Operational Semantics for Declarative Networking

Juan A. Navarro and Andrey Rybalchenko

Max Planck Institute for Software Systems

Abstract. Declarative Networking has been recently promoted as a high-level programming paradigm to more conveniently describe and implement systems that run in a distributed fashion over a computer network. It has already been used to implement various networked systems, e.g., network overlays, Byzantine fault tolerance protocols, and distributed hash tables. Declarative Networking relies upon a rule-based programming language that resembles Datalog and allows one to declaratively specify the flow of networking events. However, the presence of asynchronous communication, distribution, and imperative modification of the program state in Declarative Networking applications have been an obstacle for defining its semantics. Currently, the reference semantics is determined by the runtime environment only, which hinders further application development and makes any efforts to develop program analysis and verification tools impossible. In this paper, we propose an operational semantics for Declarative Networking that addresses these problems. The semantics is parameterized to keep open a design space required at the current stage of the language development. We also report on our first experience with an interpreter for Declarative Networking applications that implements the proposed semantics.

Keywords: Declarative networking, programming language semantics, distributed systems.

1 Introduction

Design and implementation of distributed systems is a challenging task that requires research efforts from various perspectives. In addition to improvements achieved by applying more sophisticated communication protocols, novel system designs, and more efficient algorithms, programming languages can make a significant impact on the implementation process by supporting the programmer with adequate constructs and primitives, e.g., control statements, type systems, and libraries [1–6]. In this line of research, Declarative Networking stands out as a high-level programming paradigm to more conveniently describe and implement distributed applications that run over computer networks [5].

The leading thought behind the Declarative Networking approach is to carry over declarative programming techniques inspired by Datalog to the domain of systems and networking applications. It builds upon a rule-based programming language called P2 that allows the programmer to declaratively specify the flow

of networking events. Compared to the traditional approaches that use general purpose imperative languages, e.g., C++ [7] and Java [8], the implementations written in P2 reduce the code base size by several orders of magnitude, while improving its clarity and succinctness.

The distinctive features offered by Declarative Networking attracted interest in the networking and distributed systems community, in both academia and industry. A growing number of implementation efforts have chosen P2 as the programming language. The main applications are various network protocols, including sensor networks, Byzantine fault tolerance, and distributed hash tables, see e.g. [9, 4, 10, 11]. The literature describing the resulting systems attributes their success, to a large extent, to the Declarative Networking paradigm.

The initial success and increasing adoption of Declarative Networking encourages the development of program analysis and verification tools for applications written in P2. These tools require a program semantics as a starting point, e.g., in order to simulate the execution of P2 programs on symbolic inputs, or to trace the flow of communication events through a sequence of rule invocations.

Unfortunately, a well-defined semantics for P2 has not been identified yet. Asynchronous communication, distribution, and presence of imperative modifications of the program state have been an obstacle. An additional source of complexity comes from its database-oriented setting that uses distributed query processing machinery as a basic vocabulary to define semantics. The existing specifications are incomplete and represented in an informal style that allows contradicting interpretations, as we show by examples in Section 3. Currently, P2 semantics is implicitly determined by the runtime environment [12], which in turn deviates from the descriptions in the literature [13, 5]. This state of affairs hinders further development of Declarative Networking applications and makes any efforts to provide program analysis and verification tools for P2 impossible.

In this paper, we propose a parameterized operational semantics for Declarative Networking, which addresses the open questions about the semantics of P2 programs. The semantics is given by a state transition system and is represented by an algorithm that defines the transition relation of a given P2 program. The algorithm contains a collection of parameters that determine the main characteristics of P2 computations, e.g., when the effect of rule application is propagated to the program state. We avoided a presentation of the algorithm with some fixed valuation of the parameters, since any commitment to a particular set of design choices might be premature at this stage of the language's development.

In order to show the applicability of our approach we have also developed a P2 interpreter which simulates the execution of the parametrized semantics presented in this paper. This allows one to experiment with different choices of semantics and have a better understanding on the impact that these parameters have. This implementation, moreover, is a first stepping stone towards the development of verification and symbolic execution tools. Moreover, after coupling it with a networking back-end, it will also provide a full-fledged P2 interpreter.

In summary, our proposed semantics generalizes and unifies both specifications as presented in the literature and determined by the runtime environment.

This can be used as a starting point for the development of new interpreters, verification and analysis tools built on top of a formally defined semantics.

Related work. The challenges of distributed programming are addressed by active efforts in development of adequate programming languages and extensions. The recent developments include Acute [6], Alice [14], Curry [3], Erlang [1], JoCaml [2], Mace [4], and P2 [5]. The recurrent theme is to provide high-level programming abstractions for dealing with distributed computation and communication. Most of these languages follow functional and logic programming paradigms and their combinations. Mace is an exception that provides means for the specification of distributed protocols as transition systems that are compiled to C++. Statically typed functional languages Acute, Alice, and JoCaml extend the type discipline to values that are communicated over the network [2, 15, 6]. Curry, which is based on multiple paradigms, strives for a seamless integration of distribution in the context of logical variables, non-determinism, and search.

Many of the above efforts provide experimental platforms for studying distributed programming languages. Moreover, the languages Erlang, Mace, and P2 have been proven successful from the application development perspective. Erlang is widely used to develop telecommunication software, while Mace and P2 have gained increasing interest in the systems and networking community. The main applications are network protocols, including overlays, sensor networks, Byzantine fault tolerance, and distributed hash tables, see e.g. [9, 4, 10, 11].

In comparison with the above languages, P2 stands out due to its simplicity and declarative foundations, while providing sufficient capabilities to develop state-of-the-art networking applications. Its increasing adoption by systems and networking researchers motivates our interest in its semantics, which provides a foundation for the development of program analysis tools for P2.

We note that besides P2, Datalog has been the basis for the development of other successful domain specific languages, e.g., for mining software artefacts using relational queries [16] and pointer alias analysis for imperative programs [17]. Tuples, which are atomic pieces of data in Datalog, are successfully applied as a communication primitive for distributed programming, as pioneered by the Linda system [18].

Our work can be seen as a parallel to what SLD resolution does for logic programming systems such as Prolog. Although any implementation may follow a particular strategy for the rule evaluation, any such strategy must conform to SLD resolution, which serves as the reference semantics of Prolog systems. Similarly, we wish to provide such a semantics foundation for declarative networking programming languages.

2 P2 by example

In this section, we briefly present the P2 language used for Declarative Networking. Our description follows [10].

Program states. P2 programs manipulate tables, i.e., sets of tuples, as in relational databases. We distinguish between *materialized* tuples that are stored in a distributed fashion among nodes in the network, and *event* tuples that carry data between nodes and signal the occurrence of a particular event at a node.

A P2 program starts with a declaration of materialized tables. It lists the materialized tables, and specifies the primary keys for each of them. We consider as events all tuples in tables that are not declared as materialized. For example, the declarations

```
materialize(neighbor, keys(1,2)).
materialize(sequence, keys(1)).
```

specify that whenever a *neighbor* or *sequence* tuple is produced by a node, it should also be stored in a table with the corresponding name. The *keys* declaration specifies the fields that define the primary key of each table. At any time during execution, the runtime system ensures that there is at most one tuple stored for any valuation of the tuple positions appearing in the key declaration. In our example, the declaration requires that there is at most one *sequence* tuple for each value of the first position.

The declaration of materialization can also constrain the lifetime and quantity of tuples that a table can store [10]. Since these constraints are seldom in the existing P2 code base and can be simulated within the P2 language, we choose to omit them for clarity of presentation.

Program rules. Rules are the main component specifying computations of a P2 program. They are represented by constructs of the form '*head* :− *body*.' where *body* is a list of *predicates* applied to variables and constants, and *head* is a predicate applied to a subset of variables that appear in the body of the rule. The order of appearance of predicates in the rule, and of rules in the program text is irrelevant.

For example, we consider the following rule.

```
refresh(@X) :- periodic(@X, E, 3).
```

The *periodic* predicate in the body of the rule is a special built-in event predicate. It is automatically generated every 3 seconds by the P2 runtime environment at the node with the address X, and is instantiated with a unique value E. An optional fourth parameter can be used in the *periodic* predicate to indicate how many times should the event be generated. With respect to the declaration above, both *refresh* and *periodic* are events (i.e. they are not materialized). An intuitively reading of this rule is "generate a *refresh* event tuple at node X whenever there is a *periodic* event tuple at node X with the values E and 3." Note a convention that the first field of predicates appearing in the rule denotes the address of the node where the corresponding tuple resides. It is additionally marked by the '@' symbol.

As another example, consider the following two rules.

```
sequence(@X, NewSeq) :- refresh(@X), sequence(@X, CurSeq),
    NewSeq := CurSeq + 1.
send_updates(@X) :- refresh(@X).
```

The first rule specifies that every time that a *refresh* event is seen at the node X, the current value stored at the materialized *sequence* table is read, incremented, and a tuple with the new value is inserted into the *sequence* table at the same node. Since the primary key of the sequence table includes only the address field, each node can store at most one tuple in this table. The insertion of the new tuple into the table will implicitly remove the previous tuple with the old value. The second rule in the example produces a new *send_updates* event tuple every time there is a *refresh* event tuple at the node X.

So far, we have only seen rules that are evaluated at a single network node. The following rule illustrates how distributed computation is performed in P2.

```
update(@Y, X, S) :- send_updates(@X), neighbor(@X, Y),
    sequence(@X, S).
```

Intuitively, this rule can be read as "every time there is a *send_updates* at a node X, for every *neighbor* tuple stored at X with value Y and every *sequence* tuple stored at X with value S send an *update* event tuple to the node Y with the values X and S." In a networked system, an execution of this rule at a node X notifies its neighbors about the current sequence number of X.

Finally, we introduce a few more features of P2.

```
delete neighbor(@X, Y) :- purge(@X),
    last_update(@X, Y, LastTime), f_now(@X) - LastTime > 20.
```

The keyword *delete* appears in the head of the rule. It is used in P2 to request deletion of tuples from a table whenever the body of the rule is satisfied. The built-in function *f_now* returns the current wall-clock time of a node. In order to execute this rule, assume that *purge* is an event that is periodically generated at the node X and that *last_update* is another materialized table storing the time stamp of the last update event received from a node Y. Then, this rule will remove Y from the set of X's neighbors if it has not received an update from Y within an interval of 20 seconds.

3 From declaration to execution

The P2 language can greatly simplify network protocol development, however the ease with which a P2 program can be turned into a working implementation is often overstated in previous work on Declarative Networking. In many situations, the existing description of the language is not precise enough to determine how a P2 should be interpreted. In this section, we illustrate such cases by examples.

Event creation vs. effect. First, we consider the program given in Figure 1. It consists of the rules presented in the previous section. As described above, every time that a *refresh* event is generated the node's sequence number is incremented and a *send_updates* event tuple is generated. Then, the *update* event will forward the current sequence number to all neighbor nodes. Though, at this point we need to decide what should be the value of the *current* sequence number. Should it be the value before or after executing the increment?

```
materialize(neighbor, keys(1,2)).
materialize(sequence, keys(1)).

refresh(@X) :- periodic(@X, E, 3).

sequence(@X, NewSeq) :- refresh(@X), sequence(@X, CurSeq),
    NewSeq := CurSeq + 1.
send_updates(@X) :- refresh(@X).

update(@Y, X, S) :- send_updates(@X), neighbor(@X, Y),
    sequence(@X, S).
```

Fig. 1. An example P2 program in which a node increments and sends its sequence number to all of its neighbors. Various event processing schemes are conceivable where the sequence number value either before or after the increment is sent by the last rule.

There might exist reasons to prefer one choice over the other, or even to declare that the P2 runtime environment can make an arbitrary choice among the two options. Unfortunately, the existing work on the P2 language does not address such corner cases. The documentation of the P2 runtime [12] does not go beyond an informal introduction to the language, leaving open ambiguities such as the one presented here. A recent work [5] gives a formal definition of both the syntax and semantics of a subset of the P2 language that does not deal with event tuples, i.e. all tables are materialized. Thus, it does not clarify how interactions between event processing and table updates should be handled.

Algorithms 5.1 and 5.3 in [13] implicitly suggest that updates are immediately applied after evaluating each individual rule. In our example this means that the sequence number is incremented before sending the update. An experimental evaluation using the current implementation of P2 exhibits the opposite semantics in which events that are addressed to the same node that generated it, so-called *internal events*, are propagated and evaluated before any updates are applied to the materialized store.[1] This means that we observed an update event that contains the old sequence number.

Internal vs. external events. Under-specified semantics can lead to other significant deviation between possible outcomes. See the example shown in Figure 2. The presented program maintains three materialized tables that are initialized after the declaration. Besides a *neighbor* table, every node contains ten *store* tuples and a sequence number that is initialized to zero.

The first rule specifies that a node will increment its sequence number each time that it receives a *ping* event. The second rule causes a node, upon receiving a *broadcast* event, to send ten *ping* events to each neighbor node. Finally, the

[1] We used `runStagedOverlog` executable from the P2 distribution that is compiled from the revision 2114 of the publicly available code from the anonymous SVN server `https://svn.declarativity.net/p2/trunk/`.

```
materialize(neighbor, keys(1,2)).
materialize(store, keys(1,2)).
materialize(sequence, keys(1)).

neighbor(@X, "node1").
neighbor(@X, "node2").
neighbor(@X, "node3").

store(@X, 1).
store(@X, 2).
...
store(@X, 10).

sequence(@X, 0).

sequence(@X, New) :- ping(@X), sequence(@X, Old),
    New := Old + 1.

ping(@Y) :- broadcast(@X), neighbor(@X, Y), store(@X, _).

broadcast(@X) :- periodic(@X, E, 5, 1), X = "node1".
```

Fig. 2. In this example, a node *node1* sends ten ping messages to each neighbor. Upon receiving a ping message, each node increments own sequence number.

last rule causes the node *node1* to generate a *broadcast* event after five seconds of activity. After all events have been sent, received, and processed by the corresponding nodes, one would expect that the program reaches a state in which every node stores the sequence number ten. However, this is not the case for the current P2 implementation.

The reason for a different outcome is rooted in the fact that *node1* sends ten *ping* events to itself, whereas other nodes receive them from *node1*. At the node *node1*, events will be processed simultaneously. They will refer to the current sequence value zero, and each rule invocation will result in updating it to one. Meanwhile, all other nodes will receive and process the incoming ping events one after another, and iteratively increment their respective sequence numbers, as we would expect, from zero to ten.

We argue that the observed behavior of the P2 program is unexpected, since the first two rules do not contain any predicates that should be evaluated differently on different nodes.

4 P2 programs

In this section, we present a definition of P2 programs, which is used to define their operational semantics in Section 5.

A distributed P2 program $P = \langle \mathcal{L}, \mathcal{D}, \mathcal{K}, \mathcal{R}, S_0 \rangle$ consists of

- $\mathcal{L}$: a set of predicate symbols,
- $\mathcal{D}$: a set of data elements,
- $\mathcal{K}$: a keys-declaration,
- $\mathcal{R}$: a set of declarative rules,
- S_0: an initial state.

Predicates and tuples. Each predicate symbol in $p \in \mathcal{L}$ is associated with an arity n that is strictly greater than zero. A *predicate* is an expression of the form $p(v_1, \ldots, v_n)$, where $p \in \mathcal{L}$ is a predicate symbol of arity n and $v_1, \ldots, v_n$ are variables from a set of variables $\mathcal{V}$. We will also often use the notation $p(\boldsymbol{v})$, where $\boldsymbol{v}$ is a sequence of variables of the appropriate length. A *tuple* is obtained by, given a predicate $p(\boldsymbol{v})$, applying a substitution $\sigma\colon \mathcal{V} \to \mathcal{D}$ to maps all the variables in the predicate to values from the data domain. The assumption that the arity of each predicate is strictly greater than zero is due to the convention that the first argument of a predicate as well as the first position of a corresponding tuple represent its *address*. Henceforth, we shall omit the '@' symbol.

The set of predicate symbols is partitioned into two disjoint sets of materialized and event predicate symbols $\mathcal{M}$ and $\mathcal{E}$, respectively. We have

$$\mathcal{L} = \mathcal{M} \uplus \mathcal{E} \ .$$

Tuples obtained from materialized predicates are called materialized tuples. Similarly, we obtain event tuples by applying substitutions to event predicates.

Key declarations. A *keys-declaration* is a function $\mathcal{K}$ that maps each materialized predicate symbol $p \in \mathcal{M}$ of arity n to a subset $\mathcal{K}(p) \subseteq \{1, \ldots, n\}$ of indices of its fields. We assume that $1 \in \mathcal{K}(p)$ for all materialized predicates, i.e., the address of a predicate is always an element of its key. Given a materialized tuple $m = p(c_1, \ldots, c_n)$, we write $\mathcal{K}{\downarrow}m$ to denote the tuple obtained by removing all fields that are not included into the set $\mathcal{K}(p)$. For example, given $m = p(c_1, c_2, c_3, c_4)$ and $\mathcal{K}(p) = \{1, 3\}$, we obtain $\mathcal{K}{\downarrow}m = p(c_1, c_3)$.

We say that a set of materialized tuples M is *keys-inconsistent* if M contains a pair of tuples m and m' such that $m \neq m'$ but $\mathcal{K}{\downarrow}m = \mathcal{K}{\downarrow}m'$, i.e., it contains two tuples with the same key but different values. Otherwise, we say that the set of materialized tuples is *keys-consistent*.

Rules. In order to avoid some of the concerns with the interpretation of P2 programs discussed in the previous section, we include an action specification as part of the rule declaration. Formally, an *action* is a keyword in the set $\mathcal{A} = \{\mathsf{add}, \mathsf{delete}, \mathsf{send}, \mathsf{exec}\}$. These keywords correspond to adding and deleting materialized tuples from tables, as well as sending external events and executing internal events. A *rule* in a P2 program

$$\underbrace{\alpha\ h(\boldsymbol{v})}_{\text{head}}\ :-\ \underbrace{\overbrace{t(\boldsymbol{v}_0)}^{\text{optional}}, m_1(\boldsymbol{v}_1), \ldots, m_n(\boldsymbol{v}_n).}_{\text{body}}$$

consists of a head and a body separated by the ':−' symbol such that

– $\alpha \in \mathcal{A}$ is an action and $h(\boldsymbol{v})$ is a predicate,
– the body may contain an event predicate $t(\boldsymbol{v}_0)$ called *trigger*,
– the rest of the body consists of materialized predicates $m_i(\boldsymbol{v}_i)$, for $1 \leq i \leq n$,
– all variables in the head must appear in the body of the rule.

We require the following correspondence between the action and head predicate:

– if the rule action is add or delete then $h(\boldsymbol{v})$ must be a materialized predicate, and otherwise $h(\boldsymbol{v})$ must be an event predicate,
– if the rule action is exec then all predicates appearing in the rule must have the same address, i.e., either the same variable or the same constant at the first position.

If a trigger is present in the rule then we call it a *soft-rule*. Otherwise we say that the rule is a *materialized-rule*. Intuitively, triggers are used to control which rules have to be evaluated and when, i.e. a soft-rule is not evaluated until an event that matches its trigger is seen by a node. Materialized rules (without triggers) are evaluated whenever an update is made to any of the predicates on its body.

Note that a rule can contain predicates referring to different addresses (except for exec rules). We need to make an additional assumption on the interplay between addresses appearing in the rule to ensure the possibility of its execution in a distributed setting, as formalized by the following definitions.

Given a rule r, let x and y be the addresses of two predicates in the body of r. We say that x is *linked* to y, denoted $x \rightsquigarrow y$, if there is a predicate $p(x, \boldsymbol{v})$ in the body of r such that y occurs among the set of parameters $\boldsymbol{v}$. x is *connected* to y if $x \rightsquigarrow^* y$, where $\rightsquigarrow^*$ is the reflexive and transitive closure of $\rightsquigarrow$. An address x is a *source* in the body of r, if $x \rightsquigarrow^* y$ for all addresses y that also appear in the body of r. Finally, we say that a rule is *well-connected* if its body has at least one source. From now on, we shall only consider P2 programs whose rules are well-connected.

Local and basic rules. Execution in distributed setting requires a further distinction of P2 rules. We say that a rule is *local* if all predicates in its body have the same address and i) either the rule action is send, or ii) the address of the head is equal to the address of the predicates in the body. In particular, exec rules are always local. Finally, a *basic* rule is both soft and local. We say that a P2 program is *local* (resp. *basic*) if it only contains local (resp. basic) rules.

We illustrate different rule kinds on the following example. Here, we assume materialized predicates m, n, and p together with event predicates e and t.

r1: send $e(x, y) :- m(x, z), m(y, z)$.
r2: delete $m(w, v) :- e(x, y, v), n(y, v, z), m(z, w)$.
r3: send $t(y) :- m(x, v), n(x, v, y)$.
r4: add $m(x, w) :- m(x, u), m(x, v), p(x, u, v, w)$.
r5: add $m(x, w) :- t(x), m(x, u), m(x, v), n(x, u, v, w)$.

The rule *r1* is invalid since x and y are disconnected in its body. All other rules are well-connected. Rules *r3*–*r5* are local. Soft rules are *r2* and *r5*, while all other rules are materialized. The only basic rule *r5*, i.e., it is soft and local.

States. We define a *state* of a P2 program to be a pair $S = \langle M, E \rangle$ that consists of a *materialized store* M and an *external event queue* E. The store M is a keys-consistent set of materialized tuple. The queue E is a multi-set of event tuples. The *initial* state S_0 shall be used to start the program execution, as described in the next section.

5 Semantics of basic P2

We present the operational semantics for P2 programs, i.e., we show how program rules are evaluated with respect to the current state of the program and define the resulting state. In this section, we only consider P2 programs that contain only basic rules. Section 6 presents a transformation from an arbitrary program to a basic one. Such incremental approach simplifies the exposition and separates the rule localization from rule execution.

Figure 3 shows a procedure EVALUATE for the execution of basic P2 programs. It consists of two nested loops. We refer to an iteration of the outer and inner loops as *step* and *round*, respectively. The state of the program under execution is maintained by the pair $S = \langle M, E \rangle$, which contains the materialized store and the external event queue. We use an auxiliary function UPDATE that updates the materialized store. Note that this function is non-deterministic, since a P2 program can attempt to simultaneously add several tuples that are not keys-consistent. In such case, the resulting materialized store depends on the choice of tuples in line 2 of UPDATE.

At each step, some events are selected from the external event queue, and then several rounds are executed to compute the effects of applying the program rules triggered by the selected events. We leave several choices open as parameters of the execution procedure, e.g., how many events are selected for processing at each step and when updates are actually applied to the materialized store.

It is important to note is that although EVALUATE is described from the global perspective on the state of the program, its adaption to the distributed perspective is straightforward. In the distributed perspective, each node executes the same procedure EVALUATE. Since rules are basic, they can be executed locally without requiring any information about the tuples stored at other nodes. The only change required in the exposition of the procedure is at line 10, where events may be sent over the network using an appropriate transport mechanism instead of being added directly to the local event queue E.

The following are the design choices left open in the EVALUATE procedure:

- **External selection**: We do not specify which events and how many of them are selected at line 3 from the external event queue to be processed at each step. Two possible choices are: (1) to non-deterministically select **one** event from the queue, or (2) to select **all** current events in the queue.
- **Internal selection**: Similarly, we do not specify which events to select from the internal event queue I at line 6. Again, we could select either (1) **one** non-deterministically chosen, or (2) **all** events in the queue.

```
    procedure EVALUATE
    input
        P = ⟨ℒ, 𝒟, 𝒦, ℛ, S₀⟩ : a basic program
    vars
        M : materialized store
        E : external event queue
        △,▽ : set of tuples to add and delete
        I, J : internal event queues
    begin
 1:     ⟨M, E⟩ := S₀
 2:     while E ≠ ∅ do                                          ▷ step loop
 3:         I := select and remove elements from E
 4:         ⟨△,▽⟩ := ⟨∅, ∅⟩
 5:         while I ≠ ∅ do                                      ▷ round loop
 6:             J := select and remove elements from I
 7:             for each rule α h(v) :− t(v₀), m₁(v₁), ..., mₙ(vₙ) ∈ ℛ
 8:                 and subst. σ such that t(v₀)σ ∈ J and mᵢ(vᵢ)σ ∈ M
    do
 9:                     case a of
10:                         send: E := E ∪ {h(v)σ}
11:                         exec: I := I ∪ {h(v)σ}
12:                         add: △ := △ ∪ {h(v)σ}
13:                         delete: ▽ := ▽ ∪ {h(v)σ}
14:                     end case
15:             if update after each round then
16:                 M := UPDATE(M, 𝒦, △,▽)
17:                 ⟨△,▽⟩ := ⟨∅, ∅⟩
18:             if
19:             if use only one cycle then break while
20:         done                                                ▷ end round loop
21:         E := E ∪ I
22:         if update after each step then M := UPDATE(M, 𝒦, △,▽)
23:     done                                                    ▷ end step loop
    end.

    function UPDATE
    input
        M : a set of materialized tuples
        𝒦 : a keys-declaration
        △,▽ : sets of tuples to add and delete
    begin
 1:     M := M \ ▽
 2:     for each m ∈ △ do
 3:         M := (M \ {m′ | 𝒦↓m = 𝒦↓m′}) ∪ {m}
    end.
```

Fig. 3. Parametrized procedure for evaluating basic P2 rules, and its auxiliary function to update the materialized store.

– **Update**: As the execution proceeds, the procedure uses the pair of sets of tuples $\langle \triangle, \triangledown \rangle$ to record the updates that have to be applied to the materialized store. These updates can be atomically applied either: (1) at the end of every **round** in line 15, or (2) at the end of every **step** in line 22.
– **Number of cycles**: EVALUATE processes events using **two** cycles, viz., the round and the step cycles. By breaking the internal loop at line number 19, we achieve an execution behavior with only **one** evaluation cycle.

These design choices are independent, although some combinations of parameters are not viable. For example, if considering only **one cycle**, then the choice when to apply the materialized updates becomes irrelevant. Moreover, only the cumulative effect of both selection functions is important, i.e., if they select in conjunction either **one** or **all** events to process at each iteration.

If the variant with **two cycles** is used with **step updates**, then the internal selection function becomes irrelevant. As no changes to the materialized store are performed within the evaluation of rounds, the sets E and $\triangle, \triangledown$ do not depend, when exiting the inner loop, on the particular choice of internal selection.

Emulating P2 implementation. To the best of our knowledge, the set of parameters required to emulate the semantics currently implemented in P2 corresponds to the version with **two** evaluation cycles, selecting **one** external event for processing each step, fully propagating the internal events in rounds until fix-point, and updating only after the end of the **step**.

Recall that the original definition of P2 rules, see [10], does not specify the actions, except for rules whose action is delete. If the head of the rule contains a materialized predicate then the rule is implicitly assumed to have a add action. We observe that in our formalization a basic rule of the form

$$h(x, \boldsymbol{v}) :- t(y, \boldsymbol{v}_0), m_1(\boldsymbol{v}_1), \ldots, m_n(\boldsymbol{v}_n).$$

where $h(x, \boldsymbol{v})$ is an event predicate corresponds to the pair of rules below.

$$\mathsf{send}\ h(x, \boldsymbol{v}) :- t(y, \boldsymbol{v}_0), m_1(\boldsymbol{v}_1), \ldots, m_n(\boldsymbol{v}_n), x \neq y.$$
$$\mathsf{exec}\ h(x, \boldsymbol{v}) :- t(y, \boldsymbol{v}_0), m_1(\boldsymbol{v}_1), \ldots, m_n(\boldsymbol{v}_n), x = y.$$

The explicit treatment of rule actions makes visible whether the effect of evaluating these rules is different depending on whether x and y are equal addresses. Furthermore, if we would like a node to send a message to itself, but behave as if the message was received from the network, we have now the possibility to write a rule

$$\mathsf{send}\ h(x, \boldsymbol{v}) :- t(y, \boldsymbol{v}_0), m_1(\boldsymbol{v}_1), \ldots, m_n(\boldsymbol{v}_n).$$

This scheme can be used to modify the program given in Figure 2 such that all nodes exhibit the same behavior, i.e., count up to ten.

6 Reduction to basic rules

The procedure EVALUATE for the execution of P2 program, as presented in the previous section, assumed that the input program consists of basic rules. In this

section, we relax this assumption and show how any program with well-connected rule can be transformed into a basic one. The transformation proceeds in two steps by first making the rules local, and then turning materialized rules into soft rules. Since these transformation can be automatically performed by the P2 runtime environment, they liberate the programmer from the burden of ensuring the local availability of tuples necessary to rule execution and event handling to signal updates to the materialized store.

Rule localization. Let r be a well connected rule of the form

$$\alpha\ h(\boldsymbol{v}) :- p_1(x, \boldsymbol{v}_1), \ldots, p_{k-1}(x, \boldsymbol{v}_{k-1}), p_k(y_k, \boldsymbol{v}_k), \ldots, p_n(y_n, \boldsymbol{v}_n).$$

such that $x \neq y_i$ for each $k \leq i \leq n$, and $x \rightsquigarrow y_k$. Let $\boldsymbol{v}'$ be a sequence of the variables that appear in the set $\{\boldsymbol{v}_0, \ldots, \boldsymbol{v}_{i-1}\}$, and let $|\boldsymbol{v}'|$ denote its length. In particular, we have that y_k occurs in $\boldsymbol{v}'$, and there is at most one event predicates among $p_1, \ldots, p_{k-1}$ (recall that a rule can have at most one trigger). If all these predicates are materialized we define β to be `add` action and q be a fresh materialized predicate of arity $|\boldsymbol{v}'| + 2$ with keys-declaration $\mathcal{K}(q)$ that contains all predicate fields, i.e., $\mathcal{K}(q) = \{1, \ldots, |\boldsymbol{v}'| + 2\}$. Otherwise, we have $\beta =$ `send` and q is a fresh event predicate symbol of arity $|\boldsymbol{v}'| + 2$.

Now the original rule r is replaced by the pair of rules

$$\begin{aligned} \beta\ q(y_k, x, \boldsymbol{v}') &:- p_1(x, \boldsymbol{v}_1), \ldots, p_{k-1}(x, \boldsymbol{v}_{k-1}). \\ \alpha\ h(\boldsymbol{v}) &:- q(y_k, x, \boldsymbol{v}'), p_k(y_k, \boldsymbol{v}_k), \ldots, p_n(y_n, \boldsymbol{v}_n). \end{aligned}$$

Note that the first rule has a local body, and the second rule is a well-connected rule with exactly one address variable less than the original rule r. By iteratively applying the above transformation to the non-local rules in the program, we obtain a P2 program whose rules have local bodies.

After applying this transformation some rules may still not be local, since the address in the head of a materialized update may not be equal to the address in the rule body. Such rules have the form

$$\alpha\ h(y, \boldsymbol{v}) :- p_1(x, \boldsymbol{v}_1), \ldots, p_n(x, \boldsymbol{v}_n).$$

where $x \neq y$ and α is either `add` or `delete` action. We replace each of these rules by the pair below

$$\begin{aligned} \mathsf{send}\ u(y, \boldsymbol{v}) &:- p_1(x, \boldsymbol{v}_1), \ldots, p_n(x, \boldsymbol{v}_n). \\ \alpha\ h(y, \boldsymbol{v}) &:- u(y, \boldsymbol{v}). \end{aligned}$$

where u is a fresh event predicate symbol of arity $|\boldsymbol{v}| + 1$. Now all rules are local.

Rule softening. We replace all materialized rules in the program, after rule localization procedure was applied, by soft rules as follows. First, for each materialized predicate m we create a fresh event predicate $\hat{m}$ of the same arity. Then, we replace each materialized rule of the form

$$\alpha\ h(\boldsymbol{v}) :- m_1(\boldsymbol{v}_1), \ldots, m_n(\boldsymbol{v}_n).$$

by n soft rules such that for each $1 \leq i \leq n$ we insert

$$\alpha\ h(\boldsymbol{v}) :- \hat{m}_i(\boldsymbol{v}_i), m_1(\boldsymbol{v}_1), \ldots, m_n(\boldsymbol{v}_n).$$

where the new update event $\hat{m}_i$ serves as the rule trigger.

Finally, we create rules that generate update events whenever a new tuple is inserted to a table. For this purpose for each add rule of the form

$$\mathsf{add}\ m(\boldsymbol{v}) :- t(\boldsymbol{v}_0), m_1(\boldsymbol{v}_1), \ldots, m_n(\boldsymbol{v}_n).$$

we create a fresh event predicate u of the same arity as m, and replace the rule by the three basic rules below.

$$\begin{aligned}
\mathsf{exec}\ u(\boldsymbol{v}) &:- t(\boldsymbol{v}_0), m_1(\boldsymbol{v}_1), \ldots, m_n(\boldsymbol{v}_n).\\
\mathsf{add}\ m(\boldsymbol{v}) &:- u(\boldsymbol{v}).\\
\mathsf{send}\ \hat{m}(\boldsymbol{v}) &:- u(\boldsymbol{v}).
\end{aligned}$$

We note that more sophisticated methods can be defined to localize and soften a set of well connected rules. For example, such methods may seek for opportunities to distribute the evaluation of the body among different nodes. We leave the development of optimized localization and softening methods for future work, since this paper focuses on a definition of semantics for basic rules. Our localization method can be viewed as a generalization of the *rule localization rewrite* procedure defined by Loo et al. [5] for rules containing at most two different addresses.

Moreover, similar to the treatment of localization in [5] our semantics together with the transformations is *eventually consistent* under the *bursty update* network model for P2 programs that contains only materialized rules with add actions. (Such rules define almost-Datalog programs, since they still contain function symbols.) This kind of consistency means that if after a burst of updates the network eventually quiesces then the models defined by our semantics correspond to those of the standard semantics of Datalog.

7 Conclusions

We presented a definition and operational semantics for the P2 programming language, which provides a programming foundation for Declarative Networking. Our work addresses questions that were left open by the existing literature on Declarative Networking. The main contribution of our semantics is in its utility as a starting point for the development of program analysis and verification tools for Declarative Networking, as well as advancing the evolution of the P2 language, its interpreters and runtime environments.

References

1. Armstrong, J.: Making reliable distributed systems in the presence of software errors. PhD thesis, KTH (2003)

2. Fournet, C., Fessant, F.L., Maranget, L., Schmitt, A.: JoCaml: A language for concurrent distributed and mobile programming. In: Advanced Func. Prog., Springer (2002)
3. Hanus, M.: Distributed programming in a multi-paradigm declarative language. In: PPDP, Springer (1999)
4. Killian, C.E., Anderson, J.W., Braud, R., Jhala, R., Vahdat, A.: Mace: language support for building distributed systems. In: PLDI, ACM (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: Language, execution and optimization. In: SIGMOD, ACM (2006) 97–108
6. Sewell, P., Leifer, J.J., Wansbrough, K., Nardelli, F.Z., Allen-Williams, M., Habouzit, P., Vafeiadis, V.: Acute: High-level programming language design for distributed computation. J. Funct. Program. **17**(4–5) (2007) 547–612
7. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup protocol for Internet applications. IEEE/ACM Trans. Netw. **11**(1) (2003) 17–32
8. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware, Springer (2001)
9. Chu, D., 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, ACM (2007)
10. Loo, B.T., Condie, T., Hellerstein, J.M., Maniatis, P., Roscoe, T., Stoica, I.: Implementing declarative overlays. In: SIGOPS, ACM (2005) 75–90
11. Singh, A., Das, T., Maniatis, P., Druschel, P., Roscoe, T.: BFT protocols under fire. In: NSDI, USENIX (2008)
12. Condie, T., Gay, D.E., Loo, B.T., et al.: P2: Declarative networking website (2008)
13. Loo, B.T.: The Design and Implementation of Declarative Networks. PhD thesis, UC Berkeley (2006)
14. Rossberg, A., Botlan, D.L., Tack, G., Brunklaus, T., Smolka, G.: Alice through the looking glass. In: Trends in Func. Prog., Intellect (2004)
15. Rossberg, A., Tack, G., Kornstaedt, L.: Status report: HOT pickles, and how to serve them. In: Workshop on ML, ACM (2007) 25–36
16. Beyer, D., Noack, A., Lewerentz, C.: Efficient relational calculation for software analysis. Trans. on Soft. Eng. **31**(2) (2005) 137–149
17. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI, ACM (2004)
18. Carriero, N., Gelernter, D.: Linda in context. Commun. ACM **32**(4) (1989)