Reachability in Database-driven Systems with Numerical Attributes under Recency Bounding

Parosh Aziz Abdulla
Uppsala University
parosh@it.uu.se

C. Aiswarya
Chennai Mathematical Institute
aiswarya@cmi.ac.in

Mohamed Faouzi Atig
Uppsala University
mohamed_faouzi.atig@it.uu.se

Marco Montali
Free Univ. of Bozen/Bolzano
montali@inf.unibz.it

ABSTRACT

A prominent research direction of the database theory community is to develop techniques for verification of database-driven systems operating over relational and numerical data. Along this line, we lift the framework of database manipulating systems [3] which handle relational data to also accommodate numerical data and the natural order on them. We study an under-approximation called recency bounding under which the most basic verification problem –reachability, is decidable. Even under this under-approximation the reachability space is infinite in multiple dimensions – owing to the unbounded sizes of the active domain, the unbounded numerical domain it has access to, and the unbounded length of the executions. We show that, nevertheless, reachability is ExpTime complete. Going beyond reachability to LTL model checking renders verification undecidable.

KEYWORDS

database-driven systems; reachability; numerical constraints; recency

ACM Reference Format:
Parosh Aziz Abdulla, C. Aiswarya, Mohamed Faouzi Atig, and Marco Montali. 2019. Reachability in Database-driven Systems with Numerical Attributes under Recency Bounding. In *38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3294052.3319705

PODS'19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6227-6/19/06…$15.00
https://doi.org/10.1145/3294052.3319705

1 INTRODUCTION

Verification of dynamic, database-driven systems operating over relational data, is a central task to support the integrated management of master data and business processes. Over the years, many researchers in the field of principles of data management have attacked this difficult problem, producing a flourishing plethora of formal, integrated models for processes and data, and corresponding technical results on identifying limits of decidability [12].

Within this broad literature, two main trends can be identified. A first line of research splits the data component of the database-driven system into two parts: a read-only part that is immutable and can only be queried by the process, and a "controlled" read-write part that can instead be updated while the process unfolds. Parameterized verification problems are then studied by checking whether the process satisfies a temporal property of interest irrespectively of the data contained in the read-only part (see, e.g., [10, 20, 26]). In this spectrum, verification has been studied also in the presence of numerical attributes and corresponding arithmetic constraints [17, 21]. In this challenging setting, decidability essentially requires to bound how the result of an arithmetic operation propagates to further arithmetic operations. In [17], this is ensured through the notion of *feedback freedom* (which prevents a variable to be updated based on the previous value stored in the same variable). In [21], instead, where the framework handles task hierarchies, this is achieved by controlling how parent tasks invoke their subtasks, in particular requiring that each subtask is called at most once between internal transitions of its parent task.

In a second, parallel line of research, the process can freely update the entire data component, and verification is studied by considering how the process evolves when starting from an initial, fixed database instance. Even without the parameterized flavor discussed above, verification turns out to be extremely challenging also in this setting, since it requires to analyze a transition system containing infinitely many

different states, each one related to a different database instance. Towards decidability, two main strategies have been studied.

The first strategy is to focus on *state-bounded* systems, that is, systems where the size of each database instance cannot exceed a pre-defined, possibly unknown bound [5, 7]. Under this assumption, it is possible to construct a faithful, finite-state abstraction of the transition system that cannot be distinguished from the original one even by sophisticated temporal logics based on the first-order μ-calculus [13].

The second strategy is to consider *under-approximate verification*. Under-approximation has been proven very efficient for finding bugs in software verification [25]. Instead of checking a temporal property of interest over all the runs induced by the process, only a subset of the runs is considered. In [3], this notion has been applied for the first time ever to the verification of dynamic database-driven systems. In particular, processes operating over data are formalized there in a so-called *Database Manipulating Systems (DMS)*. A DMS consists of a relational database, and a set of actions that query and modify that database. Queries can be made using first order logic over the relational schema and they can be used as guards that enable/disable actions. Actions bring changes over the database that can result in 1) deleting facts from the relations, 2) adding new facts involving existing data elements to the relations, 3) adding new facts involving new data elements (that were not present in the current database instance) into the relations. The new data insertion is the key to model user-inputs or freshly generated data such as timestamps / UIDs. As in the other formal models for data-aware processes, this sophisticated modeling comes at the price of imminent undecidability of verification: even the most simple forms of verification, such as propositional reachability, enjoy decidability by putting severe restrictions on the modeling power of DMS [4].

In this setting, under-approximation is a good compromise between feasible verification and modeling power. In [3], the subset of runs carved out by under-approximation corresponds to the one where the answers to the queries posed over the data component are all returned from the most recent B elements introduced in the system. We call this *recency-bounded verification*. The set of database instances that we may encounter while abiding to recency-bounded runs is still infinite. Thus the coverage is far superior than any finite state abstraction. Furthermore, bounding recency is a natural choice for under-approximation – it appears naturally in processes where services are prioritized on a last-come-first-serve basis. In [3] we showed that not just reachability but also model checking against very powerful monadic second order logic that can reason about temporal properties of the database instances is decidable under the bounded recency assumption.

In spite of these encouraging results, little is known about verification of database-driven systems starting from an initial, fixed database instance, *when the system is equipped with numerical attributes*. The only existing result has been obtained for state-bounded systems [14]. The result essentially shows that, due to the state-bounded restriction, the implicit comparison predicate $\leq$ can be actually lifted into an explicit binary relation that suitably reconstructs comparison on all and only data objects present in the current active domain.

Unfortunately, this approach cannot be exploited under the recency-bounded restriction. For one reason the relations are total and hence appropriate update of the explicit relation would include adding facts involving *all* the unbounded elements in the current active domain, which would indeed violate the bound on recency. Even the case of a non-total relation such as "*smaller-but-closest* in the active domain" is problematic because the partner element may be very far away according to the recency and hence not accessible within the bound.

In some sense this has to be expected and permitted, because if we give the ability to build and maintain chains (eg. "successor" relations of two total orders), that could also easily lead to undecidability defeating our purpose.

This paper takes form at this juncture to precisely address this issue: *how to verify database-driven systems with numerical attributes under recency bounding*. First of all, we extend the framework of [3] so as to tackle processes operating over interpreted data and implicit relations over them, in particular numbers with implicit order $(\mathbb{Q}, \leq)$. Secondly, we study recency-bounded verification in this richer setting. Even without going to arithmetics, the extension to DMSs with numerical attributes makes this problem extremely difficult to attack: we cannot anymore resort to the technical machinery exploited in [3], but need to develop completely novel techniques.

Towards this we assume an uninterpreted data domain which is equipped with a numerical attribute. This is not an injective association, and hence there may be multiple elements with the same numerical value. Further we assume the numerical values to range from a dense domain such as rational numbers ($\mathbb{Q}$).

We lift the framework of DMS to handle dense orders on the numerical values as implicit relations. The action guards can check constraints on the numerical values. The constraints are Boolean combinations of inequality checks. Thus the guarding queries can be written as a union of conjunctive queries with possibly negated atoms over the extended vocabulary which consists of the schema as well as the implicit order relation.

Our main result states that recency-bounded reachability is decidable, in ExpTime, even with this rich feature. We also

show a matching lower-bound thus establishing the optimality of our procedure. Further, we show that going beyond reachability towards model checking against temporal logics such as LTL-FO compromises decidability.

Our proof technique is to construct a pushdown system (with finite states and stack alphabet) which gives an elegant symbolic abstraction of the recency bounded-DMS. This is non-trivial because the pushdown system has to maintain the numerical constraints faithfully throughout, in addition to the relation instances. The numerical values are not injectively assigned to elements, and they are not limited by recency bounding or freshness, making it possible that numerical values that are far outside the recency bound can also contribute to the constraints that need to be maintained.

The number of states and the size of the stack alphabet of our symbolic pushdown system is exponential in the bound B, number of free variables in actions and the number of relations in a schema. We assume the arity of the relations in the schema is fixed to have the reachability in ExpTime. In fact the degree of the polynomial in the exponent is proportional to the maximum arity of the relations.

Outline. We will first give an overview of the problem and the ExpTime procedure in Section 2. Then we define the model and the problem more formally and state our results in Section 4, followed by an example (Section 5) to illustrate the modeling capabilities. Then we continue with the details of the ExpTime procedure in Sections 6 and 7. The proof of lower bound and undecidability are moved to the appendix for want of space. We discuss the related work and conclude the paper with further research directions.

2 OVERVIEW

In this section, we give a high-level but detailed overview of the problem and our solution. First, we introduce *Database Manipulating Systems (DMS)*, the recency semantics, and the reachability problem. Then, we describe a symbolic representation that allows to translate the reachability problem under the recency semantics to the reachability problem for standard Push-Down Automata (PDA). More precisely, we show how the transitions of a PDA can simulate the actions of a DMS while preserving reachability properties.

2.1 Model

We consider *Database Manipulating Systems (DMS)* where the elements from the data domain are equipped with numerical values. As usual, the database provides a set of *relation names* each with an *arity*. An *instance* I of the database consists of a *relation instance* for each relation name R. A relation instance of R consists of a set of *facts* each of which is a vector of data elements whose length is equal to the arity of R. The *active domain* $\textsc{Adom}(I)$ of I is the set of data elements that are part

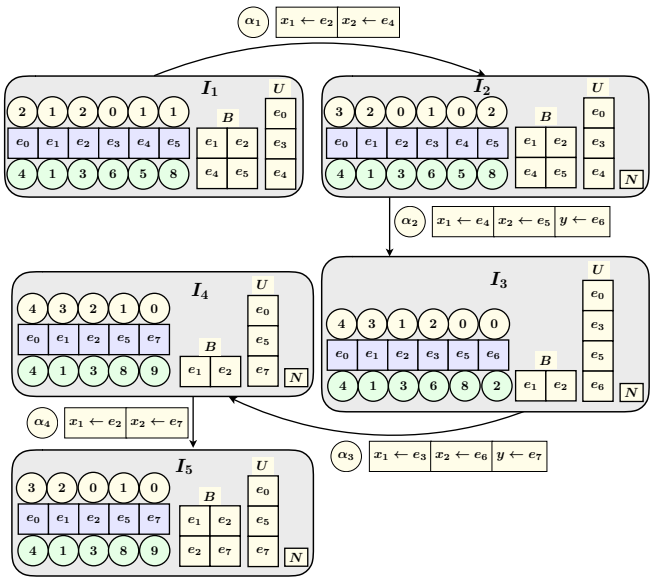

Figure 1: Database instances.

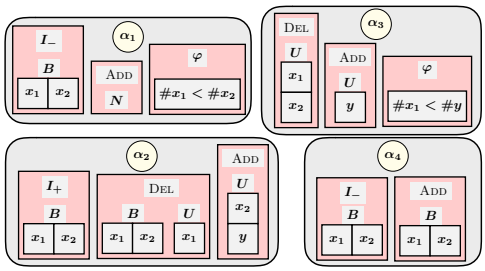

Figure 2: Actions.

of some fact in I. Fig. 1 depicts five instances $I_1, \ldots, I_5$. Let us consider I_1. The active domain contains six data elements, namely $e_0, \ldots, e_5$. The circle below each data element gives its numerical value which in general is a rational number. The numerical value of e_0 is given by $\#e_0 = 4$. For the time being, we will not explain the circles above the data elements (they will be used later when explaining the recency semantics.) The database has three relations names, namely B (for Binary) with arity 2, U (for Unary) with arity 1, and N (for Nullary) with arity 0. The instance I_1 contains the relation instances $\{\langle e_1, e_2\rangle, \langle e_4, e_5\rangle\}$ for B, and $\{\langle e_0\rangle, \langle e_3\rangle, \langle e_4\rangle\}$ for U. A nullary relation is treated as a proposition whose value is true if it occurs in the instance, and false otherwise. The value of N is false in I_1 and true in I_2.

The operational semantics of a DMS is defined as a transition relation on the set of instances. The relation is derived through a set of *actions*. Fig. 2 shows examples of four actions $\alpha_1, \ldots, \alpha_4$. Each action is defined using two sets of variables: the *old* variables that are instantiated to elements that are

currently in the active domain, and the *new* variables that are instantiated to elements that will be added to the instance. In this section, we denote the sets of old and new variables using x resp. y (possibly with subscripts). The sets of old and new variables in α_2 are given by $\{x_1, x_2\}$ and $\{y\}$ respectively. The action may contain five parts (some of which may be missing), as follows. (i) The *positive database instance* I_+ requires that some facts should be present in the instance. The action α_2 requires that there should be a fact (corresponding to the instantiation of) $\langle x_1, x_2\rangle$ in the instance of B. (ii) The *negative database instance* I_- requires that some facts should not be present in the instance. The action α_1 requires that there should not be a fact $\langle x_1, x_2\rangle$ in the instance of B. (iii) The *delete part* DEL deletes some facts from the database. The action α_2 deletes the fact $\langle x_1, x_2\rangle$ from B, and the fact $\langle x_1\rangle$ from U. (iv) The *add part* ADD adds some facts to the database. The action α_3 adds a fact with a new data element to the unary relation U. (v) The *constraint* φ compares the numerical values of the data elements using the relations $\leq$ and $<$. The compared elements may be old or newly added. The action α_3 requires that the numerical value of the newly added element is larger that the one represented by x_1

Before applying an action to an instance, its variables are instantiated. Consider the application of α_2 to I_2 leading to I_3 as depicted in Fig. 1. We instantiate x_1 and x_2 to data elements in the active domain, namely e_4 and e_5, while we instantiate y to a new data element, namely e_6. Notice that the action will remove the fact $\langle e_4, e_5\rangle$ from B and the fact $\langle e_4\rangle$ from U, and adds the facts $\langle e_5\rangle$ and $\langle e_6\rangle$ to U. In particular, the element e_4 will not be in the active domain any more.

The *reachability problem* asks, given by a DMS, an instance I, and a proposition q, whether we can apply a sequence of actions to I to obtain an instance I' that contains q.

2.2 Recency Semantics

The recency semantics corresponds to an under-approximation where we only consider "recent" data elements when instantiating the variables when performing actions. To that end, we equip each data element in an instance with a *recency measure* that orders the elements according to the number of steps that have elapsed since they last participated in an action (i.e., since they were last used to instantiate a variable in action). More precisely, an element e_1 has a smaller recency measure than an element e_2 if the last time e_1 participated in an action was after the last time e_2 participated in an action. In particular, all the elements that participated in the latest action have recency measures that are equal to 0. The *band* of measure k is the set of elements with recency measures equal to k. The recency semantics is parameterized by a *recency bound* B. At any point of a run of the DMS, we identify the *current recency window* to be the set of elements whose recency measures do not exceed the bound B. When performing an action, we are only allowed to instantiate variables with elements in the current recency window. We reset the recency measures of all the elements that participate in the action to 0, while we increase the recency measures of all the other elements by 1. Notice that the recency measure of an element e may decrease after an action either because e participates in the action, or because the action removes all the elements in some band whose measure is smaller than the measure of e. In Fig. 1, we assume that the recency bound is $B = 2$. The recency measures of the elements are given in the circles above the element. In I_2, the band of measure 2 is given by the elements e_1 and e_5; and all the elements except e_0 are in the current recency window. When we perform an action, say α_2, we are only allowed to use elements with recency bounds up to 2 (e.g., e_4 and e_5 together with the new element e_6). However, we are not allowed to instantiate x_1 or x_2 by e_0 since its recency measure in I_2 is equal to 3. Since the recency measures may increase or decrease during a given run, some elements may repeatedly leave and enter the current recency window. The element e_1 is inside the window in I_1, I_2, and I_5, and outside the window in I_3 and I_4. An instance of the reachability problem under the recency semantics is defined in a similar manner to the original semantics. However, we are also given a recency bound B, and we are asked whether we can reach I' applying actions under the recency restriction.

Notice that, both under the standard and the recency semantics, we are solving the reachability problem for a transition system that is infinite in multiple dimensions. This is true since there is no bound on the number of objects that are generated during a given run. Furthermore, the objects have numerical values that are rational numbers, and they may participate in unbounded numbers of facts during the run.

In this paper, we solve the reachability problem by reducing it to the reachability problem for PDA. Given a DMS $\mathcal{D}$, we extract a PDA $\mathcal{B}$, called a *layered database*, such that $\mathcal{B}$ can faithfully simulate $\mathcal{D}$ wrt. reachability properties.

2.3 Layers

Simulating a DMS under the recency semantics using a PDA is not trivial. The reason is that in a PDA the symbols inside the stack are totally independent, while in the case of a database, there may exist dependencies between objects belonging to recency windows that are arbitrary far apart. One source of this dependency is that the objects may participate in common facts. Another source is that there may be constraints on their relative numerical values. A main challenge here is to store dependencies among arbitrary sets of objects

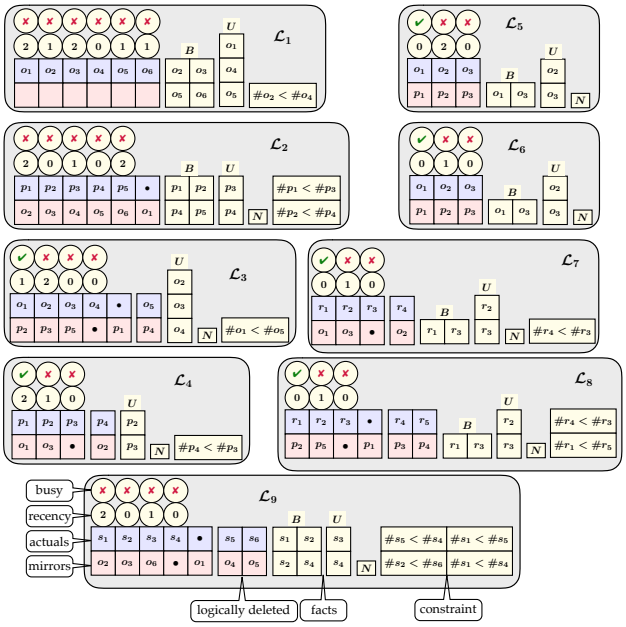

Figure 3: Layers.

using only a finite stack alphabet. In order to solve this problem, we will capture the dependency between two objects that are in different places inside the stack by maintaining a "chain" of implicit dependencies between successive stack symbols leading from the first object to the second object. To that end, we will introduce an intricate encoding where each stack symbol, called a *layer*, represents one recency window. The layers correspond to recency measures that increase from the top to the bottom of the stack (see Fig. 3 and Fig. 4.) The top-most layer represents elements whose recency measures range from 0 to B. The next layer represents elements whose recency measures range from 1 to $B+1$, and so on. According to the recency semantics, we only need to consider the objects in the current recency window, and hence in our simulation we need only to consider the top-most layer when mimicking an action in the DMS (while we still need to maintain dependencies with other objects inside the stack, as explained above). A layer maintains different types of information about the objects (see layer $\mathcal{L}_9$ in Fig. 3.) These information types are introduced one by one below.

Actual Objects. First, the layer uses a set of *actual objects* representing elements of the data domain that lie in the current recency window, i.e., objects whose recency measures were up to the bound B when the layer was created. In Fig. 3, the layer $\mathcal{L}_1$ represents the recency window just before the action α_1 is performed in Fig. 1. Therefore, its objects correspond to the data elements in the recency window of I_1 in Fig. 1. The layer contains 6 actual objects $o_1, \ldots, o_6$

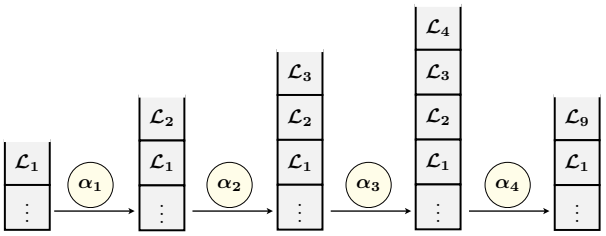

Figure 4: Actions of the layered database.

corresponding to the elements $e_0, \ldots, e_5$ respectively. Similarly, the layer $\mathcal{L}_2$ represents the recency window just after α_1 has been performed. The layer contains 5 actual objects $p_1, \ldots, p_5$ corresponding to the elements $e_1, \ldots, e_5$ respectively. The element e_0 is not represented in $\mathcal{L}_2$ since it is outside the recency window when $\mathcal{L}_2$ is created (its recency measure is 3 which exceeds the bound $B = 2$). Notice that the set of elements represented in two successive layers may overlap. For instance, the elements $e_1, \ldots, e_5$ are represented both in $\mathcal{L}_1$ and in $\mathcal{L}_2$. As we shall see later, the overlapping of elements between the successive layers will enable us to maintain the chain of dependencies among objects, thus avoiding the loss of precision. Consequently, the result of our simulation will be exact wrt. the recency semantics.

Mirrors. Since there is an overlap between the elements represented by successive layers, a layer also keeps a set of *mirror* objects that are copies of the actual objects from the next layer (the layer below the current layer in the stack). In Fig. 3, the layer $\mathcal{L}_2$ has six mirror objects $o_1, \ldots, o_6$ that are copies of the actual objects in the next layer in the stack, namely $\mathcal{L}_1$. In Fig. 3, we draw each mirror object below the object it mirrors. For instance, the object o_2 mirrors the object p_1 in $\mathcal{L}_2$ since they both represent the concrete data element e_1 in Fig. 1. The object o_1 does not mirror any object in $\mathcal{L}_2$ since it corresponds to the element e_0 in I_2 which is outside the recency window. In $\mathcal{L}_3$ (which corresponds to I_3), the object o_4 is not mirrored by any object. The reason is that o_4 corresponds to the element e_6 that is newly created (and hence not part of the previous layer $\mathcal{L}_2$). The fact that we keep mirror objects and also record which objects they mirror in the layer, allows to keep track of dependencies between objects (say o_1 and o_2) that lie in layers that are far apart in the stack (such objects belong to different recency windows.) This is possible since o_1 will be identified with its mirror, which in turn will be identified with its own mirror, and so on, until we reach a layer which contains both o_2 and a copy of o_1.

Actions. Fig. 4 depicts a run in the symbolic simulation that mimics the concrete run of Fig. 3. When performing an action, we check the top-most layer $\mathcal{L}$ in the stack, since

$\mathcal{L}$ represents the current recency window. We instantiate the variables of the actions using the actual objects of $\mathcal{L}$, and perform a sequence of operations on $\mathcal{L}$ to obtain a new layer $\mathcal{L}'$. In Figure 4, we perform α_1 on $\mathcal{L}_1$ by instantiating the variables x_1 and x_2 with the objects o_3 resp. o_5, and derive $\mathcal{L}_2$ as follows. We create new actual objects, make the actual objects in $\mathcal{L}_1$ mirror objects in $\mathcal{L}_2$, and define the correspondence between the actual and the mirror objects. Furthermore, we define the recency measures of the new actual objects. We add the proposition N as required by α_1, and add the numerical constraint in α_1 to the numerical constraint in $\mathcal{L}_1$, thus obtaining $\mathcal{L}_2$. We push $\mathcal{L}_2$ to the stack since it will now represent the current recency window and hence it should be the top stack symbol.

The transition from $\mathcal{L}_2$ to $\mathcal{L}_3$ through α_2 can be explained analogously. However, here we need three new aspects of our encoding namely *Busy flags*, *purification*, and *logical deletion*.

Busy Flags. The object p_1 in $\mathcal{L}_2$ is not copied as an actual object in $\mathcal{L}_3$. The reason is that p_1 corresponds to the object e_1 whose recency measure is 3 when $\mathcal{L}_3$ is created and hence it is outside the corresponding recency window. However, the object p_1 is involved in the fact $\langle p_1, p_2\rangle$ in $\mathcal{L}_2$ (which reflects the fact $\langle e_1, e_2\rangle$). Since p_1 is not an actual object in $\mathcal{L}_3$, the above fact will not be present in $\mathcal{L}_3$. This means that the object p_2 is not involved in any facts in $\mathcal{L}_3$. Therefore, we may mistakenly discard p_2 in the *purification* procedure that removes "useless" objects and that will be described below. To guard against accidental removal of objects which appear to be useless when only considering the current layer, we provide each actual object with a Boolean flag, called busy which tells whether the copies of the object are involved in facts further down the stack. In this case we set the flag of p_2 to true (in the figure we use the symbols ✓ and ✗ to represent the values true resp. false.)

Purification. In the transition from $\mathcal{L}_3$ to $\mathcal{L}_4$, the action α_2 deletes the facts $\langle o_2\rangle$ and $\langle o_4\rangle$ from U in $\mathcal{L}_3$. This means that o_2 and o_4 become *useless* in the since that (i) they do not participate in any facts in $\mathcal{L}_4$, and (ii) their busy-flags are false which means that they are not involved in facts further down the stack. In such a case, we *purify* the layer by removing the objects, i.e., not copying them as actual objects in $\mathcal{L}_4$.

Logical Deletion. In the previous paragraph, there is a difference between o_2 and o_4, since o_2 has a mirror p_3 in $\mathcal{L}_3$ while o_4 does not. In fact, o_2 is part of a chain of dependencies between objects in different layers as follows: (i) When applying α_3 to obtain $\mathcal{L}_4$ from $\mathcal{L}_3$, we instantiate the variables x_1 and y to the object o_2 in $\mathcal{L}_3$ and the new object p_3 in $\mathcal{L}_4$ (this corresponds to instantiating x_1 and y to e_3 and e_7 in the concrete run of Fig. 1.) To satisfy the numerical constraint in α_3, the numerical value of the object o_2 of $\mathcal{L}_3$ should be smaller than the numerical value of the new object p_3 of $\mathcal{L}_4$. (ii) The object p_3 of $\mathcal{L}_2$ mirrors the object o_2 of $\mathcal{L}_3$, and hence their numerical values are equal. (iii) According to the numerical constraint of $\mathcal{L}_2$, the numerical value of the object p_1 of $\mathcal{L}_2$ is smaller than the numerical value of the object p_3 of $\mathcal{L}_2$. From (i), (ii), and (iii) we conclude that the numerical value of the object p_1 of $\mathcal{L}_2$ is smaller than the numerical value of the object p_3 of $\mathcal{L}_4$. Notice that we have created a constraint on objects that do not lie in the same layer, and in fact not even in successive layers (the layers $\mathcal{L}_2$ and $\mathcal{L}_4$). The chain of dependencies between objects may be arbitrarily long. However, all such dependencies will be preserved using our mechanism. Here, we would have wrongly broken the chain if we had deleted the useless object o_2 of $\mathcal{L}_3$. To preserve precision, we keep a list of *logically deleted* objects together with their mirrors. These are objects that are useless but that have a mirror in the previous layer. The object p_4 will therefore be added as a logically deleted in $\mathcal{L}_4$ with mirror o_2, while o_4 that does not have a mirror in $\mathcal{L}_3$ will be completely deleted. The logically deleted objects will not be copied to the next layer. The fact that we remove useless objects and do not transfer logically deleted objects to the next layer are crucial for maintaining the finiteness of the stack alphabet.

Compactification. When we apply the action α_4 to $\mathcal{L}_4$ we get the layer $\mathcal{L}_5$ in which the band with measure 1 has disappeared. We *compactify* $\mathcal{L}_5$ by re-defining the recency measures of the higher bands (2 in this case) so that we are consistent with the definition of a recency measure (the object o_2 should have a recency measure 1 rather than 2.) This results in the layer $\mathcal{L}_6$.

Merging. Although $\mathcal{L}_6$ is compact in the sense described above, it does not represent the current recency window, since it missing the objects that currently have recency measure 2. Such objects may exist in layers that are currently in the stack. For instance the concrete element e_1 has a recency measure 2 after the application of α_4 (Fig. 1). However, this element is not represented in $\mathcal{L}_6$. In fact, its representative can be found first in layer $\mathcal{L}_2$ (object p_1). Therefore, we cannot push $\mathcal{L}_6$ to the stack (otherwise, we would violate the invariant that the top-most stack symbol represents the current recency window.) Therefore, we pop the current top-most stack symbol, which is the layer $\mathcal{L}_4$ and *merge* it with $\mathcal{L}_6$. The purpose of the merging operation is to search for objects that belong to the current recency window but are not part of $\mathcal{L}_6$. The operation "merges" the mirrors of $\mathcal{L}_6$ with the corresponding actual objects in $\mathcal{L}_4$, thus obtaining $\mathcal{L}_7$. For instance, the mirror object p_1 represents the same object as the actual object p_1 in $\mathcal{L}_4$. This means that the mirror of p_1 in $\mathcal{L}_4$, namely o_1 will be the mirror of the corresponding element

in r_1 in $\mathcal{L}_7$. We update the recency measures, the busy-flags, and merge the facts and the numerical constraints in $\mathcal{L}_4$ and $\mathcal{L}_6$. In such a manner the layer $\mathcal{L}_7$ represents a *summary* of the layers $\mathcal{L}_4$ and $\mathcal{L}_6$. Notice that $\mathcal{L}_7$ is still missing one band. Therefore, we repeat the above operation now popping $\mathcal{L}_3$ and merging it with $\mathcal{L}_7$ to obtain $\mathcal{L}_8$. Finally, we pop $\mathcal{L}_2$ and merge it with $\mathcal{L}_8$ obtaining $\mathcal{L}_9$. Since $\mathcal{L}_9$ has all the bands, we push it to the stack.

Notice that the size of the stack may increase and decrease as layers are created and deleted, reflecting that the recency measures of the different objects change during a run.

3 PRELIMINARIES

Basic Notation. We use $\mathbb{B} = \{\text{true}, \text{false}\}$, $\mathbb{N}$, and $\mathbb{Q}$ to denote the sets of Booleans, natural numbers and rational numbers respectively. For a natural number $n \in \mathbb{N}$, we use $\widehat{n}$ to denote the set $\{0, 1, \ldots, n\}$.

For sets A and B, we write $f : A \to B$ to denote that f is a (possibly partial) function that maps elements from A to B. We write $f(a) = \perp$ to denote that f is undefined for a, and write $f : A \xrightarrow{\bullet} B$ to indicate that the function f is total. For a finite set A, we use $|A|$ to denote the size of A. For a relation $R \subseteq A \times B$, and a set $C \subseteq A$, we use $R\big|_C$ to denote the relation $R \cap (C \times B)$. In particular, for a function $f : A \to B$, we use $f\big|_C$ to denote the restriction of f to C.

For a set A, we use A^* to denote the set of words over A and use $w_1 \bullet w_2$ to denote the concatenation of the words w_1 and w_2.

Numerical Constraints. We will work with sets of objects each with a numerical value in $\mathbb{Q}$. As we shall see later, these objects can either be members of the data domains in (concrete) database instances, or be used in defining symbolic encodings of such instances. A *numerical constraint* φ over a set $O \subseteq \mathcal{O}$ of objects, is a set of inequalities each of the form $\#o_1 \sim \#o_2$ where $o_1, o_2 \in \mathcal{O}$ and $\sim \in \{\leq, <\}$. We say that φ is *saturated* if whenever $(\#o_1 \sim_1 \#o_2) \in \varphi$ and $(\#o_2 \sim_2 \#o_3) \in \varphi$ then $(\#o_1 \sim_3 \#o_3) \in \varphi$ where $\sim_3$ is $\leq$ iff both $\sim_1$ and $\sim_2$ are $\leq$. We define saturate (φ) to be the smallest numerical constraint (wrt. set inclusion) φ' such that $\varphi \subseteq \varphi'$ and φ' is saturated. We say that φ is consistent if it does not contain two inequalities $\#o_1 \sim_1 \#o_2$ and $\#o_2 \sim_2 \#o_1$ such that either $\sim_1$ is $<$ or $\sim_2$ is $<$. For a numerical constraint φ and a set $O \subseteq \mathcal{O}$, we use $\varphi\big|_O$ to be the numerical constraint we get by removing all inequalities involving objects $o \notin O$.

4 MODEL

We consider databases where elements from the infinite data domain Δ are equipped with numerical values. The numerical value of each element in Δ is given by a function $\# : \Delta \to \mathbb{Q}$. The schema, as usual, contains a finite set of relation names with their arities: $\mathcal{R} = \{R_1/a_1, R_2/a_2, \ldots, R_{|\mathcal{R}|}/a_{|\mathcal{R}|}\}$. An *instance* of a relation R/a (or a relation instance) over the domain Δ is a set of a-tuples from Δ. A nullary relation is a proposition. In the rest of the paper, we fix the data domain Δ

A *database instance* over a schema $\mathcal{R}$ gives an instance of all the relations in the schema. Formally, a database instance I over a schema $\mathcal{R} = \{R_1/a_1, R_2/a_2, \ldots, R_{|\mathcal{R}|}/a_{|\mathcal{R}|}\}$ is given by $I = \left\langle R_1^I, R_2^I, \ldots, R_{|\mathcal{R}|}^I \right\rangle$ where $R_i^I \subseteq \Delta^{a_i}$ is an instance of the relation R_i/a_i in the schema. A *fact* in I is a vector that occurs in some relation instance in I, i.e., it is a vector $\langle e_1, \ldots, e_a \rangle \in R^I$ for some relation name R/a. The *active domain* of a database instance I, denoted $\textsc{Adom}(I)$, is the set of elements from the domain present in the database instance. That is, $\textsc{Adom}(I)$ is the set of data elements e such that there is some fact $\langle e_1, \ldots, e_a \rangle$ in I, and $e = e_i$ for some $i : 1 \leq i \leq a$. For a proposition $N/0$, we say N *occurs* in I if N^I is *true*.

DMS. A *Database Manipulating Systems (DMS)* $\mathcal{D}$ is given by a finite set $\textsc{Acts}$ of *actions*. An action $\alpha \in \textsc{Acts}$ is a tuple $\langle \overline{x}, \overline{y}, I_+, I_-, \textsc{Del}, \textsc{Add}, \varphi \rangle$ where $\overline{x}$ and $\overline{y}$ are disjoint sets of variables, I_+, I_-, and $\textsc{Del}$ are database instances over $\overline{x}$, $\textsc{Add}$ is a database instance over $\overline{x} \cup \overline{y}$, and φ is a numerical constraint over $\overline{x} \cup \overline{y}$.

Semantics. The operational semantics of a DMS is defined to be an (infinite-state) transition system, where each configuration is a database instance over $\mathcal{R}$. Consider a database instance I and an action $\alpha = \langle \overline{x}, \overline{y}, I_+, I_-, \textsc{Del}, \textsc{Add}, \varphi \rangle$. A substitution $\nu : \overline{x} \cup \overline{y} \to \Delta$ is said to *instantiate* α at I if $\nu(x) \in \textsc{Adom}(I)$ for all $x \in \overline{x}$, and $\nu(y) \notin \textsc{Adom}(I)$ for all $y \in \overline{y}$. Let $\nu(I_+)$ be the database instance obtained by substituting every occurrence of a variable x in I_+ by $\nu(x)$, and similarly for $\nu(I_-)$, $\nu(\textsc{Del})$, $\nu(\textsc{Add})$, and $\nu(\varphi)$. For database instances I_1 and I_2, an action $\alpha = \langle \overline{x}, \overline{y}, I_+, I_-, \textsc{Del}, \textsc{Add}, \varphi \rangle$, and a substitution ν that instantiates α at I_1, we write $I_1 \xrightarrow{\alpha, \nu} I_2$ if $\nu(\varphi)$ holds and the following three conditions are satisfied for each relation name $R \in \mathcal{R}$: (i) $R^{\nu(I_+)} \subseteq R^{I_1}$. (ii) $R^{\nu(I_-)} \cap R^{I_1} = \emptyset$. (iii) $\langle e_1, \ldots, e_n \rangle \in R^{I_2}$ iff $\langle e_1, \ldots, e_n \rangle \in R^{I_1 - \nu(\textsc{Del}) \cup \nu(\textsc{Add})}$. We write $I_1 \xrightarrow{\alpha} I_2$ if $I_1 \xrightarrow{\alpha, \nu} I_2$ for some ν, and write $I_1 \to I_2$ if $I_1 \xrightarrow{\alpha} I_2$ for some α. We use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\to$.

Reachability. The *reachability problem* asks, given a DMS $\mathcal{D}$, a database instance I, and a proposition $q \in \mathcal{R}$, whether there is a database instance I' such that $I \xrightarrow{*} I'$ and q occurs in I'.

The reachability problem is undecidable, even without numerical values and over very restrictive schema[3, 4]. We will hence resort to an under-approximate reachability problem, in which an instantiating substitution is limited to use only the recent elements.

Recency Semantics. Fix a *recency bound* $B \in \mathbb{N}$. An *augmented database instance* is a pair $J = \langle I, \text{rcy} \rangle$ where I is a database instance and $\text{rcy} : \textsc{Adom}(I) \to \mathbb{N}$ defines, for each element $e \in \textsc{Adom}(I)$, its *recency measure* $\text{rcy}(e)$. We say that J is *compact* if for every $e \in \textsc{Adom}(I)$ and $j : 0 \leq j < \text{rcy}(e)$, there is an $e' \in I$ where $\text{rcy}(e') = j$. For $J = \langle I, \text{rcy} \rangle$, we define $\text{compactify}(J) := \langle I, \text{rcy}' \rangle$ where $\text{rcy}' : \textsc{Adom}(I) \to \mathbb{N}$ is the unique function such that $\langle I, \text{rcy}' \rangle$ is compact and $\text{rcy}'(e) \leq \text{rcy}'(e')$ iff $\text{rcy}(e) \leq \text{rcy}(e')$ for all $e, e' \in \textsc{Adom}(I)$. A substitution $\nu : \overline{x} \cup \overline{y} \to \Delta$ is said to *instantiate* α at J if (i) ν instantiates α at I, and (ii) $\text{rcy}(\nu(x)) \leq B$ for all $x \in \overline{x}$. For augmented instances $J_1 = \langle I_1, \text{rcy}_1 \rangle$ and J_2, and a substitution ν that instantiates α at J_1, we write $J_1 \xrightarrow{\alpha, \nu}_B J_2$ to denote that there is a $J_3 = \left\langle I_3, \text{rcy}_3 \right\rangle$ such that (i) $I_1 \xrightarrow{\alpha, \nu} I_3$. (ii) $\text{rcy}_3(e) = 0$ if $e = \nu(x)$ for some $x \in \overline{x}$. (iii) $\text{rcy}_3(e) = \text{rcy}_1(e) + 1$ if $e \in \textsc{Adom}(I_1) - \nu(\overline{x})$. (v) $\text{rcy}_3(e) = 0$ if $e = \nu(y)$ for some $y \in \overline{y}$. (v) $J_2 = \text{compactify}(J_3)$. We define $\xrightarrow{\alpha}_B$, $\to_B$, and $\xrightarrow{*}_B$ in a similar manner to $\to$.

An instance of the reachability problem under the recency semantics is defined in a similar manner to above. However, we are also given a bound B and we are asked whether there is a database instance I' such that $I \xrightarrow{*}_B I'$ and q occurs in I'.

Our main theorem states that

Theorem 4.1. *Recency bounded reachability problem is ExpTime-complete.*

The lower bound is proved by a reduction from the intersection-non-emptiness problem of a pushdown automaton and n finite state automata. Our reduction does not need numerical constraints at all, showing that recency bounded reachability is ExpTime hard already for the DMS *à la* [3].

The upper bound is established by constructing an exponential sized pushdown automaton and translating the reachability problem to control state reachability problem in the constructed pushdown automaton. The construction is given in Sections 6 and 7.

We will also show that the decidability does not carry forward to the case of recency bounded model checking of LTL. We recall the definition of LTL-FO language and the recency bounded model checking problem in Appendix A.

Theorem 4.2. *Recency bounded model checking of DMS with numerical constraints against LTL-FO is undecidable.*

Proof is given in Appendix A. It is by a reduction from the control state reachability problem of Minsky machines.

5 EXAMPLE

To illustrate the main modelling features of our approach, we use a marketplace for recycling electronic devices. Users may bring broken devices to the marketplace, and such devices can then be fixed by reconditioning their broken/missing components using functioning components of other devices. Fully repaired devices can then be sold in the marketplace at a discounted price.

For simplicity, we consider only a single type of device, consisting of two components: a memory, and a cpu. Each device may differ, though, not only depending on which components are actually functioning, but also on the technical properties of such components, and on the ranges supported for such properties. For example, a device may come with a functioning memory of 1GB and no cpu, and with a working range of 32-64GB for its memory, and of 200-300MHz for its cpu.

Assuming that devices are identified using a code, the above data can be stored using the following schemas:

- Unary relation schemas to represent devices and their current status. In particular, $\text{Device}(c)$ indicates that c is (the code of) a device, $\text{Ok}(c)$ that c is fully operational, $\text{NoMem}(c)$ (resp., $\text{NoCpu}(c)$) that device c is missing the memory (resp., the cpu).
- Binary relation schemas to capture the properties of the working components. For simplicity, we focus on the simple properties of storage capacity and speed. Specifically, $\text{Capacity}(c, v)$ indicates that device c has a functioning memory whose capacity is $\#v$ GB, while $\text{Speed}(c, v)$ captures that device c has a functioning cpu whose speed is $\#v$ MGHz.
- Ternary relation schemas to capture the ranges for the memory capacity and the cpu speed that the device can support. Specifically, $\text{SupCapacity}(c, \text{min}, \text{max})$ and $\text{SupSpeed}(c, \text{min}, \text{max})$ respectively indicate that #min and #max are the minimum and maximum values that are compatible with device c in terms of memory capacity and cpu speed.

To introduce a new product into the marketplace, one has to indicate the device code, the device status, and the other properties related to the design. Since depending on the device status, certain properties are needed whereas others are not applicable, this input task is captured in the DMS as a set of actions, one per device status type. Below, we show the action for inserting into the marketplace a device with a missing cpu:

- $\varphi = \{\#y^c_{\text{min}} \leq \#y^c_{\text{max}}, \#y^m_{\text{min}} \leq \#y^m, \#y^m \leq \#y^m_{\text{max}}\}$
- $\textsc{Add} = \{\ \text{Device}(y), \text{NoCpu}(y), \text{Capacity}(y, y^m),$
  $\text{SupCapacity}(y, y^m_{\text{min}}, y^m_{\text{max}}), \text{SupSpeed}(y, y^c_{\text{min}}, y^c_{\text{max}})\ \}$
- $\textsc{Del} = \emptyset$

A similar action can be defined to insert a device with a missing memory.

As introduced above, devices can be combined so as to obtain a functioning device as a result. We define a *recondition* action capturing how a broken device can be made

operational by incorporating the missing component from another device. In our simple representation of devices, we assume that it is easier to extract the memory component from a device rather than the cpu, and hence a broken device x_1 missing memory is reconditioned with the memory taken from another broken device x_2. As a result, x_1 becomes fully operational, whereas x_2 is removed from the marketplace since it does not have any other component that can be reused. This reconditioning can only be applied if the memory offered by x_2 is compatible with the memory range supported by x_1. Technically, the recondition action is then defined as follows:

- $I_+ = \{\mathsf{NoMem}(x_1), \mathsf{NoCpu}(x_2), \mathsf{Capacity}(x_2, x^2_{\mathsf{mem}}), \mathsf{SupCapacity}(x_1, x^{m1}_{\mathsf{min}}, x^{m1}_{\mathsf{max}})\}$
- $\varphi = \{\#x^{m1}_{\mathsf{min}} \leq \#x^2_{\mathsf{mem}}, \#x^2_{\mathsf{mem}} \leq \#x^{m1}_{\mathsf{max}}\}$
- Add = { $\mathsf{Ok}(x_1), \mathsf{Capacity}(x_1, x^2_{\mathsf{mem}})$ }
- Del = { $\mathsf{NoMem}(x_1), \mathsf{Device}(x_2), \mathsf{NoCpu}(x_2), \mathsf{Capacity}(x_1, x^2_{\mathsf{mem}}), \mathsf{SupCapacity}(x_2, x^{m2}_{\mathsf{min}}, x^{m2}_{\mathsf{max}}), \mathsf{SupSpeed}(x_2, x^{s2}_{\mathsf{min}}, x^{s2}_{\mathsf{max}})$ }

Finally, a fully operational device can be sold, which results in removing the device from the database. If we were following the recency bounded semantics, this deletion would enable less recent devices to pop up into the recency window.

Notice that, in this context, the bound of recency can be interpreted as a marketplace where recently introduced devices are put in a front-end showcase and can be used for reconditioning, whereas older devices are stored in a back-end warehouse. If the showcase gets free space (obtained by removing useless devices after a reconditioning action, or by selling functioning devices), then the most recent elements currently stored in the warehouse are moved into the showcase.

We consider two interesting safety properties to be checked on the marketplace recycling process.

The first property aims at ensuring that the device code is a key in all the relation schemas forming the marketplace database schema. This property can be turned into a propositional reachability property by introducing a proposition *key_viol* that is inserted in the database whenever one of the relations contains two distinct tuples that agree on the device code. This, in turn, is done by introducing one action per relation schema, whose query checks the violation of the key constraint on that relation, and whose effect is to raise the *key_viol* flag. For example, for relation Capacity we get:

- $I_+ = \{\mathsf{Capacity}(x, x^1_{\mathsf{mem}}), \mathsf{Capacity}(x, x^2_{\mathsf{mem}})\}$
- $\varphi = \{\#x^1_{\mathsf{mem}} \neq \#x^2_{\mathsf{mem}}\}$
- Add = { *key_viol* }, Del = $\emptyset$

If the marketplace DMS reaches *key_viol*, then it violates the desired property.

The second property adopts a similar strategy used for the first one, this time to ascertain that no device in the marketplace contains components that are out of range. Towards this, we have actions that raise the `hit` flag whenever a faulty device (i.e., a device with memory and/or cpu plugged in but out of range) is spotted in the marketplace:

- $I_+ = \{\mathsf{Capacity}(x, x_{mem}), \mathsf{SupCapacity}(x, x^m_{\mathsf{min}}, x^m_{\mathsf{max}})\}$
- $\varphi = \{\#x_{\mathsf{mem}} \leq \#x^m_{\mathsf{min}}\}$
- Add = { `hit` }, Del = $\emptyset$

We will have a similar action with different numerical constraint: $\varphi = \{\#x^m_{\mathsf{min}} \leq \#x_{\mathsf{mem}}\}$, and two more for the case of cpu.

If the marketplace DMS reaches `hit`, then it violates the property.

6 LAYERS

We give the definition and the different operations on layers.

6.1 Definition

We assume a finite set $\mathcal{O}$ of *objects*. In Appendix C we show that taking an $\mathcal{O}$ with $3m(B+2)$ elements will be sufficient for our encoding, where m is the maximum number of variables used in any action.

A *layer* $\mathcal{L}$ is a tuple $\langle A, D, \mathrm{rcy}, I, \mathrm{busy}, \varphi, M, g\rangle$ where $A \subseteq \mathcal{O}$ is the set of *actual objects*, $D \subseteq \mathcal{O}$ is a set of objects that have been *logically deleted* where $A \cap D = \emptyset$, $\mathrm{rcy} : A \to \widehat{B+1}$ defines the recency measures of the actual objects, I is a database instance over A, $\mathrm{busy} : A \to \mathbb{B}$ indicates whether a given actual object is involved in facts with objects whose recency measures are larger than B, φ is a numerical constraint over $A \cup D$, M is the set of *mirror objects* where $(A \cup D) \cap M = \emptyset$, and $g : A \cup D \to M$ is a function such that (i) for each $o \in M$ there is an o' with $g(o') = o$, and (ii) $g(o) \neq \bot$ if $o \in D$. The actual objects will have recency measures that are up to the bound B in the "stable" states of the symbolic simulation. However, during the simulation, the system may enter transient states in which the recency bounds of some actual objects become equal to $B+1$. Objects whose recency measures remain equal to $B+1$ will be "moved" to the next recency window. The injection g tells us which actual and mirror objects correspond to the same members of the data domain.

We say that $\mathcal{L}$ is *consistent* if φ is consistent. Given that the set $\mathcal{O}$ of objects is finite, there are only finitely many layers. In fact, the number of layers will be bounded by $2^{O(|\mathcal{R}|(mB)^{\mathrm{amax}})}$ where amax is the maximum arity of any relation in the schema. Details of this counting are given in Appendix C.

6.2 Compactification

For a database instance I over $\mathcal{O}$, we write $o \in I$ if o occurs in some fact in I. An actual object $o \in A$ is said to be *useless* in $\mathcal{L}$ if $\text{busy}(o) = \text{false}$ and $o \notin I$. Intuitively, a useless object is not involved in any fact whether in the current or in the other recency windows. The logically deleted objects are useless but have a mirror in the layer. An object is said to be *useful* if it is not useless. We use $\mathcal{L}\cdot\text{useless}$ and $\mathcal{L}\cdot\text{useful}$ to denote the sets of useless and useful objects in $\mathcal{L}$ respectively. We say that $\mathcal{L}$ is *pure* if A does not contain any useless objects and all the objects in D have a mirror in $\mathcal{L}$. For a layer $\mathcal{L}_1 = \langle A_1, D_1, \text{rcy}_1, I_1, \text{busy}_1, \varphi_1, M_1, g_1\rangle$, we define $\text{purify}(\mathcal{L}_1) := \langle A_2, D_2, \text{rcy}_2, I_2, \text{busy}_2, \varphi_2, M_2, g_2\rangle$ where

- $A_2 = \mathcal{L}_1\cdot\text{useful}$. We keep only the useful objects in the set of actual objects.
- $D_2 = \{o \in \mathcal{L}_1\cdot\text{useless} \mid g_1(o) \neq \bot\}$. The logically deleted objects are those that are useless but have a mirror.
- $\text{rcy}_2 = \text{rcy}_1\big|_{A_2}$. We restrict the recency function to the useful objects.
- $R^{I_2} = R^{I_1}\big|_{A_2}$ for each relation name R, where a is the arity of R. We do not include facts that contain useless objects.
- $\text{busy}_2 = \text{busy}_1\big|_{A_2}$, i.e., we restrict the busy function to the useful objects.
- $\varphi_2 = \varphi_1\big|_{A_2 \cup D_2}$. In the numerical constraint, we keep both the useful and logically deleted objects (but throw away constraints involving objects that are useless and that do not have mirrors in the layer). The reason for keeping the logically deleted objects is that they may impose numerical constraints on the objects that are in the next layer, but not in the current layer. This will be more clear in the definition of the merge operation below.
- $M_2 = M_1$. The mirror objects are not affected.
- $g_2 = g_1$. The injection to mirror objects is not changed (although some objects in the domain of g_2 may have moved to the logically deleted objects).

We define $\text{present}(\mathcal{L}) := \{i \mid (1 \leq i \leq B+1) \wedge (\exists o \in A.\ \text{rcy}(o) = i)\}$, i.e., it is the set of measures that are present among the actual objects of $\mathcal{L}$. We define $\text{rcy}(\mathcal{L}) := \max\{\text{rcy}(o) \mid o \in A\}$, i.e., it is the maximal recency measure that appears in $\mathcal{L}$.

A layer is said to be *compact* if $j \in \text{present}(\mathcal{L})$ implies that $i \in \text{present}(\mathcal{L})$ for every $i : 0 \leq i < j$. In other words, there is no gap below the recency measures of the actual objects in $\mathcal{L}$. For a layer $\mathcal{L}_1 = \langle A_1, D_1, \text{rcy}_1, I_1, \text{busy}_1, \varphi_1, M_1, g_1\rangle$, we define $\text{compactify}(\mathcal{L}_1)$ to be the unique compact layer $\langle A_2, D_2, \text{rcy}_2, I_2, \text{busy}_2, \varphi_2, M_2, g_2\rangle$ such that $A_2 = A_1$, $D_2 = D_1$, $I_2 = I_1$, $M_2 = M_1$, and $\text{rcy}_2(o) \leq \text{rcy}_2(o')$ iff $\text{rcy}_1(o) \leq \text{rcy}_1(o')$ for all actual objects $o, o' \in A_1$. In other words, we get $\mathcal{L}_2$ from $\mathcal{L}_1$ by putting objects with successor recency measures next to each other.

Consider a compact layer $\mathcal{L} = \langle A, D, \text{rcy}, I, \text{busy}, \varphi, M, g\rangle$. We say that $\mathcal{L}$ is *under-loaded* if $\text{rcy}(\mathcal{L}) < B$, i.e., there is no actual object $o \in A$ such that $\text{rcy}(o) = B$. We say that $\mathcal{L}$ is *balanced* if $\text{rcy}(\mathcal{L}) = B$, i.e., there is an actual object $o \in A$ such that $\text{rcy}(o) = i$ but there there is no actual object $o \in A$ such that $\text{rcy}(o) = B+1$. Finally, we say that $\mathcal{L}$ is *over-loaded* if $\text{rcy}(\mathcal{L}) = B+1$, i.e., there is an actual object $o \in A$ such that $\text{rcy}(o) = B+1$.

6.3 Running Actions

We describe how to apply an action to a layer $\mathcal{L}_1 = \langle A_1, D_1, \text{rcy}_1, I_1, \text{busy}_1, \varphi_1, M_1, g_1\rangle$. Consider an action $\alpha = \langle \overline{x}, \overline{y}, I_+, I_-, \textsc{Del}, \textsc{Add}, \varphi\rangle$, and a substitution $\nu : (\overline{x} \cup \overline{y}) \to O^{old} \cup O^{new}$ where $O^{old} \subseteq A_1$, $O^{new} \subseteq \mathcal{O}$, $O^{new} \cap A_1 = \emptyset$, $\nu(x) \in O^{old}$ if $x \in \overline{x}$, and $\nu(y) \in O^{new}$ if $y \in \overline{y}$. Intuitively, the substitution maps the variables corresponding to old objects to members of A_1. This is consistent with the fact that we use A_1 to represent data belonging to the current recency window. Furthermore, the set O^{new} contains objects that correspond to the new data that will be created by the rule. We introduce a rule that explains the application of the action α to $\mathcal{L}$ under the substitution ν. The application will be captured by a transition relation $\xrightarrow{\alpha,\nu}$ on layers. The transition relation will be defined in several steps. First, we define an intermediate transition relation $\overset{\alpha,\nu}{\rightsquigarrow}$ that applies the action without purifying or compacting the resulting layer. Consider two layers $\mathcal{L}_1 = \langle A_1, D_1, \text{rcy}_1, I_1, \text{busy}_1, \varphi_1, M_1, g_1\rangle$ and $\mathcal{L}_2 = \langle A_2, D_2, \text{rcy}_2, I_2, \text{busy}_2, \varphi_2, M_2, g_2\rangle$. Define $A_1' = A_1 - O^{old}$. We write $\mathcal{L}_1 \overset{\alpha,\nu}{\rightsquigarrow} \mathcal{L}_2$ to denote that there is a bijection $h : A_2 \to A_1 \cup O^{new}$ such that the conditions below are satisfied. Notice that h means that the actual objects of $\mathcal{L}_2$ are renamings of the actual objects that are already in $\mathcal{L}_1$ and the new objects that are introduced by α.

- $A_2 \cap A_1 = \emptyset$.
- $D_2 = \emptyset$. We will identify the logically deleted objects when we purify $\mathcal{L}_2$ in the next step below.
- $\text{rcy}_2(o) = 0$ if $h(o) \in O^{old} \cup O^{new}$, and $\text{rcy}_2(o) = \text{rcy}_1(o) + 1$ if $h(o) \in A_1'$, i.e, we reset the recency measures of the objects that are involved in the action α to zero, and increase the recency measures of the other objects by one.
- $R^{\nu(I_+)} \subseteq R^{I_1}$ and $R^{\nu(I_-)} \cap R^{I_1} = \emptyset$ for all relation names R. We check that all the facts in I_+ are present, and that all the facts in I_- are absent from $\mathcal{L}_1$.
- $\langle o_1, \ldots, o_n\rangle \in R^{I_2}$ iff $\langle h(o_1), \ldots, h(o_n)\rangle \in R^{I_1 - \nu(\textsc{Del}) \cup \nu(\textsc{Add})}$ for all relation names R. We remove the facts corresponding to Del and add the facts corresponding to Add.
- $\text{busy}_2(o) = \text{false}$ if $h(o) \in O^{new}$ and $\text{busy}_2(o) = \text{busy}_1(h(o))$ if $h(o) \in A_1$. The busy-flags of the new objects are set to false since they are not involved in any

facts outside the current layer. The busy-flags of the other objects are maintained.

- $\varphi_2 = \{(o_1 \sim o_2) |\ (h(o_1) \sim h(o_2)) \in \text{saturate}(\varphi \cup \nu(\psi))\}$. We add the new numerical query to the existing numerical constraint, and saturate the result. Notice that this may mean that the layer becomes inconsistent. As we will see later, we will block runs that reach inconsistent layers from the analysis.
- $M_2 = A_1$. The mirror objects in $\mathcal{L}_2$ are the actual objects in $\mathcal{L}_1$. Notice that $M_2 \cap A_2 = \emptyset$ since $A_2 \cap A_1 = \emptyset$.
- $g_2(o) = o'$ if $o' \in A_1$ and $h(o) = o'$. The actual objects in $\mathcal{L}_1$ will mirror the corresponding objects in $\mathcal{L}_2$.

The layer $\mathcal{L}_2$ may contain useless objects. Such objects are those involved in facts all of which are included in ν (Del) and which are not involved in relations with objects outside the current recency window (i.e., their busy-flag is false). Therefore, we will purify $\mathcal{L}_2$. Assume that $\mathcal{L}_3 = \text{purify}(\mathcal{L}_2)$ which means that the useless objects will be removed from the set of actual objects. Since, we have removed the useless objects, the layer $\mathcal{L}_3$ may not be compact, and hence we need to compactify $\mathcal{L}_3$. We define the transition relation $\rightarrow$ such that $\mathcal{L}_1 \xrightarrow{\alpha,\nu} \mathcal{L}_2$ if there is a layer $\mathcal{L}_3$ where $\mathcal{L}_1 \overset{\alpha,\nu}{\rightsquigarrow} \mathcal{L}_3$ and compactify (purify ($\mathcal{L}_3$)). The layer $\mathcal{L}_2$ is compact, but it may be under-loaded, balanced, or over-loaded. We will take these three cases into consideration when we later perform our simulation.

6.4 Merging and Truncation

As we saw above, applying $\overset{\alpha,\nu}{\rightsquigarrow}$ may result in a layer which can be under- or over-loaded. In both cases, we define operations that allow getting balanced layers. First, we consider the case of under-loading. An under-loaded-layer does not contain sufficient information about the current recency window since it does not contain all elements whose recency measures are up to B. In order to retrieve all these elements we *merge* the layer with the next layer. This will give access more objects without exceeding the bound B. Below, we define the merge operation.

Consider two layers $\mathcal{L}_1 = \langle A_1, D_1, \text{rcy}_1, I_1, \text{busy}_1, \varphi_1, M_1, g_1 \rangle$ and $\mathcal{L}_2 = \langle A_2, D_2, \text{rcy}_2, I_2, \text{busy}_2, \varphi_2, M_2, g_2 \rangle$ such that $\mathcal{L}_1$ is compact and under-loaded while $\mathcal{L}_2$ is compact and balanced. We define their merging $\mathcal{L}_1 \otimes \mathcal{L}_2$ in two steps. Define $A'_2 := \{o \in A_2 |\ o \notin M_1\}$, i.e., it is the set of actual objects in $\mathcal{L}_2$ that do not mirror any objects in $\mathcal{L}_1$. Also, define $D'_1 := \{o \in D_1 |\ g_2(g_1(o)) \neq \bot\}$, i.e., it is the set of logically deleted objects that have mirrors in $\mathcal{L}_2$ (notice that such objects have by construction mirrors in $\mathcal{L}_1$.) First we define $\mathcal{L}_1 \odot \mathcal{L}_2$ to be the set of layers including each layer $\mathcal{L}_3 = \langle A_3, D_3, \text{rcy}_3, I_3, \text{busy}_3, \varphi_3, M_3, g_3 \rangle$ such that there is an injection $h : A_3 \cup D_3 \overset{\bullet}{\rightarrow} A_1 \cup A'_2 \cup D'_1 \cup D_2$ satisfying the following conditions:

- $A_3 \cap D_3 = \emptyset$ and $(A_3 \cup D_3) \cap M_2 = \emptyset$. We need these conditions since they are required by the definition a layer. In particular, the second condition ensures that the set of actual objects A_3 will not intersect with the set of mirror objects M_3 (below, we will define M_3 to be the same as M_2).
- $h\big|_{A_3}$ is a bijection from A_3 to $A_1 \cup A'_2$. The actual objects in $\mathcal{L}_3$ correspond to the actual objects in $\mathcal{L}_1$ together with the actual objects in $\mathcal{L}_2$ whose recency measures are B.
- $h\big|_{D_3}$ is a bijection from D_3 to $D'_1 \cup D_2$. The logically deleted objects in $\mathcal{L}_3$ correspond to the logically deleted objects in $\mathcal{L}_2$ together with logically deleted objects in $\mathcal{L}_1$ that have mirrors in $\mathcal{L}_2$.
- $\text{rcy}_3(o) = \text{rcy}_1(o)$ if $h(o) \in A_1$, and $\text{rcy}_3(o) = \text{rcy}_2(o) = B$ if $h(o) \in A'_2$. The objects in $\mathcal{L}_3$ take their recency measures from the corresponding objects in $\mathcal{L}_1$ and $\mathcal{L}_2$.
- For every relation name R, we have that $\langle o_1, \ldots, o_n \rangle \in R^{I_3}$ iff one of the following two conditions is satisfied:
  - There are objects $o'_1, \ldots, o'_n \in A_1$ such that $\langle o'_1, \ldots, o'_n \rangle \in R^{I_1}$ and $h(o_i) = o'_i$ for all $i : 1 \leq i \leq n$. In this case, the fact $\langle o'_1, \ldots, o'_n \rangle$ involves objects that are all in the current recency window (defined by $\mathcal{L}_1$). Therefore, we copy the fact from $\mathcal{L}_1$ to $\mathcal{L}_3$ (while renaming according to h).
  - There are objects $o'_1, \ldots, o'_n \in A_2$ such that the following three conditions are satisfied:
    - $\langle o'_1, \ldots, o'_n \rangle \in R^{I_2}$,
    - $o'_i \in A'_2$ for some $i : 1 \leq i \leq n$.
    - For each $j : 1 \leq j \leq n$. either $o'_i \in M_1$ and $g(h(o_i)) = o'_i$, or $o_i \in A'_2$ and $h(o_i) = o'_i$.

    In this case, the fact $\langle o'_1, \ldots, o'_n \rangle$ involves at least one element from A'_2, and hence it cannot be found in $\mathcal{L}_1$. Therefore, we copy the fact from $\mathcal{L}_2$. Notice that (i) all the objects $o'_1, \ldots, o'_n$ belong to A_2, (ii) each object o'_i either belongs to A'_2 or mirrors some object in A_1, and (iii) an object in A_2 belongs either to M_1 or A'_2.
- $\text{busy}_3(o)$ is defined as follows:
  - If $h(o) = o' \in A_1$ and $g_1(o') = \bot$ then $\text{busy}_3(o) = \text{false}$. The object o does not have a mirror object in the recency window defined by $\mathcal{L}_2$, and hence it cannot be part of facts that belong to $\mathcal{L}_2$ (or later layers).
  - If $h(o) \in A_1$ and $g_1(h(o)) = o'$ then $\text{busy}_3(o) = \text{busy}_2(o')$. The object o has a mirror in $\mathcal{L}_1$, and hence its busy-flag is defined by $\mathcal{L}_2$. Notice that the flag of the object may be true in $\mathcal{L}_1$ only because it is involved in a fact with objects whose recency measures are B in $\mathcal{L}_2$.
  - If $h(o) = o' \in A'_2$ then $\text{busy}_3(o) = \text{busy}_2(o')$. The object is part of $\mathcal{L}_2$ (but not $\mathcal{L}_1$) and hence its busy-flag is defined accordingly.

- $\varphi_3 = \{o_1 \sim o_2 |\ (h(o_1) \sim h(o_2)) \in \text{saturate}(\varphi_1 \cup \varphi_2)\} \cap (A_3 \cup D_3)$. In other words, we take the union of the numerical constraints, and saturate the result. Notice that it is important to include the objects that are logically deleted in $\mathcal{L}_1$ and that do not have mirrors in $\mathcal{L}_2$. when we saturate $\varphi_1 \cup \varphi_2$. The reason is that they may imply constraints among the other objects. However, after performing the saturation, these objects may be safely deleted. The definition of φ_3 ensures that the deletion will take place since we only include objects in $A_3 \cup D_3$.
- $M_3 = M_2$. The set of mirror objects are the same as in $\mathcal{L}_2$
- g_3 is defined as follows:
  - If $o \in A_3$, $h(o) \in A_1$, and $g_1(h(o)) = \perp$ then $g_3(o) = \perp$. If an object does not have a mirror in $\mathcal{L}_1$ then the corresponding object will not have a mirror in $\mathcal{L}_3$.
  - If $o \in A_3$, $h(o) \in A_1$, and $g_1(h(o)) \neq \perp$ then $g_3(o) = g_2(g_1(h(o)))$. If an object has a mirror in $\mathcal{L}_1$, and the mirror has in turn a mirror in $\mathcal{L}_2$, then the latter defines the mirror of o in the new layer $\mathcal{L}_3$.
  - If $o \in A_3$ and $h(o) \in A'_2$ then $g_3(o) = g_2(h(o))$. If an object is part of $\mathcal{L}_2$ then its mirror object is defined accordingly.
  - If $o \in D_3$ and $h(o) \in D'_1$ then $g_3(o) = g_2(g_1(h(o)))$. If an object is logically deleted in $\mathcal{L}_2$ but its mirror (in $\mathcal{L}_1$) has a mirror in $\mathcal{L}_2$, then the latter defines the mirror of o in $\mathcal{L}_3$.
  - If $o \in D_3$ and $h(o) \in D_2$ then $g_3(o) = g_2(h(o))$. If an object is logically deleted in $\mathcal{L}_2$ then the mirror of the corresponding object in $\mathcal{L}_3$ will be taken to be the same mirror.

Now we define the merging of $\mathcal{L}$ and $\mathcal{L}'$ by $\mathcal{L} \otimes \mathcal{L}' :=$ compactify $(\mathcal{L} \odot \mathcal{L}')$. Notice that $\mathcal{L} \otimes \mathcal{L}'$ can still be underloaded (but not over-loaded).

To handle the case where a layer is over-loaded we will use an operation that *truncates* the layer by removing the objects whose recency measures are equal to $B+1$. Consider a compact layer $\mathcal{L}_1 = \langle A_1, D_1, \text{rcy}_1, I_1, \text{busy}_1, \varphi_1, M_1, g_1 \rangle$ such that $\text{rcy}(\mathcal{L}_1) = B+1$. We define *truncate* $(\mathcal{L}_1) := \langle A_2, D_2, \text{rcy}_2, I_2, \text{busy}_2, \varphi_2, M_2, g_2 \rangle$, such the following conditions are satisfied:

- $A_2 = \{o \in A_1 |\ \text{rcy}(o) \leq B\}$. We include only the actual objects whose recency measures are up to the bound B.
- $D_2 = D_1$, i.e., we keep all the logically deleted objects.
- $\text{rcy}_2 = \text{rcy}_1\big|_{A_2}$. We restrict the recency function to the actual objects in $\mathcal{L}_2$.
- $R^{I_2} = R^{I_1}\big|_{A_1}$ for each relation name R. We only include facts that contain objects from the set A_2, and remove the rest of the facts.
- $\text{busy}_2(o) =$ true iff either $\text{busy}_1(o) =$ true or there is a relation name R and a fact $\langle o_1, \ldots, o_n \rangle \in R^{I_1}$ such that $o_i = o$ and $\text{rcy}_1(o_j) = B+1$ for some $i, j : 1 \leq i \neq j \leq n$. The busy-flag of an object is set to true if it is already true or if it is part of a fact that also involves an object whose recency measure is $B+1$. The reason for the latter case is that the second object will not be part of the current layer any more.
- $\varphi_2 = \varphi_1\big|_{A_2}$. We restrict the numerical constraint to the objects in A_2.
- $M_2 = \{o \in M_1 |\ \exists o' \in A_2.\ g_1(o) = o'\}$. We include only objects that mirror some object in the set A_2.
- $g_2 = g_1\big|_{A_2}$. We restrict the injection g_2 as expected.

7 LAYERED DATABASES

In this section, we present our symbolic simulation of DMS under the recency semantics. We will represent a database instance symbolically using a push-down automaton where the stack carries a word of layers. The layers in the stack encode recency windows with increasing recency measures, and in particular. the top-most layer represents the current recency window.

Push-Down Automata. We recall the classical model of Push-Down Automata (PDA). A PDA $\mathcal{P}$ is a tuple $\langle Q, q_{init}, \Gamma, T \rangle$ where Q is the set of *states*, $q_{init} \in Q$ is the *initial state*, Γ is the (finite) stack alphabet, and T is the set of *transitions*. A transition $t \in T$ is a triple $\langle q_1, op, q_2 \rangle$ where $q_1, q_2 \in Q$ are states and op is an *operation* of one of three forms: (i) *nop* is an empty operation that does not change the content of the stack, (ii) *push* (a), where $a \in \Gamma$, adds a to the top of the stack, (iii) *pop* (a), where $a \in \Gamma$, removes the top-most stack symbol if this symbol is equal to a. A *configuration* β is a pair $\langle q, w \rangle$ where $q \in Q$ is the (local) state of the automaton, and $w \in \Gamma^*$ is the content of the stack. We define a transition relation $\rightsquigarrow_{\mathcal{P}}$ on the set of configurations as follows. For configurations $\beta_1 = \langle q_1, w_1 \rangle$ and $\beta_2 = \langle q_2, w_2 \rangle$, and a transition $t = \langle q_1, op, q_2 \rangle$, we write $\beta_1 \overset{t}{\rightsquigarrow}_{\mathcal{P}} \beta_2$ to denote that one of the following three conditions is satisfied: (i) $op = nop$ and $w_1 = w_2$, (ii) $op = push(a)$ and $w_2 = a \bullet w_1$, or (ii) $op = pop(a)$ and $w_1 = a \bullet w_2$. We define $\rightsquigarrow_{\mathcal{P}} := \cup_{t \in T} \overset{t}{\rightsquigarrow}_{\mathcal{P}}$, and define $\overset{*}{\rightsquigarrow}_{\mathcal{P}}$ to be the reflexive transitive closure of $\rightsquigarrow_{\mathcal{P}}$. For a configuration β and a state q, we write $\beta \overset{*}{\rightsquigarrow}_{\mathcal{P}} q$ to denote that there is a word $w \in \Gamma^*$ such that $\beta \overset{*}{\rightsquigarrow}_{\mathcal{P}} \langle q, w \rangle$. An instance of the *reachability problem* consists of an (initial) configuration β_{init} and a (target) state q_F and we are asked whether $\beta \overset{*}{\rightsquigarrow}_{\mathcal{P}} q_F$.

Consider a DMS $\mathcal{D}$, a recency bound B, and a proposition p. The *layered database* $\mathcal{B}$ induced by $\mathcal{D}$ is a push-down automaton $\langle Q, q_{init}, \Gamma, T \rangle$ defined as follows.

Stack Alphabet. The stack alphabet Γ consists of the set of all compact layers, in addition to the *stack bottom* symbol Btm. By construction, the bottom symbol of the stack will be Btm, while all the other symbols will be compact layers. All the layers, except the one next to the bottom of the stack will

be balanced. The layer next to the bottom of the stack may be under-loaded. This happens when all the objects represent data elements that are in the current recency window. No over-loaded layers will ever be pushed to the stack.

Initial Configuration. Let J be database instance over Δ, with $\textsc{Adom}(J) = \{e_1, \ldots, e_n\}$. We will encode J as a layer $\text{encode}(J) = \langle A, D, \text{rcy}, I, \text{busy}, \varphi, M, g\rangle$ over the set objects O. For this, we may assume that $|\textsc{Adom}(J)| < B$, as we can keep $\max\{|\textsc{Adom}(J)|, B\}$ as the real B. [1] $A = \{o_1, \ldots, o_n\}$, $D = \emptyset$, $\text{rcy}(o) = 0$ for all actual objects $o \in A$, $\langle o_1, \ldots, o_n\rangle \in R^I$ iff $\langle e_1, \ldots, e_n\rangle \in R^J$ for every relation name R, $\text{busy}(o) =$ false for all actual objects $o \in A$, $(o_1 \sim o_2) \in \varphi$ iff $\#e_1 \sim \#e_2$, $M = \emptyset$, and $g(o) = \perp$ for all objects $o \in A \cup D$.

The initial configuration is defined by $\langle q_{init}, \text{encode}(J) \bullet \textsc{Btm}\rangle$

States. In addition to q_{init} and the target state q_F, the set Q contains the state Origin which is the "stable state" from which we perform the actions of $\mathcal{D}$ on the top-most layer in the stack. Furthermore, the set Q contains a number of "transient states" that are used to mimic the actions. More precisely, for each compact layer $\mathcal{L}$, the set Q contains the following states:

- [Fetched, $\mathcal{L}$]: we have fetched the layer $\mathcal{L}$ from the stack.
- [Restored, $\mathcal{L}$]: We have put back the layer $\mathcal{L}$ to the stack.
- [Derived, $\mathcal{L}$]: we have derived the layer $\mathcal{L}$ by applying an action to a layer that we have fetched from the stack.
- [Merging, $\mathcal{L}_1, \mathcal{L}_2$]: we are about to merge the layers $\mathcal{L}_1$ and $\mathcal{L}_2$.

Transitions. The set T contains the following transitions.

- $\langle q_{init}, nop, \textsc{Origin}\rangle$. We move from the initial state to the state Origin from which we start the simulation.
- $\langle \textsc{Origin}, pop(\mathcal{L}), [\textsc{Fetched}, \mathcal{L}]\rangle$ for each compact layer $\mathcal{L}$. From the state Origin, we can non-deterministically fetch a layer $\mathcal{L}$ from the stack.
- $\langle [\textsc{Fetched}, \mathcal{L}], push(\mathcal{L}), [\textsc{Restored}, \mathcal{L}]\rangle$ for each compact and balanced layer $\mathcal{L}$. We restore the stack by immediately pushing pack the layer we have fetched in the previous step. The purpose of this transition (together with the previous transition) is to find out the top-most layer $\mathcal{L}$. Notice that $\mathcal{L}$ is stored in the definition of the state (each layer $\mathcal{L}$ has its own [Fetched, $\mathcal{L}$]-state.) We also have a transition $\langle [\textsc{Fetched}, \mathcal{L}], nop, [\textsc{Restored}, \mathcal{L}]\rangle$ if $\mathcal{L}$ is under-loaded.
- $\langle [\textsc{Restored}, \mathcal{L}_1], nop, [\textsc{Derived}, \mathcal{L}_2]\rangle$ for all compact layers $\mathcal{L}_1$ and $\mathcal{L}_2$ where $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ and $\mathcal{L}_2$ is consistent. We consider the successors of $\mathcal{L}_1$ wrt. $\rightarrow$.
- $\langle [\textsc{Derived}, \mathcal{L}], push(\mathcal{L}), \textsc{Origin}\rangle$ for each compact and balanced layer. If the layer is balanced we simply push it to the stack. An action has been performed, and we return back to the state Origin from which we look for the next action.
- $\langle [\textsc{Derived}, \mathcal{L}_1], push(\mathcal{L}_2), \textsc{Origin}\rangle$ for all compact layers $\mathcal{L}_1$ and $\mathcal{L}_2$ such that $\mathcal{L}_1$ is over-loaded and $\mathcal{L}_2 = truncate(\mathcal{L}_1)$. In case the derived layer is over-loaded, we truncate it first before we push it to the stack.
- $\langle [\textsc{Derived}, \mathcal{L}_1], pop(\mathcal{L}_2), [\textsc{Merging}, \mathcal{L}_1, \mathcal{L}_2]\rangle$ where $\mathcal{L}_1$ is compact and under-loaded, and $\mathcal{L}_2$ is either a compact layer or equal to Btm. Since $\mathcal{L}_1$ is under-loaded it needs to be merged with another layer. Therefore, we fetch $\mathcal{L}_2$ from the stack.
- $\langle [\textsc{Derived}, \mathcal{L}], nop, q_F\rangle$ if the target proposition p occurs in $\mathcal{L}$.
- $\langle [\textsc{Merging}, \mathcal{L}_1, \mathcal{L}_2], nop, [\textsc{Derived}, \mathcal{L}_3]\rangle$ for all compact layers $\mathcal{L}_1$ and $\mathcal{L}_2$ where $\mathcal{L}_3 \in \mathcal{L}_1 \otimes \mathcal{L}_2$. We have derived the layer $\mathcal{L}_3$ through the merging of the layers $\mathcal{L}_1$ and $\mathcal{L}_2$. We also have the transition $\langle [\textsc{Merging}, \mathcal{L}, \textsc{Btm}], push(\mathcal{L}), \textsc{Origin}\rangle$.

Correctness. Corresponding to any path $\pi_{\mathcal{D}}$ starting from the initial database instance J in the B-bounded configuration graph of $\mathcal{D}$, we have a path $\pi_{\mathcal{B}}$ in the configuration graph of our constructed PDA $\mathcal{B}$. Further, the length of $\pi_{\mathcal{D}}$ and the number of stable states in $\pi_{\mathcal{B}}$ are in bijection. Symmetrically, corresponding to any path $\pi_{\mathcal{D}}$ in the configuration graph of our constructed PDA which starts in the initial configuration and ends in a stable state, there exist at least one path $\pi_{\mathcal{D}}$ starting from J in the B-bounded configuration graph of the DMS. This can be proved by induction on the length of the paths. Further, in any database instance on a path $\pi_{\mathcal{D}}$, a proposition p occurs if, and only if, it also occurs in the corresponding stable state of $\mathcal{B}$ in the corresponding path $\pi_{\mathcal{B}}$. Thus, the target p is reachable in $\mathcal{D}$ if and only if the state q_F is reachable in $\mathcal{B}$.

Complexity: The PDA we construct is of size $n2^{O(|\mathcal{R}|(mB)^{\text{amax}})}$ where m is the maximum number of variables used in any action, n is the number of actions in the DMS and amax is the maximum arity of any relation in the schema. Control state reachability in pushdown systems is in linear time, and hence recency bounded propositional reachability can be done in $n2^{O(|\mathcal{R}|(mB)^{\text{amax}})}$. This is exponential time assuming amax is a constant. Detailed complexity analysis can be found in Appendix C.

This justifies the ExpTime upper bound stated in Theorem 4.1. The lower bound is proved in Appendix B by a reduction from the language intersection non-emptiness problem of a pushdown automaton and n finite state automata.

[1] Note that since we are doing under-approximate verification, it will only increase coverage and not change the ExpTime upper bound we obtain.

8 RELATED WORK

Formal verification is very tempting due to its unmatched precision and the guarantee of correctness. Recent years have seen attempts to bring formal methods to the verification of business processes. This has resulted in a body of research papers throwing theoretical insight to a practical tool [23, 24] for data-centric work flow. Different formalisms to model database driven systems have been proposed in the literature.

The line of research followed in [10, 20, 26] assumes the system can start from an arbitrary initial database instance and evolve, as long as the instances agree to a constraint. This framework can now-a-days support rich features such as numerical data, task creation, hierarchical structure and limited concurrency. By introducing appropriate restrictions they get decidability for verification, which is ready to be put in practice [23].

In another line of research, in the works of [5, 6, 14, 18] the system starts from a fixed initial database and the actions result in bulk operations that add/delete an unbounded number of facts at a time. Two versions of bounded reachability has been studied in [5, 7] where the length of the run and size of the active domain are bounded respectively. These bounding strategies are an under-approximation, but they only admit a bounded number of distinct database instances modulo renaming.

The closest related work is indeed [3]. In [3] we showed that model checking of a weaker class of DMS which does not allow numerical values against a very expressive specification language (MSO-FO) is decidable. MSO-FO can relate data objects at different instances using global quantifiers. The proof idea was to encode the "data equality" using "same origin" in MSO, which crucially depended on the assumption on history-freshness – a newly inserted value must be different from any value ever used in the history of a run. Thus, it allowed to "interpret" the runs in nested words (analogous to tree-interpretation) on which MSO is decidable.

Our model, on the other hand, allows numerical values to be repeated freely, not restricted by freshness or recency. In fact we treat numerical values differently from the object of the data domain precisely to give it this freedom, which is crucial for most modelling applications. In the presence of non-fresh objects the proof from [3] fails, as the interpretation is not any more a nested word, but a data - nested word (analogous to "data-tree" [8]). This would make model checking against LTL undecidable (cf. Appendix A).

Hence we develop new proof techniques to handle this case by means of symbolic representation by layers and we capture the dynamics by a pushdown automaton. With this we could relax the restriction of history-freshness on data objects to local-freshness. Further the notion of recency is relaxed to recently *accessed* as opposed to recently *added*. Also, multiple elements are allowed to have the same recency. This increases the coverage of the under-approximation.

Pushdown systems with numerical values have been studied extensively in the literature [1, 2, 11, 16] but we have a much richer structure because we have to deal with unbounded database instances.

Formalisms for database-driven dynamic systems such as DMS are very similar, on the surface, to those used in planning and reasoning about actions (such as STRIPS and ADL). All such formalisms employ a first-order language (and fragments thereof) to capture the executability of actions and the updates they induce. In addition, the existence of a plan that brings the system from an initial to a target state can be formulated as a reachability problem. There is, however, a fundamental difference: planning and action languages are usually interpreted over a finite object domain, so that they simply provide a compact way for representing propositional action theories. To the best of our knowledge, the few approaches that attack planning problems over infinite object domains (such as [15, 22]) introduce conditions towards decidability that are different from those studied here, and that in fact make the resulting system *essentially finite state*.

9 CONCLUSIONS AND FUTURE WORK

We have lifted the framework of DMS [3] to handle elements from a dense domain with an implicit total order between them. We use an elegant symbolic encoding to show that the reachability problem under bounded recency restriction can be decided in ExpTime. We also give a matching lower-bound. We further show that going beyond reachability to model checking against liner-time properties is undecidable.

There are several directions for future research. An immediate step would be to see how the decidability/complexity varies if we allow full first-order guards in the actions.

It would be interesting to see which other interpreted data and relations can still give us decidable verification. In particular, whether it would be possible to extend the decidability results to the discrete domain. Another direction would be to use time as interpreted data and allow guards with timing constraints. Timing constraints are crucial for many database-driven transactions such as in banking domains or for security purposes. It is worth exploring the status of decidability and complexity in these settings. Another question is to see whether we could go beyond relations to interpreted functions such as arithmetic on numerical values.

Acknowledgments

C. Aiswarya has been partially supported by DST Inspire. Marco Montali has been partially supported by the UNIBZ projects REKAP and DACOMAN.

REFERENCES

[1] Parosh Aziz Abdulla, C. Aiswarya, and Mohamed Faouzi Atig. 2017. Data Multi-Pushdown Automata. In *CONCUR (LIPIcs)*, Vol. 85. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 38:1–38:17.

[2] P. A. Abdulla, M. F. Atig, G. Delzanno, and A. Podelski. 2013. Push-Down Automata with Gap-Order Constraints. In *FSEN (LNCS)*, Vol. 8161. Springer, 199–216.

[3] Parosh Aziz Abdulla, C. Aiswarya, Mohamed Faouzi Atig, Marco Montali, and Othmane Rezine. 2016. Recency-Bounded Verification of Dynamic Database-Driven Systems. In *PODS*. ACM, 195–210.

[4] Parosh Aziz Abdulla, C. Aiswarya, Mohamed Faouzi Atig, Marco Montali, and Othmane Rezine. 2018. Complexity of Reachability for Data-Aware Dynamic Systems. In *ACSD*. IEEE Computer Society, 11–20.

[5] Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Alin Deutsch, and Marco Montali. 2013. Verification of Relational Data-Centric Dynamic Systems with External Services. In *PODS*. ACM Press.

[6] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. 2012. Verification of GSM-Based Artifact-Centric Systems through Finite Abstraction. In *ICSOC (LNCS)*, Vol. 7636. Springer, 17–31.

[7] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. 2014. Verification of Agent-Based Artifact Systems. *J. Artif. Intell. Res.* 51 (2014), 333–376.

[8] Mikolaj Boja'nczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. 2009. Two-variable logic on data trees and XML reasoning. *J. ACM* 56, 3 (2009), 13:1–13:48. https://doi.org/10.1145/1516512.1516515

[9] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. 2006. Two-Variable Logic on Words with Data. In *LICS*. 7–16.

[10] Mikolaj Bojanczyk, Luc Segoufin, and Szymon Torunczyk. 2013. Verification of Database-Driven Systems via Amalgamation. In *PODS*.

[11] B. Bollig, A. Cyriac, P. Gastin, and K. Narayan Kumar. 2012. Model Checking Languages of Data Words. In *FoSSaCS'12 (LNCS)*, Vol. 7213. Springer, 391–405.

[12] Diego Calvanese, Giuseppe De Giacomo, and Marco Montali. 2013. Foundations of Data-Aware Process Analysis: A Database Theory Perspective. In *PODS*. ACM Press.

[13] Diego Calvanese, Giuseppe De Giacomo, Marco Montali, and Fabio Patrizi. 2018. First-order μ-calculus over generic transition systems and applications to the situation calculus. *Inf. Comput.* 259, 3 (2018), 328–347.

[14] Diego Calvanese, Giorgio Delzanno, and Marco Montali. 2015. Verification of Relational Multiagent Systems with Data Types. In *AAAI*. AAAIP.

[15] Diego Calvanese, Marco Montali, Fabio Patrizi, and Michele Stawowy. 2016. Plan Synthesis for Knowledge and Action Bases. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, Subbarao Kambhampati (Ed.). IJCAI/AAAI Press, 1022–1029. http://www.ijcai.org/Abstract/16/149

[16] Lorenzo Clemente and Slawomir Lasota. 2015. Reachability Analysis of First-order Definable Pushdown Systems. In *CSL 2015, (LIPIcs)*, Vol. 41. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 244–259.

[17] E. Damaggio, A. Deutsch, and V. Vianu. 2011. Artifact Systems with Data Dependencies and Arithmetic. In *ICDT*.

[18] Giuseppe De Giacomo, Riccardo De Masellis, and Riccardo Rosati. 2012. Verification of Conjunctive Artifact-Centric Services. *Int. J. Cooperative Inf. Syst.* 21, 2 (2012), 111–140. https://doi.org/10.1142/S0218843012500025

[19] Stéphane Demri and Ranko Lazić. 2009. LTL with the freeze quantifier and register automata. *ACM TOCL* 10, 3 (2009).

[20] Alin Deutsch, Richard Hull, Fabio Patrizi, and Victor Vianu. 2009. Automatic verification of data-centric business processes. In *ICDT*.

[21] A. Deutsch, Y. Li, and V. Vianu. 2016. Verification of Hierarchical Artifact Systems. In *Proc. of PODS*. ACM Press, 179–194.

[22] Jörg Hoffmann, Piergiorgio Bertoli, Malte Helmert, and Marco Pistore. 2009. Message-Based Web Service Composition, Integrity Constraints, and Planning under Uncertainty: A New Connection. *J. Artif. Intell. Res.* 35 (2009), 49–117. https://doi.org/10.1613/jair.2716

[23] Yuliang Li, Alin Deutsch, and Victor Vianu. 2017. VERIFAS: A Practical Verifier for Artifact Systems. *Proc. VLDB Endow.* 11, 3 (Nov. 2017), 283–296.

[24] Riccardo De Masellis, Chiara Di Francescomarino, Chiara Ghidini, Marco Montali, and Sergio Tessaris. 2017. Add Data into Business Process Verification: Bridging the Gap between Theory and Practice. In *AAAI*.

[25] Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *TACAS*. Springer Berlin Heidelberg, 93–107.

[26] Victor Vianu. 2009. Automatic Verification of Database-Driven Systems: a New Frontier. In *Proc. of ICDT*. 1–13.

A LTL-FO MODEL CHECKING IS UNDECIDABLE

LTL-FO has been considered the apt temporal logic for evolving database driven systems. The reduction we present here will go through for a fragment of LTL-FO called LTL-UCQ where atomic formula can only check for the presence of a fact in the database instance.

The syntax of LTL-UCQ is given by:

$$\psi := R(u_1, \dots u_a) \mid \#(u) \leq \#(v) \mid \psi \wedge \psi \mid \neg\psi \mid \mathsf{X}\psi \mid \mathsf{F}\psi \mid \mathsf{P}\psi \mid \psi \mathsf{U} \psi \mid \exists u \psi$$

where R/a is in the schema.

A formula ψ is evaluated on a sequence $\rho = I_0, I_1, I_2 \dots$ of database instances and an index $i \in \mathbb{N}$ together with a valuation $\nu : \mathsf{free}(\psi) \to \textsc{Adom}(\rho)$ of the free variables to the elements of the active domain of the database instance at i where $\textsc{Adom}(\rho) = \bigcup_i \textsc{Adom}(I_i)$. The semantics is given below:

$$\rho, i, \nu \models R(u_1, \dots u_a) \quad \text{if } \langle \nu(u_1), \dots \nu(u_a) \rangle \in R^{I_i}$$

$$\rho, i, \nu \models \#(u) \leq \#(v) \quad \text{if } \#(\nu(u)) \leq \#(\nu(v))$$

$$\rho, i, \nu \models \exists u \psi \quad \text{if } \exists\, e \in \textsc{Adom}(I_i) \text{ such that } I_i, \nu' \models \psi$$

$$\text{where } \nu'(v) = \begin{cases} \nu(v) & \text{if } v \neq u \\ e & \text{if } v = u \end{cases}$$

The semantics of the standard temporal operators are as expected.

The recency bounded model checking problem asks, given a DMS S, and initial database instance I_0, a recency bound b and an LTL-UCQ sentence ψ, do we have $\rho, 0 \models \psi$ for all recency b - bounded runs $\rho = I_0, I_1, I_2, \dots$ of S ?

Theorem A.1. *Recency bounded model checking problem of DMS against LTL-UCQ is undecidable.*

We will prove the undecidability of the model checking problem by demonstrating a reduction from the control state reachability problem in a two-counter machine (Minsky machines).

Two-counter machines. A two counter machine (2CM) is a tuple (States, init, Trans) where States is the finite set of states, init $\in$ States is the initial state, and Trans $\subseteq$ States $\times$ Ops $\times$ States is the set of transitions with $Ops = \{\mathtt{inc}_1, \mathtt{dec}_1, \mathtt{ifz}_1, \mathtt{inc}_2, \mathtt{dec}_2, \mathtt{ifz}_2\}$ denoting increment, decrement and zero-test on counter 1 and 2. We may assume, without loss of generality, that there are no transitions of the form (q, op, init).

The semantics is given by an infinite configuration graph whose set of nodes is States $\times \mathbb{N} \times \mathbb{N}$. A configuration (q, n_1, n_2) denotes that the machine is in state q, counter 1 has value n_1 and counter 2 has value n_2. We write $(q, n_1, n_2) \rightarrow (q', n_1', n_2')$ if one of the following conditions holds:

(1) $(q, \mathtt{inc}_1, q') \in Trans$, $n_1' = 1 + n_1$ and $n_2' = n_2$
(2) $(q, \mathtt{inc}_2, q') \in Trans$, $n_1' = n_1$ and $n_2' = 1 + n_2$
(3) $(q, \mathtt{dec}_1, q') \in Trans$, $n_1 > 0$, $n_1' = n_1 - 1$ and $n_2' = n_2$
(4) $(q, \mathtt{dec}_2, q') \in Trans$, $n_2 > 0$, $n_1' = n_1$ and $n_2' = n_2 - 1$
(5) $(q, \mathtt{ifz}_1, q') \in Trans$, $n_1' = n_1 = 0$ and $n_2' = n_2$
(6) $(q, \mathtt{ifz}_1, q') \in Trans$, $n_1' = n_1$ and $n_2' = n_2 = 0$

We denote the reflexive-transitive closure of $\rightarrow$ by $\rightarrow^*$.

The control-state reachability problem for two counter machines asks, given a 2CMand a target state $q_{\text{target}} \in$ States, do we have $(\mathsf{init}, 0, 0) \rightarrow^* (q_{\text{target}}, n_1, n_2)$ for some $n_1, n_2 \in \mathbb{N}$? The problem is well known to be undecidable.

Construction of the DMS. Our schema consists of a unary relations curr and propositions denoting state and the current operation. That is schema $\mathcal{R} = \{P_q/0 \mid q \in \text{States}\} \cup \{P_{op}/0 \mid op \in \text{Ops}\} \cup \{\mathtt{curr}/1\}$. A new element gets added in every action, and will be deleted in the subsequent one. Thus we maintain an invariant that there will be exactly one element in the active domain always (except for the very initial database instance). Corresponding to every transition $t = (q, op, q') \in$ Trans we have an action α_t with

- $I_+ = \{P_q, \mathtt{curr}(x)\}$
- DEL $= \{P_q, \mathtt{curr}(x), \{P_{op'}\}_{op' \in \text{Ops}}\}$
- ADD $= \{P_{q'}, \mathtt{curr}(y), P_{op}\}$

We need an extra action α_{init} to handle the very first transition, as at that time the active domain is empty.

- $I_+ = \{P_{\mathsf{init}}\}$
- DEL $= \{P_{\mathsf{init}}, \{P_{op'}\}_{op' \in \text{Ops}}\}$
- ADD $= \{P_{q'}, \mathtt{curr}(y), P_{op}\}$

Notice that the assumption of having no transitions with init as the target state is important to maintain the invariant of only one element in the active domain.

The DMS would run from the initial database instance (I_0) consisting of an empty active domain and a single proposition P_{init} being true. It also maintains an invariant that an any instance apart from I_0, exactly one proposition corresponding to the state, and one proposition corresponding to the type of the operation is true. The element added to the active domain keeps an identity of the operation which we will exploit using LTL for capturing faithful simulations.

LTL-UCQ formula characterising faithful simulation of 2CM. Notice that the DMS constructed could allow any sequence of actions as far the control states permit them. But these sequence of actions can take counters to negative values and zero-tests to be performed when the counter is not zero. We will rule out these spurious runs using the LTL-FO formulae. Our formula for faithful simulation (ψ_{faithful}) will be a conjunction of the following:

(1) $\neg(\mathsf{F}(\exists u \ (\mathsf{F} \ \exists v \ (\#u = \#v \wedge \mathsf{F}(\exists u \ \ \#u = \#v)) \ ) \ ))$
There must not be three elements with the same numerical value.
(2) $\mathsf{G}\ (\exists u \ \ \mathtt{dec}_1 \wedge \mathtt{curr}(u) \Rightarrow \mathsf{P}\ (\exists v \ \ \mathtt{inc}_1 \wedge \mathtt{curr}(v) \wedge \#u = \#v))$
Every decrement on counter 1 must have a corresponding increment on counter 1 before.
(3) $\mathsf{G}(\exists u \ \ \mathtt{dec}_2 \wedge \mathtt{curr}(u) \Rightarrow \mathsf{P}(\exists v \ \ \mathtt{inc}_2 \wedge \mathtt{curr}(v) \wedge \#u = \#v))$
Similarly for counter 2.
(4) $\mathsf{G}(\exists u \ \ \mathtt{inc}_1 \wedge \mathtt{curr}(u) \Rightarrow \neg \ \mathtt{ifz}_1 \mathsf{U}(\exists v \ \ \mathtt{dec}_1 \wedge \mathtt{curr}(v) \wedge \#u = \#v)$
Every increment on counter 1 must be decremented before a zero test on counter 1.
(5) $\mathsf{G}(\exists u \ \ \mathtt{inc}_2 \ \wedge \ \mathtt{curr}(u) \ \ \Rightarrow \ \ (\neg \ \mathtt{ifz}_1) \ \mathsf{U} \ (\exists v \ \mathtt{dec}_2 \ \wedge \ \mathtt{curr}(v) \wedge \#u = \#v)$
Similarly for counter 2.

Finally to check if a faithful run reaches the target state, we want to check if the formula $\psi_{\text{faithful}} \wedge \mathsf{F} q_{\text{target}}$ is satisfied by at least one run of the DMS. Since model checking asks whether all runs satisfy the given formula, we tweak the formula to $\psi_{\text{faithful}} \implies \mathsf{G} \neg q_{\text{target}}$. The model checking answers *no* if an only if the q_{target} is reachable by a faithful simulation (same as, q_{target} is reachable in 2CM). Thus the model-checking against LTL is undecidable.

Remark. Alternate proof: With recency bound 1, we can simulate a generic automaton generating all data words [9, 19]. The recency is applied only to objects, their values are completely free (to repeat or to be fresh). Using LTL-UCQ we can express Freeze-LTL [19] on datawords, for which it is undecidable to check satisfiability.

B LOWER BOUND

In this section we show that the recency bounded reachability problem for DMS is ExpTime-hard, already with a recency bound $B = 1$ starting from an initial database having a singleton active domain. Further our schema contains only unary relations, our actions employ only simple UCQN guards, and *do not* involve numerical constraints.

We will give a reduction from the intersection non-emptiness problem of the languages of a pushdown automaton and m finite state automata. The input is a pushdown automaton $\mathcal{A}_0$ and m finite state automata $\mathcal{A}_1, \ldots \mathcal{A}_m$, over the same finite alphabet Σ. The question is whether there exists a word $w \in \Sigma^*$ such that w is accepted by all $\mathcal{A}_i$.

Let the pushdown automaton $\mathcal{A}_0 = \langle \mathsf{States}_0, \Gamma, \mathsf{Trans}_0, \mathsf{init}_0, \mathsf{Acc}_0 \rangle$. We have $\mathsf{Trans}_0 \subseteq \mathsf{States}_0 \times \Sigma \times \Gamma \times \Gamma^* \times \mathsf{States}_0$, indicating the source state, the letter being read, the top of the stack symbol being popped, the sequence of symbols to be pushed onto the stack, and the target state. We define the set $\mathit{ToPush} \subseteq \Gamma^*$ to be the suffixes of the sequence of strings to be pushed to the stack in any transition of $\mathcal{A}_0$. That is, $\mathit{ToPush} = \{\alpha \mid \alpha \in \Gamma^* \text{ and } \exists (p, a, \gamma, \beta\alpha, p') \in \mathsf{Trans}_0$ for some $p, p' \in \mathsf{States}_0, a \in \Sigma$, and $\beta \in \Gamma^*\}$. Let the finite state automata be $\mathcal{A}_i = \langle \mathsf{States}_i, \mathsf{Trans}_i, \mathsf{init}_i, \mathsf{Acc}_i \rangle$ for $i : 1 \leq i \leq m$. Without loss of generality, we assume that the set of states States_i are disjoint for all $i : 0 \leq i \leq m$. We also assume that $|\mathsf{Acc}_i| = 1$, by letting the automata to be non-deterministic.

We will construct a DMS over a schema consisting of only unary relations and nullary relations (propositions). Our schema will contain:

- a unary relation U_γ for each stack symbol $\gamma \in \Gamma$,
- a proposition P_s for each state $s \in \bigcup_i \mathsf{States}_i$,
- a proposition P_a for each letter $a \in \Sigma$,
- a proposition $P_{\mathcal{A}_i}$ for each automaton $\mathcal{A}_i, i : 0 \leq i \leq m$
- a proposition P_α, for each $\alpha \in \mathit{ToPush}$ and
- two special values $P_\$$ and P_{Acc}.

That is, the schema $\mathcal{R} = \{U_\gamma/1 \mid \gamma \in \Gamma\} \cup \{P_s/0 \mid s \in \bigcup_i \mathsf{States}_i\} \cup \{P_a/0 \mid a \in \Sigma\} \cup \{P_{\mathcal{A}_i}/0 \mid 1 \leq i \leq m\} \cup \{P_\alpha/0 \mid \alpha \in \mathit{ToPush}\} \cup \{P_\$/0, P_{\mathsf{Acc}}/0\}$.

We will design the DMS to simulate a run of each of the automata on the same word, in synchronous rounds. In a round, every automaton will perform a transition and all of them will agree on the letter being read. A round is realized by a sequence of actions which simulate each automata in a round-robin fashion.

As an invariant we maintain that, there will always be exactly one proposition from $\{P_s/0 \mid s \in \mathsf{States}_i\}$ which is set to true, for each i. This will represent a global state, if we were doing a product construction of the automata.

Further, throughout a round exactly one and the same proposition from the set $\{P_a/0 \mid a \in \Sigma\}$ will be set to true. At the very beginning of a round, no proposition from the set $\{P_a/0 \mid a \in \Sigma\}$ is set to true. This enables us to simulate a synchronous transition in the product automata using a round.

Another invariant would be that every element present in the active domain will be present in exactly one unary relation from $\gamma \in \Gamma$. Thus the active domain along with the recency information represents a stack which is needed to simulate the run of the pushdown automaton $\mathcal{A}_0$.

At the beginning of a round the control will be given to the pushdown automaton $\mathcal{A}_0$, by setting $P_{\mathcal{A}_0}$ to true and every other $P_{\mathcal{A}_i}$ $(i \neq 0)$ would be false. The DMS would assert that no P_a is set to true, and simulate a transition of the pushdown automaton (it may take more than one action, we describe this below) and eventually give the control to the finite state automaton $\mathcal{A}_1$. The proposition P_a set to true by $\mathcal{A}_0$ would continue to remain true, and a consistent transition of $\mathcal{A}_1$ labelled by letter a would be simulated. Then the control will be given to $\mathcal{A}_2$ and so on. Finally once a transition of $\mathcal{A}_m$ is simulated the proposition P_a is set to false and DMS moves to a special stage intended to check for acceptance. This is marked by the special proposition $P_\$$. In this stage, an action will check if all automata are in Acc_i and if so, will set the target proposition P_{Acc} true, and else will set $P_{\mathcal{A}_0}$ to true in order to start the next round.

Formally, the DMS will have the following actions.

(1) $\alpha_{\text{Accept-yes}}$
- $I_+ = \{P_\$, \{P_{\mathsf{Acc}_i}\}_i\}$, $I_- = \emptyset$
- $\textsc{Add} = \{P_{\mathsf{Acc}}\}$, $\textsc{Del} = \emptyset$

An accepting configuration sets the target proposition P_{Acc} true.

(2) $\alpha_{\text{Accept-no-i}}$
- $I_+ = \{P_\$\}$, $I_- = \{P_{\mathsf{Acc}_i}\}$
- $\textsc{Del} = \{P_\$\}$, $\textsc{Add} = \{P_{\mathcal{A}_0}\}$

If not accepting as witnessed by Automaton $\mathcal{A}_i$, start the next round by giving the control to $\mathcal{A}_0$. We have one such transition for each i.

(3) α_t
- $I_+ = \{P_{\mathcal{A}_0}, P_q, U_\gamma(x)\}$, $I_- = \{\{P_a\}_{a \in \Sigma}\}$
- $\textsc{Del} = \{P_q, U_\gamma(x)\}$, $\textsc{Add} = \{P_{q'}, P_a, P_\alpha\}$

if $t = (q, a, \gamma, \alpha, q') \in \mathsf{Trans}_0$
Checks if the transition is enabled, if yes sets the letter, updates the state, pops the top of the stack and remembers to push α to the stack by simulation. The control still is with $\mathcal{A}_0$ as $P_{\mathcal{A}_0}$ is still true.

(4) $\alpha_{\gamma\alpha}$
- $I_+ = \{P_{\mathcal{A}_0}, P_{\gamma\alpha}\}$, $I_- = \emptyset$
- $\textsc{Del} = \{P_{\gamma\alpha}\}$, $\textsc{Add} = \{U_\gamma(y), P_\alpha\}$

if $\gamma\alpha \in \mathit{ToPush}$

If $\gamma\alpha$ needs to be pushed to the stack, it first inserts an element to the unary relation U_γ and remembers that it has to still push α.

(5) α_ϵ
- $I_+ = \{P_{\mathcal{A}_0}, P_\epsilon\}, \quad I_- = \emptyset$
- $\textsc{Del} = \{P_\epsilon, P_{\mathcal{A}_0}\}$, $\textsc{Add} = \{P_{\mathcal{A}_1}\}$

If no more symbols needs to be pushed into the stack, then the control is given to $\mathcal{A}_1$.

(6) α_t
- $I_+ = \{P_{\mathcal{A}_i}, P_q, P_a\}, \quad I_- = \emptyset$
- $\textsc{Del} = \{P_{\mathcal{A}_i}, P_q\}$, $\textsc{Add} = \{P_{q'}, P_{\mathcal{A}_{i+1}}\}$

if $t = (q, a, q') \in \mathsf{Trans}_i$. We have one such transition for each $i : 1 \le i \le m-1$.
Perform a transition labelled by the chosen letter and give control to the next automaton.

(7) α_t
- $I_+ = \{P_{\mathcal{A}_m}, P_q, P_a\}, \quad I_- = \emptyset$
- $\textsc{Del} = \{P_{\mathcal{A}_m}, P_q, P_a\}$, $\textsc{Add} = \{P_{q'}, P_{\mathcal{A}_\$}\}$

if $t = (q, a, q') \in \mathsf{Trans}_m$.
Once the round is finished, un-set the letter, and set the special proposition $P_\$$ (which will in turn check if the configuration is accepting, cf. Action in Item (1)).

Initially, the propositions P_{init_i} and the special proposition $P_\$$ are set to true, and all other propositions are set to false. Further all unary relations are empty except $U_\perp$ which contains one element, to mark the bottom of the stack. This defines I_0. The recency bound is 1. The target proposition is P_{Acc}.

Notice that I_0 ensures that all automata are in their respective initial states initially, and that the stack only contains the stack bottom symbol. A run of recency bound 1 will make sure that last-in-first-out policy of stacks is maintained on the elements of active domain along the run. Finally, if P_{Acc} is reachable then the automata have a non-empty intersection (the word whose length is the number of rounds in the run of the DMS, and the ith letter being the letter chosen at the ith round). Conversely, if the the automata have a non-empty intersection then there will be a run of the DMS which is recency bounded by 1.

Notice that the size of the schema is linear in the size of the input. The size of the DMS is polynomial in the size of the input. Thus we have a polynomial time reduction from the ExpTime complete problem to the bounded recency reachability problem thus establishing the ExpTime hardness of the latter.

C COMPLEXITY ANALYSIS

Complexity of our decision procedure will depend on the following parameters:

- m: the maximum number of variables used in any action
- amax: the maximum of the arities of the relations in the schema. That is, amax $= \max\{a \mid R/a \in \mathcal{R}\}$
- $|\mathcal{R}|$: number of relations in the schema
- B: the bound on recency and
- n: the number of actions in the DMS.

First we notice that the maximum number of elements that can be in a band (all having same value for rcy) is bounded by m. Next, we can observe that in a layer, the set A contains only the first $B+2$ bands. Hence the number of elements in A is bounded by $m \cdot (B+2)$. The sets D and M are also similar, in the worst case we need $3 \cdot m \cdot (B+2)$ objects to be distributed across these. We can reuse the names for a different layer as long the conditions are satisfied. Hence we can establish a bound on the size of O. Let $N = |O| = 3 \cdot m \cdot (B+2)$.

Now, keeping N as a loose upper bound, the number of different A, D, M possible are all 2^N. Number of different rcy is bounded by $(B+2)^N$. Number of different g is bounded by N^N. The number of different database instance over O is bounded by $2^{|\mathcal{R}| \cdot N^{\text{amax}}}$. Hence the number of different layers possible is the product of these numbers, which is $2^{O(|\mathcal{R}| N^{\text{amax}})}$ or equivalently $2^{O(|\mathcal{R}|(mB)^{\text{amax}})}$.

Thus the number of states of the constructed PDA, as well as the stack alphabet be bounded by the same number. The number of transitions of the PDA is $O(n)$ where n is the number of actions in the DMS. Further the construction takes time $2^{O(|\mathcal{R}|(mB)^{\text{amax}})}$.

Finally reachability in pushdown systems is in linear time, and hence recency bounded propositional reachability can be done in $2^{O(|\mathcal{R}|(mB)^{\text{amax}})}$. Notice that this is exponential time assuming amax is a constant.

Remark. We will assume that the number of elements in the active domain of the initial database $|\textsc{Adom}(I_0)|$ is smaller than the recency bound. Otherwise, in the complexity analysis above we need to replace B by $\max\{|\textsc{Adom}(I_0)|, B\}$.