DEDALUS: Datalog in Time and Space

Peter Alvaro* William R. Marczak* Neil Conway*
Joseph M. Hellerstein* David Maier† Russell Sears*

*University of California, Berkeley †Portland State University

{palvaro,wrm,nrc,hellerstein,sears}@cs.berkeley.edu maier@cs.pdx.edu

ABSTRACT

Recent research has explored using Datalog-based languages to express a distributed system as a set of logical invariants [2, 21]. Two properties of distributed systems proved difficult to model in Datalog. First, the state of any such system evolves with its execution. Second, deductions in these systems may be arbitrarily delayed, dropped, or reordered by the unreliable network links they must traverse. Previous efforts addressed the former by extending Datalog to include updates, key constraints, persistence and events, and the latter by assuming ordered and reliable delivery while ignoring delay. These details have a semantics outside Datalog, which increases the complexity of the language or its interpretation, and forces programmers to think operationally. We argue that the missing component from these previous languages is a notion of *time*.

In this paper we present **DEDALUS**, a foundation language for programming and reasoning about distributed systems. DEDALUS reduces to a subset of Datalog [33] with negation, aggregate functions, successor and choice, and admits an explicit representation of time into the logic language. We show that DEDALUS provides a declarative foundation for the two signature features of distributed systems: mutable state, and asynchronous processing and communication. Given these two features, we address three important properties of programs in a domain-specific manner: a notion of *safety* appropriate to non-terminating computations, *stratified* monotonic reasoning with negation over time, and efficient evaluation over time via a simple execution strategy. We also provide conservative syntactic checks for our temporal notions of safety and stratification. Our experience implementing full-featured systems in variants of Datalog suggests that DEDALUS is well-suited to the specification of rich distributed services and protocols, and provides both cleaner semantics and richer tests of correctness.

1. INTRODUCTION

In recent years, there has been a resurgence of interest in Datalog as the foundation for applied, domain-specific languages in a wide variety of areas, including networking [22], distributed systems [6, 9], natural language processing [12], robotics [5], compiler analysis [16], security [15, 19, 37] and computer games [36]. The resulting languages have been promoted for their compact and natural representations of tasks in their respective domains, in many cases leading to code that is orders of magnitude shorter than equivalent imperative programs. Another stated advantage of these languages has been the ability to directly capture intuitive specifications of protocols and programs as executable code.

While most of these efforts were intended to be "declarative" languages, many chose to extend Datalog with operational features natural to their application domain. These operational aspects, though familiar, limit the ability of the language designers to leverage the rich literature on Datalog: program checks like safety and stratifiability, and optimizations like magic sets and materialized recursive view maintenance. In addition, in many of these languages the blend of operational and declarative constructs leads to semantic ambiguities. This is of particular interest to us in the context of networking and other distributed systems, both because we have considerable practical experience with these languages [2, 22], and because others have examined the semantic ambiguities of these languages in some depth [26, 29].

In this paper we reconsider declarative programming for distributed systems from a model-theoretic perspective. We introduce a declarative language called DEDALUS[1] that enables the specification of rich distributed systems concepts without recourse to operational constructs. DEDALUS is a subset of a language with well-studied features: Datalog enhanced with negation, aggregate functions, choice, and a successor relation. DEDALUS provides a model-theoretic foundation for the two key features of distributed systems: mutable state, and asynchronous processing and communication. We show how these features are captured in DEDALUS via a natural incorporation of *time* as an attribute of Datalog predicates.

Given the ability to express programs with these two features, we address three important properties of DEDALUS programs: a temporal notion of *safety* appropriate to long-running services and protocols, *stratified* monotonic reasoning with negation over time, and efficient evaluation via a simple execution strategy. We also provide conservative syntactic checks for our temporal notions of safety and stratification.

We begin by defining DEDALUS$_0$, a restricted sublanguage of Datalog (Section 3). We show how DEDALUS$_0$ supports state update in Section 4, prove temporal safety and stratifiability properties of DEDALUS$_0$ in Section 5, and describe a simple, efficient evalua-

[1] DEDALUS is intended as a precursor language for **Bloom**, a high-level language for programming distributed systems that will replace Overlog in the **BOOM** project [2]. As such, it is derived from the character Stephen Dedalus in James Joyce's *Ulysses*, whose dense and precise chapters precede those of the novel's hero, Leopold Bloom. The character Dedalus, in turn, was partly derived from Daedalus, the greatest of the Greek engineers and father of Icarus. Unlike Overlog, which flew too close to the sun, Dedalus remains firmly grounded.

tion scheme (Section 6). Finally, we introduce DEDALUS by adding support for asynchrony to DEDALUS$_0$ in Section 7. Throughout, we demonstrate the expressivity and practical utility of our work with specific examples, including a number of building-block routines from classical distributed computing, such as sequences, queues, distributed clocks, and reliable broadcast. We also discuss the correspondence between DEDALUS and our prior work implementing full-featured distributed services in somewhat more operational Datalog variants [2, 22].

2. MOTIVATION

2.1 Distributed Systems and Logic Languages

Distributed logic languages promise to significantly raise the level of abstraction at which distributed systems are currently implemented, allowing programmers to specify systems as a set of invariants over local state, and rules that describe how state changes and moves across a network. This approach leads to succinct, executable specifications whose faithfulness to the original pseudocode may be visually verified. Moreover, the first-order logic foundations of such languages, in which programs are expressed as a collection of logical implications, lends itself (at least in principle) to powerful formal verification techniques [34, 35] and assertional reasoning [2].

Although the original work in declarative networking was aimed at routing protocols for which soft state, best-effort messaging and eventually consistent semantics are acceptable, it wasn't long before researchers began attempting to exploit distributed logic languages to implement nontrivial systems that enforce distributed invariants [23] or enforce transactional consistency or global ordering [3]. In order to achieve atomic semantics with regard to side-effecting operations like disk writes and messages, the fully pipelined operation of P2 was replaced with a *chain of fixpoints* semantics. All rules are expressed as straightforward Datalog, and evaluation proceeds in three phases:

1. Input from the external world, including network messages, clock interrupts and host language calls, is collected.
2. Time is frozen, the union of the local store and the batch of events is taken as EDB, and the program is run to fixpoint.
3. The deductions that cause side effects (e.g., updates, messages and host language callbacks) are dealt with.

This "framing" of atomic deduction with imperative constructs exposes more clearly the semantic distinction between the two extremes of *events*, ephemeral tuples that hold for exactly one timestep or fixpoint computation, *hard state* or tuples that persist in relations across timesteps, and the semantically ambiguous family of tuple types that exist between the two extremes.

Consider a distributed system in which participants announce their presence with *heartbeat* messages; participants that want to reason about who is present must remember the message content and the time of its arrival. It is natural to represent the messages themselves as events, and the log of messages and timestamps as a persistent table:

```
heartbeat_cache(@Host, Peer, Time) ←
  heartbeat(@Host, Peer), local_time(Time);
```

Although the Overlog language permits the transparent intermixing of the table types, their interaction is often unintuitive, and compromises the reading of rules as logical implications. The state of the system after the arrival of some number of *heartbeat* events is clearly not a minimal model of the given input database, which does not contain the events that (for one moment) caused the derivation of *heartbeat_cache* tuples.

2.2 Distributed Idioms

Despite their problematic logical interpretation, events and hard state tables are desirable primitives for implementing distributed systems. Network messages and timers are naturally represented as ephemeral tuples that are only instantaneously true, but which may be joined with persistent state to deduce new tuples. A notion of events makes it straightforward to represent many useful distributed programming idioms, including counters or *sequences* and *queues*, and the interaction of events and hard state allow the simulation of "soft state" useful for best-effort caching in network protocols and heartbeat messages in distributed systems. For example, an Overlog sequence might be specified in the following fashion:

```
seq(To, X+1) ←
  seq(To, X), ping(To, From);
```

The structure and intent of the rule above are fairly obvious: when a *ping* event occurs, the current value X of the sequence*seq* should be incremented. Upon closer inspection, however, we see that the rule has an unambiguous meaning only under an operational interpretation of the system's semantics: precisely because *ping* is ephemeral, we know that the arrival of a *ping* tuple will "trigger" the rule, and the insertion of the new sequence value will not retrigger it, causing an infinite chain of deductions. The queue pattern, which enables ordered processing of a set, is slightly more complicated:

```
r1
min_elmt(0) ←
  queue(0, _);
r2
deq(Host, X) ←
  queue(0, X), min_elmt(0),
  deq_request(Host);
r3
delete queue(0, X) ←
  queue(0, X), min_elmt(0),
  deq_request(_);
```

Again, the semantics are clear only because of the ephemerality of *deq_request*, and because it is either true or false – either an item is atomically dequeued and deleted, or neither thing happens. As in the sequence example, the operational interpretation of events as triggers is necessary to ensure that the rules do not represent a circular definition: the event always drives the *r2's* evaluation, and hence the result of *r3's* evaluation (a new min_elmt tuple) does not cause another evaluation of *r2*. In order to implement a construct that is ubiquitous in other languages, we found that we had to forfeit the logical reading of rules. It is no longer the case that if the premises hold the conclusions must hold: instead, the above examples must be read as ECA (event, condition, action) rules, and their conclusions after the event's disappearance (e.g., the value of the sequence or the contents of the queue at any arbitrary point in time) are essentially groundless facts.

notes – paa

all three are well-intentioned examples, but the logical readings of all three are hopeless and the semantics are implementation-dependent.

in P2, event relations are implemented as queues, and as such enforced a discipline that 1.) the queue must be the driving or 'delta' relation in any join plan, and 2.) zero or or one tuple are dequeued from the event at a given fixpoint. in JOL, this was relaxed and events, while true for only one fixpoint, are dequeued in batches, causing bugs in many P2 ports that relied on (2). the semantic ambiguities persisted, however.

finally, asynchrony. in the global program we imagine that there is a rule:

```
heartbeat(@Host, Peer, Time) ← master(@Peer, Host), timer(@P
```

but clearly we cannot read this as logical implication or conclude anything wrt the timing of its arrival at Host.

3. DEDALUS$_0$

We take as our foundation the language Datalog¬ [33]: Datalog enhanced with negated subgoals. We will be interested in the classes of statically stratifiable and modularly stratifiable [30] programs, which we revisit below. For conciseness, when we refer to "Datalog" below our intent is to admit negation — i.e., Datalog¬.

As a matter of notation, we refer to a countably infinite universe of constants C—in which $C_1, C_2, \ldots$ are representations of individual constants—and a countably infinite universe of variable symbols $\mathcal{A} = A_1, A_2, \ldots$. We will capture time in DEDALUS$_0$ via an infinite relation `successor` isomorphic to the successor relation on the integers; for convenience we will in fact refer to the domain $\mathbb{Z}$ when discussing time, though no specific interpretation of the symbols in $\mathbb{Z}$ is intended beyond the fact that `successor(x,y)` is true iff $y = x + 1$.

3.1 Syntactic Restrictions

DEDALUS$_0$ is a restricted sublanguage of Datalog. Specifically, we restrict the admissible schemata and the form of rules with the four constraints that follow.

Schema: We require that the final attribute of every DEDALUS$_0$ predicate range over the domain $\mathbb{Z}$. In a typical interpretation, DEDALUS$_0$ programs will use this final attribute to connote a "timestamp," so we refer to this attribute as the *time suffix* of the corresponding predicate.

Time Suffix: In a well-formed DEDALUS$_0$ rule, every subgoal uses the same existential variable $\mathcal{T}$ as its time suffix. A well-formed DEDALUS$_0$ rule must also have a time suffix $\mathcal{S}$ as its rightmost head attribute, which must be constrained in exactly one of the following two ways:

1. The rule is said to be *deductive* if $\mathcal{S}$ is bound to the value $\mathcal{T}$; that is, the body contains the subgoal $\mathcal{S} = \mathcal{T}$.
2. The rule is said to be *inductive* if $\mathcal{S}$ is the successor of $\mathcal{T}$; that is, the body contains the subgoal `successor(`$\mathcal{T}$`, `$\mathcal{S}$`)`.

In Section 7 when we consider DEDALUS—a superset of DEDALUS$_0$—we will introduce a third kind of rule to capture asynchrony.

EXAMPLE 1. *The following are examples of well-formed deductive and inductive rules, respectively.*
deductive: `p(A, B, S)` ← `e(A, B,` $\mathcal{T}$`),` $\mathcal{S} = \mathcal{T}$`;`

inductive: `q(A, B, S)` ← `e(A, B,` $\mathcal{T}$`), successor(`$\mathcal{T}$`, `$\mathcal{S}$`);`

Positive and Negative Predicates: For every extensional predicate `r` in a DEDALUS$_0$ program P, we add to P two distinguished predicates `r_pos` and `r_neg` with the same schema as `r`. We define `r_pos` using the following rule:

r_pos(A_1, A_2,[...],A_n,$\mathcal{S}$) ←
  r(A_1, A_2,[...],A_n,$\mathcal{T}$), S=$\mathcal{T}$;

That is, for every extensional predicate `r` there is an intensional predicate `r_pos` that contains at least the contents of `r`. Intuitively, this rule allows extensional facts to serve as ground for `r_pos`, while enabling other rules to derive additional `r_pos` facts.

The predicate `r_pos` may be referenced in the body or head of any DEDALUS$_0$ rule. We will make use of the predicate `r_neg` later to capture the notion of mutable state; we return to it in Section 4.2. Like `r_pos`, the use of `r_neg` in the heads and bodies of rules is unrestricted.

Guarded EDB: No well-formed DEDALUS$_0$ rule may involve any extensional predicate, except for a rule of the form above.

3.2 Abbreviated Syntax and Temporal Interpretation

We have been careful to define DEDALUS$_0$ as a subset of Datalog; this inclusion allows us to take advantage of Datalog's well-known semantics and the rich literature on the language.

DEDALUS$_0$ programs are intended to capture temporal semantics. For example, a fact, `p(`$C_1 \ldots C_n$`,` C_{n+1}`)`, with some constant C_{n+1} in its time suffix can be thought of as a fact that is true "at time C_{n+1}". Deductive rules can be seen as *instantaneous* statements: their deductions hold for predicates agreeing in the time suffix and describe what is true "for an instant" given what is known at that instant. Inductive rules are *temporal*—their consequents are defined to be true "at a different time" than their antecedents.

To simplify DEDALUS$_0$ notation for this typical interpretation, we introduce some syntactic "sugar" as follows:

- *Implicit time-suffixes in body predicates:* Since each body predicate of a well-formed rule has an existential variable $\mathcal{T}$ in its time suffix, we optionally omit the time suffix from each body predicate and treat it as implicit.
- *Temporal head annotation:* Since the time suffix in a head predicate may be either equal to $\mathcal{T}$, or equal to $\mathcal{T}$'s successor, we omit the time suffix from the head—and its relevant constraints from the body—and instead attach an identifier to the head predicate of each temporal rules, to indicate the change in time suffix. A temporal head predicate is of the form:

  `r(`A_1`,`A_2`,[...],`A_n`)@next`

  The identifier `@next` stands in for `successor(`$\mathcal{T}$`,S)` in the body.
- *Timestamped facts:* For notational consistency, we write the time suffix of facts (which must be given as a constant) outside the predicate. For example:

  `r(`A_1`,`A_2`,[...],`A_n`)@`C

EXAMPLE 2. *The following are "sugared" versions of deductive and inductive rules from Example 1, and a temporal fact:*
deductive: `p(A, B)` ← `e(A, B);`

inductive: `q(A, B)@next` ← `e(A, B);`

fact: `e(1, 2)@10;`

4. STATE IN LOGIC

> *Time is a device that was invented to keep everything from happening at once.*[2]

Given our definition of DEDALUS$_0$, we now address the persistence and mutability of data across time—one of the two signature features of distributed systems for which we provide a model-theoretic foundation.

The intuition behind DEDALUS$_0$'s `successor` relation is that it models the passage of (logical) time. In our discussion, we will say that ground atoms with lower time suffixes occur "before" atoms with higher ones. The constraints we imposed on DEDALUS$_0$ rules restrict how deductions may be made with respect to time. First,

[2]Graffiti on a wall at Cambridge University [1].

rules may only refer to a single time suffix variable in their body, and hence *cannot join across different "timesteps"*. Second, rules may specify deductions that occur concurrently with their ground facts or in the next timestep—in DEDALUS$_0$, we rule out induction "backwards" in time or "skipping" into the future.

This notion of time allows us to consider the contents of the EDB—and hence a minimal model of the IDB—with respect to an "instant in time": we simply bind the time suffixes ($\mathcal{T}$) of all body predicates to a constant. Because this produces a sequence of models (one per timestep), it gives us an intuitive and unambiguous way to declaratively express persistence and state changes across time. In this section, we give examples of language constructs that capture state-oriented motifs such as persistent relations, deletion and update, sequences, and queues.

4.1 Simple Persistence

A fact in predicate p at time $\mathcal{T}$ may provide ground for deductive rules at time $\mathcal{T}$, as well as ground for deductive rules in timesteps greater than $\mathcal{T}$, provided there exists a *simple persistence rule* of the form:

`p_pos(`A_1`,`A_2`,[...],`A_n`)@next ← p_pos(`A_1`,`A_2`,[...],`A_n`);`

A simple persistence rule of this form ensures that a p fact true at time i will be true $\forall j \in \mathbb{Z} : j >= i$.

4.2 Mutable State

To model deletions and updates of a fact, it is necessary to break the induction in a simple persistence rule. Adding a `p_neg` subgoal to the body of a simple persistence rule accomplishes this:

$\text{p\_pos}(A_1, A_2, [...], A_n)@\text{next} \leftarrow$
$\quad \text{p\_pos}(A_1, A_2, [...], A_n),$
$\quad \neg\ \text{p\_neg}(A_1, A_2, [...], A_n);$

If, at any time k, we have a fact `p_neg(`C_1`,`C_2`,[...],`C_n`)@k`, then we do not deduce a `p_pos(`C_1`,`C_2`,[...],`C_n`)@k+1` fact. By induction, we do not deduce a `p_pos(`C_1`,`C_2`,[...],`C_n`)@j` fact for any $j > k$, unless this `p_pos` fact is re-derived at some timestep $j > k$ by another rule. This corresponds to the intuition that a persistent fact, once stated, is true until it is retracted.

EXAMPLE 3. *Consider the following* DEDALUS$_0$ *program and ground facts:*

```
p_pos(A, B)@next ← p_pos(A, B), ¬p_neg(A, B);

p(1,2)@101;
p(1,3)@102;
p_neg(1,2)@300;
```

It is easy to see that the following facts are true: `p(1,2)@200`*,* `p(1,3)@200`*,* `p(1,3)@300`*. However,* `p(1,2)@301` *is false because it was "deleted" at timestep* `300`*.*

Since mutable persistence occurs frequently in practice, we provide the *persist* macro, which takes two arguments: a predicate name and its arity. The macro expands to the corresponding mutable persistence rule, and rewrites the current program in such a way that any references to the given predicate (say p) in rule bodies or heads are replaced by references to its positive relation (e.g. *p_pos*), except for rules in which the keyword *delete* appears before p in the head, which are replaced with *p_neg*. For example, the above `p_pos` persistence rule may be equivalently specified as `persist[p, 2]`.

Mutable persistence rules enable *updates*. For some time $\mathcal{T}$, an update is any pair of facts:

$\text{p\_neg}(C_1, C_2, [...], C_n)@\mathcal{T};$
$\text{p\_pos}(D_1, D_2, [...], D_n)@\mathcal{T} + 1;$

Intuitively, an update represents deleting an old value of a tuple and inserting a new value. Every update is *atomic across timesteps*, meaning that the old value ceases to exist at the same timestep in which the new value is derived—timestep $\mathcal{T} + 1$ in the above definition.

4.3 Sequences

One may represent a database sequence—an object that retains and monotonically increases a counter value—with a pair of inductive rules. One rule increments the current counter value when some condition is true, while the other persists the value of the sequence when the condition is false. We can capture the increase of the sequence value without using arithmetic, because the infinite series of `successor` has the monotonicity property we require:

```
seq(B)@next ← seq(A), successor(A,B), event(_);
seq(A)@next ← seq(A), ¬event(_);
```

Note that these two rules produce only a single value of `seq` at each timestep, but they do so in a manner slightly different than our standard persistence template.

4.4 Queues

While sequences are useful constructs for generating or imposing an ordering on tuples, programs will in some cases require that tuples are processed in a particular (partial) order associated with specific timesteps. To this end, we introduce a queue template, which employs inductive persistence and aggregate functions in rule heads to process tuples according to a data-dependent order, rather than as a set.

Aggregate functions simplify our discussion of queues. Mumick and Shmueli observe correspondences in the expressivity of Datalog with stratified negation and stratified aggregation functions [28]. Adding aggregation to our language does not affect its expressive power, but is useful for writing natural constructs for distributed computing including queues and ordering.

In DEDALUS$_0$ we will allow aggregate functions $\rho_1 - \rho_n$ to appear in the head of a deductive rule of the form:

$\text{p}(A_1, \ldots, A_n, \rho_1(A_{n+1}), \ldots, \rho_m(A_{n+m})) \leftarrow$
$\quad q_1\ (A_1, \ldots, A_n, A_{n+1}), \ldots, q_m(A_1, \ldots, A_n, A_{n+m});$

According to this rule, the predicate p contains one row for each satisfying assignment of $A_1, \ldots, A_n$ — akin to the distinct "groups" of SQL's "GROUP BY" notation.

Consider a predicate `priority_queue` that represents a series of tasks to be performed in some predefined order. Its attributes are a string representing a user, a job, and an integer indicating the priority of the job in the queue:

```
priority_queue('bob', 'bash', 200)@123;
priority_queue('eve', 'ls', 1)@123;
priority_queue('alice', 'ssh', 204)@123;
priority_queue('bob', 'ssh', 205)@123;
```

Observe that all the time suffixes are the same. Given this schema, we note that a program would likely want to process `priority_queue` events individually in a data-dependent order, in spite of their coincidence in logical time.

In the program below, we define a table `m_priority_queue` that serves as a queue to feed `priority_queue`. The queue must persist across timesteps because it may take multiple timesteps to drain it. At each timestep, for each value of **A**, a single tuple is projected into `priority_queue` and deleted (atomic with the projection) from `m_priority_queue`, changing the value of the aggregate calculated at the subsequent step:

```
persist[m_priority_queue, 3]

% find the min priority
omin(A, min<C>) ←
  m_priority_queue(A, _, C);

% feed p in the next step
% with the items of min priority
priority_queue(A, B, C)@next ←
  m_priority_queue(A, B, C),
  omin(A, C);

% delete from the next step
% those items of min priority
delete m_priority_queue(A, B, C) ←
  m_priority_queue(A, B, C),
  omin(A, C);
```

Under such a queueing discipline, deductive rules that depend on `priority_queue` are constrained to consider only min-priority tuples at each timestep per value of the variable **A**, thus implementing a per-user FIFO discipline. To enforce a global FIFO ordering over `priority_queue`, we may redefine `omin` and any dependent rules to exclude the **A** attribute.

A queue establishes a mapping between DEDALUS$_0$'s timesteps and the priority-ordering attribute of the input relation. By doing so, we take advantage of the monotonic property of timestamps to enforce an ordering property over our input that is otherwise very difficult to express in a logic language. We return to this idea in our discussion of temporal "entanglement" Section 7.5.2.

5. STRATIFICATION AND SAFETY

In the previous section we demonstrated that DEDALUS$_0$ can capture intuitive notions of persistence and mutability of state via a stylized use of Datalog. However, the alert reader will note that even very simple DEDALUS$_0$ programs make for unusual Datalog: among other concerns, persistence rules produce derivations for an infinite number of values of the time suffix. Traditional Datalog interpreters, which work against static databases, would attempt to enumerate these values, making this approach impractical.

However, in the context of distributed systems and networks, the need for non-terminating "services" or "protocols" is very common. In this section we show that expressing distributed systems properties such as persistence and mutable state in logic does not require dispensing with familiar notions of safety and stratification: we take traditional notions of acceptable Datalog programs, and extend them in a way that admits sensible non-terminating programs.

5.1 Stratification in DEDALUS$_0$

We first turn our attention to the semantics of programs with negation. As we will see, the inclusion of time introduces a "source of monotonicity" in programs that allows for clean minimal model semantics in some surprising cases, and enables purely syntactic monotonicity checks for a broad class of temporal programs.

LEMMA 1. *A* DEDALUS$_0$ *program without negation has a unique minimal model.*

PROOF. A DEDALUS$_0$ program without negation is a pure Datalog program. Every pure Datalog program has a unique minimal model. □

We define syntactic stratification of a DEDALUS$_0$ program the same way it is defined for a Datalog program:

Definition 1. A DEDALUS$_0$ program is *syntactically stratifiable* if there exists no cycle with a negative edge in the program's predicate dependency graph.

We may evaluate such a program in *stratum order* as described in the Datalog literature [33]. It is easy to see that any syntactically stratified DEDALUS$_0$ instance has a unique minimal model because it is a syntactically stratified Datalog program.

However, many programs we are interested in expressing are not syntactically stratifiable. Fortunately, we are able to define a syntactically checkable notion of *temporal stratifiability* of DEDALUS$_0$ programs that maps to a subset of *modularly stratifiable* [30] Datalog programs.

Definition 2. The *deductive reduction* of a DEDALUS$_0$ program P is the subset of P consisting of exactly the deductive rules in P.

Definition 3. A DEDALUS$_0$ program is *temporally stratifiable* if its deductive reduction is syntactically stratifiable.

LEMMA 2. *Any temporally stratifiable* DEDALUS$_0$ *instance P has a unique minimal model.*

EXAMPLE 4. *A simple temporally stratifiable* DEDALUS$_0$ *program that is not syntactically stratifiable.*

```
persist[p, 3]

r1
p(A, B, T) ←
  insert_p(A, B, T);

r2
delete p(A, B, T) ←
  p(A, B, T),
  delete_p(T);
```

In the DEDALUS$_0$ *program above,* insert_p *and* delete_p *are captured in EDB relations. This reasonable program is unstratifiable because $p \succ p\_neg \wedge p\_neg \succ p$. But because the successor relation is constrained such that $\forall A, B, successor(A, B) \rightarrow B > A$, any such program is modularly stratified on* successor. *Therefore, we have $p_n \not\succ^* p\_neg_n \not\succ^* p_{n+1}$; informally, earlier values do not depend on later values.*

5.2 Temporal Safety

Next we consider the issue of infinite results raised in the introduction to this section. In traditional Datalog, this is a well-studied concern. A Datalog program is considered *safe* if it has a finite minimal model, and hence has a finite execution. Safety in Datalog is traditionally ensured through the following syntactic constraints:

1. No functions are allowed.
2. Variables are *range restricted*: all attributes of the head goal appear in a non-negated body subgoal.
3. The EDB is finite.

These constraints ensure that the Herbrand Universe is finite: any atom that may be deduced by a safe program may only take its attributes from the set of all constant symbols appearing in the program and EDB. In fact, the set of all possible assignments of these constants to predicate attributes, representing every possible interpretation, is itself finite.

Since our definition of `successor` violates these rules, and indeed leads to an infinite set of facts, DEDALUS$_0$ programs violate this definition of safety. However, `successor` models time, not computation; later sections explain how DEDALUS implementations avoid enumerating the contents of `successor` at runtime. This section introduces a notion of termination that allows us to reason about the safety of DEDALUS$_0$ programs.

A DEDALUS$_0$ program containing only deductive rules is informally equivalent to a Datalog program in which all predicates have no time suffix. If all the rules in such a program meet the conditions above, then clearly we would like them to meet DEDALUS$_0$'s definition of safety.

Definition 4. A rule is *instantaneously safe* if it is deductive, function-free and range-restricted. A DEDALUS$_0$ program is instantaneously safe if its deductive reduction is instantaneously safe.

The `successor` relation complicates the discussion of safety, as it introduces the countably infinite set $\mathbb{Z}$ to our universe of constants.

Consider the following DEDALUS$_0$ program, which derives a single, persistent fact:

EXAMPLE 5. *An unsafe* DEDALUS$_0$ *instance?*

```
persist[p, p_neg 2]
p(1, 2)@123;
```

The single ground fact will cause one deduction for each tuple in `successor`*. Since* `successor` *is infinite, the corresponding Datalog program is unsafe.*

However, observe that each of these deductions produces a tuple that changes only in its time suffix. We find it useful to distinguish between unsafe programs and programs that, given a finite EDB, eventually derive only tuples that are equivalent except in their time suffixes.

Definition 5. Two sets of ground atoms Γ and Γ' are *equivalent modulo time* if each atom $\gamma \in \Gamma$ has a corresponding atom $\gamma' \in \Gamma'$ such that γ and γ' have the same predicate symbol, and the same assignment of constants to attributes for all attributes except the time suffix.

Definition 6. We say a DEDALUS$_0$ instance is *quiescent at time T* if the set of all atoms with time suffix T is equivalent modulo time to the set of all atoms with time suffix $T-1$.

OBSERVATION 1. *A* DEDALUS$_0$ *instance that is quiescent at time T will be quiescent until timestamp of the next EDB fact V, i.e. for all* $U \in \mathbb{Z} : V > U >= T$*. If no EDB fact has a timestamp greater than T, then the instance will be henceforth quiescent.*

Definition 7. A DEDALUS$_0$ instance with finite EDB is *temporally safe* if it is henceforth quiescent after some time T.

Definition 8. Given the depends-on relation $>$ and its transitive closure $>^*$, an intensional predicate e in a program P is called an *instantaneous predicate* if for every predicate p for which $e >^* p$ (ie, e depends transitively on p), either p appears in the head of no inductive rules, or the body of each inductive rule with head p contains at least one positive instantaneous predicate.

We propose the following conservative test for temporal safety. A program is guaranteed to be temporally safe if every rule is either:

1. An instantaneously safe rule, or
2. An inductive rule in which the head predicate occurs also in the body with the same variable bindings for all attributes save the time suffix, or
3. An inductive rule that has at least one instantaneous predicate as a positive subgoal in the body.

While a temporally safe program is henceforth quiescent after some time T, a temporally unsafe program changes infinitely. Note that the DEDALUS$_0$ program in Example 5 is temporally safe because the basic persistence macro creates a rule that satisfies the second condition above.

LEMMA 3. *A temporally stratifiable* DEDALUS$_0$ *instance is temporally safe if it has a finite EDB and every rule is one of the kinds 1-3 above.*

EXAMPLE 6. *A* DEDALUS$_0$ *instance with a temporally unsafe deductive rule.*

```
p(A, B) ← q(A);
```

The program above has a temporally unsafe deductive rule that corresponds to an unsafe rule in Datalog: it is not range-restricted. The head variable B could range over an infinite set of constants.

EXAMPLE 7. *A* DEDALUS$_0$ *instance that is temporally unsafe due to infinite oscillation.*

```
flip_flop(B, A)@next ← flip_flop(A, B);
flip_flop(0, 1)@1;
```

The above program exemplifies temporally unsafe induction. Even though it contains no function symbols, and all variables are range-restricted, it entails infinite oscillation of the p *predicate.*

We can imagine interesting examples of temporally unsafe programs, and do not forbid them in DEDALUS$_0$.

6. EVALUATION

In previous sections, we extended the notions of stratifiability and safety to DEDALUS$_0$ programs. In this section, we address the third and final property of DEDALUS$_0$ programs that we want to ensure—efficient execution.

Unfortunately, the direct application of traditional bottom-up Datalog execution strategies like semi-naive evaluation results in a rather literal and inefficient notion of the idea of "persistence." If a fact is true across a long sequence of timesteps, bottom-up evaluation will persistently "re-derive" that fact inductively for each timestep, and the number of derivations in a program will be infinite simply to maintain persistence in time. Instead, we would like an incremental evaluation strategy that allows an external agent to examine the state of the database at any timestep t without requiring $O(t)$ inductive derivations for persistence. The intuitive strategy would be to use a memory device, "storing" a fact on first derivation and "deleting" it at the timestep that the induction is broken. In this section we derive such a strategy via a combination of program rewriting and an operational evaluation loop, in the style of semi-naive evaluation.

6.1 Temporal Evaluation Over Storage

The traditional description of semi-naive evaluation takes a recursive Datalog program, rewrites it to a non-recursive "delta" program, and executes that program in a loop bracketed by state modifications. In that spirit, we present a strategy we call "temporal evaluation," which takes a DEDALUS$_0$ program, rewrites it to a Datalog program that refers to a single timestep, and executes that program in a loop—once per timestep—bracketed by state modifications. Algorithm 1 presents this strategy. Note that the DEDALUS$_0$ rules are written in their native "unsugared" syntax because we rewrite them into Datalog that strays from DEDALUS$_0$ conventions:

Observe that the final loop of Algorithm 1 binds the time suffix of each rule by replacing it with a constant value from a previous timestep. This "marches" through time in order, skipping steps

Algorithm 1 Temporal Evaluation

// rewrite program
foreach persistent predicate $P(A_1, \ldots, A_n)$ **do**
  // include "old facts" in the current timestep
  addRule $P(A_1, \ldots, A_n, \mathcal{T}) \leftarrow P\_store(A_1, \ldots, A_n)$.
  // identify "new" derived facts
  addRule $\Delta_P^+(A_1, \ldots, A_n) \leftarrow P(A_1, \ldots, A_n, \mathcal{T}), \neg P(A_1, \ldots, A_n, \mathcal{T} - 1)$.
  // identify "new" facts that are "to-be-deleted"
  addRule $\Delta_P^- \leftarrow P\_neg(A_1, \ldots, A_n, \mathcal{T})$.
end for
foreach rule R **do**
  foreach persistent predicate $P(A_1, \ldots, A_n, \mathcal{T})$ in R's body **do**
    substitute $P\_store(A_1, \ldots, A_n)$ for $P(A_1, \ldots, A_N, \mathcal{T})$
  end for
end for
// evaluate program starting at "minimum" EDB timestamp
let $t = u \in \mathbb{Z} : \exists P(A_1, \ldots, A_n, u), \nexists Q(B_1, \ldots, B_m, v) : v < u$
repeat
  replace $\mathcal{T}$ with t in the rewritten program, and compute a fixpoint of the result via semi-naive evaluation
  foreach persistent predicate $P(A_1, \ldots, A_n)$ **do**
    $P\_store(A_1, \ldots, A_n) :=$
      $P\_store(A_1, \ldots, A_n) \cup \Delta_P^+(A_1, \ldots, A_n)$
    $P\_store(A_1, \ldots, A_n) :=$
      $P\_store(A_1, \ldots, A_n) \setminus \Delta_P^-(A_1, \ldots, A_n)$
  end for
  // t becomes the least larger timestamp with an atom
  if $\exists u \in \mathbb{Z} : u > t, \exists P(A_1, \ldots, A_n, t), \nexists v : v > t \wedge v < u$ **then**
    let $t = u$
  else
    let $t = \emptyset$
  end if
until $t = \emptyset$

that have no changes. A simple proof by induction shows that for each timestep t, the temporal evaluation yields a database that corresponds to the minimal model of the original Dedalus$_0$ program with the `successor` relation truncated to the prefix ending at t.

7. ASYNCHRONY

Until now we have restricted our discussion to Dedalus$_0$. In this section we introduce Dedalus, a superset of Dedalus$_0$ that also admits the *choice* construct [14] to bind time suffixes. Choice allows us to model the inherent nondeterminism in communication over unreliable networks that may delay, lose or reorder the results of logical deductions. We also describe a syntactic convention to employ this communication model for "horizontal partitions" of relations on different machines.

7.1 Choice

An important property of distributed systems is that individual computers cannot control or observe the temporal interleaving of their computations with other computers. One aspect of this uncertainty is captured in network delays: the arrival "time" of messages cannot be directly controlled by either sender or receiver. In this section, we enhance our language with a traditional model of nondeterminism from the literature to capture these issues: the *choice* construct as defined by Greco and Zaniolo [14].

The subgoal `choose((`X_1`), ((`X_2`))` may appear in the body of a rule, where X_1 and X_2 are vectors whose constituent variables occur elsewhere in the body. Such a subgoal enforces the functional dependency $X_1 \rightarrow X_2$, "choosing" a single assignment of values to the variables in X_2 for each variable in X_1.

The choice construct is nondeterministic. In a model-theoretic interpretation of logic programming, a nondeterministic program must have a multiplicity of stable models—that is it must be unstratifiable. Greco and Zaniolo define choice in precisely this fashion: the choice construct is expanded into an unstratifiable strongly-connected component of rules, and each possible choice is associated with a different model. Each such model has a unique, non-deterministic assignment that respects the given functional dependencies. In our discussion, it may be helpful to think of one such model chosen nondeterministically—a non-deterministic "assignment of timestamps to tuples."

7.2 Distribution Model

The choice construct will capture the non-determinism of multiple communicating agents in a distributed system, but we want to use it in a stylized way to model typical notions of distribution. To this end Dedalus adopts the "horizontal partitioning" convention introduced by Loo et al. and used in many subsequent efforts [24]. To represent a distributed system, we consider some number of agents, each running a copy of the same program against a disjoint subset (*horizontal partition*) of each predicate's contents. We require one attribute in each predicate to be used to identify the partitioning for tuples in that predicate. We call such an attribute a *location specifier*, and prefix it with a # symbol in Dedalus.

Finally, we constrain Dedalus rules so that the location specifier variable in each body predicate be the same—i.e. the body contains tuples from exactly one partition of the database, logically colocated (on a single "machine"). If the head of the rule has the same location specifier variable as the body, we call the rule "local," since its results can remain on the machine where they are computed. If the head has a different variable in its location specifier, we call the rule a *communication rule*. We now proceed to our model of the asynchrony of this communication, which is captured in a syntactic constraint on the heads of communication rules.

7.3 Asynchronous Rules

In order to represent the nondeterminism introduced by distribution, we admit a third type of rule, called an *asynchronous* rule. A rule is asynchronous if the relationship between the head time suffix $\mathcal{S}$ and the body time suffix $\mathcal{T}$ is unknown. Furthermore, $\mathcal{S}$ (but not $\mathcal{T}$) may take on the special value $\top$ which means "never." Derivation at $\top$ indicates that the deduction is "lost," as time suffixes in rule bodies do not range over $\top$.

We model network nondeterminism using the choice construct to choose from a value in the special `time` predicate, which is defined using the following Datalog rules:

```
time(⊤);
time(S) ← successor(S, _);
```

Each asynchronous rule with head predicate `p(`$A_1, \ldots, A_n$`)` has the following additional subgoals in its body:

`time(`$\mathcal{S}$`), choose((`$A_1, \ldots, A_n, \mathcal{T}$`), (`$\mathcal{S}$`))`,

where $\mathcal{S}$ is the timestamp of the rule head. Note that our use of `choose` incorporates all variables of each head predicate tuple, which allows a unique choice of $\mathcal{S}$ for each head tuple.

Example 8. *A well-formed asynchronous* Dedalus *rule:*

```
r(A, B, S) ←
  e(A, B, T),
  time(S), choose((A, B, T)), (S));
```

We admit a new temporal head annotation to sugar the rule above. The identifier `async` implies that the rule is asynchronous, and stands in for the additional body predicates. The above example expressed using `async` is:

EXAMPLE 9. *A sugared asynchronous* DEDALUS *rule:*

```
r(A, B)@async ← e(A, B);
```

7.4 Asynchrony and Distribution in DEDALUS

As a syntactic constraint of DEDALUS, the *communication rules* introduced in the previous section (rules that differ in head and body location specifiers) are required to be asynchronous. This restricts our model of communication between agents in two important ways. First, by restricting bodies to a single agent, the only communication modeled in DEDALUS occurs via communication rules. Second, because all communication rules are asynchronous, agents may only learn about time values at another agent by receiving messages (with unbounded delay) from that agent. Note that this model says nothing about the relationship between the agents' clocks; they could be non-monotonically increasing, or they could respect a global order.

7.5 Temporal Monotonicity

Nothing in our definition of asynchronous rules prevents tuples in the head of a rule from having a timestamp that precedes the timestamp in the rule's body. This is a significant departure from DEDALUS$_0$, since it violates the monotonicity assumptions upon which we based both Algorithm 1 and our proof of temporal stratification. On an intuitive level, it may also trouble us that rules can derive head tuples that exist "before" the body tuples on which they are grounded; this violates intuitive notions of causality and admits the possibility of temporal paradoxes.

We have avoided restricting DEDALUS to rule out such issues, as doing so would reduce its expressiveness. Recall that simple monotonic Datalog (without negation) is insensitive to the values in any particular attribute. Hence DEDALUS programs without negation are also well-defined regardless of any "temporal ordering" of deductions: in monotonic programs, even if tuples with timestamps "in the future" are used to derive tuples "from the past," there is an unambiguous least minimal model. In Section 5.1 we showed that the monotonicity of time suffixes achieved by inductive rules ensures a unique minimal model even for non-monotonic DEDALUS$_0$ programs.

7.5.1 Practical Implications

Given this discussion, in practice we are interested in three asynchronous scenarios: (a) monotonic programs (even with non-monotonicity in time), (b) non-monotonic programs whose semantics guarantee monotonicity of time suffixes and (c) non-monotonic programs where we have domain knowledge guaranteeing monotonicity of time suffixes. Each represents practical scenarios of interest.

The first category captures the spirit of many simple distributed implementations that are built atop unreliable asynchronous substrates. For example, in some Internet publishing applications (weblogs, online fora), it is possible due to caching or failure that a "thread" of discussion arrives out of order, with responses appearing before the comments they reference. In many cases a monotonic "bag semantics" for the comment program is considered a reasonable interface for readers, and the ability to tolerate temporal anomalies simplifies the challenge of scaling a system through distribution.

The second scenario is achieved in DEDALUS$_0$ via the use of `successor` for the time suffix. The asynchronous rules of DEDALUS require additional program logic to guarantee monotonic increases in time for predicates with dependencies. In the literature of distributed computing, this is known as a *causal ordering* and is enforced by distributed clock protocols. We review one classic protocol in the DEDALUS context in Section 7.6; including this protocol into DEDALUS programs ensures temporal monotonicity.

Finally, certain computational substrates guarantee monotonicity in both timestamps and message ordering—for example, some multiprocessor cache coherency protocols achieve this. When temporal monotonicity is given, the proofs of temporal stratification and Algorithm 1 both apply.

7.5.2 Entanglement

Consider the asynchronous rule below:

```
p(A, B, N)@async ←
  q(A, B)@N;
```

Due to the `async` keyword in the rule head, each *p* tuple will take some unspecified time suffix value. Note however that the time suffix *N* of the rule body appears also in an attribute of *p* other than the time suffix, recording a binding of both the time value of the deduction and the time value of its consequence. We call such a binding an *entanglement*. Note that in order to write the rule it was necessary to not sugar away the time suffix in the rule body.

Entanglement is a powerful construct. It allows a rule to reference the logical clock time of the deduction that produced one (or more) of its subgoals; this supports protocols that reason about partial ordering of time across machines. More generally, it exposes the infinite `successor` relation to attributes other than the time suffix, allowing us to express concepts such as infinite sequences.

7.6 Lamport Clocks

Recall that DEDALUS allows program executions to order message timestamps arbitrarily, violating intuitive notions of causality by allowing deductions to "affect the past." This section explains how to implement Lamport clocks [17] atop DEDALUS, which allows programs to ensure temporal monotonicity by reestablishing a causal order despite derivations that flow backwards through time.

Consider a rule `p(A,B)@async ← q(A,B)`. By rewriting it to:

```
persist[p, 2]
p_wait(A, B, N)@async ← q(A, B)@N;
p_wait(A, B, N)@next ← p_wait(A, B, N)@M, N ≥ M;
p(A, B)@next ← p_wait(A, B, N)@M, N < M;
```

we place the derived tuple in a new relation `p_wait` that stores any tuples that were "sent from the future" with their sending time "entangled"; these tuples stay in the `p_wait` predicate until the point in time at which they were derived. Conceptually, this causes the system to evaluate a potentially large number of timesteps (if N is significantly less than the timestamp of the system when the tuple arrives). However, if the runtime is able to efficiently evaluate timesteps when the database is quiescent, then instead of "waiting" by evaluating timesteps, it will simply increase its logical clock to match that of the sender. In contrast, if the tuple is "sent into the future," then it is processed using the timestep that receives it.

This manipulation of timesteps and clock values is equivalent to conventional descriptions of Lamport clocks, except that our Lamport clock implementation effectively "advances the clock" by preventing derivations until the clock is sufficiently advanced, by temporarily store incoming tuples in the `p_wait` relation.

We gloss over one detail here: Lamport clocks rely upon a "tie-breaking" function to ensure that no two events have the same timestamp. In DEDALUS, such a function could be implemented via another use of `choice`, or by a program convention like appending a unique node identifier to each timestamp to prevent "ties."

7.7 Reliable Broadcast

Distributed systems cope with unreliable networks by using mechanisms like broadcast and consensus protocols, timeouts and retries, and often hide the nondeterminism behind these abstractions. DEDALUS supports these notions, achieving encapsulation of nondeterminism while dealing explicitly with the uncertainty in the model. Consider the simple broadcast protocol below:

```
sbcast(#Member, Sender, Message)@async ←
  smessage(#Agent, Sender, Message),
  members(#Agent, Member);

sdeliver(#Member, Sender, Message) ←
  sbcast(#Member, Sender, Message);
```

Assume the table `members` is a persistent relation given to us, containing the broadcast membership list. The protocol is straightforward: if a tuple appears in `smessage` (an EDB predicate), then it will be sent to all members (a multicast). The interpretation of the non-deterministic choice implied by the `@async` rule indicates that order and delivery (i.e., finite delay) are not guaranteed.

The program shown below makes use of the multicast primitive provided by the previous program, and uses it to implement a basic reliable broadcast using a textbook mechanism [27] that assumes any node that fails to receive a message sent to it has failed. When broadcast completes, all nodes that have not failed have received the message.

Our simple two-rule broadcast program is augmented with the following rules, so that if a node receives a message, it also multicasts it to every member *before* delivering the message locally:

```
smessage(Agent, Sender, Message)  ←
  rmessage(Agent, Sender, Message);

buf_bcast(Sender, Me, Message)  ←
  sdeliver(Me, Sender, Message);

smessage(Me, Sender, Message)  ←
  buf_bcast(Sender, Me, Message);

rdeliver(Me, Sender, Message)@next  ←
  buf_bcast(Sender, Me, Message);
```

Note that all network communication is initiated by the `@async` rule from the original simple broadcast. The `@next` is required in the `rdeliver` definition in order to prevent nodes from taking actions based upon the broadcast before it is guaranteed to meet the reliability guarantee.

Implementing other disciplines like FIFO and atomic broadcast and consensus are similar exercises, requiring the use of ordered queueing and sequences.

8. RELATED WORK

8.1 Updateable State

Many deductive database systems, including LDL [8] and Glue-Nail [11], admit procedural semantics to deal with updates using an assignment primitive. In contrast, languages proposed by Cleary and Liu [10, 20, 25] retain a purely logical interpretation by admitting temporal extensions into their syntax and interpreting assignment or update as a composite operation across timesteps [20] rather than as a primitive. We follow the latter approach, but differ in several significant ways. First, we model persistence explicitly in our language, so that like updates, it is specified as a composite operation across timesteps. Partly as a result of this, we are able to enforce stricter constraints on the allowable time suffixes in rules: a program may only specify what deductions are visible in the current timestep, the immediate next timestep, and *some* future timestep, as opposed to the free use of intervals allowed in rules in Liu et al. Our simple inductive approach to persistence obviates the need to evaluate stabbing queries on time "ranges."

U-Datalog [7] addresses updates using syntax annotations that establish different interpretations for the set of updated relations and the IDB, interpreting update atoms as constraints and using constraint logic programming techniques to test for inconsistent derivations. Similarly, Timed Concurrent Constraint Programming (TCCP) [31, 32] handles nonmonotonic constructs in a CLP framework by outputting a new (possibly diminished) store and constraint program at each timestep.

Our temporal approach to representing state change most closely resembles the Statelog language [13]. We were unaware of the Statelog work when formalizing the notions of persistence via induction and temporal stratification, and were surprised (and encouraged) to observe very similar ideas in Statelog. By contrast, our motivation is the logical specification and implementation of distributed systems, and our principal contribution is the use of time to model both local state change and communication over unreliable networks.

Lamport's TLA+ [18] is a language for specifying concurrent systems in terms of constraints over valuations of state, and temporal logic that describes admissible transitions. Two distinguishing features of DEDALUS with respect to TLA+ is our minimalist use of temporal constructs (next and later), and our unified treatment of temporal and other attributes of facts, enabling the full literature of Datalog to be applied both to temporal and instantaneous properties of programs.

8.2 Distributed Systems

Significant recent work ([2, 6, 9, 22], etc.) has focused on applying deductive database languages extended with networking primitives to the problem of specifying and implementing network protocols and distributed systems. Implementing distributed systems entails a data store that changes over time, so any useful implementation of such a language addresses the updateable state issue in some manner. Existing distributed deductive languages like NDlog and Overlog adopt the *chain of fixpoints* interpretation discussed in Section 2.1.

Unfortunately, the language descriptions of systems following this "observe, think, act" model give no careful specification of how and when deletions and updates should be made visible, so the third step is a "black box." Loo et al. [21] proved that classes of programs with certain monotonicity properties (i.e., programs without negation or fact deletion) are equivalent (specifically, eventually consistent) when evaluated globally (via a single fixpoint computation) or in a distributed setting in which the *chain of fixpoints* interpretation is applied at each participating node, and no messages are lost. Navarro et al. [29] proposed an alternate syntax that addressed key ambiguities in Overlog, including the *event creation vs. effect* ambiguity. Their solution solves the problem by introducing procedural semantics to the interpretation of the augmented Overlog programs. A similar analysis was offered by Mao [26].

9. CONCLUSION

Datalog has inspired a variety of recent applied work, which touts the benefits of declarative specifications for practical implementations. We have developed substantial experience building significant distributed systems [2, 4, 9, 22] using hybrid declarative/imperative languages such as Overlog [22]. While our experience with those languages was largely positive, the combination of Datalog and imperative constructs often clouded our understanding of the "correct" execution of single-node programs that performed state updates. This work developed in large part as a reaction to the semantic

difficulties presented by these distributed logic languages.

Through its reification of time as data, DEDALUS allowed us to achieve the goal of a declarative language without sacrificing critically expressive features for the distributed systems domain. We believe that DEDALUS is as expressive as Overlog, whose operational semantics [2] are essentially the same as those described in Algorithm 1. Formalizing this intuition is difficult because the semantics of Overlog are not well specified. Instead, we are currently validating our practicality by "porting" many of our Overlog programs to DEDALUS.

In DEDALUS, state update and communication differ from logical deductions only in terms of timing. In the local case, this allows us to express state update without giving up the clean semantics of Datalog; unlike Datalog extensions that use imperative constructs to provide such functionality, each DEDALUS rule expresses a logical invariant that will hold over all program executions. However, interactions with external processes, and primitives such as asynchronous and unreliable communication introduce nondeterminism which DEDALUS models with `choose`. Our hope is that modeling external processes and events with a single primitive will simplify formal program verification techniques for the distributed systems domain. Two natural directions in this vein are to determine for a given DEDALUS program whether Church-Rosser confluence holds for all models produced by `choice`, or to capture finer-grained notions like serializability of such models with respect to transaction identifiers embedded in EDB facts.

10. ACKNOWLEDGMENTS

Ras Bodík and Tyson Condie were intimately involved in discussions surrounding the development of DEDALUS. We are also indebted to Mark Utting and Erik Meijer for conversation and inspiration from their Starlog and LINQ experiences respectively, and to Phil Bernstein for suggestions on future work. Thanks to Jesse Trutna and Kuang Chen for comments on the paper. This work was supported by NSF grants 0803690, 0722077 and 0713661, Air Force Office of Scientific Research award 22178970-41070-F, and gifts from Yahoo Research, IBM Research and Microsoft Research.

11. REFERENCES

[1] H. Abelson and G. J. Sussman, editors. *Structure and Interpretation of Computer Programs*. McGraw Hill, second edition, 1996.

[2] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears. BOOM: Data-Centric Programming in the Datacenter. Technical Report UCB/EECS-2009-113, EECS Department, University of California, Berkeley, Jul 2009.

[3] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I do declare: consensus in a logic language. *SIGOPS Oper. Syst. Rev.*

[4] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I do declare: Consensus in a logic language. In *NetDB*, 2009.

[5] M. P. Ashley-Rollman et al. Declarative programming for modular robots. In *Workshop on Self-Reconfigurable Robots/Systems and Applications*, 2007.

[6] N. Belaramani, J. Zheng, A. Nayate, R. Soulé, M. Dahlin, and R. Grimm. Pads: A policy architecture for distributed storage systems. In *NSDI*, 2009.

[7] E. Bertino, B. Catania, and R. Gori. Enhancing the Expressive Power of the U-Datalog Language. *Theory and Practice of Logic Programming*, 1(1):105–122, 2001.

[8] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL System Prototype. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):76–90, 1990.

[9] D. C. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *5th ACM Conference on Embedded networked Sensor Systems (SenSys)*, 2007.

[10] J. G. Cleary, M. Utting, and R. Clayton. Data Structures Considered Harmful. In *Australasian Workshop on Computational Logic*, 2000.

[11] M. A. Derr, S. Morishita, and G. Phipps. The Glue-Nail Deductive Database System: Design, Implementation, and Evaluation. *The VLDB Journal*, 3:123–160, 1994.

[12] J. Eisner, E. Goldlust, and N. A. Smith. Dyna: a declarative language for implementing dynamic programs. In *Proc. ACL*, 2004.

[13] W. M. Georg Lausen, Bertram Ludäscher. On active deductive databases: The statelog approach. In *Transactions and Change in Logic Databases*, pages 69–106, 1998.

[14] S. Greco and C. Zaniolo. Greedy Algorithms in Datalog with Choice and Negation. In *JICSLP'98: Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, pages 294–309, Cambridge, MA, USA, 1998. MIT Press.

[15] T. Jim. Sd3: A trust management system with certified evaluation. *Security and Privacy, IEEE Symposium on*, 0:0106, 2001.

[16] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *PODS*, 2005.

[17] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[18] L. Lamport. The temporal logic of actions. *ACM Toplas*, 16(3):872–923, May 1994.

[19] N. Li and J. Mitchell. Datalog with constraints: A foundation for trust-management languages. In *International Symposium on Practical Aspects of Declarative Languages*, 2003.

[20] M. Liu and J. Cleary. Declarative Updates in Deductive Databases. *Journal of Computing and Information*, 1:1435–1446, 1994.

[21] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *ACM SIGMOD*, New York, NY, USA, 2006.

[22] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Communications of the ACM*, 52(11):87–95, 2009.

[23] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.

[24] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *SIGCOMM*, 2005.

[25] L. Lu and J. G. Cleary. An Operational Semantics of Starlog. In *Proc. Principles and Practice of Declarative Programming*, pages 131–162. Springer-Verlag, 1999.

[26] Y. Mao. On the declarativity of declarative networking. In *NetDB*, 2009.

[27] S. Mullender, editor. *Distributed Systems*. Addison-Wesley,

second edition, 1993.
[28] I. S. Mumick and O. Shmueli. How expressive is stratified aggregation? *Annals of Mathematics and Artificial Intelligence*, 15(3-4):407–435, Sept. 1995.
[29] J. A. Navarro and A. Rybalchenko. Operational Semantics for Declarative Networking. In *PADL*, pages 76–90, 2009.
[30] K. A. Ross. Modular Stratification and Magic Sets for DATALOG Programs with Negation. In *PODS*, pages 161–171, New York, NY, USA, 1990. ACM.
[31] V. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 22(5-6):475–520, 1996.
[32] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of Timed Concurrent Constraint Programming. In *LICS*, pages 71–80, 1994.
[33] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990.
[34] A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative network verification. In *PADL '09: Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages*, 2009.
[35] A. Wang, L. Jia, C. Liu, B. Thau, L. Oleg, and S. P. Basu. Formally verifiable networking.
[36] W. White et al. Scaling games to epic proportions. In *SIGMOD*, 2007.
[37] W. Zhou, Y. Mao, B. T. Loo, and M. Abadi. Unified declarative platform for secure netwoked information systems. In *ICDE*, pages 150–161, 2009.

APPENDIX

.1 Proof of Lemma 2

Proof. **Case 1:** P consists of only deductive rules. In this case, P's deductive reduction is P itself. We know P is syntactically stratifiable, thus it has a unique minimal model.

Case 2: P consists of both deductive and inductive rules. Assume that P does not have a unique minimal model. This implies that P is not syntactically stratifiable. Thus, there must exist some cycle through at least one predicate q involving negation. Furthermore, this cycle must involve an inductive rule, as P is temporally stratified.

Since the time suffix in the head of an inductive rule is strictly greater than the time suffix of its body, no atom may depend negatively on itself—it may only depend negatively on atoms in the previous timestep. Thus, P' is modularly stratified over time, using the definition of modular stratification according to Ross et al. [30]. This guarantees a unique minimal model achievable via standard bottom-up fixpoint execution. □

A. PROOF OF OBSERVATION 1

Proof. A Dedalus$_0$ program admits only instantaneous and inductive rules, which derive new tuples at the same time as their ground tuples, or in the immediate next timestep. Thus, the set of tuples true at time T is completely determined by any tuples true at time $T-1$, and any EDB facts true at time T. Observe that the integer value of the timestep does not influence the derivation.

If the instance is quiescent at T, then given $\mathbf{A}$, the set of atoms with timestamp $T-1$, and the EDB at T, the program entails $\mathbf{A}$ at timestamp T. Thus in the absence of EDB facts at $T+1$, it entails $\mathbf{A}$ at $T+1$. □

B. PROOF OF LEMMA ??

Proof. Assume the program is temporally unsafe. That is, there exists no time T such that $\forall U >= T$, the set of all atoms with timestamp U is equivalent modulo time to the set of all atoms with timestamp $T-1$. Let E be the maximum timestamp of any fact in the EDB.

Observe that every rule r of kind 3 may only entail a finite number of facts—as the EDB is finite—and thus may entail no facts at a timestamp greater than some maximum timestamp $V_r <= E+1 \in \mathbb{Z}$. Since a Dedalus$_0$ program has a finite set of rules we know $\exists V \in \mathbb{Z} : \forall r : V >= V_r$, and thus $V <= E+1$.

We now consider times T such that $T > E+1$. By the above argument, no rules of kind 3 entail any facts with a timestamp greater than $E+1$. Recall that no EDB atoms are true at any timestamp greater than E. Thus, any facts with timestamp greater than $E+1$ are entailed by rules of kind 1 or 2.

Consider the set of equivalence classes modulo time of all possible atoms, $\mathbf{A}$, given the Herbrand universe. We know $\mathbf{A}$ is finite, as the Herbrand Universe is finite. Therefore, if the program is temporally unsafe, then $\mathbf{B}$, the set of atoms entailed by the program, both contains and excludes infinitely many members of at least one equivalence class in $\mathbf{A}$ (i.e., something "infinitely oscillates in time" between being true and false). Since the program has finitely many rules, at least one rule must entail infinitely many atoms (from at least one of the equivalence classes from $\mathbf{A}$). Thus, it is easy to see that there must be a cycle that contains some predicate P and $\neg P$.

We show there exists such a cycle containing only rules of kind 1, which implies that the program is temporally unstratifiable. In order for such a cycle to exist, P must transitively depend on $\neg P$, and $\neg P$ must transitively depend on P. Thus, the program contains a rule J_1 with $\neg P$ in its body, and some predicate R in its head, as well as a rule J_2 that is transitively dependent on R, with P in its head.

Case 1: $P \neq R$. In this case, J_1 must be of kind 1, as for any $Q \neq P$, a rule of kind 2 with P in the head may not directly entail Q given P. J_2 must also be of kind 1—if it is of kind 2, then it necessarily contains P in its body, so it cannot entail P unless P is entailed by some other rule. If J_2 contains R in its body, then the program is syntactically unstratifiable. But if J_2 does not contain R in its body, then it contains some predicate S transitively entailed by R; wlog the body contains R. Thus, the program is syntactically unstratifiable.

Case 2: $P = R$. In this case, J_1 and J_2 are the same rule: $P \leftarrow \neg P$. Thus, the program is syntactically unstratifiable.

Thus, the program is temporally unstratifiable, which contradicts our assumption. □