On the Declarativity of Declarative Networking

Yun Mao
AT&T Labs - Research
180 Park Ave, Florham Park, NJ, USA
maoy@research.att.com

ABSTRACT

Initiated by the declarative networking project, rule-based declarative programming languages have gained increasing popularity in building complex networked systems across multiple application domains. This paper investigates the *declarativity* of those systems. First, by analyzing the language semantics, we classify rules into deductive rules and Event-Condition-Action (ECA) rules, and reveal their different levels of declarativities. Then, we use case studies to show that ECA rules that are less declarative are dominantly used in most of the proposed systems. As a result, the benefit of declarative programming is undermined. We identify the key factors that cause the low declarativity effect, and present our ongoing work towards addressing those challenges.

1. INTRODUCTION

Designing and implementing complex distributed systems has long been known as a challenging, tedious, and error-prone task. Among many proposed frameworks that try to tackle the problem from different perspectives [5, 17], declarative networking [10] is a unique approach that promotes declarative, data-driven programming to concisely specify and implement distributed protocols and services.

The original vision behind declarative networking is to write (recursive) database queries to declaratively specify the routing tables of nodes in a rule-based language derived from Datalog [12]. As a result, distributed data flows that are automatically generated by the query planner virtually implement the sending and receiving of control packets in traditional routing protocols. Since then, numerous proposals have derived the idea from routing to broader domains, such as overlay protocols [11], Byzantine fault tolerance (BFT) protocols [18], sensor networks [3], data and control-plane composition [13], storage systems [1], wireless networks [8], and security policies [21].

Acknowledging the increasing popularity and initial success of declarative networking and its derivatives, this paper begins with investigating the following question: *how declarative are these systems?* Having a high degree of declarativity is important in carrying out the declarative approach and explore its benefit to the full potential. Unfortunately, our analysis shows a surprisingly low level of declarativity in most proposed systems. To this end, we identify the key factors that cause the low declarativity effect, and present our ongoing work, the WIND language that aims at addressing those challenges.

This paper makes the following contributions: (1) by analyzing the semantics of OverLog, the language constructs of both high and low declarativity are identified; (2) a diverse group of declarative programs in different application domains are analyzed to demonstrate the dominance of low declarativity in existing declarative systems; (3) we analyze the root causes and challenges behind the low declarativity effect, and present our ongoing work, the WIND language, that aims at addressing those challenges.

2. DECLARATIVE PROGRAMMING

Declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow. More informally, declarative programming is about specifying *what* is to be computed, not necessarily *how* it is to be computed. This is in contrast to imperative programming, which requires a detailed description of the algorithm of computation.

There are two major components in declarative programming: a programming language and its execution system. A program is written in the language as the description of the computing task, and the execution system *automatically* figures out the control logic, i.e. the procedural implementations based on the given program. For example, writing database queries in SQL is considered as declarative programming. Other typical declarative languages include Datalog, Haskell, etc.

In practice, many programming languages are hybridized with both declarative and imperative language constructs. For example, OCaml known as a functional (hence declarative) language allows programmers to write statements with side-effects. While the `SELECT` statement in SQL is declarative, the `INSERT`, `UPDATE` and `DELETE` statements are imperative.

To analyze a hybrid language, we use two criteria to assess the *declarativity* of its language constructs: (1) from the

language's perspective, a construct with high declarativity has a declarative semantics that gives users a way to understand the program without thinking about how the program is to be executed, while a construct with low declarativity is best understood by its operational semantics during execution; (2) from the execution system's perspective, there is a non-trivial algorithm that automatically translates the declarations into implementations for a construct with high declarativity, while no such algorithms exist for a construct with low declarativity.

3. DECLARATIVE NETWORKING

Declarative networked systems are written in languages derived from Datalog. In this section, we start from the background on Datalog, then use OverLog and its execution system P2 [11, 4] to study the language features and semantics. OverLog is chosen because many of the proposed languages (NDlog [10], Snlog [3], Mozlog [13], R/Overlog [1]) are derived from OverLog [11] without major semantic difference that matters to the declarativity analysis in this paper.

3.1 Background on Datalog

A Datalog program consists of a set of deductive *rules*. Each rule has a rule head and a rule body, which are separated by the deduction symbol(":-"). A predicate is a relation name with variables or constants as arguments. A rule body is a list of predicates or Boolean expressions that conjunctively derive the predicate in the corresponding rule head. A relation can be either in an extensional database (EDB) or an intensional database (IDB). EDB relations are the inputs of the program and remain constant during the evaluation. IDB relations are deduced based on EDB (and possibly IDB) relations according to the rules. As a result, EDB relations can never appear in a rule head.

```
r1 reachable(S,D) :- link(S,D).
r2 reachable(S,D) :- link(S,N), reachable(N,D).
```

In the above example, `link` is an EDB relation, representing directional link in a graph. `reachable` is an IDB relation derived from `link` that represents all-pair reachability. Intuitively, each Datalog rule can be explained as a first-order logic assertion as "if everything in the rule body is true then the rule head is true". We refer such explanations as *intuitive conditions*. In the example, rule `r1` computes all pairs of nodes reachable within a single hop from all input links, and rule `r2` expresses that "if there is a link from `S` to `N`, and `N` can reach `D`, then `S` can reach `D`." These rules are declarative because they only specify what conditions the `reachable` relation should satisfy, rather than procedures to compute it.

Unfortunately, such intuitive readings of rules are not rigorous enough to eliminate unwanted results. For example, one can construct a `reachable` relation that includes all possible pairs of nodes in the network. No matter which nodes are connected as in `link`, the intuitive conditions (if the body is true, the head is also true) are always satisfied. To address this problem, Datalog adopts the *least model semantics* [16]. Informally, least model semantics dictates that a tuple is not included in an IDB relation unless it is necessary to satisfy one of the rules defining it.

3.2 OverLog

OverLog introduces the following major extensions.

Horizontal partitioning: The data model in OverLog consists of relational tables that are partitioned across the nodes in a network. Each relation in a rule must have one attribute that is preceded by an @ sign, which is called the location specifier of the relation. Location specifiers describe the horizontal partitioning of the relation: each tuple is stored at the address found in its location specifier attribute. Tuples must be stored at the address of their location specifier, and hence some of its derived tuples in rule evaluation process are sent across the network to achieve this physical constraint. As a result, network communication is implicit in OverLog. In the evaluation process, all rules are first transformed to "localized rules" such that the location specifiers in all predicates in a rule body are instantiated by the same variable [10].

Hard-state and events: Each OverLog table has a soft-state lifetime (from zero to infinity) that determines how long a tuple in that table remains stored before it is automatically deleted. In this paper we only consider two representative cases: zero-lifetime tables as event tables, and infinite-lifetime tables as hard-state tables.

Based on types of the predicates, we classify rules into four categories: H-H rule (or hard-state rule) that contains only hard-state predicates, E-E rule that has an event predicate in the rule head and an event predicate in the rule body, E-H rule that has an event predicate in the rule head but all hard-state predicates in the rule body, and H-E rule that has a hard-state rule head and an event predicate in the rule body. [1] All four kinds of rules can be informally explained by the intuitive conditions (the rule body implies the rule head). However, they follow different semantics.

H-H rules closely resemble the original least model semantics in Datalog. Because EDB relations in OverLog may not remain constant (e.g. a `link` relation that represents connectivity in a dynamic network), IDB relations should be viewed as continuous queries over the tables in EDB, which introduces consistency concerns. P2 achieves *eventual consistency* within a bursty update model [10]: the assumption is that after a burst of updates in EDB, the network eventually quiesces (does not change) for a time long enough to allow all the queries in the system to reach a fixpoint where nothing changes. At this point, the least model semantics is satisfied.

Rules that involve events (i.e. E-H, H-E, and E-E rules) do not have the same semantics as hard-state rules. We explain how these rules are evaluated in P2 to illustrate their operational semantics as follows, and refer interested readers to [14] for a detailed discussion.

- E-E rule: if at time t_1, the rule body is true, then $\exists t_2$ $(t_2 > t_1)$, at which the event predicate in the rule head is true.

[1] This classification is specialized for cases where only event and hard-state predicates are used. A more general classification that include soft-state predicates with non-zero lifetime can be found in [9]. H-H, E-E, E-H, and H-E rules are subsets of hard-state, pure soft-state, derived soft-state, and archival soft-state rules, respectively.

- H-E rule: if at time t, the rule body is true, then before the next fixpoint, a tuple is inserted in the head relation such that the rule head is true.
- E-H rule: In the simple case where there is only one hard-state predicate in the rule body, if at time t_1 a new tuple is inserted in the predicate, then $\exists t_2$ ($t_2 > t_1$), at which the event predicate in the rule head is true. If there are multiple predicates in the rule body in the form of `e :- h1, h2, ...`, one can rewrite this rule to an H-H rule `h :- h1, h2, ...` and an E-H rule `e :- h.`, and explain them based on the simple case semantics.

Deletions and updates: OverLog supports deletions and updates to tuples in materialized tables. An H-E rule can be prefaced by the keyword `delete`, and the rule body specifies facts to be deleted. Each materialized table must also have a primary key specification. Any derived tuple for that relation that matches an existing tuple on the primary key is intended to replace that existing tuple.

4. DECLARATIVITY

Having presented the features and semantics of OverLog, we analyze its declarativity and discuss how they are used in existing systems.

4.1 Analysis

We analyze the declarativity of OverLog based on the criteria described in Section 2 by examining the following questions. To what extent can a programmer understand the outcome of the semantics without thinking about the evaluation process? Is there a non-trivial algorithm that translates the declaration into the implementation?

High declarativity (1) *horizontal partitioning*: with location specifiers in the predicates, the programmer can understand where the tuples are stored without thinking about how to ship them via network. The rule localization algorithm is used to automatically generate the evaluation process that carries out the implicit network communication.

(2) *H-H rules*: using the least model semantics and eventual consistency, a programmer can reason about what is the result of the IDB relations without thinking about the evaluation process. Each IDB relation is considered as a distributed materialized view, and the incremental view maintenance algorithm is the one that does the translation from declarations to implementations by P2.

Low declarativity (1) *H-E rules, deletions and updates*: the semantics of such rules only define insertion/update/deletion operations in the process of the evaluation. There is no declarative semantics like the least model semantics to understand what the relations in the rule head should be. These rules are Event-Condition-Action (ECA) rules that when an event (the event relation in the rule body) occurs, evaluate the condition (the rest of the rule body), and execute the action (the rule head) if the condition is satisfied. ECA rules are considered less declarative than deductive rules [20]. They are also known as *triggers* in a traditional DBMS. Using triggers are usually not encouraged if their functions can be implemented declaratively otherwise (e.g. using constraints) [15].

(2) *E-E, E-H rules*: similar to H-E rules, E-E and E-H rules can also be viewed as ECA rules, where the actions are to send events instead of to manipulate hard-state relations. This explanation naturally fits the operation semantics. It is inappropriate to think declaratively in the eventual consistency model, because eventually all event relations are empty due to the zero-lifetime policy. Nevertheless, one might be able to apply a less obvious declarative thinking if she considers multiple snapshots of the system state. For example, in an E-E rule "`e2:-e1,h1`", at t_1 when `e1` is received and `h1` is true. At t_2, `e2` becomes true. If one combines the snapshot of IDB relations `e2` at t_2 and the snapshot of EDB relations `e1,h1` at t_1, then the least model semantics applies. Compared with eventual consistency, it is less straightforward to think declaratively across multiple snapshots.

(3) *Mixing H-H and H-E rules*: if a hard-state relation appears in rule heads of both H-H rules and H-E rules, the declarative semantics of H-H rules do not apply. One can only use operational semantics of the H-H rules (i.e. the implementation of the incremental view maintenance algorithm) to analyze with the H-E rules. As a result, if the underlying implementation changes, the program behavior might change as well. Although such mixtures are considered as "bugs" [9], they are used occationally in practice (see Section 4.2).

To summarize, H-H rules are *deductive rules* with high declarativity, while E-E, E-H and H-E rules are best understood as *ECA rules* with low declarativity.[2]

4.2 Case study

We examine 5 representative OverLog programs that are written by 3 research groups: path vector routing protocol [12], Narada overlay mesh [11], Chord DHT [11], the Zyzzyva BFT protocol [18], and the Coda file system [1]. Two classification methods are used: (1) each rule is classified as one of the H-H, H-E, E-H, and E-E rules; (2) each rule is classified as either a localized rule (where all location specifiers in the rule body is the same variable) or a non-localized rule. The goal is to quantify rules with high declarativity. Recall from Section 4.1 that H-H rules have declarative semantics and take advantage of the incremental view maintenance algorithm. Non-localized rules leverage the concept of distributed database with horizontal partioning, and utilize the localization algorithm.

As shown in Table 1, the results demonstrate a surprisingly low level of declarativity. First, other than the path vector program, rules with low declarativity (E-E, H-E, E-H rules) significantly outnumber the rules with high declarativity (H-H rules). To make things worse, the rule head predicates in the two H-H rules in Chord and one of the H-H rules in Zyzzyva are also used as head predicates in other H-E rules, which effectively eliminates the declarative semantics of those H-H rules. Second, three programs (Narada, Zyzzyva, Coda) have zero non-localized rules. Even in the two programs with non-localized rules, they only count as a relatively small amount. This might suggest that the

[2]We by no means suggest that E-E, E-H and H-E rules are equally (low) declarative. In fact, in our belief E-E rules have medium declarativity. The detailed discussion is omitted due to space constraints.

Table 1: Declarativity case study of OverLog

	total	H-H (mixed)	E-E	H-E	E-H	localized	non-localized
Path Vector	4	4 (0)	0	0	0	3	1
Narada	17	0 (0)	7	9	1	17	0
Chord	46	2 (2)	20	22	2	43	3
Zyzzyva	147	4 (1)	86	41	16	147	0
Coda	42	0 (0)	19	20	3	42	0

programmers tend to shy away from the declarative distributed query evaluation, and may want more control for the dataflow over the network.

4.3 Discussion

We argue that three major factors lead to the low declarativity effect:

Language factor Deductive rules in OverLog may not be expressive enough to represent what a user wants. E.g., OverLog does not support negation so that one has to use H-E rules to delete tuples from relations as needed. Besides, because the semantics is based on first-order logic, it is very difficult, if possible, to describe rules that involve time (such as something must happen after something else). Therefore, one has no choice but to use ECA rules to encode such logic.

Planner factor Even if the language is expressive to declare all wanted goals, the planner in the execution system might not be sophisticated enough to meet a wide range of requirements in building complex networked systems. For example, in P2, a declarative query can only be stored as materialized views. If one needs to evaluate a query on the fly, or the query planner is not capable of generating an acceptable plan to execute the query, she has no choice but to encode such a query in ECA rules.

Human factor Most system and networking programmers are accustomed to the imperative thinking style. When the language exposes such a way of programming, people naturally move towards their comfort zones. As a result, when it is possible to write rules with either deductive rules or ECA rules, ECA rules with low declarativity might be more likely to be chosen. In addition, rules in OverLog with the same syntax have different semantics and different declarativities based on the types of the predicates. A programmer without thorough understandings of the semantic subtleties might not even realize that using ECA rules could lower the declarativity of the program.

Besides the fundamental factors described above, there are other practical reasons to consider. For example, building a system like P2 is no trivial task. Like most of the research prototypes, it contains bugs and performance issues. Although some of the problems are not related to declarativity[3], we argue that in general the level of complexity does have a positive correlation with the level of declarativity, because high declarative components tend to have complex algorithms that are fundamentally harder to implement correctly. For instance, the incremental view maintenance algorithm for recursive queries in a distributed environment is much harder to implement correctly, compared with implementing an ECA rule. P2 is known to not be able to update recursive hard-state IDB relations correctly when there are EDB tuple deletions in certain situations [4]. To get around with the potentially subtle bugs, ECA rules with less or no bugs are favored.

[3]For example, P2 internally embeds a stack-based virtual machine named PEL VM to implement dynamic code execution. The push/pull dataflow engine adopted from Click [7] also adds complexity. Both design decisions might incur performance penalty.

5. WIND LANGUAGE

Having presented the low declarativity effect, in this section, we discuss the design philosophy and features of the Wind (WIND Is Not Datalog) language with the goal of encouraging declarative thinking without compromising the power from the imperative features.

5.1 Philosophy

Neither declarative nor imperative programming is fundamentally more superior than the other. In our problem domain, deductive rules are usually more succinct, easier to understand. On the flip side, ECA rules, while less declarative, are more flexible to achieve things that deductive rules cannot. The design goal of Wind is to encourage declarative thinking without compromising the power provided by the imperative features.

We argue that it is unlikely for a language with only deductive rules to be practical. If we were to be able to eliminate ECA rules to raise declarativity, all factors in Section 4.3 must be solved. We acknowledge that some of the factors might be solved relatively easily. For example, introducing stratified negation improves the expressiveness of the language; adding syntactic salt to differentiate deductive rules and ECA rules could help users to better understand their differences in semantics; putting more engineering effort could improve performance and reduce the number of bugs. However, it is unclear whether a sophisticated planner would emerge to fit in a design space much larger than that of a traditional DBMS. For example, in a distributed system, the optimization goal of the planner is well beyond query execution time. Consistency, throughput, latency, power consumption, etc, are all valid concerns in different settings, which demand specialized optimizers.

Instead, Wind does not preclude imperative features. The design philosophy of Wind is to encourage programmers to *think declaratively, act freely, and verify optionally.* That is, the programmer must be able to provide a declarative specification of the program in a modular fashion, and can freely choose either to use the planner from the execution system to figure out the implementation plan if possible, or to realize the implementation with imperative features of the language. Optionally, one should be able to verify the cus-

tomized implementation against the specification, with the help of a model checker or theorem prover. State maintained by imperative code should also be able to be exportable by a declarative interface to interact with the rest of the code.

5.2 Features

Relations and rules WIND provides data structures including tables, events, and views. All of them are relations with statically *typed* attributes. One of the attributes in a relation is annotated as the location specifier. Views are IDB relations that are deduced from EDB relations, which include tables or views. Both deductive rules and ECA rules are supported. Deductive rules inherit the Datalog syntax with stratified negation. A deductive rule head predicate must be a view. In the execution system of WIND, views are materialized and incrementally maintained. ECA rules adopt a different syntax: `on event() [,conditions..] => action`. Events include user-defined events, system events (e.g. timer expired) or database events (e.g. tuple insertion or deletion in a table or view). Actions include sending user-defined events, performing database operations (e.g. inserting or deleting tuples in tables but not views), and system actions. Tuples in views are not allowed to be manipulated in ECA rule actions to avoid the "mixed declarativity" problem in OverLog where H-E and H-H rules with same predicates as rule head co-exist. It is intentional to have different syntax for deductive rules and ECA rules to highlight their differences in terms of semantics and declarativity.

Virtual views The default views are materialized views and are incrementally maintained. Virtual views provide the programmability to do query on-demand, caching, and more sophisticated optimization that is not supported by the default query planner in the execution system. For example, the lookup operation in Chord on a node `NI` takes a key `K` as input and returns the node `RI` which is responsible for the key. It is too expensive to maintain a materialized view `lookup(@NI,K,RI)` to hold all possible query results because the key space is too large. However, using virtual views, one can conduct query on-demand once the key `K` is known.[4]

A virtual view predicate $v(K_1, \cdots, K_n, R_1, \cdots, R_m)$ has a set of attributes $K_1, \cdots, K_n$ which are bounded to input values at runtime. The remaining attributes $R_1, \cdots, R_m$ represent the return values from invoking the virtual view query plan given the input values. In principle, this is akin to having a function interface that takes the bounded attributes as input, and returns the unbounded attributes. The implementation of a virtual view is written in ECA rules.

Specifications To bridge the declarativity gap between ECA rules and deductive rules, WIND allows a programmer to write specifications of relations (or specs in short). These specs resemble deductive rules (so that they are declarative), but they are not used by the execution system to generate implementations. Instead the implementations are still dictated by the ECA rules (not shown). Writing specs not only improves readability, but creates the possibility of verification—they can be fed to a theorem prover or a model checker as liveness properties [6] to verify the correctness of the implementations.

```
spec{
  succ(@NI, min<Dist>, SI) :-
    node(@NI, NodeKey), node(@SI, SuccKey),
    Dist := f_ring_dist(SuccKey,NodeKey),
    groupBy(NI);
}
```

The code above exemplifies the spec of the successor definition in the Chord protocol. It declares that the successor of a node is the closest node in the ring in the forwarding direction, where the distance is calculated by the function `f_ring_dist`. Note that because there is a distributed cross product in the query, a standard query planner would have produced a very expensive implementation to maintain `succ`. The Chord protocol is essentially a scalable incremental maintenance plan for this particular query.

Modules To improve code readability and reuse, rules are grouped by modules in WIND. Similar to Coral [19], a module specifies the public EDB and IDB relations (i.e. tables, events, views and virtual views) and the corresponding type information of a set of rules. EDB relations are the input of the module, and cannot be modified in the module. On the other hand, IDB relations are the output of the module, and can only be changed by the rules inside the module. Relations that are neither in IDB nor in EDB are considered private relations that are not visible to code outside of the module. A module can be composed by instantiating all of its EDB relations, e.g. from the IDB relations of existing modules.

5.3 Implementation

Different from P2 which dynamically load and interpret rules at runtime but similar to DSN [3], programs in WIND are statically compiled into binary executables. The WIND compiler is written in OCaml. Its front-end does parsing and query planning, and generate query plans in intermediate code format. The backend takes the plans and generate C++ code and then use GNU gcc compiler to finally emit machine code for execution. One of the up side of this design choice is performance gains. The benchmark on our preliminary prototype shows from 10 to 1000 times performance improvement over P2, depending on the workload. The other benefit is the reuseability of the frontend. Different compiler backends can be plugged in to produce code for model checkers, or to use implementation languages other than C++. The potential down side is that once the code is compiled, it is less flexible at runtime. For example, currently the generated code has fixed query plans so that it is hard to dynamically adapt plans based on different factors at runtime.

6. CONCLUSIONS AND DIRECTIONS

Using semantic analysis and case studies, we have showed that the declarativity of recent declarative networked systems has been decreasing. We argue that language expressiveness, planner sophistication and human factors are the keys causing such an effect. In WIND, we propose new language constructs such as virtual views and specifications to

[4]In fact, this implementation of virtual view behaves exactly like query evaluation with magic-set rewrite.

bridge the gap between declarative specifications and imperative implementations.

Our effort is merely the beginning in the area. Many interesting challenges and opportunities exist. For example, some DB concepts such as declarative constraints and transactions have not been fully explored. New application domains such as network management [2], cloud and mobile computing deserve more attention. More benefits of declarative programming could be taken advantage of, such as implicit parallelism for multi-core architecture, and metaprogramming [4] to embed planning algorithms to create specialized virtual views. Indeed, designing such a declarative language and its execution system demands collaborative inter-disciplinary efforts, including databases, networking, systems, programming languages and software engineering.

Acknowledgments

The author would like to thank Mary Fernández, Divesh Srivastava, Trevor Jim, Pamela Zave, Wenchao Zhou, Changbin Liu, Mike Dahlin, and the anonymous reviewers for their helpful comments on the draft of the paper, and Nalini Belaramani for providing the source code of Coda in OverLog.

7. REFERENCES

[1] N. Belaramani, J. Zheng, A. Nayte, M. Dahlin, and R. Grimm. PADS: A Policy Architecture for building Distributed Storage systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI'09)*, Apr. 2009.

[2] X. Chen, Y. Mao, Z. M. Mao, and J. E. van der Merwe. DECOR: DEClaritive network management and OpeRation. In *Proceedings of the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO)*, 2009.

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

[4] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita Raced: Metacompilation for Declarative Networks. In *Proceedings of Conference on Very Large Data Bases (VLDB)*, 2008.

[5] C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: Language Support for Building Distributed Systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[6] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Detecting Liveness Bugs in Systems Code. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI'07)*, Cambridge, MA, April 2007.

[7] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[8] C. Liu, Y. Mao, M. Oprea, P. Basu, , and B. T. Loo. A declarative perspective on adaptive manet routing. In *Proceedings of the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO)*, Seattle, WA, August 2008.

[9] B. T. Loo. The Design and Implementation of Declarative Networks (Ph.D. Dissertation). Technical Report UCB/EECS-2006-177, University of California at Berkeley, 2006.

[10] 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 *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 2006.

[11] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[12] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, Philadelphia, PA, 2005.

[13] Y. Mao, B. T. Loo, Z. G. Ives, and J. M. Smith. MOSAIC: Unified Declarative Platform for Dynamic Overlay Composition. In *Proceedings of the 4th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Madrid, Spain, Dec 2008.

[14] J. A. N. Pérez and A. Rybalchenko. Operational semantics for declarative networking. In *Proceedings of International Symposium on Practical Aspects of Declarative Languages (PADL)*, Jan. 2009.

[15] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, third edition, 2002.

[16] R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[17] P. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-williams, P. Habouzit, and V. Vafeiadis. Acute: high-level programming language design for distributed computation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 15–26, 2005.

[18] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT Protocols Under Fire. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI'08)*, Apr 2008.

[19] The Coral Project. http://www.cs.wisc.edu/coral/.

[20] J. Widom. Deductive and active databases: Two paradigms or ends of a spectrum? In *Proc. of 1st Workshop on Rules in Database Systems*, pages 306–315, 1993.

[21] W. Zhou, Y. Mao, B. T. Loo, and M. Abadi. Unified declarative platform for secure networked information systems. In *Proceedings of IEEE Conference on Data Engineering (ICDE)*, pages 150–161, 2009.