Confluence Analysis for Distributed Programs: A Model-Theoretic Approach

William R. Marczak[1], Peter Alvaro[1], Neil Conway[1], Joseph M. Hellerstein[1], and David Maier[2]

[1] University of California, Berkeley
[2] Portland State University

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 introduce the *ultimate model*, which captures non-deterministic eventual outcomes of distributed programs. After showing the question of confluence undecidable for DEDALUS, we identify restricted sub-languages that guarantee confluence while providing adequate expressivity. We observe that the semipositive restriction DEDALUS$^+$ guarantees confluence while capturing PTIME, but show that its restriction of negation makes certain simple and practical programs difficult to write. To remedy this, we introduce DEDALUS^S, a restriction of DEDALUS that allows a kind 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 [1]. This has led to activity in language and system design (e.g., [2–5]), as well as formal models for distributed computation using such languages (e.g., [6–8]).

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 [9], we 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 [10] corresponding to all possible temporal permutations of messages. In [9], we show these models equivalent to all possible executions in an operational semantics akin to those in 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.

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 its *ultimate models*. We also identify restricted sub-languages of DEDALUS that ensure a model-theoretic notion of confluence: the existence of a unique ultimate model for any program 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.

A natural step in this direction is to restrict Dedalus to its semi-positive subset, a language we call Dedalus$^+$. This is inspired in part by the CALM theorem [11, 1, 12], which established a connection between confluence and monotonicity. However, we note that this restriction makes common distributed systems tasks difficult to achieve.

We achieve a more comfortable balance between expressive power, ease of programming and guarantees of confluence in Dedalus^S, which admits a controlled use of negation that draws inspiration from both stratified negation in logic programming, and coordination protocols from distributed computing. We present the model-theoretic semantics of Dedalus^S, and give it an operational semantics by compiling Dedalus^S programs into stylized Dedalus programs that augment the original code with "coordination" rules that effectively implement distributed stratified evaluation. 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 [13].

Due to space restrictions, proofs and additional examples are available in [14].

2 Dedalus

Dedalus extends Datalog to model the critical semantic issue from asynchronous distributed computing: asynchrony across nodes. We use a restricted version of Sacca and Zaniolo's *choice* construct [10], interpreted under the stable model semantics, to model program behaviors. Our use of the stable model semantics induces 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.

We begin this section by reviewing the syntax of Dedalus first presented in Alvaro *et al*. [15]. We then review the model-theoretic semantics for Dedalus [9].

2.1 Syntax

Preliminaries Let $\mathcal{U}$ be an infinite universe of values, with $\mathbb{N} = \{0, 1, 2, \ldots\} \subset \mathcal{U}$.

A *relation schema* is a pair $R^{(k)}$ where R is a relation name and k its arity. A *database schema* $\mathcal{S}$ is a set of relation schemas. Any relation name occurs at most once in a database schema. We assume the existence of an order: every database schema contains the relation schema $<^{(2)}$. 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 $\texttt{R(c}_1\texttt{, ..., c}_n\texttt{)}$. 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*(φ), a set *pos*(φ) of *positive body atoms*, a set *neg*(φ) of *negative body atoms*, a set of inequalities *ineq*(φ) of the form $x < y$, and a set of choice operators *cho*(φ) applied to the variables. The conventional syntax for a rule is:

$\texttt{head}(\varphi) \leftarrow f\_1, \ldots, f\_n, \neg g\_1, \ldots, \neg g\_m,\ \texttt{ineq}(\varphi),\ \texttt{cho}(\varphi)$.

where $f_i \in pos(\varphi)$ for $i = 1, \ldots, n$ and $g_i \in neg(\varphi)$ for $i = 1, \ldots, m$.

Dedalus maintains the usual Datalog safety restrictions: every variable symbol `V` in a rule must appear in some atom in *pos*. For readability, we will use the underscore symbol (_) to represent a variable that appears only once in a rule.

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

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 `node(d)`. We denote a spatial fact over $R^{(n)}$ by $\texttt{R(d, c}_1\texttt{, ..., c}_n\texttt{)}$. A *spatial relation instance* for a relation schema $\texttt{r}^{(n)}$ is a set of spatial facts for $\texttt{r}^{(n+1)}$. 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 $\texttt{r}^{(n)} \in (\mathcal{S} \setminus \{<\})$ we include a relation schema $\texttt{r}^{(n+2)}$ 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), $\texttt{node}^{(1)}$ (finite), $\texttt{time}^{(1)}$ (infinite) and $\texttt{timeSucc}^{(2)}$ (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 $\texttt{R(d, t, c}_1\texttt{, ..., c}_n\texttt{)}$.

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

$$\begin{aligned}
&\texttt{p(L,T,}\overline{\texttt{W}}\texttt{)} \leftarrow \texttt{b}_1\texttt{(L,T,}\overline{\texttt{X}_1}\texttt{), ..., b}_l\texttt{(L,T,}\overline{\texttt{X}_l}\texttt{), }\neg\texttt{c}_1\texttt{(L,T,}\overline{\texttt{Y}_1}\texttt{), ..., }\neg\texttt{c}_m\texttt{(L,T,}\overline{\texttt{Y}_m}\texttt{),}\\
&\qquad\qquad \texttt{node(L), time(T), } ineq(\varphi)\texttt{.}\\
&\texttt{p(L,S,}\overline{\texttt{W}}\texttt{)} \leftarrow \texttt{b}_1\texttt{(L,T,}\overline{\texttt{X}_1}\texttt{), ..., b}_l\texttt{(L,T,}\overline{\texttt{X}_l}\texttt{), }\neg\texttt{c}_1\texttt{(L,T,}\overline{\texttt{Y}_1}\texttt{), ..., }\neg\texttt{c}_m\texttt{(L,T,}\overline{\texttt{Y}_m}\texttt{),}\\
&\qquad\qquad \texttt{node(L), time(T), timeSucc(T,S), } ineq(\varphi)\texttt{.}\\
&\texttt{p(D,S,}\overline{\texttt{W}}\texttt{)} \leftarrow \texttt{b}_1\texttt{(L,T,}\overline{\texttt{X}_1}\texttt{), ..., b}_l\texttt{(L,T,}\overline{\texttt{X}_l}\texttt{), }\neg\texttt{c}_1\texttt{(L,T,}\overline{\texttt{Y}_1}\texttt{), ..., }\neg\texttt{c}_m\texttt{(L,T,}\overline{\texttt{Y}_m}\texttt{),}\\
&\qquad\qquad \texttt{node(L), time(T), time(S), choice((L, T, }\overline{\texttt{B}}\texttt{),(S)), node(D), } ineq(\varphi)\texttt{.}
\end{aligned}$$

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

The first kind is a *deductive* rule, the second an *inductive* rule, and the last an *asynchronous rule*. The last two kinds of rules are collectively called *temporal* rules.

The use of a single location specifier and timestamp in rule bodies corresponds to considering deductions that can be evaluated at a single node at a single point in time.

The `choice` construct is from Saccà and Zaniolo [10] and is used to model the fact that the network may arbitrarily delay, re-order, and batch messages. We use the causality rewrite of Alvaro *et al*. [9], which restricts `choice` in the following way: 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). Intuitively, the causality constraint rules out models corresponding to impossible executions, in which effects are perceived before their causes. Full details about `choice` and the causality constraint are available in a companion paper [9].

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

Syntactic Sugar The restrictions on timestamps and location specifiers suggest a natural syntactic sugar that omits boilerplate usage of timestamp attributes and location specificers, as well as the use of `node`, `time`, `timeSucc`, and `choice` in rule bodies. Example deductive, inductive, and asynchronous rules are shown below.

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

In any 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, the head location specifier can be accessed in a similar manner in the head atom, as shown in the following rule.

```
p(#D,L,W)@async ← b(#L,D,W), ¬c(#L,L).
```

The head and body location specifiers are `D` and `L` respectively. `D` may appear in the body, `L` may appear in the head, and `L` may appear duplicated in the body.

2.2 Semantics

We only consider Dedalus programs whose deductive rules are syntactically stratified.

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 $\mathtt{r}(\mathtt{d},\mathtt{c}_1,\ldots,\mathtt{c}_n) \in \mathcal{E}$, the fact $\mathtt{r}(\mathtt{d},\mathtt{t},\mathtt{c}_1,\ldots,\mathtt{c}_n) \in \mathcal{K}$ for all $\mathtt{t} \in \mathbb{N}$. Intuitively, EDB facts "exist at all timesteps."

We refer to a Dedalus program together with an EDB instance as a Dedalus *instance*. The behavior of a Dedalus program can be viewed as a mapping from EDB instances to spatio-temporal database instances. We use the *stable model semantics* to describe this mapping. Intuitively, there is a one to one correspondence between stable models and values for timestamps for all messages that obey the causality rewrite [9].

Example 1. 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 are exactly the stable models: $\{\texttt{node(n1)}\} \cup \{\texttt{p(n1,i)} \mid \texttt{i} \in S\} \cup \{\texttt{q(n1,i)} \mid \texttt{i} \in \mathbb{N}\}$.

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. The causality constraint rules out elements of the power set with finite cardinality [9].

Ultimate Models Stable models highlight uninteresting temporal differences that may not be "eventually" significant. Intuitively, there would be different stable models for different message orderings, even when those orderings were not meaningful because they represented some commutative operation. An example appears in Appendix F of [14]. 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 $\mathcal{S}^O$.

Let $\Diamond\Box$ map spatio-temporal database instances $\mathcal{T}$ to spatial database instances. 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 2. For Example 1 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.

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.

Example 3. A simple asynchronous marriage ceremony:

```
i_do(X)@async ⟵ i_do_edb(X).
runaway() ⟵ ¬i_do(bride), i_do(groom).
runaway() ⟵ ¬i_do(groom), i_do(bride).
runaway()@next ⟵ runaway().
i_do(X)@next ⟵ i_do(X).
```

The intended meaning of the program is that 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 match this specification. Any stable model where `i_do(groom)` and `i_do(bride)` disagree in their first chosen timestamps yields an ultimate model containing `runaway()`. If the votes are assigned the same timestamp, the ultimate model does not contain `runaway()`. See Appendix A of [14] for a version of this example involving asynchrony.

In this case, there is a preferred model where negation is not applied to a set until its content has been fully determined. This is akin go the notion of a perfect model in Datalog. Typically, a programmer would induce this preferred model by inserting *coordination* code (e.g., voting or consensus between all communicating agents) to ensure that there are no outstanding messages in flight, before applying a summarizing operation like negation.

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). 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

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 [16, 17]. Confluence may be viewed as a specialization of the more general notion of consistency of distributed state.

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

This result is hardly surprising, as confluence is defined over all EDB instances. Another symptom of DEDALUS being "too big" a language is its expressive power.

Lemma 2. DEDALUS *subsumes PSPACE.*

3.2 DEDALUS$^+$

Distributed programs that produce non-deterministic output or have exponential runtimes are often undesirable. Since checking for confluence in DEDALUS is undecidable, we present a restriction of DEDALUS that allows only confluent programs and prove that it captures exactly PTIME.

A DEDALUS program is *semipositive* if the ¬ symbol is only used on EDB relations. 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(X̄)@next ← p(X̄)`. The language of semipositive DEDALUS programs with guarded asynchrony is called DEDALUS$^+$.

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.*

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

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 $\mathcal{A}(P)$ to be the program 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 $pos(\varphi)$. 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$^+$ (see Appendix E of [14]). Not only are DEDALUS$^+$ programs confluent, but they also capture exactly PTIME.

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.*

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

4 DEDALUS^S

The marriage program from Example 3 uses IDB negation to determine the truth value of `runaway`, and is thus not directly expressible in DEDALUS$^+$. 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 4.

While the rewrite is straightforward, a majority of the program's rules need to be modified. Note that since Example 4 is in DEDALUS$^+$, it is confluent; therefore, it is not subject to the non-deterministic output observed in the original program (Example 3).

Example 4. An asynchronous marriage ceremony without IDB negation:

```
i_dont(X)@async ← ¬i_do_edb(X).
runaway() ← i_dont(bride).
runaway() ← i_dont(groom).
runaway()@next ← runaway().
i_dont(X)@next ← i_dont(X).
```

Programs involving negation of recursion, such as the distributed garbage collection program presented in Appendix B of [14], present a more difficult problem, as negation must be pushed down through recursion. The best known techniques for this may result in unacceptable overhead as they involve doubling the arity of 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 stratified IDB negation. As one might expect, Dedalus^S retains the benefits of Dedalus$^+$. We provide an operational semantics for Dedalus^S, based on the one for Dedalus [9], 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. After some preliminary definitions, we introduce a semantics for stratifiable Dedalus programs.

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 [9] or `choice` [10] 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 $P_n(\ldots P_1(P_0(E))\ldots)$, obtained by a stratum-order evaluation.

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.*

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.* [9]. 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,`$c_1, \ldots, c_n$`)` $\in \mathcal{M}$ implies that `p(l,t,`$c_1, \ldots, c_n$`)` $\in \mathcal{M}$ for all timestamps `s` $>$ `t`. Intuitively, `p_done()` is true when the content of `p` is *sealed* (henceforth unchanging).

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()`. For IDB relations `p`, we present `p_done()` for non-async-recursive nodes and async recursive nodes separately.

Non-Async-Recursive Nodes For non-async-recursive nodes, we compute a `done` fact for each rule, then collate these into a `done` fact for each relation. The `done` fact for a deductive rule is true when all of the relations in the body of the rule are henceforth unchanging. We assume guarded asynchrony applies to the rules in this section.

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

The rule for `p_done()` is: `p_done()` $\leftarrow$ r_1`_done(), ..., `r_{i_p}`_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 $p_1, \ldots, p_{i_q}$. For each rule $1 \leq j \leq i_p$ in P with head relation `p`, handle rule j according to the cases below.

Deductive: Add the rule: r_j`_done()` $\leftarrow$ p_1`_done(), ..., `p_{i_q}`_done()`.

Asynchronous: For an asynchronous rule, we need to ensure that there are no messages that have not yet been delivered, before we derive r_j`_done()`. We do this by adding rules to record all sent messages, and rules for receivers to send acknowledgements back to senders. When a sender has received an acknowledgement for each sent message, and there are no more messages to send, he indicates this to the receiver. In the vacuous case where a sender has no messages to send to a receiver, he also indicates this to the receiver. When a receiver has been notified by all nodes that there are no in-flight messages, he can derive r_j`_done()`. The rules to express this protocol are in Appendix D of [14].

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

For each rule j, add the rules for asynchronous rules shown in Appendix D of [14], except for the last two rules. Instead write:

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

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 in the head of any asynchronous rule in the following protocol. The rules shown below begin the first round of voting. Nodes vote $\texttt{complete\_1}_i$ if $\texttt{all\_ack}_i$ is true—if they have no outstanding unacknowledged messages. Votes are sent to the master.

```
not_node_min(L1) ← node(L1), node(L2), L2 < L1.
node_min(L) ← ¬not_node_min(L), node(L).
start_round_1_i() ← node_min(#L,L), ¬round_1_i().
round_1_i()@next ← start_round_1_i().
round_1_i()@next ← round_1_i(), ¬start_round_2_i().
vote_1_i(#N)@async ← start_round_1_i(), node(N).
complete_1_i(#M,N)@async ← vote_1_i(#N), all_ack_i(#N), node_min(#N,M).
incomplete_1_i(#M,N)@async ← vote_1_i(#N), ¬all_ack_i(#N), node_min(#N,M).
```

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

```
complete_k_i(N)@next ← complete_k_i(N), ¬start_round_1_i().
incomplete_k_i(N)@next ← incomplete_k_i(N), ¬start_round_1_i().
```

To count votes, we assume the following rules are instantiated for $k = 1$ and 2. Round 1 is restarted if some node votes $\texttt{incomplete\_1}_i$ in round 1—i.e., it has an outstanding unacknowledged message – or $\texttt{incomplete\_2}_i$ 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 starts 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 ensures that $\neg\texttt{all\_ack}_i\texttt{()}$ is true if a message is sent, as 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. Nodes 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 $\texttt{complete\_2}_i$.

$\texttt{done\_recursion}_i\texttt{()} \leftarrow \neg\texttt{not\_all\_recv\_2}_i\texttt{()},\ \neg\texttt{not\_all\_comp\_2}_i\texttt{()}.$

Finally, for every relation $\texttt{p} \in L(i)$, add the rule: $\texttt{p\_done()} \leftarrow \texttt{done\_recursion}_i\texttt{()}.$

This program transformation produces a Dedalus$^+$ program equivalent to any Dedalus^S program. 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* $\texttt{p\_done(1,t)} \in \mathcal{M}$*, then* $\texttt{p\_done(1,s)} \in \mathcal{M}$ *for all timestamps* $\texttt{s} > \texttt{t}$*. Furthermore, if* $\texttt{p\_done(1,t)} \in \mathcal{M}$*, then* $\texttt{p(1,s,}c_1,\ldots,c_n\texttt{)} \in \mathcal{M}$ *implies that* $\texttt{p(1,t,}c_1,\ldots,c_n\texttt{)} \in \mathcal{M}$ *for all timestamps* $\texttt{s} > \texttt{t}$*.*

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. See Appendix C of [14] for an example of applying the program transformation $\mathcal{P}$.

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 [18]. In the protocol from the program transformation $\mathcal{P}$, 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.

5 Related Work

The purely declarative semantics of Dedalus, based on the reification of logical time into facts, are close in spirit and interpretation to Statelog [19], the languages proposed by Cleary and Liu [20–22], and work in temporal deductive databases [23].

Significant recent work ([2–5]) has focused on using deductive database languages extended with networking primitives to specify 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*. [24]. In general, however, the language extensions proposed in much of this prior work added expressivity and domain applicability but compromised declarativity, making formal analysis difficult [25, 7].

Recently, Ameloot *et al*. explored Hellerstein's CALM theorem using relational transducers [6]. They proved that monotonic first-order queries are exactly those queries that can be computed in a coordination-free fashion using transducers. Some of their assumptions differ from ours—for example, they assume that all messages sent by a node are multicast to a fixed neighbor set, whereas Dedalus permits arbitrary unicast.

Abiteboul *et al*. recently proposed Webdamlog [12], 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 [9], 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.

References

1. Hellerstein, J.M.: The Declarative Imperative: Experiences and Conjectures in Distributed Logic. SIGMOD Rec. **39** (September 2010) 5–19
2. Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J.M., Sears, R.C.: BOOM Analytics: Exploring Data-centric, Declarative Programming for the Cloud. In: EuroSys. (2010)
3. Belaramani, N., Zheng, J., Nayate, A., Soulé, R., Dahlin, M., Grimm, R.: PADS: A policy architecture for distributed storage systems. In: NSDI. (2009)
4. Chu, D.C., Popa, L., Tavakoli, A., Hellerstein, J.M., Levis, P., Shenker, S., Stoica, I.: The design and implementation of a declarative sensor network system. In: SenSys. (2007)
5. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking. Communications of the ACM **52**(11) (2009) 87–95
6. Ameloot, T.J., Neven, F., Van den Bussche, J.: Relational Transducers for Declarative Networking. In: PODS. (2011)
7. Navarro, J.A., Rybalchenko, A.: Operational Semantics for Declarative Networking. In: PADL. (2009)
8. Pérez, J.A., Rybalchenko, A., Singh, A.: Cardinality abstraction for declarative networking applications. In: CAV. (2009)
9. Alvaro, P., Ameloot, T.J., Hellerstein, J.M., Marczak, W., Van den Bussche, J.: A Declarative Semantics for Dedalus. Technical Report UCB/EECS-2011-120, EECS Department, University of California, Berkeley (Nov 2011)
10. Saccà, D., Zaniolo, C.: Stable Models and Non-Determinism in Logic Programs with Negation. In: PODS. (1990) 205–217
11. Alvaro, P., Conway, N., Hellerstein, J.M., Marczak, W.R.: Consistency Analysis in Bloom: a CALM and Collected Approach. In: CIDR. (2011)
12. Abiteboul, S., Bienvenu, M., Galland, A., Antoine, E.: A rule-based language for web data management. In: PODS. (2011)
13. : Bloom programming language. `http://www.bloom-lang.org`
14. Marczak, W., Alvaro, P., Conway, N., Hellerstein, J.M., Maier, D.: Confluence analysis for distributed programs: A model-theoretic approach. Technical report, EECS Department, University of California, Berkeley (Jun 2012)
15. Alvaro, P., Marczak, W.R., Conway, N., Hellerstein, J.M., Maier, D., Sears, R.: Dedalus: Datalog in Time and Space. In: Proceedings of the Datalog 2.0 Workshop (to appear). (2011)
16. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: SOSP. (1995)
17. Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M., Theimer, M., Welch, B.W.: Session Guarantees for Weakly Consistent Replicated Data. In: Proceedings of the Third International Conference on Parallel and Distributed Information Systems. PDIS ’94, Washington, DC, USA, IEEE Computer Society (1994) 140–149
18. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. (1993)
19. Lausen, G., Ludäscher, B., May, W.: On active deductive databases: The statelog approach. In: Transactions and Change in Logic Databases. (1998) 69–106
20. Cleary, J.G., Utting, M., Clayton, R.: Data Structures Considered Harmful. In: Australasian Workshop on Computational Logic. (2000)
21. Liu, M., Cleary, J.: Declarative Updates in Deductive Databases. Journal of Computing and Information **1** (1994) 1435–1446
22. Lu, L., Cleary, J.G.: An Operational Semantics of Starlog. In: Proc. Principles and Practice of Declarative Programming, Springer-Verlag (1999) 131–162
23. Chomicki, J., Imieliński, T.: Temporal Deductive Databases and Infinite Objects. In: PODS. (1988) 61–73
24. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative Networking: Language, Execution and Optimization. In: SIGMOD. (2006)
25. Mao, Y.: On the declarativity of declarative networking. In: NetDB. (2009)