Confluence Analysis for Distributed Programs: A Model-Theoretic Approach

William Marczak
Peter Alvaro
Neil Conway
Joseph M. Hellerstein
David Maier

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-154
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-154.html

December 18, 2011

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
18 DEC 2011		**00-00-2011 to 00-00-2011**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Confluence Analysis for Distributed Programs: A Model-Theoretic Approach	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Building on recent interest in distributed logic programming, we take a model-theoretic approach to analyzing confluence of asynchronous distributed programs. We begin with a model-theoretic semantics for Dedalus and develop the concept of ultimate models to capture the non-deterministic eventual outcomes of distributed programs. After demonstrating the undecidability of checking confluence for Dedalus programs, we look for restricted sub-languages that guarantee confluence while providing adequate expressivity. We observe that a simple semipositive restriction called Dedalus+ guarantees confluence while capturing PTIME, but demonstrate that the limited use of negation in Dedalus+ makes certain simple and practical programs very di cult to express. To remedy this, we introduce DedalusS , a restriction of Dedalus that allows a natural use of negation in the spirit of stratified negation, but retains the confluence of Dedalus+ and similarly captures PTIME.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**14**	

Acknowledgement

This work was supported by NSF grants 0917349, 0803690, 0722077, 0713661 and 0435496, Air Force Office of Scientific Research award 22178970-41070-F, the Natural Sciences and Engineering Research Council of Canada, and gifts from Yahoo Research, Microsoft Research and NTT Multimedia Communications Laboratories.

Confluence Analysis for Distributed Programs: A Model-Theoretic Approach

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

Building on recent interest in distributed logic programming, we take a model-theoretic approach to analyzing confluence of asynchronous distributed programs. We begin with a model-theoretic semantics for Dedalus and develop the concept of *ultimate models* to capture the non-deterministic eventual outcomes of distributed programs. After demonstrating the undecidability of checking confluence for Dedalus programs, we look for restricted sub-languages that guarantee confluence while providing adequate expressivity. We observe that a simple semipositive restriction called Dedalus$^+$ guarantees confluence while capturing PTIME, but demonstrate that the limited use of negation in Dedalus$^+$ makes certain simple and practical programs very difficult to express. To remedy this, we introduce Dedalus^S, a restriction of Dedalus that allows a natural use of negation in the spirit of stratified negation, but retains the confluence of Dedalus$^+$ and similarly captures PTIME.

1. INTRODUCTION

In recent years there has been optimism that declarative languages grounded in Datalog can provide a clean foundation for distributed programming [24]. This has led to activity in language and system design (e.g., [4, 9, 14, 32]), as well as formal models for distributed computation using such languages (e.g., [8, 36, 37]).

The bulk of this work has presented or assumed a formal operational semantics based on transition systems and traces of input events. A model-theoretic semantics for these languages has been notably absent. In a related paper [3], we have developed a model-theoretic semantics for Dedalus, a distributed logic language based on Datalog, in which the "meaning" of a program is precisely the set of stable models [38] that can arise via all possible temporal permutations of messages. In the same paper, we demonstrate an equivalence of these models with all possible executions in a operational semantics akin to those in the prior literature.

In this paper we take advantage of the availability of declarative semantics to explore the correctness of distributed programs. Specifically, we address the desire to ensure deterministic program outcomes—confluence—in the face of inherently non-deterministic timings of computation and messaging. This is a matter of widespread practical concern in distributed systems, often cast in terms of "eventual consistency" [40, 39], and grounded in foundational issues of time and clocks in the theory of distributed computing [27].

Using our model-theoretic semantics for Dedalus, we can reason about the set of possible outcomes of a distributed program, based on what we define as their *ultimate models*. This formal framework enables us to declaratively describe the potential for non-deterministic outcomes in Dedalus programs. It also allows us to identify restricted sub-languages of Dedalus that ensure a model-theoretic notion of confluence: the existence of a unique ultimate model for any program expressible in that sub-language. The next question then is to identify a sub-language that ensures confluence without unduly constraining expressivity—both in terms of both computational power, and the ability to employ familiar programming constructs.

One natural step in this direction is to drop back from the expressive power of Dedalus to a monotonic subset: a language we call Dedalus$^+$ that disallows negation of IDB predicates. This is inspired in part by the CALM theorem [5, 24], which established a connection between confluence and monotonicity; subsequent formalizations proved that monotonic distributed programs are indeed guaranteed to be confluent [1, 8]. In terms of expressivity, Immerman's celebrated result regarding the collapse of the fixpoint hierarchy established that PTIME is captured by a similar monotonic language: semipositive Datalog (Datalog$\neg$ where negation is restricted to EDB relations) augmented with an ordering over the universe [25]. Put together, these results lead to a rather startling conclusion: Dedalus$^+$ shows that it is possible to express any polynomial-time distributed algorithm (surely the vast majority of useful distributed code!) in an eventually consistent manner.

This result is intriguing but not necessarily useful. In particular, it does not guarantee that Dedalus$^+$ or similar monotonic languages can be used to express *natural* declarations of programs. Perhaps this is why, despite Immerman's complexity results over 25 years ago, there has been ongoing interest in the topic of negation in logic programs. More specifically, we have found that Dedalus$^+$ is quite unnatural to use in many cases that interest us—we demonstrate this below via a practical system component (distributed garbage collection) that is easily written in Dedalus, but would be quite convoluted in Dedalus$^+$.

Given this background, we seek a more comfortable balance between expressive power, ease of programming and guarantees of confluence. We achieve this via a controlled use of negation that draws inspiration from both stratified negation in logic, and coordination protocols from distributed computing. We present a language

called DEDALUS^S whose semantics allow negated predicates, but subject to a closed-world assumption that these predicates are evaluated on their "complete", unchanging state. To make this practical in a distributed context, we then show that an operational semantics for DEDALUS^S can be achieved by compiling DEDALUS^S programs into stylized DEDALUS programs that augment the original code with "coordination" rules to establish completion of subgoals as needed. The operational semantics of the resulting DEDALUS programs are then given by the prior literature. The result is a sub-language that retains many of the features we desire: PTIME expressivity, guarantees of confluence, and an intuitive and familiar use of negation. We believe the result is practically useful—indeed, DEDALUS^S corresponds closely to Bloom, a practical programming we have used to implement a broad array of distributed systems [10].

Our technical contributions in this paper include the following: (a) the definition of ultimate models as a declarative framework for assessing outcomes of DEDALUS programs and the undecidability of confluence for DEDALUS programs (Section 3), (b) the introduction of DEDALUS$^+$, its expressive power, and proof that it can only express programs with unique ultimate models (Section 3.2), (c) the introduction of DEDALUS^S, examples of its use, and model-theoretic proof of its confluence (Section 4, 4.1), and (d) an operational semantics for DEDALUS^S achieved via a compilation of DEDALUS^S programs into judiciously "coordinated" DEDALUS programs (Section 4.2, 4.3).

2. DEDALUS

DEDALUS extends Datalog by adding *spatial* and *temporal* attributes to every relation. The critical semantic issue from distributed computing that we wish to capture is non-determinism of message timing across nodes. This non-determinism is modeled simply by the use of a restricted version of Sacca and Zaniolo's *choice* construct [38]. We use the stable model semantics [21] to assign meanings to the behaviors of DEDALUS programs over time. In a companion paper [3], we prove these stable models are equivalent to traces in a variation of the *network transducer* model—an operational formalism for distributed systems—and thus argue that DEDALUS is a reasonable model for distributed systems.

We believe that the stable model semantics are inappropriate to represent the output of a distributed system, as they contain a potentially infinite number of distinctions that are not meaningful in an "eventual" sense. Thus, we introduce the *ultimate model* semantics as a representation of program output. As one might imagine, even in the ultimate model semantics, some programs may have multiple ultimate models that correspond to meaningful distinctions between stable models.

We begin this section by reviewing the syntax of DEDALUS first presented in Alvaro *et al*. [6]. We then review the model-theoretic semantics for DEDALUS [3], using a similar exposition for simplicity.

2.1 Syntax

2.1.1 Preliminary Definitions

We assume an infinite universe $\mathcal{U}$ of values. We assume $\mathbb{N} \subset \mathcal{U}$.[1]

A *relation schema* is a pair (R, k) where R is a relation name and k its arity. We also write (R, k) as $R^{(k)}$. A *database schema* $\mathcal{S}$ is a set of relation schemas. Any relation name occurs at most once in a database schema.

As in Immerman [25], we assume the existence of an order: every database schema contains the relation schema $<^{(2)}$.[2] Later, we will explain how $<$ is populated.

A *fact* over a relation schema $R^{(n)}$ is a pair consisting of the relation name R and an n-tuple $(c_1, \ldots, c_n)$, where each $c_i \in \mathcal{U}$. We denote a fact over $R^{(n)}$ by `R(c1, ..., cn)`.

A *relation instance* for relation schema $R^{(n)}$ is a set of facts whose relation is R. A *database instance* maps each relation schema $R^{(n)}$ to a corresponding relation instance for $R^{(n)}$.

A *rule* φ consists of several distinct components: a head atom $head(\varphi)$, a set $pos(\varphi)$ of atoms, a set $neg(\varphi)$ of atoms, a set of inequalities $ineq(\varphi)$ of the form $x < y$, and a set of choice operators $cho(\varphi)$ applied to the variables. Intuitively, we use choice operators to model real-world non-determinism due to network asynchrony. The elements of $pos(\varphi)$ are called *positive body atoms*, and the elements $neg(\varphi)$ are called *negative body atoms*.

The conventional syntax for a rule is:

$$head(\varphi) \leftarrow f_1, \ldots, f_n, \neg g_1, \ldots, \neg g_m, ineq(\varphi), cho(\varphi).$$

where $f_i \in pos(\varphi)$ for $i = 1, \ldots, n$ and $g_i \in neg(\varphi)$ for $i = 1, \ldots, m$. The following is an example of a rule over schema $\mathcal{S}$ with $ineq(\varphi) = \emptyset$ and $cho(\varphi) = \emptyset$.

$$\mathtt{p}(\overline{\mathtt{W}}) \leftarrow \mathtt{b}_1(\overline{\mathtt{X}_1}), \ \ldots, \ \mathtt{b}_l(\overline{\mathtt{X}_l}), \ \neg\mathtt{c}_1(\overline{\mathtt{Y}_1}), \ \ldots, \ \neg\mathtt{c}_m(\overline{\mathtt{Y}_m}).$$

where $\mathtt{p}, \mathtt{b}_1, \ldots, \mathtt{b}_l, \mathtt{c}_1, \ldots, \mathtt{c}_m$ are relations in $\mathcal{S}$, and $\overline{\mathtt{W}}$, $\overline{\mathtt{X}_i}$ and $\overline{\mathtt{Y}_j}$ denote a tuple (of the appropriate arity) consisting of constants from $\mathcal{U}$ or variable symbols.

The relation name $<$ may only appear in *ineq*; in particular, $<$ may not appear in any atom in *head*, *pos*, or *neg*.

2.1.2 Safety

DEDALUS maintains the usual Datalog safety restrictions: every variable symbol `V` in a rule must appear in some atom in *pos*.

For a variable symbol `V` that appears exactly once in exactly one *neg* atom, and does not appear elsewhere in the rule, there is a straightforward rewrite defined in Ullman [41] that brings the rule into compliance with the safety restriction. An example of the rewrite appears below.

Example 1. The unsafe rule: `p(X) ← q(X), ¬r(X,Y).`
is rewritten into the following two rules that obey the safety constraint:

```
p(X) ← q(X), ¬r'(X).
r'(X) ← r(X,Y).
```

where $\mathtt{r}'^{(1)}$ is a relation schema that does not otherwise appear in the program.

For readability, we will use the underscore symbol (_) to represent a variable that appears only once in a rule.

2.1.3 Spatial and Temporal Extensions

Given a database schema $\mathcal{S}$, we use $\mathcal{S}^+$ to denote the schema obtained as follows. For each relation schema $\mathtt{r}^{(n)} \in (\mathcal{S} \setminus \{<\})$, we include a relation schema $\mathtt{r}^{n+1}$ in $\mathcal{S}^+$. The additional column added to each relation schema is called the *location specifier*. By convention, the location specifier is the first column of the relation. $\mathcal{S}^+$ also includes $<^{(2)}$, and a relation schema $\mathtt{node}^{(1)}$: the finite set of node identifiers that represents all of the nodes in the distributed system. We call $\mathcal{S}^+$ a *spatial schema*.[3]

A *spatial fact* over a relation schema $R^{(n)}$ is a pair consisting of the relation name R and an $(n+1)$-tuple $(d, c_1, \ldots, c_n)$ where each $c_i \in \mathcal{U}$, $d \in \mathcal{U}$, and $\mathtt{node}(d)$. We denote a spatial fact over $R^{(n)}$ by `R(d, c1, ..., cn)`. A *spatial relation instance* for a relation

[1] $\mathbb{N} = \{0, 1, 2, \ldots\}$

[2] We will often write $<$ in infix notation.

[3] The terms *spatial* and *spatio-temporal* are borrowed from Ameloot [7].

schema $\mathtt{r}^{(n)}$ is a set of spatial facts for $\mathtt{r}^{(n+1)}$. A *spatial database instance* is defined similarly to a database instance.

Given a database schema $\mathcal{S}$, we use $\mathcal{S}^*$ to denote the schema obtained as follows. For each relation schema $\mathtt{r}^{(n)} \in (\mathcal{S} \setminus \{<\})$ we include a relation schema $\mathtt{r}^{(n+2)}$ in $\mathcal{S}^*$. The first additional column added is the location specifier, and the second is the *timestamp*. By convention, the location specifier is the first column of every relation in $\mathcal{S}^*$, and the timestamp is the second. $\mathcal{S}^*$ also includes $<^{(2)}$ (finite), $\mathtt{node}^{(1)}$ (finite), $\mathtt{time}^{(1)}$ (infinite) and $\mathtt{timeSucc}^{(2)}$ (infinite), We call $\mathcal{S}^*$ a *spatio-temporal schema*.

A *spatio-temporal fact* over a relation schema $R^{(n)}$ is a pair consisting of the relation name R and an $(n+2)$-tuple $(d, t, c_1, \ldots, c_n)$ where each $c_i \in \mathcal{U}$, $d \in \mathcal{U}, t \in \mathcal{U}$, `node(d)`, and `time(t)`. We denote a spatial fact over $R^{(n)}$ by `R(d, t, c1, ..., cn)`.

A *spatio-temporal relation instance* for relation schema $\mathtt{r}^{(n)}$ is a set of spatio-temporal facts for $\mathtt{r}^{(n+2)}$. A *spatio-temporal database instance* is defined similarly to a database instance; in any spatio-temporal database instance, $\mathtt{time}^{(1)}$ is mapped to the set containing a `time(t)` fact for all $\mathtt{t} \in \mathbb{N}$, and $\mathtt{timeSucc}^{(2)}$ to the set containing a `timeSsucc(x,y)` fact for all $y = x + 1$, $(x, y \in \mathbb{N})$.

We will use the notation `f@t` to mean the spatio-temporal fact obtained from the spatial fact `f` by adding a timestamp column with the constant `t`.

A *spatio-temporal rule* over a spatio-temporal schema $\mathcal{S}^*$ is a rule of one of the following three forms:

1. A *deductive* rule φ:

$$\mathtt{p(L,T,\overline{W})} \leftarrow \mathtt{b}_1\mathtt{(L,T,\overline{X_1})}, \ \ldots, \ \mathtt{b}_l\mathtt{(L,T,\overline{X_l})}, \\ \neg\mathtt{c}_1\mathtt{(L,T,\overline{Y_1})}, \ \ldots, \ \neg\mathtt{c}_m\mathtt{(L,T,\overline{Y_m})}, \ \mathtt{node(L)}, \\ \mathtt{time(T)}, \ \mathit{ineq}(\varphi).$$

2. An *inductive* rule φ:

$$\mathtt{p(L,S,\overline{W})} \leftarrow \mathtt{b}_1\mathtt{(L,T,\overline{X_1})}, \ \ldots, \ \mathtt{b}_l\mathtt{(L,T,\overline{X_l})}, \\ \neg\mathtt{c}_1\mathtt{(L,T,\overline{Y_1})}, \ \ldots, \ \neg\mathtt{c}_m\mathtt{(L,T,\overline{Y_m})}, \ \mathtt{node(L)}, \\ \mathtt{time(T)}, \ \mathtt{timeSucc(T,S)}, \ \mathit{ineq}(\varphi).$$

3. An *asynchronous* rule φ:

$$\mathtt{p(D,S,\overline{W})} \leftarrow \mathtt{b}_1\mathtt{(L,T,\overline{X_1})}, \ \ldots, \ \mathtt{b}_l\mathtt{(L,T,\overline{X_l})}, \\ \neg\mathtt{c}_1\mathtt{(L,T,\overline{Y_1})}, \ \ldots, \ \neg\mathtt{c}_m\mathtt{(L,T,\overline{Y_m})}, \ \mathtt{node(L)}, \\ \mathtt{time(T)}, \ \mathtt{time(S)}, \ \mathtt{choice((L,\ T,\ \overline{B}),(S))}, \\ \mathtt{node(D)}, \ \mathit{ineq}(\varphi).$$

The last two kinds of rules are collectively called *temporal* rules.

In the rules above, $\overline{\mathtt{B}}$ is a tuple that contains all of the distinct variable symbols in $\overline{\mathtt{X_1}}, \ldots, \overline{\mathtt{X_l}}, \overline{\mathtt{Y_1}}, \ldots, \overline{\mathtt{Y_m}}$. The variable symbols `D` and `L` may appear in any of $\overline{\mathtt{W}}, \overline{\mathtt{X_1}}, \ldots, \overline{\mathtt{X_l}}, \overline{\mathtt{Y_1}}, \ldots, \overline{\mathtt{Y_m}}$, whereas `T` and `S` may not. Head relation name `p` may not be `time`, `timeSucc`, or `node`. Relations $\mathtt{b}_1, \ldots, \mathtt{b}_l, \mathtt{c}_1, \ldots, \mathtt{c}_m$ may not be `timeSucc`, `time`, or `<`.

The use of a single location specifier and timestamp in rule bodies intuitively corresponds to considering deductions that can be evaluated at a single node at a single point in time. Inductive rules use the `timeSucc` relation to carry the results of deductions into the next visible timestep.

Note that asynchronous rules are the only kinds of rules with $cho \neq \emptyset$. The `choice` construct is from Saccà and Zaniolo [38]. The $\mathtt{choice((\overline{X}),\ (\overline{Y}))}$ construct represents the constraint that the variables in $\overline{\mathtt{Y}}$ be functionally dependent on the variables in $\overline{\mathtt{X}}$. Due to variable binding restrictions, only asynchronous rules may have a different value for the head location specifier than the body location specifier. Intuitively, different values for the head and body location specifiers represents cross-node communication; a binding of `L`, `T`, and $\overline{\mathtt{B}}$ represents a message being sent from location `L` to location `D`. To model the fact that the network may arbitrarily delay, re-order, and batch messages, any single value of head timestamp `S` is permissible.

We use the causality rewrite of Alvaro *et al*. [3], which introduces the following causality constraint: a message sent by a node x at local timestamp s cannot cause another message to arrive in the past of node x (i.e., at a time before local timestamp s).[4] Intuitively, the causality constraint rules out models corresponding to impossible executions, in which effects are perceived before their causes. Full details are available in a companion paper [3].

A Dedalus *program* is a finite set of causally rewritten spatio-temporal rules over some spatio-temporal schema $\mathcal{S}^*$.

2.1.4 Syntactic Sugar

The restrictions on timestamps and location specifiers suggest a natural syntactic sugar to improve readability. We annotate inductive head relations with `@next` and asynchronous head relations with `@async`; deductive rules have no head annotation. These annotations allow us to omit the boilerplate usage of `node`, `time`, `timeSucc` and `choice` in rule bodies, as well as the timestamp attributes from rule heads and bodies. We also omit location specifiers by default, but refer to them if necessary, as described later. Using this syntactic sugar, below are examples of the three kinds of rules listed above.

Example 2. Example deductive, inductive, and asynchronous rules.

1. Deductive:

$$\mathtt{p(\overline{W})} \leftarrow \mathtt{b}_1\mathtt{(\overline{X_1})}, \ \ldots, \ \mathtt{b}_l\mathtt{(\overline{X_l})}, \ \neg\mathtt{c}_1\mathtt{(\overline{Y_1})}, \ \ldots, \\ \neg\mathtt{c}_m\mathtt{(\overline{Y_m})}.$$

2. Inductive:

$$\mathtt{p(\overline{W})@next} \leftarrow \mathtt{b}_1\mathtt{(\overline{X_1})}, \ \ldots, \ \mathtt{b}_l\mathtt{(\overline{X_l})}, \ \neg\mathtt{c}_1\mathtt{(\overline{Y_1})}, \ \ldots, \\ \neg\mathtt{c}_m\mathtt{(\overline{Y_m})}.$$

3. Asynchronous:

$$\mathtt{p(\overline{W})@async} \leftarrow \mathtt{b}_1\mathtt{(\overline{X_1})}, \ \ldots, \ \mathtt{b}_l\mathtt{(\overline{X_l})}, \ \neg\mathtt{c}_1\mathtt{(\overline{Y_1})}, \ \ldots, \\ \neg\mathtt{c}_m\mathtt{(\overline{Y_m})}.$$

In any kind of rule, the body location specifier can be accessed by including a variable symbol or constant prefixed with `#` as any body atom's first argument. In asynchronous rules only, the head location specifier can be accessed by including a variable symbol or constant prefixed with a `#` as the head atom's first argument. The example below shows an example of `#` in an asynchronous rule.

Example 3. The head and body location specifiers are bound to `D` and `L` respectively. Note how `D` may appear in the body, `L` may appear in the head, and `L` may appear duplicated in the body.

$$\mathtt{p(\#D,L,W)@async} \leftarrow \mathtt{b(\#L,D,W)}, \ \neg\mathtt{c(\#L,L)}.$$

2.2 Semantics

We restrict our attention to Dedalus programs whose deductive rules are syntactically stratified.

[4] Note that in other presentations of Dedalus (e.g., [6]), message timestamps are chosen from $\mathbb{N} \cup \top$, where $\top$ represents a special value indicating that the message was dropped by the network. In this paper, we assume reliable delivery of messages.

An *input schema* $\mathcal{S}^I$ for a DEDALUS program P with spatio-temporal schema $\mathcal{S}^*$ is a subset of P's spatial schema $\mathcal{S}^+$. Every input schema contains the `node` relation; we will not explicitly mention the presence of `node` when detailing an input schema. A relation in $\mathcal{S}^I$ is called an *EDB relation*. All other relations are called *IDB*.

An *EDB instance* $\mathcal{E}$ is a spatial database instance that maps each EDB relation `r` to a finite spatial relation instance for `r`. The *active domain* of an EDB instance $\mathcal{E}$ for a program P is the set of constants appearing in $\mathcal{E}$ and P. Every EDB instance maps the < relation to a total order over its active domain.

We can view an EDB instance as a spatio-temporal database instance $\mathcal{K}$. For every $\texttt{r(d,c}_1\texttt{,...,c}_n\texttt{)} \in \mathcal{E}$, the fact $\texttt{r(d,t,c}_1\texttt{,...,c}_n\texttt{)} \in \mathcal{K}$ for all $\texttt{t} \in \mathbb{N}$. Intuitively, EDB facts "exist at all timesteps."

We refer to a DEDALUS program together with an EDB instance as a DEDALUS *instance*. The semantics of a DEDALUS program can be viewed as a mapping from EDB instances to spatio-temporal database instances.

Recall that `choice` is only used in asynchronous rules, to model the fact that the network may arbitrarily delay, re-order, and batch messages. Saccà and Zaniolo [38] propose the *stable model semantics* as a natural interpretation of `choice`, and we provide the model-theoretic details elsewhere [3]. Intuitively, each stable model is a spatio-temporal database instance that defines a possible function for `choice` that obeys the causality rewrite; every possible function that obeys the causality rewrite defines a stable model. In other words, each different causal choice of timesteps for a DEDALUS instance corresponds to a different stable model of that instance.

Example 4. Take the following DEDALUS program with input schema {`q`}. Assume the EDB instance is {`node(n1), q(n1)`}.

```
p(#L)@async ← q(#L).
```

Let the power set of X be denoted $\mathcal{P}(X)$. For each $S \in \mathcal{P}(\mathbb{N} \setminus \{0\})$, where $|S| = |\mathbb{N}|$, the following is a stable model:

$$\{\texttt{node(n1)}\} \cup \{\texttt{p(n1,i)} \mid \texttt{i} \in S\} \cup \{\texttt{q(n1,i)} \mid \texttt{i} \in \mathbb{N}\}$$

These are the only stable models of the instance. Since `q` is part of the input schema, it is true at every time. Every time involves a separate choice of time for `p`, which must be later than time 0. Elements S of the power set with finite cardinality are ruled out, due to the causality constraint [3].

2.2.1 Ultimate Models

The stable model semantics is a suitable model-theoretic characterization of the behavior of a program in that there is a correspondence between stable models and traces in an operational formalism based on network transducers [3]. However, stable models highlight uninteresting temporal differences that may not be "eventually" significant, such as in the following example:

Example 5. Take the following DEDALUS program with input schema {`q`}. The program determines whether two values, `c1` and `c2` "arrive" at the same time. Assume the EDB instance is {`node(n1), q(n1,c1), q(n1,c2)`}.

```
p(#L,X)@async ← q(#L,X), ¬r(#L,X).
r(X)@next ← q(X).
r(X)@next ← r(X).
concurrent() ← p(n1,c1), p(n1,c2).
concurrent()@next ← concurrent().
```

For each $\texttt{s}, \texttt{t} \in \mathbb{N}$, the following is a stable model:

$$\begin{aligned}
&\{\texttt{q(n1,i,c1)}, \texttt{q(n1,i,c2)} \mid \texttt{i} \in \mathbb{N}\} \cup \\
&\{\texttt{node(n1)}, \texttt{p(n1,s,c1)}, \texttt{p(n1,t,c2)}\} \cup \\
&\{\texttt{r(n1,i,c1)}, \texttt{r(n1,i,c2)} \mid \texttt{i} \in \mathbb{N} \setminus \{0\}\} \\
&\{\texttt{concurrent(n1,i)} \mid \texttt{i} \in \mathbb{N} \wedge \texttt{s} \leq \texttt{i}\} \text{ if } \texttt{s} = \texttt{t} \cup
\end{aligned}$$

These are the only stable models of the instance. Since `q` is part of the input schema, `q` facts are true at every time. By the rules, `r` facts are true at every time except time `0`. Thus, there is only one choice of head timestamp for `p` for each value of `q`'s second argument—this choice corresponds with a body timestamp of `0`. If these choices are the same, then `concurrent()` is true at all timestamps afterwards.

However, note that while the specific values of `s` and `t` are unimportant in terms of the eventual contents of the `concurrent` relation, there are different stable models for each of these choices. Intuitively, we do not want these "intermediate" temporal behaviors that are not eventually significant, to differentiate program outputs.

In order to rule out such behaviors from the output, we will define the concept of an *ultimate model* to represent a program's "output."

An *output schema* for a DEDALUS program P with spatio-temporal schema $\mathcal{S}^*$ is a subset of P's spatial schema $\mathcal{S}^+$. We denote the output schema as $\mathcal{S}^O$.

Recall that a stable model defines a spatio-temporal database instance, which is a mapping from every relation `r` in $\mathcal{S}^*$ to a spatio-temporal relation instance for `r`, which itself is a set of spatio-temporal facts for `r`. We define the *eventually always true* function $\Diamond\Box$, which maps a spatio-temporal database instance $\mathcal{T}$ to a spatial database instance $\Diamond\Box\mathcal{T}$. For every spatio-temporal fact $\texttt{r(p,t,c}_1\texttt{,...,c}_n\texttt{)} \in \mathcal{T}$, the spatial fact $\texttt{r(p,c}_1\texttt{,...,c}_n\texttt{)} \in \Diamond\Box\mathcal{T}$ if relation `r` is in $\mathcal{S}^O$ and $\forall \texttt{s}.\, (\texttt{s} \in \mathbb{N} \wedge \texttt{t} < \texttt{s}) \Rightarrow (\texttt{r(p,s,c}_1\texttt{,...,c}_n\texttt{)} \in \mathcal{T})$.

The set of *ultimate models* of a DEDALUS instance I is $\{\Diamond\Box(\mathcal{T}) \mid \mathcal{T}$ is a stable model of I$\}$. Intuitively, an ultimate model contains exactly the facts in relations in the output schema that are eventually always true in a stable model.

Note that an ultimate model is always finite because of the finiteness of the EDB, the safety conditions on rules, the restrictions on the use of `timeSucc` and `time`, and the prohibition on binding timestamps to non-timestamp attributes. A DEDALUS program only has a finite number of ultimate models for the same reason.

Example 6. For Example 4 with $\mathcal{S}^O = \{\texttt{p}\}$, there are two ultimate models: {} and {`p(n1)`}. The latter corresponds to an element of the power set S such that $\exists x.\, \forall y.\, (y > x) \Rightarrow (y \in S)$. The former corresponds to an element S that does not have this property.

For Example 5 with $\mathcal{S}^O = \{\texttt{concurrent()}\}$, there are two ultimate models: {} and {`concurrent()`}. The former corresponds to choices of timestamp for `c1` and `c2` that are not equal, whereas the latter corresponds to equal choices of timestamp.

3. REFINING DEDALUS

DEDALUS can express a broad class of distributed systems but this flexibility comes at a cost. As we have shown, a DEDALUS program may have multiple ultimate models. However, it is often desirable to ensure that a program has a single, deterministic output, regardless of non-determinism in its behavior.

Having defined the DEDALUS language, we will refer to two running examples for the remainder of the paper.

Example 7. A simple asynchronous marriage ceremony:

```
groom_i_do()@async ← groom_i_do_edb().
bride_i_do()@async ← bride_i_do_edb().
runaway() ← ¬bride_i_do(), groom_i_do().
runaway() ← ¬groom_i_do(), bride_i_do().
runaway()@next ← runaway().
groom_i_do()@next ← groom_i_do().
bride_i_do()@next ← bride_i_do().
```

In a classic paper, Gray notes the similarity between distributed commit protocols and marriage ceremonies [22]. For simplicity (and felicity), Example 7 presents a simple asynchronous voting program with a fixed set of members: a bride and a groom. The marriage is off (`runaway()` is true) if one party says "I do" and the other does not.

However, the DEDALUS program as given does not correctly implement such a vote. Any stable model where `groom_i_do()` and `bride_i_do()` disagree in their first chosen timestamps yields an ultimate model containing `runaway()`. By contrast, if the votes are assigned the same timestamp, the ultimate model does not contain `runaway()`. In operational terms, this program exhibits a race condition: when the EDB contains "I do" votes from both parties, the truth value of `runaway()` depends on the (non-deterministic) times at which their messages are delivered.

Example 8. Distributed garbage collection:

```
addr(Addr)@async ← addr_edb(Addr).
refers_to(#M, Src, Dst)@async ←
        local_ptr_edb(#N, Src, Dst), master(#M).
refers_to(Src, Dst)@next ← refers_to(Src, Dst).
reach(Src, Dst) ← refers_to(Src, Dst).
reach(Src, Next) ← reach(Src, Dst),
        refers_to(Dst, Next).
garbage(Addr) ← addr(Addr), root_edb(Root),
        ¬reach(Root, Addr).
garbage(Addr)@next ← garbage(Addr).
```

Example 8 presents a simple garbage collection program for a distributed memory system. Each node manages a set of pointers and forwards this information to a central master node. The master computes the set of transitively reachable addresses; if an address is not reachable from the root address, it can be garbage collected. For simplicity, we assume that each node owns a fixed set of pointers, stored in the EDB relation `local_ptr_edb`.

This more complicated example suffers from the same ambiguity as the marriage ceremony presented previously. While the program has an ultimate model corresponding to executions in which `garbage` is not computed until the transitive closure of `refers_to` has been fully determined (i.e., after all messages have been delivered), it also has ultimate models corresponding to executions in which `garbage` is "prematurely" computed. When `garbage` is computed before all the `refers_to` messages have been delivered, there is a correctness violation: reachable memory addresses appear in the `garbage` relation.

Note that for both examples, there is a single ultimate model corresponding to the execution in which negation is not applied to a set until the content of the set has been fully determined. This "preferred" model is akin to the perfect model computed by a centralized Datalog evaluator that evaluates rules in stratum order [41], applying the closed-world assumption to relations only when it is certain that they will no longer change. Unfortunately, in an asynchronous distributed system it is difficult to distinguish the absence of a message (e.g., the `bride_i_do` or some expected `refers_to` messages) from channel delay. Hence both programs above are underspecified insofar as they conclude, as soon as they receive any messages, that no others will arrive. In practice, a programmer could remediate the problem by augmenting their programs with *coordination* code that enforces a computation barrier. This technique generally entails a protocol (e.g., voting or consensus) that takes place between all communicating agents to ensure that there are no outstanding messages in flight.

In the remainder of this section, we explore the aspects of DEDALUS that allow such ambiguities and propose a restricted language DEDALUS$^+$ that rules them out (but complicates the specification of programs like our examples above). In Section 4, we consider a different language—DEDALUS^S—that allows relatively intuitive program specifications like our examples, but narrows their interpretation to a single, "preferred" model.

3.1 Problems with DEDALUS

Definition 1. A DEDALUS program is *confluent* if, for every EDB instance, it has a single ultimate model. A program that is not confluent is *diffluent*.

Confluence is a desirable, albeit conservative, correctness property for a distributed program. A program that is confluent produces deterministic output despite any non-deterministic behaviors that might occur during its execution. For example, if we could show that a data replication protocol was confluent, we could prove a version of the commonly desired property that all replicas be "eventually consistent" after all messages have been delivered [40, 39]. Confluence may be viewed as a specialization of the more general notion of consistency of distributed state, which the CALM theorem [24] argues is strongly connected with the model-theoretic property of logical monotonicity.

Unfortunately, confluence is an undecidable property of DEDALUS programs:

LEMMA 1. *Confluence of* DEDALUS *programs is undecidable.*

This result is perhaps not surprising, as confluence is defined over all EDB instances. We present a proof in the appendix.

Another symptom of DEDALUS being "too big" a language is its expressive power: it subsumes PSPACE.

LEMMA 2. DEDALUS *subsumes PSPACE.*

PROOF. We show how to write the PSPACE-complete Quantified Boolean Formula (QBF) problem [20] in DEDALUS. Since DEDALUS is closed under first-order reductions and QBF is PSPACE-complete under first-order reductions, we have that PSPACE $\subseteq$ DEDALUS. Details are in the appendix. □

3.2 DEDALUS$^+$

Distributed programs that produce non-deterministic outputs or have runtimes exponential in their inputs are often undesirable in practice. Since checking for confluence in DEDALUS is undecidable in general, we may instead ask whether a more constrained language will exclude such undesirable programs. We will present a restriction of DEDALUS that allows only confluent programs and prove that it captures exactly PTIME.

Definition 2. A DEDALUS program is *semipositive* if the ¬ symbol only appears on EDB relations in the program.

Definition 3. A DEDALUS program P has *guarded asynchrony* if for every relation `p` appearing in the head of an asynchronous rule, the program P has a rule `p(`$\overline{\texttt{X}}$`)@next ← p(`$\overline{\texttt{X}}$`).`

We will refer to the language of semipositive DEDALUS programs with guarded asynchrony as DEDALUS$^+$.

3.2.1 Confluence

To show that all Dedalus$^+$ programs are confluent, we begin by showing that Dedalus$^+$ programs are *temporally inflationary*: if a stable model of a Dedalus$^+$ instance contains a spatio-temporal fact `f@t`, then it also contains `f@t+1` (and thus the ultimate model contains `f`).

Lemma 3. Dedalus$^+$ *programs are temporally inflationary.*

Proof. Consider a *derivation tree* for `f@t`: a finite tree of instantiated (variable-free) rules, where negation only occurs at the leaves. Note that the instantiated head atom, as well as every instantiated body relation, is a spatio-temporal fact. The tree's root is some instantiated rule with `f@t` in its head. A node has one child node for each body fact: the child node contains an instantiated rule with the fact in its head—if the body fact's relation does not appear in the head of any rule, then the corresponding node contains just the fact, and is a leaf node. The leaves of the tree are instantiated EDB facts.

For the moment, we assume that every fact has a unique derivation tree. Multiple derivation trees are easy to handle—simply repeat the following process for each tree.

If the relation of `f` is EDB, or appears in the head of an asynchronous rule, then the lemma holds by definition of Dedalus$^+$. Assume some stable model contains `f@t` and not `f@t+1`. Thus, if the rule is inductive (resp. deductive), then for some child of `f@t`, call it `g@t-1` (resp. `g@t`), the fact `g@t` (resp. `g@t+1`) is not in the stable model. Inductively proceed down the tree, at each step going to a node whose relation does not appear in the head of an asynchronous rule. However, the path will eventually terminate at a leaf node providing a contradiction, because facts at leaf nodes are either EDB or negated EDB, meaning that they exist at all timestamps, or they are one of `time`, `timeSucc`, or <, which also exist at all timestamps. □

A consequence of temporal inflation is that all Dedalus$^+$ programs are confluent.

Theorem 1. Dedalus$^+$ *programs are confluent.*

Proof. Towards a proof by contradiction, consider a Dedalus$^+$ program that induces two ultimate models $\mathcal{U}_1, \mathcal{U}_2$ for some EDB. Without loss of generality, there must be a spatial fact `f`, such that $\texttt{f} \in \mathcal{U}_1$ and $\texttt{f} \notin \mathcal{U}_2$.

Recall that if spatial fact `f` is in some ultimate model, then for some $\texttt{t}_0 \in \mathbb{N}$, there is some stable model that contains `f@t` for all $\texttt{t} > \texttt{t}_0$.

Consider a derivation tree for $\texttt{f@t}_0$ in any stable model that yields $\mathcal{U}_1$. Again, for simplicity, we assume uniqueness of this derivation tree. For some child of $\texttt{f@t}_0$, call it `g@s`, for all stable models that yield $\mathcal{U}_2$ there is no `r` such that `g@r` is in the stable model by Lemma 3. Continue traversing the tree, at each step picking such a `g`. Eventually, the traversal terminates at an EDB node, leading to a contradiction. □

Since a Dedalus$^+$ program has a unique ultimate model, the specific `choice` of values for timestamps does not affect the ultimate model. In particular, we can assume that the `timeSucc` of the body timestamp is always chosen:

Corollary 1. *Define the program transformation $\mathcal{A}(P)$ to be the transformation that, converts every asynchronous rule φ of* Dedalus$^+$ *program P into an inductive rule by undoing the causality and* `choice` *rewrites, dropping the* `choice` *operator, and adding* `timeSucc(T,S)` *to pos(φ). Then, the ultimate model of $\mathcal{A}(P)$ is the same as the ultimate model of P.*

Of course, there are confluent Dedalus programs not in Dedalus$^+$. For example:

Example 9. A confluent Dedalus program that is not a Dedalus$^+$ program.

```
b(#N, I)@async ← b_edb(#L, I).
b(I)@next ← b(I), ¬dequeued(I).
b_lt(I, J) ← b(I), b(J), I < J.
dequeued(I)@next ← b(I), ¬b_lt(_, I),
        b_lt(_, _).
```

Any instance of this program has a single ultimate model in which `b()` (at all nodes) contains the highest element in `b_edb()` according to the order <. Thus it is confluent, but the program uses IDB negation and does not have guarded asynchrony.

3.2.2 Computational Complexity

Not only are Dedalus$^+$ programs confluent, but they also capture exactly PTIME. We will prove this by showing an equivalence to semipositive Datalog programs, which are known to capture exactly PTIME over ordered structures [26].

First, we note that inductive rules in Dedalus$^+$ can be "converted" into deductive rules without affecting the ultimate model.

Lemma 4. *Define the program transformation $\mathcal{I}(P)$ in the following way: in every inductive rule of* Dedalus$^+$ *program P—except any basic persistence rule for a relation that appears in the head of an asynchronous rule—remove the* `timeSucc(T,S)` *body atom, and replace all instances of the variable* `S` *with the variable* `T`*. The ultimate model of $\mathcal{I}(P)$ is the same as the ultimate model of P.*

Proof. Note that by Lemma 3, $\mathcal{I}(P)$ is inflationary. The proof proceeds similarly to the proof of Lemma 3—there is some fact in $\mathcal{U}_1$ but not $\mathcal{U}_2$; we consider a derivation tree for this fact in any stable model; it must be the case that some child fact of the parent does not appear in any stable model for $\mathcal{U}_2$ (by Lemma 3). We inductively repeat the procedure, and discover that in order for the fact to be absent from $\mathcal{U}_1$, the EDB must be different, which is a contradiction. □

Theorem 2. Dedalus$^+$ *captures exactly PTIME.*

Proof. First we apply Corollary 1 to rewrite asynchronous rules as inductive rules. Then, we convert all inductive rules into deductive rules using Lemma 4. Since all rules are deductive, there is a unique stable model, which is also the same for every timestamp.

Consider removing the timestamp attributes from all relations, and thus the `time` relations from the bodies of all rules. The result is a Datalog program with EDB negation. Its minimal model is exactly the ultimate model of the single-timestep Dedalus$^+$ program.

In the other direction, it is clear that we can encode any Datalog program with EDB negation in Dedalus$^+$ using deductive rules; the ultimate model coincides with the minimal model of the Datalog program. □

4. Dedalus^S

Returning to our running examples, it is easy to see that neither program is directly expressible in Dedalus$^+$. The marriage program from Example 7 uses IDB negation to determine the truth value of `runaway`. To avoid using IDB negation, we can rewrite the program to "push down" negation to the EDB relations `groom_i_do` and `bride_i_do`, and then derive the `runaway` IDB relation positively as shown in Example 10. While the rewrite is straightforward, a majority of the program's rules need to be modified. Note that since

Example 10 is written in Dedalus$^+$, the program must be confluent; therefore, it is not subject to the non-deterministic output observed for the original marriage program (Example 7).

Example 10. An asynchronous marriage ceremony without IDB negation:

```
groom_i_dont()@async ← ¬groom_i_do_edb().
bride_i_dont()@async ← ¬bride_i_do_edb().
runaway() ← groom_i_dont().
runaway() ← bride_i_dont().
runaway()@next ← runaway().
groom_i_dont()@next ← groom_i_dont().
bride_i_dont()@next ← bride_i_dont().
```

The garbage collection program from Example 8 is likewise outside Dedalus$^+$ due to IDB negation but it presents a rather more difficult problem, as negation must be pushed down through recursion. The rules for positively computing the negation of a transitive closure are not particularly intuitive, and expressing the negation of an arbitrary recursive computation is even more difficult [25]. Furthermore, the best known strategies involve at least a doubling in the arity of the relations.

In general, the restriction of negation to EDB relations presents a significant barrier to expressing practical programs. In this section, we introduce Dedalus^S, an extension of Dedalus$^+$ that allows a limited form of IDB negation but retains the benefits of Dedalus$^+$: Dedalus^S also captures PTIME exactly and allows only confluent programs. We show that Dedalus^S and Dedalus$^+$ are equivalently expressive. Then we provide an operational semantics for Dedalus^S, based on the one for Dedalus [3], inspired by coordination protocols from distributed systems.

4.1 Safe IDB Negation

The stratified semantics for logic programs with negation is both intuitive and corresponds to common distributed systems practices: negation is not applied until the negated relation is "done" being computed.

First, we define a *predicate dependency graph* (PDG). The PDG of a Dedalus program P with spatio-temporal schema $\mathcal{S}^*$ is a directed graph with one node per relation; each node i has a label $L(i)$. If node i represents relation `p`, then $L(i)$ = `p`. There is an edge from the node with label `q` to the node with label `p` if relation `p` appears in the head of a rule with `q` in its body. If some rule with `p` in the head and `q` in the body is asynchronous (resp. inductive), then the edge is said to be *asynchronous* (resp. *inductive*). If some rule with `p` in the head has ¬`q` in its body, then the edge is said to be *negated*. Collectively, asynchronous and inductive edges are referred to as *temporal edges*. The PDG does not contain nodes for the `node`, `timeSucc`, or `time` relations, or any relation introduced in the causality [3] or `choice` [38] rewrites.

Dedalus^S is the language of Dedalus programs with guarded asynchrony whose PDG does not contain any cycles through negation. As is standard, a Dedalus^S program can be partitioned into *strata*. The *stratum* of a relation `r` is the largest number of negated edges on any path from `r`.

Each stratum of an n-stratum Dedalus^S program can be viewed as a Dedalus$^+$ program. Stratum i's program, P_i, consists of all rules whose head relation is in stratum i. The output schema of P_i contains all relations in stratum $i+1$, and P_i's EDB contains all relations in stratum $j < i$. P_0's EDB contains all EDB relations. P_n's output schema contains all relations in P's output schema.

The *ultimate model* of a Dedalus^S program is the ultimate model $P_n(\ldots P_1(P_0(E))\ldots)$.

Since a Dedalus^S program is a straightforward composition of Dedalus$^+$ programs, we can apply several previous results. Note that Dedalus^S programs are temporally inflationary.

Corollary 2. *Dedalus^S programs are confluent.*

Note that every Dedalus$^+$ program is a Dedalus^S program, and every Dedalus^S program has a constant number of strata in the size of its input. Thus we have:

Corollary 3. *Dedalus^S programs capture exactly PTIME.*

Thus, Dedalus^S maintains the desirable properties of Dedalus$^+$: it is both confluent and PTIME.

4.2 Coordination rewrite

While the model-theoretic semantics of Dedalus^S are clear, its negation semantics are different than those of Dedalus. Thus, we cannot directly apply the correspondence to a distributed operational semantics in Alvaro *et al.* [3]. Fortunately, we can rewrite any Dedalus^S program to a Dedalus program.

Given a Dedalus^S program S, the *coordination rewrite* $\mathcal{P}(S)$ of S is the Dedalus program obtained by adding `p_done()` to the body of any rule in S that contains a ¬`p(...)` atom and adding rules to define `p_done()` as described below.

We will see that `p_done()` has the property that in any stable model $\mathcal{M}$ if `p_done(l,t)` $\in \mathcal{M}$, then `p_done(l,s)` $\in \mathcal{M}$ for all timestamps `s` > `t`. Furthermore, if `p_done(l,t)` $\in \mathcal{M}$, then `p(l,s,`c_1`,...,`c_n`)` $\in \mathcal{M}$ implies that `p(l,t,`c_1`,...,`c_n`)` $\in \mathcal{M}$ for all timestamps `s` > `t`. Intuitively, `p_done()` is true when the content of `p` is *sealed* (henceforth unchanging).

We will present a specification of `p_done()` after introducing some preliminary definitions.

A *collapsed PDG* of a Dedalus program P is the graph obtained by replacing each strongly connected component of the PDG of P with a single node i, such that $L(i)$ comprises the set of all relations from the component. If a strongly connected component has any asynchronous edges, we call the resulting collapsed node *async recursive*. Each node in the collapsed PDG whose label contains a relation names in $\mathcal{S}^O$ is called an *output* node. Note that a collapsed PDG is acyclic.

For EDB relations `p`, the rule for `p_done` is `p_done().`[5] For IDB relations, defining `p_done()` takes some work. Intuitively, `p_done()` for `p` $\in L(i)$ directly depends on `r_done()` for any `r` in the body of a rule with `p` in the head. Additionally, asynchronous rules take some care—while deductively defined relations are done in the same timestamp as all relations they depend on, there may be arbitrary delay before asynchronously defined relations are done.

For ease of exposition, we will first present the computation of `p_done()` for `p` in non-async-recursive nodes. We will then explain how to support async recursive nodes. We assume that all inductive rules have been rewritten to deductive rules (Lemma 4).

4.2.1 Non-Async-Recursive Nodes

For non-async-recursive nodes, we can compute a `done` fact for each rule, then collate these into `done` facts for each relation. We handle deductive and asynchronous rules separately. The `done` fact for a deductive rule is true when all of the relations in the body of the rule are henceforth unchanging. The `done` fact for an asynchronous rule is henceforth true at some local timestamp after all facts derived in the head relation are true at their respective locations. We assume guarded asynchrony applies to the rules in this section.

[5] This expression is actually a rule. Consider the unsugared form: `p_done(L,T) ← node(L), time(T).`

Let i be a non-async-recursive node. Repeat the following for each element of $\mathtt{p} \in L(i)$. Assume the rules in P with head relation $\mathtt{p}$ are numbered $1, \ldots, i_p$. The rule for `p_done()` is:

$$\mathtt{p\_done()} \leftarrow \mathtt{r}_1\mathtt{\_done()},\ \ldots,\ \mathtt{r}_{i_p}\mathtt{\_done()}.$$

Let the nodes in the collapsed PDG connected via incoming edges to node i be denoted by $E(i)$. Let the relations $\bigcup_{k \in E(i)} L(k)$ be named $\mathtt{p}_1, \ldots, \mathtt{p}_{i_q}$.

For each rule $1 \leq j \leq i_p$ in P with head relation $\mathtt{p}$, if j is:

Deductive: Add the rule:

$$\mathtt{r}_j\mathtt{\_done()} \leftarrow \mathtt{p}_1\mathtt{\_done()},\ \ldots,\ \mathtt{p}_{i_q}\mathtt{\_done()}.$$

Asynchronous: For each asynchronous rule:

$$\mathtt{p(\#N,}\overline{\mathtt{W}}\mathtt{)@async} \leftarrow \mathtt{b}_1\mathtt{(\#L,}\overline{\mathtt{X}_1}\mathtt{)},\ \ldots,\ \mathtt{b}_l\mathtt{(\#L,}\overline{\mathtt{X}_l}\mathtt{)},\ \neg\mathtt{c}_1\mathtt{(\#L,}\overline{\mathtt{Y}_1}\mathtt{)},\ \ldots,\ \neg\mathtt{c}_m\mathtt{(\#L,}\overline{\mathtt{Y}_m}\mathtt{)}.$$

add the following set of rules:

$$\mathtt{p}_j\mathtt{\_to\_send(N,}\overline{\mathtt{W}}\mathtt{)} \leftarrow \mathtt{b}_1\mathtt{(\#L,}\overline{\mathtt{X}_1}\mathtt{)},\ \ldots,\ \mathtt{b}_l\mathtt{(\#L,}\overline{\mathtt{X}_l}\mathtt{)},\ \neg\mathtt{c}_1\mathtt{(\#L,}\overline{\mathtt{Y}_1}\mathtt{)},\ \ldots,\ \neg\mathtt{c}_m\mathtt{(\#L,}\overline{\mathtt{Y}_m}\mathtt{)}.$$

$$\mathtt{p}_j\mathtt{\_to\_send\_done()} \leftarrow \mathtt{b}_1\mathtt{\_done()},\ \ldots,\ \mathtt{b}_l\mathtt{\_done()},\ \mathtt{c}_1\mathtt{\_done()},\ \ldots,\ \mathtt{c}_m\mathtt{\_done()}.$$

$$\mathtt{p}_j\mathtt{\_send(\#N,L,}\overline{\mathtt{X}}\mathtt{)@async} \leftarrow \mathtt{p}_j\mathtt{\_to\_send(\#L,N,}\overline{\mathtt{X}}\mathtt{)}.$$

$$\mathtt{p}_j\mathtt{\_ack(\#N,L,}\overline{\mathtt{X}}\mathtt{)@async} \leftarrow \mathtt{p}_j\mathtt{\_send(\#L,N,}\overline{\mathtt{X}}\mathtt{)}.$$

$$\mathtt{r}_j\mathtt{\_done\_node(\#L,N)@async} \leftarrow \mathtt{p}_1\mathtt{\_done(\#N)},\ \ldots,\ \mathtt{p}_{i_q}\mathtt{\_done(\#N)},\ \left(\forall\overline{\mathtt{X}}.\mathtt{p}_j\mathtt{\_to\_send(\#N,L,}\overline{\mathtt{X}}\mathtt{)} \Rightarrow \mathtt{p}_j\mathtt{\_ack(\#N,L,}\overline{\mathtt{X}}\mathtt{)}\right).$$

$$\mathtt{r}_j\mathtt{\_done()} \leftarrow \left(\forall\mathtt{N}.\mathtt{node(N)} \Rightarrow \mathtt{r}_j\mathtt{\_done\_node(N)}\right).$$

The first rule stores messages to be sent at the body (source's) location specifier, so the source can check whether all messages have been acknowledged. The original destination location specifier is stored as an ordinary column in the $\mathtt{p}_j$`_to_send` relation (indicated by the absence of `#`). Note that because this first rule is a deductive rule, as well as the only rule defining $\mathtt{p}_j$`_to_send`, the $\mathtt{p}_j$`_to_send` relation is done at the same time as the body relations of the first rule, as shown in the second rule. The third rule copies messages to the correct destination location specifier, while including the location specifier of the source (`L`). The fourth derives acknowledgments at the source's location specifier. The fifth rule (at the source) derives a $\mathtt{r}_j$`_done_node` fact at a node when the source has an $\mathtt{p}_j$`_ack` for each $\mathtt{p}_j$`_send`. Note that the causality constraint ensures that the timestamp chosen for each $\mathtt{r}_j$`_done_node` message is greater than any timestamp before the stable model satisfies the body of the rule. The final rule (at the destination) asserts that rule j is done once $\mathtt{r}_j$`_done_node` has been received from all nodes—intuitively, the rule is done when all messages from all nodes have been received.

The formula $\forall\overline{\mathtt{X}}.\phi(\overline{\mathtt{W}},\overline{\mathtt{X}})$ where $\phi(\overline{\mathtt{W}},\overline{\mathtt{X}})$ is of the form $\mathtt{p}(\overline{\mathtt{W}},\overline{\mathtt{X}}) \Rightarrow \mathtt{q}(\overline{\mathtt{W}},\overline{\mathtt{X}})$ translates to $\mathtt{forall}_\phi(\overline{\mathtt{W}})$, and the following rules are added:

$$\mathtt{p}_\phi\mathtt{\_min}(\overline{\mathtt{W}},\overline{\mathtt{X}}) \leftarrow \mathtt{p}(\overline{\mathtt{W}},\overline{\mathtt{X}}),\ \neg\mathtt{p}_\phi\mathtt{\_succ}(\overline{\mathtt{W}},\overline{\_},\overline{\mathtt{X}}),\ \mathtt{p}_\phi\mathtt{\_succ\_done()}.$$

$$\mathtt{p}_\phi\mathtt{\_max}(\overline{\mathtt{W}},\overline{\mathtt{X}}) \leftarrow \mathtt{p}(\overline{\mathtt{W}},\overline{\mathtt{X}}),\ \neg\mathtt{p}_\phi\mathtt{\_succ}(\overline{\mathtt{W}},\overline{\mathtt{X}},\overline{\_}),\ \mathtt{p}_\phi\mathtt{\_succ\_done()}.$$

$$\mathtt{p}_\phi\mathtt{\_succ}(\overline{\mathtt{W}},\overline{\mathtt{X}},\overline{\mathtt{Y}}) \leftarrow \mathtt{p}(\overline{\mathtt{W}},\overline{\mathtt{X}}),\ \mathtt{p}(\overline{\mathtt{W}},\overline{\mathtt{Y}}),\ \overline{\mathtt{X}} < \overline{\mathtt{Y}},\ \neg\mathtt{p}_\phi\mathtt{\_not\_succ}(\overline{\mathtt{W}},\overline{\mathtt{X}},\overline{\mathtt{Y}}),\ \mathtt{p}_\phi\mathtt{\_not\_succ\_done()}.$$

$$\mathtt{p}_\phi\mathtt{\_not\_succ}(\overline{\mathtt{W}},\overline{\mathtt{X}},\overline{\mathtt{Y}}) \leftarrow \mathtt{p}(\overline{\mathtt{W}},\overline{\mathtt{X}}),\ \mathtt{p}(\overline{\mathtt{W}},\overline{\mathtt{Y}}),\ \mathtt{p}(\overline{\mathtt{W}},\overline{\mathtt{Z}}),\ \overline{\mathtt{X}} < \overline{\mathtt{Z}},\ \overline{\mathtt{Z}} < \overline{\mathtt{Y}}.$$

$$\mathtt{forall}_\phi\mathtt{\_ind}(\overline{\mathtt{W}},\overline{\mathtt{X}}) \leftarrow \mathtt{p}_\phi\mathtt{\_min}(\overline{\mathtt{W}},\overline{\mathtt{X}}),\ \mathtt{q}(\overline{\mathtt{W}},\overline{\mathtt{X}}).$$

$$\mathtt{forall}_\phi\mathtt{\_ind}(\overline{\mathtt{W}},\overline{\mathtt{X}}) \leftarrow \mathtt{forall}_\phi\mathtt{\_ind}(\overline{\mathtt{W}},\overline{\mathtt{Y}}),\ \mathtt{p}_\phi\mathtt{\_succ}(\overline{\mathtt{W}},\overline{\mathtt{Y}},\overline{\mathtt{X}}),\ \mathtt{q}(\overline{\mathtt{W}},\overline{\mathtt{X}}).$$

$$\mathtt{forall}_\phi(\overline{\mathtt{W}}) \leftarrow \mathtt{forall}_\phi\mathtt{\_ind}(\overline{\mathtt{W}},\overline{\mathtt{X}}),\ \mathtt{p}_\phi\mathtt{\_max}(\overline{\mathtt{W}},\overline{\mathtt{X}}).$$

The first four rules above compute a total order over the facts in $\mathtt{p}_\phi$. The final three rules iterate over the total order of $\mathtt{p}_\phi$, and checking each $\mathtt{p}_\phi$ to see if `q` also holds. If `q` does not hold for any `p`, iteration will cease. However, if `q` holds for all `p` then $\mathtt{forall}_\phi$ is true.

We additionally need to add a rule for the vacuous case of the universal quantification. In general, we cannot write $\mathtt{forall}_\phi(\overline{\mathtt{W}}) \leftarrow \neg\mathtt{p}(\overline{\mathtt{W}},\overline{\_}),\ \mathtt{p\_done()}.$, because the variables in $\overline{\mathtt{W}}$ do not obey our safety restrictions. Thus, for every rule r that contains $\forall\overline{\mathtt{X}}.\phi(\overline{\mathtt{W}},\overline{\mathtt{X}})$ in its body, we must duplicate r, replacing the $\forall$ clause with the atom $\neg\mathtt{p}(\overline{\mathtt{W}},\ \overline{\_})$.

Note also that we are abusing notation for the $<$ relation. We previously defined $<$ as a binary relation, but it is easy to define a $2n$-ary version of $<$ that encodes a lexicographic ordering over n-ary relations. Here, we use $<$ to refer to the latter.

4.2.2 Async Recursive Nodes

The difficulty with a relation `p` in an async recursive node is that `r` is done when all messages have been received in the node, and all messages have been received if `p` is done. To circumvent this circular dependency, we introduce a specialized two-phase voting protocol.

Consider an async recursive node i.

Let the asynchronous rules with head relations in $L(i)$ be numbered $1, \ldots, i_p$. Add the rule:

$$\mathtt{all\_ack}_i\mathtt{()} \leftarrow \mathtt{r}_1\mathtt{\_done()},\ \ldots,\ \mathtt{r}_{i_p}\mathtt{\_done()}.$$

For each rule j, add the rules for asynchronous rules in the previous section, except for the last two rules. Instead write:

$$\mathtt{r}_j\mathtt{\_not\_done()} \leftarrow \mathtt{p}_j\mathtt{\_to\_send}(\overline{\mathtt{X}}),\ \neg\mathtt{p}_j\mathtt{\_ack}(\overline{\mathtt{X}}).$$

$$\mathtt{r}_j\mathtt{\_done()} \leftarrow \neg\mathtt{r}_j\mathtt{\_not\_done()}.$$

We perform a two-round voting protocol among the nodes; the node with the minimum identifier is the master. We assume that guarded asynchrony does not apply to the relations that appear in the head of any asynchronous rule in the following protocol. The rules shown below begin the first round of voting. Nodes vote $\mathtt{complete\_1}_i$ if $\mathtt{all\_ack}_i$ is true—intuitively, if they have no outstanding unacknowledged messages. Votes are sent to the node with minimum identifier (the master).

$$\mathtt{not\_node\_min(L1)} \leftarrow \mathtt{node(L1)},\ \mathtt{node(L2)},\ \mathtt{L2} < \mathtt{L1}.$$

$$\mathtt{node\_min(L)} \leftarrow \neg\mathtt{not\_node\_min(L)},\ \mathtt{node(L)}.$$

$$\mathtt{start\_round\_1}_i\mathtt{()} \leftarrow \mathtt{node\_min(\#L,L)},\ \neg\mathtt{round\_1}_i\mathtt{()}.$$

$$\mathtt{round\_1}_i\mathtt{()@next} \leftarrow \mathtt{start\_round\_1}_i\mathtt{()}.$$

$$\mathtt{round\_1}_i\mathtt{()@next} \leftarrow \mathtt{round\_1}_i\mathtt{()},\ \neg\mathtt{start\_round\_2}_i\mathtt{()}.$$

$$\mathtt{vote\_1}_i\mathtt{(\#N)@async} \leftarrow \mathtt{start\_round\_1}_i\mathtt{()},\ \mathtt{node(N)}.$$

$$\mathtt{complete\_1}_i\mathtt{(\#M,N)@async} \leftarrow \mathtt{vote\_1}_i\mathtt{(\#N)},\ \mathtt{all\_ack}_i\mathtt{(\#N)},\ \mathtt{node\_min(\#N,M)}.$$

$$\mathtt{incomplete\_1}_i\mathtt{(\#M,N)@async} \leftarrow \mathtt{vote\_1}_i\mathtt{(\#N)},\ \neg\mathtt{all\_ack}_i\mathtt{(\#N)},\ \mathtt{node\_min(\#N,M)}.$$

To persist votes until round 1 begins again, the following rules are instantiated for $k = 1$ and 2.

$$\mathtt{complete\_k}_i\mathtt{(N)@next} \leftarrow \mathtt{complete\_k}_i\mathtt{(N)},\ \neg\mathtt{start\_round\_1}_i\mathtt{()}.$$

$$\mathtt{incomplete\_k}_i\mathtt{(N)@next} \leftarrow \mathtt{incomplete\_k(N)},\ \neg\mathtt{start\_round\_1}_i\mathtt{()}.$$

To count votes, we assume the following rules are instantiated for $k = 1$ and 2. Round 1 is restarted if some node votes $\mathtt{incomplete\_1}_i$ in round 1—i.e., it has an outstanding unacknowledged message – or $\mathtt{incomplete\_2}_i$ in round 2.

```
recv_k_i(N) ← complete_k_i(N).
recv_k_i(N) ← incomplete_k_i(N).
not_all_recv_k_i() ← node(N), ¬recv_k_i(N).
not_all_comp_k_i() ← node(N), ¬complete_k_i(N).
start_round_1_i() ← ¬not_all_recv_k_i(),
        not_all_comp_k_i().
```

Once a node has received a $\texttt{vote\_1}_i$ vote solicitation, it also begins keeping track of whether it has sent any messages in the async recursive component; this information is erased if another $\texttt{vote\_1}_i$ solicitation is received. The causality constraint implies that $\neg\texttt{all\_ack}_i()$ is true if a message is sent, because messages cannot be instantly acknowledged.

```
sent_i() ← ¬all_ack_i().
sent_i()@next ← sent_i(), ¬vote_1_i().
```

Round 2 is started by the master if no node has an outstanding message.

```
start_round_2_i() ← ¬not_all_recv_1_i(),
        ¬not_all_comp_1_i(), node_min(#L,L).
```

The voting for round 2 is shown below. Notes vote $\texttt{incomplete\_2}_i$ if they have sent any messages since the last $\texttt{vote\_1}_i$ solicitation. Recall that any $\texttt{incomplete\_2}_i$ votes result in the protocol restarting with round 1.

```
vote_2_i(#N)@async ← start_round_2_i(), node(N).
complete_2_i(#M,N)@async ← vote_2_i(#N),
        all_ack_i(#N), ¬sent_i(#N), node_min(#N,M).
incomplete_2_i(#M,N)@async ← vote_2_i(#N),
        sent_i(#N), node_min(#N,M).
```

The entire async recursive node i is done when all nodes have voted $\texttt{complete\_2}_i$.

```
done_recursion_i() ← ¬not_all_recv_2_i(),
        ¬not_all_comp_2_i().
```

For every relation $\texttt{p} \in L(i)$, add the rule:

```
p_done() ← done_recursion_i().
```

4.3 Equivalence of coordination rewrite to DEDALUS^S

We first argue that the rules for computing p_done have the desired effect.

LEMMA 5 (SEALING). *Assume a* DEDALUS^S *program S with relation* p. *The* DEDALUS *program $\mathcal{P}(S)$ contains a relation* p_done *with the following property: in any of its stable models $\mathcal{M}$, if* $\texttt{p\_done(l,t)} \in \mathcal{M}$, *then* $\texttt{p\_done(l,s)} \in \mathcal{M}$ *for all timestamps* $\texttt{s} > \texttt{t}$. *Furthermore, if* $\texttt{p\_done(l,t)} \in \mathcal{M}$, *then* $\texttt{p(l,s,}c_1\texttt{,...,}c_n\texttt{)} \in \mathcal{M}$ *implies that* $\texttt{p(l,t,}c_1\texttt{,...,}c_n\texttt{)} \in \mathcal{M}$ *for all timestamps* $\texttt{s} > \texttt{t}$.

PROOF. We assume that $\texttt{p}_1\texttt{\_done()}$, ..., $\texttt{p}_{i_q}\texttt{\_done()}$ have the properties mentioned in the Lemma.

Clearly, p_done() has the properties mentioned in the Lemma for the deductive case.

In the asynchronous case, p_done() is similarly correct; the causality constraint implies that the timestamp on acknowledgments is later than the timestamp on the facts they acknowledge, and thus the timestamp on each node's $\texttt{r}_j\texttt{\_node\_done}$ fact is greater than the timestamp on the acknowledged facts. Thus, before a node concludes p_done(), that node has all p facts.

In the asynchronous recursive case, the causality constraint ensures that every response in the second round is received at a time greater than every response in the first round. Thus, between at least the last response of the first round and the last response of the second round, no node has outstanding messages and no node sends a message. This implies that no node ever sends a message again. □

The above Lemma implies that the ultimate model of DEDALUS^S program S is the same as the ultimate model of DEDALUS program $\mathcal{P}(S)$, as relations in lower strata are complete before higher strata rules are satisfiable.

4.4 Discussion

Applying the program transformation $\mathcal{P}$ to the garbage collection program from Example 8 results in the addition of the following rules.

Example 11. Synthesized rules for the garbage collection program:

```
refers_to_to_send(M, Src, Dst) ←
        local_ptr_edb(N, Src, Dst), master(M).
refers_to_send(#M, L, Src, Dst)@async ←
        refers_to_to_send(#L, M, Src, Dst).
refers_to_ack(#N, L, Src, Dst) ←
        refers_to_send(#L, N, Src, Dst).
refers_to_done_node(#M, N)@async ←
        local_ptr_edb_done(#N), master(#N, M),
        (∀X̄.refers_to_to_send(#N, M, X̄) ⇒
        refers_to_ack(#N, M, X̄)).
refers_to_done(M) ← (∀N.node(N) ⇒
        refers_to_done_node(M, N)).
reach_done() ← refers_to_done(),
        (∀N.node(N) ⇒ local_ptr_edb_done(N)).
```

One rule from the original program must also be rewritten to include the new subgoal reach_done:

Example 12. Garbage collection rewrite

```
garbage(Addr) ← addr_edb(Addr), root_edb(Root),
        ¬reach(Root, Addr), reach_done().
```

As we have shown, the resulting program has a single ultimate model. This model corresponds exactly with one of the ultimate models of the original program from Example 8: the model in which ¬reach is not evaluated until reach is fully determined. The rewrite has effectively forced an evaluation strategy analogous to stratum-order evaluation in a centralized Datalog program.

Note also that the rewrite code is a generalization of the "coordination" code that a DEDALUS programmer could have written by hand to ensure that the local relation refers_to is a faithful representation of global state. In distributed systems, global computation barriers are commonly enforced by protocols based on voting: the two-phase commit protocol from distributed databases is a straightforward example [23]. In the synthesized protocol shown above, every agent responsible for a fragment of the global state must "vote" that every message they send to the coordinator has been acknowledged. The coordinator must tally these votes and ensure that the vote is unanimous for all agents. If the protocol completes successfully, the coordinator may proceed past the barrier.

An explicit goal of our work with DEDALUS has been to view general distributed systems through a model-theoretic lens. From this perspective, the connection between coordination protocols that enforce barriers and stratified evaluation of logic programs is clear. Indeed, global stratification requires a coordination protocol to ensure a global consensus on set completion before negation is applied.

5. RELATED WORK

DEDALUS shares features with a long history of deductive database systems. The purely declarative semantics of DEDALUS, based on the reification of logical time into facts, are closer in spirit and interpretation to Statelog [29] and the languages proposed by Cleary and Liu [15, 30, 33] than to languages that admit procedural semantics to deal with update and deletion over time [12, 16]. Previous work in temporal deductive databases attempted to compute finite representations for periodic phenomena [13]: we reuse many of these results in DEDALUS.

Significant recent work ([4, 9, 14, 32]) has focused on applying deductive database languages extended with networking primitives to the problem of specifying and implementing network protocols and distributed systems. Theorem 1 resembles the correctness proof of "pipelined semi-naive evaluation" for distributed Datalog presented by Loo *et al*. [31]. In general, however, the language extensions proposed in much of this prior work added expressivity and domain applicability but compromised the declarative semantics of Datalog, making formal analysis difficult [35, 36]. In designing DEDALUS, we tried to recover and extend the model-theoretic analyses applicable to pure Datalog, while preserving the features appropriate to modeling loosely coupled distributed systems.

Specification languages such as TLA [28] and I/O Automata [34] employ first-order logic and set theory to model and prove properties about distributed systems, and a subset of both languages produce executable code. Like DEDALUS, TLA expresses concurrent systems in terms of constraints over valuations of state, and temporal logic that describes admissible transitions. DEDALUS differs from TLA in its minimalist use of temporal constructs (`@next` and `@async`), and in its model-theoretic semantics. I/O Automata model distributed systems at a lower level than DEDALUS, as a composition of state machines with explicitly specified transition systems. We intend to further explore the relationship of DEDALUS to these traditional distributed systems formalisms.

Recently, Ameloot *et al*. explored Hellerstein's CALM theorem using relational transducers [8]. They proved that monotonic first-order queries are exactly the set of queries that can be computed in a coordination-free fashion in that transducer formalism. Their work uses some different assumptions than ours—for example, they assume that all messages sent by a node are multicast to a fixed set of neighbors, whereas DEDALUS permits arbitrary unicast. Relational transducers have also been used to specify and show the correctness of interactive web services and electronic commerce workflows (e.g., [2, 17, 18]).

Abiteboul *et al*. recently proposed Webdamlog [1], another distributed variant of Datalog that bears many similarities to DEDALUS. They demonstrate that Webdamlog has an operational semantics similar to the operational semantics in DEDALUS [3], and provide conservative conditions for confluence based on a variant of (node-local) stratification. Our work additionally provides a model-theoretic semantics for DEDALUS^S that corresponds to the operational semantics. DEDALUS^S programs (which are guaranteed to be confluent) also admit a broader use of negation—ensured via a synthesized coordination protocol—than the stratification conditions of Webdamlog.

6. FUTURE WORK

An obvious topic for future work is to extend beyond our focus on "one-shot" executions over fixed inputs. Some distributed computations are continuous services whose semantics need to be described with respect to subsets or subsequences of their inputs and outputs. To this end, models from stream queries may be useful (e.g., [11]). Our network model makes similar assumptions of finiteness—in particular we have ignored dropped messages (infinite delays) and the standard practice of timeout logic for dealing with them. In our applied work [4, 5] we have modeled timeouts as messages that arrive asynchronously under the control of an external "clock" agent. Programs that reason about timeouts typically "seal" the contents of IDB relations based on the inherently non-deterministic subset of messages that "beat the clock." It would be interesting to characterize a useful family of ultimate models in such programs without resorting to the full power of DEDALUS.

Concurrent with this research, our team has been developing a practical language for implementing distributed systems called Bloom [10]. Bloom has built-in support for input streams, including "periodic" relations, in which tuples appear at regular (wall clock) intervals and which are the basis of timeout logic. Instead of relying on language restrictions like those presented in this paper, Bloom offers the full power of DEDALUS. However, we use the intuition of DEDALUS^S to motivate a (necessarily) conservative static analysis for confluence of Bloom programs. The analysis can mark a program as confluent if it is only uses the constructs of DEDALUS$^+$. Otherwise, the analysis alerts the programmer to uses of negation (and aggregation) that are applied over asynchronously-delivered messages or their consequences. The programmer must then manually "guard" these negative constructs with coordination logic, and manually verify the confluence of the result. This allows programmers to choose from (and implement) a wide variety of coordination protocols, as opposed to our approach here in which a compiler synthesizes a simple, generic protocol. In practice, the performance tradeoffs between these protocols can be substantial, depending on the execution environment.

As future work, it would helpful to formally characterize these practical tradeoffs, and automatically synthesize efficient and provably confluent coordination logic suited to the environment. The observation that two programs have the same complexity in a Turing Machine model does not mean they have similar network performance characteristics in the operational semantics of network transducers. We are pursuing work on a complexity model that will address this.

7. REFERENCES

[1] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. A rule-based language for web data management. In *PODS*, 2011.

[2] S. Abiteboul, V. Vianu, B. S. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *J. Comput. Syst. Sci.*, 61(2):236–269, 2000.

[3] P. Alvaro, T. J. Ameloot, J. M. Hellerstein, W. Marczak, and J. Van den Bussche. A Declarative Semantics for Dedalus. Technical Report UCB/EECS-2011-120, EECS Department, University of California, Berkeley, Nov 2011.

[4] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears. BOOM Analytics: Exploring Data-centric, Declarative Programming for the Cloud. In *EuroSys*, 2010.

[5] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR*, 2011.

[6] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in Time and Space. In *Proceedings of the Datalog 2.0 Workshop (to appear)*, 2011.

[7] T. J. Ameloot. (personal communication), 2011.

[8] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational

Transducers for Declarative Networking. In *PODS*, 2011.

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

[10] Bloom programming language. `http://www.bloom-lang.org`.

[11] B. Chandramouli, J. Goldstein, and D. Maier. On-the-fly progress detection in iterative stream queries. In *VLDB*, 2009.

[12] 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.

[13] J. Chomicki and T. Imieliński. Temporal Deductive Databases and Infinite Objects. In *PODS*, pages 61–73, 1988.

[14] 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 *SenSys*, 2007.

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

[16] 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.

[17] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *ICDT*, 2009.

[18] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *J. Comput. Syst. Sci.*, 73:442–474, 2007.

[19] H. Gaifman, H. Mairson, Y. Sagiv, and M. Y. Vardi. Undecidable Optimization Problems for Database Logic Programs. *Journal of the ACM*, 40:683–713, July 1993.

[20] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[21] M. Gelfond and V. Lifschitz. The Stable Model Semantics For Logic Programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[22] J. Gray. The transaction concept: Virtues and limitations (invited paper). In *VLDB*, pages 144–154, 1981.

[23] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[24] J. M. Hellerstein. The Declarative Imperative: Experiences and Conjectures in Distributed Logic. *SIGMOD Rec.*, 39:5–19, September 2010.

[25] N. Immerman. Relational Queries Computable in Polynomial Time. *Information and Control*, 68:86–104, 1986.

[26] N. Immerman. *Descriptive Complexity*. Springer, 1999.

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

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

[29] G. Lausen, B. Ludäscher, and W. May. On active deductive databases: The statelog approach. In *Transactions and Change in Logic Databases*, pages 69–106, 1998.

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

[31] 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 *SIGMOD*, 2006.

[32] 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.

[33] 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.

[34] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

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

[36] J. A. Navarro and A. Rybalchenko. Operational Semantics for Declarative Networking. In *PADL*, 2009.

[37] J. A. Pérez, A. Rybalchenko, and A. Singh. Cardinality abstraction for declarative networking applications. In *CAV*, 2009.

[38] D. Saccà and C. Zaniolo. Stable Models and Non-Determinism in Logic Programs with Negation. In *PODS*, pages 205–217, 1990.

[39] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

[40] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[41] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990.

APPENDIX

A. PROOF OF LEMMA 1

PROOF. Using the construction presented by Gaifman et al. [19], it is possible to write a Datalog program that encodes any two-counter machine's transition relation and an arbitrarily long finite successor relation in the EDB, and define a 0-ary output relation `accept` that is true if and only if the two-counter machine accepts and the transition and successor relations are valid. As the construction is possible in Datalog, it is also possible in DEDALUS.

We add the following rules to the construction, to non-deterministically decide whether to run the machine or not:

```
message(0)@async.
message(1)@async.
run_machine() ← message(0), message(1).
accept() ← message(0), ¬message(1),
        input_valid().
accept() ← ¬message(0), message(1),
        input_valid().
```

Note that the first two lines are actually rules.

For valid inputs, the ultimate model is `accept()` if and only if either `message(0)` and `message(1)` are assigned the same timestamp and the machine accepts, or if the timestamps are different. For invalid inputs, all ultimate models are empty.

If we could decide confluence for this program, we could decide whether there is any valid input for which an arbitrary two-counter machine halts in an accepting state. □

B. QBF IN DEDALUS

We assume that the QBF formula is in prenex normal form: $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n(x_1, \ldots, x_n)$. The textbook recursive algorithm for QBF [20] involves removing Q_1 and recursively calling the algorithm twice, once for $F_1 = Q_2 x_2 \ldots Q_n x_n(0, x_2, \ldots x_n)$ and once for $F_2 = Q_2 x_2 \ldots Q_n x_n(1, x_2, \ldots, x_n)$ for x_1. If $Q_1 = \exists$, then the algorithm returns $F_1 \vee F_2$; if $Q_1 = \forall$, then $F_1 \wedge F_2$.

The leaves of the tree of recursive calls can each be represented as an n-bit binary number, where bit i holds the value of x_i. Assume the left child of a node at depth i of the recursive call tree represents binding x_i to 0, and the right child 1.

Our algorithm is intuitively similar to a postorder traversal of this recursive call tree. Recursively, first visit the left node, then visit the right node, then visit the root. If we are visiting a leaf node, we evaluate the formula for the given variable binding and store a 0 or 1 at the node depending on whether the formula is false or true for that particular binding. If we are visiting a non-root node at level i, we apply the quantifier Q_i to the values stored in the child nodes. Even though the recursive call tree is exponential in size, we only require $O(n)$ space due to the sequentiality of the traversal.

First, we iterate through all of the n-bit binary numbers, one per timestamp. We assume that the order over the variables is such that the leftmost variable in the formula (the high-order bit) is the x_1 (the first), and the rightmost is x_n (the last). Thus, our addition is "backwards" in that it propagates carries from x_i to x_{i-1}:

```
carry(V) ← var_last(V).
one(V)@next ← carry(V), ¬one(V).
one(V)@next ← one(V), ¬carry(V).
carry(U) ← carry(V), one(V), var_succ(U, V).
```

At each timestep, we check whether the current assignment of values to the variables makes the formula true. We omit these rules for brevity. If the formula is true, then `formula_true()` is true at that timestep.

The following rules handle how nodes set their values to either 0 or 1. Note that we only require $2n$ bits of space for this step: each depth $1, \ldots, n$ in the recursive call tree has two one-bit registers (labelled by constant symbols `a` and `b`) representing the current values of the children in the traversal.

`var_sat_in` associates a depth with a given truth value (0 or 1). This value is placed into `var_sat` at depth `V` in register `a` if `a` is empty, or `b` otherwise. Once a value is placed in register `b`, it is deleted in the immediate next timestamp. As we will see later, before with this deletion, the parent node applies its quantifier to the values in the two registers.

The truth value at depth n (denoted by `var_last`) is the truth value of the formula (`formula_true()`) for the assignment of variables at the current timestep.

```
var_sat_in(V, 1) ← formula_true(), var_last(V).
var_sat(a, V, B)@next ← var_sat_in(V, B),
        ¬var_sat(_, V, _).
var_sat(b, V, B)@next ← var_sat_in(V, B),
        var_sat(a, V, _)..
var_sat(N, V, B)@next ← var_sat(N, V, B),
        ¬var_sat(b, V, _).
```

`var_sat_left_in` associates a value with the parent of a given depth. This is used for propagating the result of the quantifier application to the parent. The cases for existential (`exists`) and universal (`forall`) quantifiers are clear.

```
var_sat_in(N, U, B) ← var_sat_left_in(V, B),
        var_succ(U, V).
var_sat_left_in(vn, 1) ← exists(vn),
        var_sat(_, vn, 1).
var_sat_left_in(vn, 0) ← exists(vn),
        var_sat(a, vn, 0), var_sat(b, vn, 0).
var_sat_left_in(vn_succ, 1) ← forall(vn),
        var_sat(a, vn, 1), var_sat(b, vn, 1).
var_sat_left_in(vn_succ, 0) ← forall(vn),
        var_sat(_, vn, false).
```

Finally, the entire formula is `satisfiable(1)` (satisfiable) if the output of the first quantifier is 1, and `satisfiable(0)` (unsatisfiable) if the output of the first quantifier is 0.

```
satisfiable(B) ← var_sat_left_in(V, B),
        var_first(V).
```