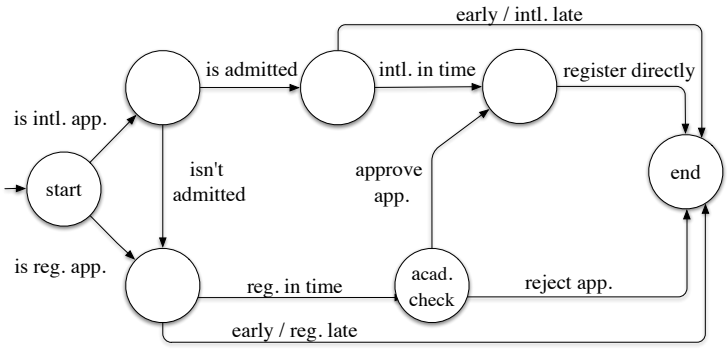


Figure 24: Student registration process after 30th September.

In principle, checking if a general process behaves as a rowo, would require checking that it does not write in a relation from which it reads. Thus, checking if a general DABP behaves as a rowo is as complex as checking stability. Still we expect that for limited parameters, such as deadlines, one can obtain a more efficient complete procedure. We leave this for future investigation.

Observations. To check stability of a rowo DABP, we construct a non-recursive Datalog program where process and data instances are encoded as a set of facts. We make several observations on why rowo encoding becomes simpler:

(i) since the written facts cannot be read, an instance cannot affect own execution or executions of the other instances by the facts it generates, and thus one can analyze produced facts by observing each instance independently;

(ii) cycles do not affect complexity since no additional fact can be produced if an instance traverses a transition more than once;

(iii) negation in the rules is always unstratified since it is on the read-only part of the database;

(iv) if n is the size of the query then the process has to insert at most n new facts to create a new query answer, and to generate those facts it needs at most n process instances.

Based on the observations above we discuss how to define the encodings for rowo DABPs based on those for the general variants.

5.2 Rowo Normal Fresh Open

First we analyze a rowo $\mathcal{B} = \langle \mathcal{P}, \mathcal{D} \rangle$ under open semantics. We adapt encodings for open positive DABPs defined in Section 4.2.

From *(ii)* we have that each instance needs not to traverse a transition more than once in order to produce the most that the transition can produce. It may need to traverse some transitions more than once to reach other transitions, but in total it is sufficient that it makes at most m^2 traversals to reach all transitions, where m is the number of transitions. Thus, it is sufficient to consider executions of a single instance of maximal size m^2, and therefore, we can eliminate recursion from traversal rules by creating a bounded derivation (as in the acyclic variant).

Based on these observations, we adapt $\Pi^{po,fr}_{\mathcal{P}}$ from Section 4.2 as follows. We introduce relations

- In^i_p for each place p in $\mathcal{P}$ and each i up to m^2, to record that a fresh instance can reach place p in i steps, that is, $\mathit{In}^i_p(\bar{s})$ is true iff a fresh instance with $\mathit{In}(\bar{s})$-record can reach place p in i steps.

Traversal Rules. For each transition t from a place q to a place p and for each i up to m we introduce a *traversal rule* as follows:

$$\mathit{In}^{i+1}_p(\overline{X}) \leftarrow \mathit{In}^i_q(\overline{X}), E_t(\overline{X})$$

Note that, as a difference from the general case, here $E_t(\overline{X})$ denotes the execution condition evaluated over the initial database (rather than on the extended database as $E'_t(\overline{X})$ would denote).

Generation Rules. Similarly, for each transition t from above we introduce the following *generation rule*:

$$R'(\bar{u}) \leftarrow \mathit{In}^i_q(\overline{X}), E_t(\overline{X}), B_t(\overline{X})$$

As pointed in the observation *(iii)* above negation in $E_t(\overline{X})$ and $B_t(\overline{X})$ does not make reasoning more complex since negation is on the base relations that are not updated by the process.

Summary. Let $\Pi^{ro,fr}_{\mathcal{P}}$ be the non-recursive Datalog program with stratified negation that encodes the rowo process $\mathcal{P}$ obtained from $\Pi^{po,fr}_{\mathcal{P}}$ substituting the traversal and generation rules with the rules above. The rest of the program is the same as in the general positive open variant.

Lemma 14. *Let $R'(\bar{u})$ be a fact defined over adom*, then the following is equivalent:*

- *there is an execution in $\mathcal{B}$ that produces $R(\bar{u})$*
- $\Pi_{\mathcal{P}}^{ro,fr} \cup \mathcal{D} \models R'(\bar{u})$

Let Π_Q^{test} be the test program based on a query Q as defined for the general variant.

Theorem 8. *The following is equivalent:*

- *Q is instable in $\mathcal{B}$ under open semantics*
- $\Pi_{\mathcal{P}}^{ro,fr} \cup \mathcal{D} \cup \Pi_Q^{test} \models$ *Instable*

5.3 Rowo Normal Arbitrary Closed

We now consider a possibly cyclic rowo $\mathcal{B} = \langle \mathcal{P}, \mathcal{I}, \mathcal{D} \rangle$ under closed semantics. We adapt the encoding from the general acyclic closed variant. The main difference is that each instance is encoded independently of the others (because of *(i)*). I.e., we encode an execution of a single instance as a tuple $\bar{\omega}$ of the form

$$\bar{\omega} = \langle o; t_{h_1}, \ldots, t_{h_i} \rangle.$$

meaning that instance o traverses first t_{h_1} then t_{h_2}, and so on.

Similarly we adapt R^i's and *State*^{i}'s from the general case such that:

- $R^i(o; t_{h_1}, \ldots, t_{h_i}; \bar{s})$ denotes that the instance o after traversing $t_{h_1}, \ldots, t_{h_i}$ produces $R(\bar{s})$; and
- *State*$^i(o; t_1, \ldots, t_i; p)$ denotes that the instance o after traversing $t_1, \ldots, t_i$ is located at place p.

Similarly to the previous variant, cycles can be dealt with bounded derivations of maximal length m^2, so i ranges from $1, \ldots, m^2$. Similarly to the general variant, we use *In*$^0(o; \bar{s})$ to associate instance o with the input record *In*$(\bar{s})$. In this way, we obtain the facts that can be produced by each instance. Then we introduce additional rules that combine facts produced by different instances. Assume we are given a query $Q(\bar{X}) \leftarrow R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ that we want to check for stability. To this end, we introduce the following relations.

- *Path* is a relation with arity $m^2 + 1$ that contains legal paths of an instance. *Path*$(o; \bar{t}, \bar{\epsilon})$ is true iff $\bar{t}$ is a legal path in $\mathcal{P}$ for instance o. For technical reasons we introduce ϵ to denote an empty transition. Then, $\bar{\epsilon}$ is vector of ϵ that we use to fill in remaining positions in *Path* ($|\bar{\epsilon}| = m - |\bar{t}|$).
- R' is an auxiliary relation of size $1 + m^2 +$ *arity*(R) that we introduce for each R in $\mathcal{B}$ to store R-facts produced by an instance. That is, $R'(o; \bar{t}, \bar{\epsilon}; \bar{s})$ in true iff $R(\bar{s})$ is produced after o traversed $\bar{t}$.
- *Exec*^{j} are relations of arity $(m^2 \times j) + j$ for every $j = 1, \ldots, k$ that combines legal paths for different n instances where n is is the number of atoms in the query. Then, *Exec*$^j(o_1, \bar{t}_1, \ldots, o_j, \bar{t}_j)$ is true iff tuple $\bar{t}_l$ is a legal path for instance o_l and if $o_h = o_l$ then $\bar{t}_h = \bar{t}_l$. This relation we use to record all combinations of instances that can contribute to create a new query answer (see point *(iv)* above), thus if two facts are produced by the same instance ($o_h = o_l$) then the facts have to be produced on the same legal path ($\bar{t}_h = \bar{t}_l$).

Again i ranges from $1, \ldots, m^2$. Now we define a program that generates those relations.

Encoding into Datalog

Initialization Rules. First we adapt the *initialization rules* for a single instance:

$$\textit{State}^0(o; q) \leftarrow \textit{true} \text{ iff } q \text{ is the starting place of instance } o, \text{ and}$$
$$R^0(O; \overline{Y}) \leftarrow \textit{In}'(O; \_), R(\overline{Y}).$$

Traversal Rules. Similarly, we adapt traversal rules to be for a single instance, as follows:

$$\textit{State}^{i+1}(O; \overline{T}, t; p) \leftarrow \textit{State}^i(O; \overline{T}; q), E_t(O)$$

for each transition t from place q to place p and $E_t(O)$ is the same as E_t except that each atom $\textit{In}(\bar{s})$ is replaced with $\textit{In}^0(O, \bar{s})$ and $\overline{T}$ is a vector of different variables of size i.

Generation Rules. For a transition t with writing rule $W_t \colon R(\bar{u}) \leftarrow B_t(\bar{s})$, the generation rules become:

$$R^{i+1}(O; \overline{T}, t; \bar{u}) \leftarrow \textit{State}^{i+1}(O; \overline{T}, t; p), B_t(O; \bar{s}).$$

Here $B_t(O)$ is obtained in the same way as $E_t(O)$.

Copy Rules. Then we adapt the copy rules as follows:

$$R^{i+1}(O; \overline{T}, t; \overline{U}) \leftarrow R^i(O; \overline{T}; \overline{U}), \text{ and}$$
$$R'(O; \overline{T}, \bar{\epsilon}; \overline{U}) \leftarrow R^i(O; \overline{T}; \overline{U})$$

where the size of ϵ-vector $|\bar{\epsilon}| = m^2 - i$.

Summary. We denote the above program as $\Pi_{\mathcal{P}}^{\textit{ro,cl}}$. Then we have that the following holds:

Lemma 15. *Let o be an instance in $\mathcal{B}$, a list of transitions $\bar{t}$ in $\mathcal{B}$ of size i, and $R(\bar{u})$ a fact, then the following is equivalent:*

- *after o traverses $\bar{t}$ the fact $R(\bar{u})$ is produced;*
- $\Pi_{\mathcal{P}}^{\textit{ro,cl}} \cup \mathcal{D} \models R^i(o; \bar{t}; \bar{u})$.

Testing Program

Combining Rules. Then we need to combine the atoms produced by the different instances, e.g.,

$$R_1^i(o_1; \bar{t}_1; \bar{s}_1)$$
$$\vdots$$
$$R_n^i(o_n; \bar{t}_n; \bar{s}_n).$$

and ensure that atoms produced by one instance are all produced following one path. To do this we need to ensure that if $o_i = o_j \Rightarrow \bar{t}_i = \bar{t}_j$. This is achieved with the *combining rules*.

First we copy all legal paths of an instance from *State*^{i} into *Path* relation:

$$\textit{Path}(O;\overline{T},\bar{\epsilon}) \leftarrow \textit{State}^i(O;\overline{T};\_) \text{ where } |\epsilon| = m^2 - i.$$

Then we initialize *Exec*^{1} with all legal paths of an instance.

$$\textit{Exec}^1(O;\overline{T}) \leftarrow \textit{Path}(O;\overline{T}).$$

Then we combine different paths in the following way. If instance O_l executes $\overline{T}_l$ and the same instance executes $\overline{T}_{i+1}$ then $\overline{T}_l$ and $\overline{T}_{i+1}$ must be the same. This is captured with the following rules:

$$\begin{aligned}\textit{Exec}^{j+1}(O_1,\overline{T}_1,\ldots,O_l,\overline{T}_l,\ldots,O_j,\overline{T}_j,O_{j+1},\overline{T}_{j+1}) \leftarrow\\ \textit{Exec}^j(O_1,\overline{T}_1,\ldots,O_l,\overline{T}_l,\ldots,O_j,\overline{T}_j),\\ \textit{Path}(O_{j+1};\overline{T}_{j+1}),\\ O_l = O_{j+1}, \overline{T}_l = \overline{T}_{j+1}\end{aligned}$$

for every $l \in 1,\ldots,i$.

If the instance O_{j+1} is different from all the other instances $O_1,\ldots,O_j$ then executing path of O_{j+1} can be any legal path

$$\begin{aligned}\textit{Exec}^{j+1}(O_1,\overline{T}_1,\ldots,O_l,\overline{T}_l,\ldots,O_j,\overline{T}_j,O_{j+1},\overline{T}_{j+1}) \leftarrow\\ \textit{Exec}^j(O_1,\overline{T}_1,\ldots,O_l,\overline{T}_l,\ldots,O_j,\overline{T}_j),\\ \textit{Path}(O_{j+1};\overline{T}_{j+1}),\\ \neg(O_1 = O_{j+1}),\neg(O_2 = O_{j+1}),\ldots,\neg(O_j = O_{j+1}).\end{aligned}$$

Q'-rule. Then, Q' collects what has been produced for relations $R_1,\ldots,R_n$ for the give query $Q(\overline{X}) \leftarrow R_1(\bar{u}_1),\ldots,R_n(\bar{u}_n)$ with the rule

$$\begin{aligned}Q'(\overline{X}) \leftarrow &\textit{Exec}^n(O_1,\overline{W}_1,\ldots,O_n,\overline{W}_n),\\ &R'_1(O_1;\overline{W}_1;\bar{u}_1),\\ &\ldots\\ &R'_n(O_n;\overline{W}_n;\bar{u}_n).\end{aligned}$$

Test Rule. The *test rule* is then as before:

$$\textit{Instable} \leftarrow Q'(\overline{X}),\neg Q(\overline{X}).$$

Summary. Let us denote with $\Pi^{\textit{test,ro}}_{\mathcal{P},Q}$ the testing program for Q defined above. The program is non-recursive Datalog with stratified negation.

Let $\mathcal{D}_{\textit{In}}$ be the database that encodes the instance part *In*.

Theorem 9. *The following are equivalent:*

- *Q is instable in $\mathcal{B}$ under closed semantics;*
- $\Pi^{\textit{ro,cl}}_{\mathcal{P}} \cup \mathcal{D}_{\textit{In}} \cup \mathcal{D} \cup \Pi^{\textit{test,ro}}_{\mathcal{P},Q} \models \textit{Instable}$.

5.4 Rowo Normal Arbitrary Open

For arbitrary rowo under open semantics, similarly to the general variants, the encoding is obtained combining the encodings for the open and the closed variants.

To combine what comes from the instances in the process and the new ones it is enough to add rules that will combine the program for normal cyclic arbitrary closed ($\Pi^{ro,cl}_{\mathcal{P},Q}$) and normal cyclic fresh open ($\Pi^{ro,fr}_{\mathcal{P},Q}$).

To this end, for a given query $Q(\overline{X}) \leftarrow R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, we introduce relations:

- B^i_Q of arity $\mathit{arity}(R_1) + \cdots + \mathit{arity}(R_n)$ that contains on the first i arguments what comes from a mixture of existing and new process instances while the others come only from existing process instances, for $i = 1, \ldots, n$.

Encoding into Non-Recursive Datalog Let $\mathcal{B} = \langle \mathcal{P}, \mathcal{I}, \mathcal{D} \rangle$ be a rowo normal arbitrary open DABP.

Now we define rules that compute relations introduced above.

First we consider what is produced by the running instances

$$\begin{aligned} B^0_Q(\overline{Y}_1, \ldots, \overline{Y}_n) \leftarrow & \mathit{Exec}^n(O_1, \overline{W}_1, \ldots, O_n, \overline{W}_n), \\ & R'_1(O_1; \overline{W}_1; \bar{u}_1), \\ & \ldots \\ & R'_n(O_n; \overline{W}_n; \bar{u}_n). \end{aligned}$$

Then, for the i-th atom we both consider the case in which it was produced by a new instance (1) and the case it was produced by the instances already in the process (2). These cases are added to the combinations obtained for the atoms from 1 to $i-1$. We do this for every $i = 1, \ldots, n$.

$$\begin{aligned} & B^i_Q(\ldots, \overline{Y}_{i-1}, \overline{Y}_i, \overline{Y}_{i+1}, \ldots) \leftarrow && (1) \\ & \qquad B^{i-1}_Q(\ldots, \overline{Y}_{i-1}, \_, \overline{Y}_{i+1}, \ldots), R'_i(\overline{Y}_i) \\ & B^i_Q(\ldots, \overline{Y}_{i-1}, \overline{Y}_i, \overline{Y}_{i+1}, \ldots) \leftarrow && (2) \\ & \qquad B^{i-1}_Q(\ldots, \overline{Y}_{i-1}, \overline{Y}_i, \overline{Y}_{i+1}, \ldots). \end{aligned}$$

Then, we add the *Q'-rule* to collect what has been produced by the process for relations $R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ as follows:

$$Q'(\overline{X}) \leftarrow B^n_Q(\overline{Y}_1, \ldots, \overline{Y}_n).$$

The above rules extend the testing program $\Pi^{test,ro}_{\mathcal{P},Q}$ for normal cyclic arbitrary closed. We denote the new testing program with $\Pi^{test,ro,op}_{\mathcal{P},Q}$.

Theorem 10. *The following are equivalent:*

- *Q is instable in $\mathcal{B}$ under open semantics;*
- $\Pi^{ro,cl}_{\mathcal{P}} \cup \Pi^{ro,fr}_{\mathcal{P}} \cup \mathcal{D}_{\mathcal{I}} \cup \mathcal{D} \cup \Pi^{test,ro,op}_{\mathcal{P},Q} \models \mathit{Instable}$.

5.5 Rowo Complexity

Data and Instance Complexity Instances and data are encoded as facts of a non-recursive Datalog program, hence instance and data complexity is in AC^0. It also means that the stability problem can be reduced to the problem of evaluating a FOL query that encodes process part over a database that stores the process instances and the database instance. Since, FOL queries correspond to SQL queries (without aggregates), this opens the possibility for an efficient reasoning on stability using established RDBMS technologies.

5.5.1 Rowo CO-NP-hardness in Process Complexity

The process complexity is even lower than program complexity for non-recursive Datalog programs.

The process complexity of checking instability is NP-complete which is as complex as evaluating CQs. This holds already for the simplest variant of rowo. Intuitively, the upper-bound follows from the observation *(iv)* above. In fact, one can construct a single CQ instead of Datalog rules, but the construction is very technical.

To show hardness we encode 3-colorability problem.

Lemma 16 (Process Complexity 3-colourability Encoding)**.** *There exist a configuration $\mathcal{C}_{ar}$, a fresh configuration $\mathcal{C}_{fr}$, and a query Q_{test} such that for every graph G one can construct a positive rowo process $\mathcal{P}_G$ in logarithmic space such that the following are equivalent:*

- *G is not 3-colorable;*
- *Q_{test} is stable in $\langle \mathcal{P}_G, \mathcal{C}_{ar} \rangle$ under closed semantics;*
- *Q_{test} is stable in $\langle \mathcal{P}_G, \mathcal{C}_{fr} \rangle$ under open semantics.*

Proof. The test query is $Q_{test} \leftarrow \textit{dummy}$, and read-only databases for both $\mathcal{C}_{ar}$ and $\mathcal{C}_{fr}$ contain exactly 6 correct colourings $E(\textit{red}, \textit{blue}), \ldots, E(\textit{green}, \textit{blue})$. Given a graph $G = (V, E)$ we construct a Boolean positive query $Q_G \leftarrow \bigwedge_{(v_i,v_j)\in E} E(x_i, x_j)$ such that G is 3-colourable iff Q_G evaluates to *true* over the read-only database. Using this query we construct a DABP with a single transition t. We set that query to be the execution condition $E_t = Q_t$ for t. In the case of closed semantics we place an instance at the beginning of t. We add the writing rule W_t: *dummy* $\leftarrow$ *true* that writes *dummy* if the transition is executed. From there the claim follows directly in both cases. □

5.5.2 Rowo CO-NP-membership in Process Complexity

First we establish an auxiliary Lemma that shows how complex is to check whether a set of facts is produced by a singleton DABP.

Lemma 17 (Complexity Singleton Rowo DABPs)**.** *For any ground atoms $A_1, \ldots, A_n$, one can decide in* NP *time, whether for a given singleton rowo DABP $\mathcal{B}$, there is a closed execution in $\mathcal{B}$ that produces $A_1, \ldots, A_n$.*

Proof. Assume the instance o is at a place $p = M_P(o)$ and it has an input record $I(\bar{s}) = M_S(o)$. To show the claim it is sufficient to guess a closed execution Υ consisting of the traversals by o, and then verify whether atoms $A_1, \ldots, A_n$ can be produced by such execution. In the case of singleton rowo DABP under closed semantics,

a closed execution is uniquely determined by a path in $\mathcal{P}$. Thus, we guess a path $t_1, \ldots,$ t_m in $\mathcal{P}$ that starts in p. This guess is polynomial in the size of $\mathcal{P}$. For all transitions on the path, we further guess assignments $\alpha_1, \ldots, \alpha_m$ for the execution conditions $E_{t_1}, \ldots, E_{t_m}$. Then for each writing rule W_{t_i} of the transition t_i we guess up to n assignments $\beta^l_{t_i}$ for $1 \leq l \leq n$, because a rule may need to produce more than one fact, but no more than the size of the query. In principle, only a subset of $W_{t_1}, \ldots, W_{t_m}$ may be needed to produce atoms $A_1, \ldots, A_n$. Wlog we can guess the assignments for all. Now we verify. Firstly, we verify whether the path can be traversed by the instance. This is, if for every execution condition E_{t_i} the ground query $\alpha_i E_{t_i}$ evaluates to *true* in $\mathcal{D} \cup \{\mathit{In}(\bar{s})\}$. Secondly, we verify whether $A_1, \ldots, A_n$ are produced on the path by checking if for every A_i there exist a writing rule $W_{t_j} = Q_{t_j} \to A_{t_j}$ and assignment $\beta^l_{t_j}$ such that the ground query $\beta^l_{t_j} Q_{t_j}$ evaluates to *true* in $\mathcal{D} \cup \{\mathit{In}(\bar{s})\}$ and A_i is equal with the head of the writing rule $\beta^l_{t_j} A_{t_j}$. Since all guesses and checks are polynomial in the size of $\mathcal{B}$ the claim follows directly. □

Based on Lemma 17 we show that checking instability is in CO-NP for closed semantics in process complexity.

Lemma 18 (Process Complexity Upper Bounds Closed)**.** *Checking stability in rowo DABPs under closed semantics is in* CO-NP *in process complexity.*

Proof. To show the upper bound in process complexity we assume a fixed query $Q(\bar{x}) \leftarrow A_1, \ldots, A_n$. We want to show that for a given rowo DABP $\langle \mathcal{P}, \mathcal{C} \rangle$ checking instability of Q under closed semantics can be decided in nondeterministic polynomial time in the size of $\langle \mathcal{P}, \mathcal{C} \rangle$. Then checking stability is in CO-NP. We proceed by guessing ground atoms that make the query instable and then checking whether those atoms can be produced. *(i)* First we guess an assignment θ for $A_1, \ldots, A_n$. For the range of θ we take constants from $\mathcal{B}$, because the DABP executes under closed semantics, and therefore, the process can only produce such atoms. Then we split the atoms on those ones that are already in the database, and those ones that need to be produced by the process. Wlog we assume that the latter ones are $\theta A_1, \ldots, \theta A_k$. *(ii)* Now we need to guess an execution that produces $\theta A_1, \ldots, \theta A_k$ using the instances from $\mathcal{B}$. To guess that it is enough to guess an execution for each instance individually and then combine those individual executions in one execution. *(iii)* For that aim, we guess m instances $o_1, \ldots, o_m$ from the configuration that should altogether produce $\theta A_1, \ldots, \theta A_k$. For each instance o_i, where $1 \leq i \leq m$, we guess a subset $\theta A^i_1, \ldots, \theta A^i_{n_i}$ of $\theta A_1, \ldots, \theta A_k$ that is produced by this instance. Subsets are guessed in such a way that they cover the superset. *(iv)* Now, we verify. From Lemma 17 we get that for an instance o_i one can check in nondeterminstic polynomial time if o_i can produce $\theta A^i_1, \ldots, \theta A^i_{n_i}$ in $\mathcal{B}$. *(v)* Finally, we verify that $\theta \bar{x} \notin Q(D)$, i.e., that the query is not stable. To conclude, the described procedure return "yes" if Q is not stable in $\langle \mathcal{P}, \mathcal{C} \rangle$, and it runs in nondeterminstic polynomial time in the size of $\langle \mathcal{P}, \mathcal{C} \rangle$. □

Now we investigate process complexity for fresh rowo under open semantics. According to the abstraction principle for values we have that to check stability in a rowo $\mathcal{B}$ it is enough to consider facts from the extended active domain adom^*.

Lemma 19 (Complexity Fresh Rowo DABPs)**.** *One can decide in* NP *time whether there is an open execution in a fresh rowo DABP $\mathcal{B}$ that produces a ground atom A that takes constants from adom^*.*

Proof. Since $\mathcal{B}$ is rowo, it is sufficient to guess one new instance at the *start* place, and then check if the instance can traverse $\mathcal{B}$ and produce A. Thus, we guess an instance o with an input record $In(\bar{s})$ where constants from $\bar{s}$ occur in $adom^*$. We update the current configuration $\mathcal{C}$ to $\mathcal{C}'$ accordingly. By this we obtain a singleton $\mathcal{B}' = \langle \mathcal{P}, \mathcal{C}' \rangle$. Then to decide the initial problem it is enough to check if $\mathcal{B}'$ can produce A under closed semantics. From Lemma 17 it follows that this can be checked in nondeterministic polynomial time in the size of $\mathcal{B}'$. Since both guess and check were polynomial in the size of $\mathcal{B}$ the claim follows from there. □

Combining results from Lemmas 18 and 19 we show process complexity for arbitrary rowo DABPs under open semantics.

Lemma 20 (Process Complexity Upper Bounds Open)**.** *Checking stability in rowo DABPs under open semantics is in* CO-NP *in process complexity.*

Proof. To decide stability in arbitrary rowo DABPs under open semantics one can combine reasoning for arbitrary rowo DABPs under closed semantics and fresh rowo DABPs under open semantics. In particular, one can extend the proof technique of Lemma 18, by splitting the ground query atoms in the step *(i)* of Lemma 18 into three parts: atoms produced by the existing instances; atoms produced by the fresh instances; and atoms that are already in the initial database. Then to check if those three part are indeed produced in that way, one can apply Lemma 18 for the first part and Lemma 19 for the second. Check for the third part can be done by simply looking into the database. The rest of the proof remains the same as in Lemma 18. □

Complexity Summary

Theorem 11 (Complexity Summary)**.** *For all variants of rowo DABPs under open and closed semantics checking stability is*

1. CO-NP*-complete in process complexity;*
2. Π_2^{P}*-complete in query and combined complexity;*
3. *in* AC^0 *in instance and data complexity.*

6 Related Work and Conclusion

6.1 Related Work

Traditional approaches for business process modeling focus on the set of activities to be performed and the flow of their execution. These approaches are known as *activity-centric*. A different perspective, mainly investigated in the context of databases, consists in identifying the set of data (entities) to be represented and describes processes in terms of their possible evolutions. These approaches are known as *data-centric*.

Activity-Centric Approaches In the context of activity-centric processes, Petri Nets (PNs) have been used for the representation, validation and verification of formal properties, such as absence of deadlock, boundedness and reachability [21, 22]. In PNs and their variants, a token carries a limited amount of information, which can be represented by associating to the token a set of variables, like in colored PNs [15]. No database is considered in PNs. From the control flow perspective, PNs model parallelism. How to model parallelism is still an open question in data-centric BP models.

Data-Centric Approaches *Transducers* [1, 20] were among the first formalisms ascribing a central role to the data and how they are manipulated. These have been extended to *data driven web systems* [9] to model the interaction of a user with a web site, which are then extended in [8]. These frameworks express insertion and deletion rules using FO formulas. The authors verify properties expressed as FO variants of LTL, CTL and CTL* temporal formulas. The verification of these formulas results to be undecidable in the general case. Decidability is obtained under certain restrictions on the input, yielding to EXPSPACE complexity for checking LTL formulas and CO-NEXPTIME and EXPSPACE for CTL and CTL* resp., in the propositional case.

Data-Centric Dynamic Systems (DCDSs) [3] describe processes in terms of guarded FO rules that evolve the database. New data can be introduced via service calls yielding an infinite state-system in the general case. The authors study the verification of temporal properties expressed in variants of μ-calculus (that subsumes CTL*-FO). They identify several undecidable classes and isolate decidable variants by assuming a bound on the size of the database at each step or a bound on the number of constants at each run. In these cases verification is EXPTIME-complete in data complexity.

Overall, both frameworks are more general than DABPs, since deletions and updates of facts are also allowed. This is done by rebuilding the database after each execution step. Further, our stability problem can be encoded as FO-CTL formula. However, our decidability results for positive DABPs under open semantics are not captured by the decidable fragments of those approaches. In addition, the authors of the work above investigate the borders of decidability, while we focus on a simpler problem and study the sources of complexity. Concerning the process representation, both approaches adopt a rule-based specification. This makes the control flow more difficult to grasp, in contrast to activity-centric approaches where the control flow has an explicit representation.

Artifact-centric approaches [14] use artifacts to model business relevant entities. An artifact is composed of a data part (information model) and a process part (life cycle) that captures how the data part evolves. In [5,12,13] the authors investigate the verification of properties of artifact-based processes such as reachability, temporal constraints, and the existence of dead-end paths. In [12,13], there is an explicit representation of the process part using Finite State Machines, and in [5] the process is represented by business rules, thus leaving the control-flow implicit. However, none of these approaches explicitly models an underlying database. Also, the authors focus on finding suitable restrictions to achieve decidability, without a fine-grained complexity analysis as in our case.

Approaches in [2] and [4], investigate the challenge of combining processes and data, however, focusing on the problem of data provenance and of querying the process structure.

In [10,17] the authors study the problem of determining whether a query over views is independent from a set of updates over the database. The authors do not consider a database instance nor a process. Decidability results for rowo DABPs under open semantics can be seen as a special case of those.

In summary, our approach to process modeling is closer to the activity-centric one but we model manipulation of data like in the data-centric approaches. Also, having process instances and DABPs facets gives finer granularity compared to data-centric approaches.

6.2 Discussion and Conclusion

Contributions Reasoning about data and processes can be relevant in decision support to understand how processes affect query answers. (1) To model processes that manipulate data we adopt an explicit representation of the control flow as in standard BP languages (e.g., BPMN). We specify how data is manipulated as annotations on top of the control flow. (2) Our reasoning on stability can be offered as a reasoning service on top of the query answering that reports on the reliability of an answer. Ideally, reasoning on stability should not bring a significant overhead on query answering in practical scenarios. Existing work on processes and data [3] shows that verification of general temporal properties is typically intractable already measured in the size of the data. (3) In order to identify tractable cases and sources of complexity we investigated different variants of our problem, by considering negation in conditions, cyclic executions, read access to written data, presence of pending process instances, and the possibility to start fresh process instances. (4) Our aim is to deploy reasoning on stability to existing query answering platforms such as SQL and Datalog. For this reason we established different encodings into suitable variants of Datalog, that are needed to capture the different characteristics of the problem. For each of them we showed that our encoding is optimal. In contrast to existing approaches, which rely on model checking to verify properties, in our work we rely on established database query languages.

Implementation Technologies Complexity results provide boundaries on the technologies that can be used to implement the reasoning on stability. In particular, when the problem requires an encoding into non-recursive Datalog (PSPACE), which is equivalent to the relational algebra, we can in principle check stability using an SQL engine. For the encoding into Datalog with stratified negation (EXPTIME) we may still be able to use an SQL engine, however, we would need SQL recursive rules to express recursion. Finally, if our problem requires unstratified negation to be encoded (CO-NEXPTIME) we need ASP technology [16] to reason about it.

Discussion and Future Work Updates and deletions in DABPs can be modeled using negation in the rules. Thus, one can show that for positive DABPs under open semantics with updates or deletions, checking stability is undecidable. Similarly, for positive cyclic DABPs under closed semantics, with updates or deletions the complexity increases to CO-NEXPTIME. In addition, our results apply also in case the initial database is not known, since an arbitrary database can be produced by a process under open semantics starting from an empty database.

As future work, *(i)* we plan to investigate parallelism. Currently, we are able to deal with it only in case instances do not interact (like in rowo). *(ii)* We plan to investigate more expressive queries than CQ, such as $CQ^{\neg}$ (conjunctive queries with negation) and FO queries (= SQL). Under open semantics we expect decidability for $CQ^{\neg}$ and undecidability for FO queries. Also, we plan to consider: *(iii)* stability of aggregate queries and aggregates in the process rules; *(iv)* quantified instability (in case a query is not stable, compute the minimal and maximal number of possible new answers); *(v)* other data quality aspects such as *data timeliness* and *currency*.

References

[1] S. Abiteboul, V. Vianu, B. S. Fordham, and Y. Yesha. Relational Transducers for Electronic Commerce. In *PODS*, pages 179–187, 1998.

[2] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting Lipstick on Pig: Enabling Database-Style Workflow Provenance. *PVLDB*, 5(4):346–357, 2011.

[3] B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, and M. Montali. Verification of Relational Data-Centric Dynamic Systems with External Services. In *PODS*, pages 163–174, 2013.

[4] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes. In *VLDB*, pages 343–354, 2006.

[5] K. Bhattacharya, C. E. Gerede, R. Hull, R. Liu, and J. Su. Towards Formal Analysis of Artifact-Centric Business Process Models. In *BPM*, pages 288–304, 2007.

[6] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving Data Quality: Consistency and Accuracy. In *VLDB*, pages 315–326, 2007.

[7] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[8] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic Verification of Data-Centric Business Processes. In *ICDT*, pages 252–267, 2009.

[9] A. Deutsch, L. Sui, and V. Vianu. Specification and Verification of Data-Driven Web Services. In *PODS*, pages 71–82, 2004.

[10] C. Elkan. Independence of Logic Database Queries and Updates. In *PODS*, pages 154–160, 1990.

[11] W. Fan, F. Geerts, and J. Wijsen. Determining the Currency of Data. *ACM Trans. Database Syst.*, 37(4):25, 2012.

[12] C. E. Gerede, K. Bhattacharya, and J. Su. Static Analysis of Business Artifact-Centric Operational Models. In *SOCA*, pages 133–140, 2007.

[13] C. E. Gerede and J. Jianwen Su. Specification and Verification of Artifact Behaviors in Business Process Models. In *ICSOC*, pages 181–192, 2007.

[14] R. Hull. Artifact-Centric Business Process Models: Brief Survey of Research Results and Challenges. In *OTM*, pages 1152–1163, 2008.

[15] K. Jensen and L. M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.

[16] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[17] A. Y. Levy and Y. Sagiv. Queries Independent of Updates. In *VLDB*, pages 171–181, 1993.

[18] E. Marengo, W. Nutt, and O. Savković. Towards a Theory of Query Stability in Business Processes. In *AMW*, volume 1189 of *CEUR Workshop Proceedings*, 2014.

[19] S. Razniewski and W. Nutt. Completeness of Queries over Incomplete Databases. *PVLDB*, 4(11):749–760, 2011.

[20] M. Spielmann. Verification of Relational Transducers for Electronic Commerce. In *PODS*, pages 92–103. ACM, 2000.

[21] W. M. P. van der Aalst. Verification of Workflow Nets. In *ICATPN*, pages 407–426, 1997.

[22] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.