# DECOR: DEClarative network management and OpeRation

Xu Chen
Department of EECS
University of Michigan
Ann Arbor, MI
chenxu@umich.edu

Yun Mao
AT&T Labs - Research
Shannon Laboratory
Florham Park, NJ
maoy@research.att.com

Z. Morley Mao
Department of EECS
University of Michigan
Ann Arbor, MI
zmao@eecs.umich.edu

Jacobus Van der Merwe
AT&T Labs - Research
Shannon Laboratory
Florham Park, NJ
kobus@research.att.com

## ABSTRACT

Network management operations are complicated, tedious and error-prone, requiring significant human involvement and expert knowledge. In this paper, we first examine the fundamental components of management operations and argue that the lack of automation is due to a lack of programmability at the right level of abstraction. To address this challenge, we present DECOR, a database-oriented, declarative framework towards automated network management. DECOR models router configuration and any generic network status as relational data in a conceptually centralized database. As such, network management operations can be represented as a series of transactional database queries, which provide the benefit of atomicity, consistency and isolation. The rule-based language in DECOR provides the flexible programmability to specify and enforce network-wide management constraints, and achieve high-level task scheduling. We describe the design rationale and architecture of DECOR and present some preliminary examples applying our approach to common network management tasks.

## Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network Management

## General Terms

Design, Languages, Management

## Keywords

Network Management, Declarative Language

## 1. INTRODUCTION

Network management and operational tasks are performed on a daily basis in all large operational networks. These operational tasks span a wide range of activities including (i) planned maintenance, *e.g.,* to upgrade or introduce new equipment, (ii) emergency repair, *e.g.,* when a natural or human induced event causes failure or malfunction, (iii) fault management, *e.g.,* to localize and replace faulty equipment, (iv) configuration management, *e.g.,* to enable new functionality or customer features, (v) traffic/performance management, *e.g.,* to deal with traffic growth and dynamic traffic events, (vi) security management, *e.g.,* dealing with security incidents like worm outbreaks and DDoS attacks, (vii) network measurement and monitoring, *e.g.,* to detect anomalies.

The scale of modern networks, the diversity of the equipment used to realize their functionality, and the inherent complexity of many of these operational tasks combined make network management and operation one of the most significant challenges faced by network operators. This state of affairs is exacerbated by the fact that networks are always "live", *i.e.,* traffic associated with the myriad of services enabled by the network is continuously being carried by the network. The implication is that operational tasks have to be performed with minimal impact on existing services.

To address these challenges, it is desirable to have as much automation as possible so that systems can be utilized to keep track of dependencies and constraints as network operational tasks are performed. However, the realization of a unifying framework to enable fully automated network operations would be a challenging task at best and in the worst case might not be feasible.

In this paper we take a modest step towards the realization of such a unifying operational framework. Fundamental to our overall approach is the recognition that automation can only be achieved in a closed-loop fashion where the operational actions are informed by the state of the network, which reflect the result of previous operational actions as well as the dynamic behavior of the network.

A significant challenge in realizing any automated operations/-management system is choosing the "right" level of abstraction: abstractions are needed in all complicated systems in order to hide unnecessary details; however, those exact same details to hide for one task might be important to expose in another task. In this work we explore the utility of a database-oriented declarative language approach to facilitate both programmability as well as the ability to realize different abstractions over the same data and thus to serve as a unifying framework towards automated network operations.

Specifically, we present DECOR, a unifying network operation management system, which models router configurations and any generic network status as relational data in a conceptually centralized database. As such, network management operations can be represented as a series of transactional database queries, which pro-

vide the benefit of atomicity, consistency, and isolation. The rule-based language in DECOR provides the flexible programmability to specify and enforce network-wide management constraints, and achieve high-level task scheduling. We describe the design rationale and architecture of DECOR and present some preliminary examples from our experiences of applying the approach to some common network management tasks.

# 2. APPROACH

In this section, we first examine the fundamental components of management operations, then present the benefits and the architectural overview of our database-oriented declarative approach to automated network management.

## 2.1 Mechanics of Network Operations

Network operations are fundamental to the well-being of today's networks. In operational networks, they are usually performed manually, or in a semi-automated fashion, via so called *method of procedure (MOP)* documents. MOPs describe the procedures to follow in order to realize specific operational tasks, often via manual *command line interface (CLI)* procedures.The procedures usually serves as a template that stitches the following four components together to achieve actual network management tasks:

*Configuration management:* The configuration of network elements collectively determines the very functionality provided by the network in terms of protocols and mechanisms involved in providing functionality such as basic packet forwarding. Configuration management, or more generically all commands executed via the operational interface of network elements, are also the primary means through which most network operational tasks are performed.

*Status checking:* Obtaining network running status is an essential part of network management [1]. As a matter of fact, the result of status-checking activities largely determines the actual progress of network operational task. As a trivial example, a BGP session configuration would only be carried out on a router after IP level connectivity to the remote BGP peer has been verified.

*External synchronization:* Today's networks are inherently managed by multiple parties. While devices can be logically accessed from a central location, field operators are essential in carrying out operations on the physical infrastructure of the networks. There are also external decision systems that can guide various types of management tasks, such as router or link maintenance [2]. From a network management system point of view, it is important to encode the capability of synchronizing with these external parties.

*High-level constraints:* While making changes to the networks, there are usually certain constraints that should never be violated. For a large ISP network with many routers and inter-links, link maintenance is performed all the time. A bottom-line constraint could be "never partition the network". This constraint could effectively restrain two (only) cross-country links being maintained at the same time.

Most of the existing work [3, 4] focus on the automation of generating configuration changes. Few, if any, effort has been made to automate checking network status, synchronizing with external entities, enforcing high-level constraints and carrying out operational procedures. As a matter of fact, today, all these aspects are mostly performed manually and thus prone to error. A unified framework can bridge these aspects, for both automation and error-resilience.

## 2.2 Abstracting Networks as Databases

In this paper we explore the utility of a database abstraction for network operations through a system called DECOR. We abstract router state and network state into tables in a conceptually centralized relational database. Programmability is naturally provided by a declarative language composed of a series of database queries. As a result, the database automatically propagates state change from database tables to routers to carry out network operations. We argue that the approach has the following advantages:

*Flexible levels of abstractions*: Managing routers as databases not only raises the abstraction to a higher level than the MOP/CLI approach, but also provides the ability to realize different abstractions over the same data by creating *views* on top of the base tables. For example, one could derive a `path` view that describes all paths established by a routing protocol based on a `link` table, which describes link relation between routers and is extracted from each router. As a result, operations and policies based on path properties can be directly specified against the derived view.

*Configuration and status unification*: Both router configurations and network status are represented as relational tables in DECOR. Therefore, it is straightforward to write queries that configure routers based on different network conditions.

*Transactional operation*: Network operations are represented as a series of transactional database queries, which provide the benefit of atomicity, consistency and isolation. Should any failures or policy violations occur, DECOR rolls back to the previous consistent state.

*Declarative policy enforcement*: DECOR enables network operators and administrators to specify high-level policies (*i.e.,* constraints). For example, one may specify that each router must have a unique interface identifier, or at least one of the two important links must be up. These policies are expressed *independently* from the authors of operation transactions, and are considered declarative in that they describe *what* should happen as opposed to *how* to enforce them during each network operation. Such enforcement mechanisms are automatically generated from the policies by DECOR.

## 2.3 Architecture

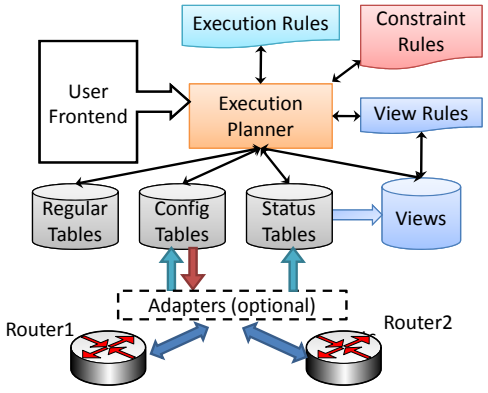


**Figure 1: DECOR architecture**

The DECOR architecture is depicted in Figure 1. In a nutshell, DECOR maintains tables and views that reflect router configurations and network status. Network operations and constraints are

expressed as rule-based database queries. They are fed into the execution planner where automated execution programs are generated to manipulate the tables and views. Relevant state changes in the tables are committed to the corresponding routers. For commodity routers that do not support the database abstraction, adapters are used to bridge the gap. A user interface is provided for operators to examine data and execute operations.

*Data model*: All state involved in operation tasks is modeled as relational data, and stored in one of the following types of tables in DECOR: i) *regular tables* are just like tables in a traditional database. Their state is not associated with any router. Such tables are typically used to store auxiliary execution state for an operation, such as the stage of a multi-stage operation; ii) *config tables* store router configuration information, such as IP addresses, protocol-specific parameters, interfaces, etc. One can read these tables to get current configuration, and also write to those tables to change the configuration. DECOR is responsible for maintaining consistency between config tables and router state. For example, an update of the `interface` table entry `interface(if_id,"down")` effectively triggers CLI commands that shut down the according interface; iii) *status tables* represent the current network state. For example, a `ping(Src,Dest,RTT)` table represents the ping result between two routers `Src` and `Dest`. These tables are read-only, and maintained in an on-demand fashion: DECOR only fetches status from the routers when relevant status table entries are referenced in a query.

*Language*: DECOR adopts a rule-based query language Mosaic [5], a variant of Datalog [6], for operators and administrators to program automated network operations. Datalog is known to be more expressive in representing recursive queries than SQL, which is desirable to describe network properties. DECOR utilizes three types of rules for different purposes: i) *execution rules* are used to define automated network operations. They are usually in the form of event-condition-actions (ECA rules). For example, a `startOp(RouterID)` event triggers the execution of an ECA rule, and depending on current router configurations and network status (*i.e.,* conditions), different actions are taken to carry out the operation. In a complicated operation, an action may trigger other events, which further lead to other actions that dictated by other execution rules; ii) *constraint rules* specify the policies of a network as the consistency conditions of the database. Any actions in execution rules should not make the database inconsistent; iii) *view rules* are used to create views that are derived from existing tables or views. Views provide different levels of abstractions (recall the example in Section 2.2).

# 3. EXAMPLES

In this section, we exemplify how to handle different types of network operations in DECOR.

## 3.1 Link Maintenance

We use the example of link maintenance with increasing sophistication to show how different aspects of network management can be expressed as declarative rules. We also give some intuition on how the execution engine picks up and executes rules to automate management operations.

**Listing 1: Rules for Router Maintenance**

```
R1 on insert Maintenance(L,"pending"), EndPoint(L,int1,int2),
   => insert interface(int1,"down"), insert interface(int2,"down");
R2 on periodic(10), Maintenance(L,"pending"),EndPoint(L,int1,int2),
    interface(int1,"down"), interface(int2,"down");
   => messageToField(L,"start"), insert Maintenance(L,"onfield");
R3 on messageFromField(L,"done"), Maintenance(L,"onfield")
   => insert Maintenance(L,"fdone");
R4 on periodic(10), Maintenance(L,"fdone"), EndPoint(L,int1,int2)
   => insert interface(int1,"up"), insert interface(int2,"up"),
    delete Maintenance(L,"fdone");
```

**Basic link maintenance procedure:** From a network operator's perspective, the basic operational procedure of link maintenance includes: 1) shut down the interfaces on both ends of the link; 2) coordinate with field team so that they work on the physical part of the link; 3) bring up the interfaces.

Listing 1 shows how to use 4 execution rules (`R1-R4`) to realize a primitive maintenance procedure. Three tables are used in the example: the `Maintenance` table contains a list of links that are under-going maintenance procedures, associated with its up-to-date procedure status; the `EndPoint` table records each link and the interface IDs of its two ends; the `interface` table is a config table to bring up or down router interfaces. Modifying the state of an interface from "up" to "down" would result in configuration changes automatically populated to the actual devices. There are two events `messageToField` and `messageFromField` in the example. They are sent and received respectively via the user front-end to interact with operators.

`R1-R4` are event-condition-action (ECA) rules. They are triggered by events, including user-defined events, system events, or database events. The actions of a rule are executed when all conditions hold. Specifically, `R1` fires when a new link maintenance task on link `L` is scheduled, indicated by the insertion event of a tuple `(L,"pending")` into the `Maintenance` table. Then the endpoint interfaces `int1` and `int2` of the link `L` are identified. Finally, it performs the actions of shutting down both interfaces by changing the `interface` table. The details of how this change is done are transparent to the rule writers.

`R2` and `R3` are used to carry out external synchronization. `periodic(10)` represents a system event that is triggered every 10 seconds. So, `R2` is periodically triggered to find a link `L` in `"pending"` state and both of its interface endpoints are already shut down, then performs the actions of notifying field team to start working and changing the state of the link L to be `"onfield"`. `messageToField` and `messageFromField` are both events for exchanging messages with the field team. `R3` is fired if a message is received from field team saying link `L` is done on their side, resulting moving the state of link `L` to `"fdone"`.

`R4` is periodically triggered to pick up a link `L` that is done with field work, identifies both of its endpoint interfaces, then performs the action of bringing them up, and removing `L` from the `Maintenance` table, indicating the completion of the task on link `L`.

Given the above rules, maintaining a link is as simple as inserting a tuple `(L,"pending")` into the `Maintenance` table, and then our system would automatically fire the rules when appropriate to finish the task. As illustrated in this example, it is very straightforward to express a procedural network operation using the declarative language. Basically, the main management target is assigned with an explicit state, which is updated as the operational stage progresses. At each stage, a set of table modification or event generation are done. A new stage is entered, if the previous stage is verified to have achieved its effect.

**Listing 2: Rules for Router Maintenance**

```
% include rules from previous listing
#include(R1,R2,R3)
```

```
// maintain a list of links that are down
V1 linkDown(L) :- EndPoint(L,intf,_), interface(intf, "down");
V2 linkDown(L) :- EndPoint(L,_,intf), interface(intf, "down");
// Shortest path routing
BP1 path(S,D,P,C) :- link(L,S,D,C), !linkDown(L), P=[L];
BP2 path(S,D,P,C) :- link(L,S,Z,C1), !linkDown(L), path(Z,D,P2,C2),
     C=C1+C2, P=[L]+P2;
BP3 bestPath(S,D,P, min<C>) :- path(S,D,P,C);
// maintain links that are used
V3 linksInUse(L) :- link(L,_,_,_), bestPath(_,_,P,_), P.contains(L);
// cost out the link, as the new first step
R5 on insert Maintenance(L,"pre-pending"), link(L,S,D,C) =>
     insert link(L,S,D,inf), insert Maintenance(L,"costout"),
          insert costSave(L,C);
// only schedule to shut down the link if it's cost out
R6 on periodic(10), Maintenance(L,"costout"), !linksInUse(L) =>
     insert Maintenance(L,"pending");
R4' on periodic(10), Maintenance(L,"fdone"), EndPoint(L,int1,int2),
        costSave(L,C), link(L,S,D,_)
     => insert interface(int1,"up"), insert interface(int2,"up"),
     delete Maintenance(L,"fdone"), insert link(L,S,D,C);
```

**Routing protocols integration:** The procedure defined in listing 1 is straightforward, yet problematic in that an interface can be shut down, even if it is still being used actively for packet forwarding, causing transient network packet loss until the routing protocol reconverges. In listing 2, we show how to make the maintenance task aware of network protocol running state.

First, we introduce several views (in rule `V1-V3,BP1-3`) to raise the level of abstraction to special links and routing paths. `V1` and `V2` are view rules that define links that are down—we consider a link to be down if one of its interface endpoint is down. `BP1-3` create a `bestPath` view that is generated by a shortest path routing protocol [7]. Basically, `BP1-2` computes the paths (`P`) with cost (`C`) between a source (`S`) and destination (`D`), in a recursive fashion. Note that we add additional dependency on `linkDown` to make sure a down link is not used. `BP3` selects the best path between any pair of source and destination. We assume the routing table is set up according to the `bestPath` view. Rule `V3` is used to derive a list of links that are currently used from the routing table.

Next, in rule `R5`, we introduce a new state of `"pre-pending"` for a link in the `Maintenance` table. To maintain a link, `(L,"pre-pending")` should be inserted to take advantage of the additional sophistication. `R5` states that for each link in `"pre-pending"` state, we first change its link cost to infinity (`inf`). This would effectively remove the link from the current routing table. `R6` states that only if the link `L` is confirmed not to be used in the routing table, can we transit it to the `"pending"` state, resulting a shut down by `R1` (included from listing 1). We use `R4'` to replace the original `R4`, adding the action to restore the link cost of `L`.

Note that this program is meant to exemplify how the network status observation can be integrated into the network operations. Our system does not require the routing protocols to be implemented declaratively. We can simply populate a status table with up-to-date network routing state and write queries based on that.

**Listing 3: Rules for Router Maintenance**

```
#include(R1,R2,R3,R4',BP1,BP2,BP3,V3,R5,R6)
// for every router S and D, there must be a path
C1 router(S), router(D) -> path(S,D,_,_);
```

**Constraint enforcement:** While the rules in the above two programs can help the careful progression of a link maintenance task, some operators may include some other rules to manipulate `interface` table in other ways. The problem is that the combination of these programs may introduce bad state, such as network partition. In this example, we introduce the usage of constraint rules. `C1` in Listing 3 is a simple way to express, for any two routers `C` and `D`, there is always a path between them. Note that constraint rules are assertions that do not change any state, unlike ECA rules where the actions do make state changes. Constraints can be used to expressed high-level policies over all the network operations. The constraints can be "do not partition the network", "do not cause traffic oscillation more than X percent", etc. When an execution rule firing has the potential of violating these constraints, that rule firing is canceled or delayed to retry at a later time.

## 3.2 Network Monitoring and Fault Diagnosis

**Listing 4: VPN Monitoring and Fault Diagnosis**

```
// periodically test the connectivity between R1 and R2
// pingResult would only store the recent N seconds of data
R7 on periodic(10), router(R1), router(R2), R1!=R2,
    ping(R1,R2,result), T=PosixTime::now()
    => insert pingResult(R1,R2,T,result);
// count how many failed pings and how many in total
V4 recentPingFail(R1,R2,count<*>) :- pingResult(R1,R2,_,result),
    result="failed", groupBy(R1,R2);
V5 recentPingTries(R1,R2,count<*>) :- pingResult(R1,R2,_,_),
    groupBy(R1,R2);
V6 recentPingFailRatio(R1,R2,r) :- recentPingFail(R1,R2,f),
    recentPingTries(R1,R2,t), r=f/t;
// if failed ping ratio is higher than a threshold trigger diagnosis
R8 on periodic(30), vpn(C1,P1,VPN), vpn(C2,P2,VPN), C1!=C2,
    P1!=P2, recentPingFailRatio(C1,C2,R), R > alert_pctg,
    ! VpnDiag(C1,C2,_), T=PosixTime::now()
    => insert VpnDiag(C1,C2,P1,P2,"diag_ce_pe",T);
R9 on insert VpnDiag(C1,C2,P1,P2,"diag_ce_pe",T),
    recentPingFailRatio(C1,P1,R1), R>alert_pctg
    => alarm(C1,C2,"down due to CE to PE link !");
R10 on insert VpnDiag(C1,C2,P1,P2,"diag_ce_pe",T),
    recentPingFailRatio(C1,P1,R1), R<=alert_pctg
    => insert VpnDiag(C1,C2,P1,P2,"diag_pe_route",T);
```

Listing 4 shows how to build a simple network connectivity monitor and further automates VPN connectivity problem diagnosis in DECOR.

`R7` is a very straightforward rule used to get raw connectivity data: it is triggered every 10 seconds for every pair of routers, a `ping` table query is issued and the ping result stored in `pingResult` table. As a status table, any query to the `ping` table is translated to a ping command on the corresponding router. `V4` and `V5` are views that count the number of failed and total ping trials between any pair of routers based on the `pingResult` table. `V6` calculates the failure ratio between all pairs of routers within the recent N seconds. This exemplified DECOR's capability of building high-level abstraction over relatively low-level data elements.

`R8` monitors VPN connectivity by firing every 30 seconds and finding two CE routers `C1` and `C2`, that are within the same VPN but connecting to different PEs (`P1` and `P2`): if the ping failure ratio is between the two CEs is higher than a pre-defined threshold, an automatic diagnosis procedure on this pair of CEs is started. Note that, `!VpnDiag(C1,C2,_)` is used as a condition to prevent launching a diagnosis procedure for the same pair of CEs twice.

VPN diagnosis is very complicated and involves in multiple steps to narrow down the problem. For brevity, we only show one step from an online tutorial [8] in the example. In this step, we need to verify if the CE `C1` can reach the PE `P1` correctly. `R9` and `R10` check the failure ratio between `C1` and `P1`: 1) if the ratio is higher than a threshold, `R9` is fired, meaning that the problem is confirmed to the connectivity loss between CE and PE and thus an

alarm is generated; 2) otherwise, `R10` is fired, moving on to next stage diagnosis `"diag_pe_route"`, which tries to determine if the CE router's loopback IP exists in the PE router's VRF table.

A wide range of network monitoring and follow-up automated response can be expressed similarly. For example, the following rule can be used to monitor link usage and perform rate-limiting automatically: `on periodic(10), LinkUsage(L,R), R>0.8 => RateLimit(L).`

# 4. CHALLENGES

In the preceding sections, we have described an architecture that leverages a database abstraction for network management, and illustrated some of the benefits through examples. Building such a system, however, has to deal with many challenges that do not exist in traditional DBMS. We discuss them in this section.

**Synchronization issues:** While DECOR uses database tables as a new layer of abstraction, there is the possibility that the database tables and the actual network status are out-of-sync. This is particularly true when we use adapters to interact with non-declarative systems or components.

On the one hand, network changes (*e.g.,* configuration modification) need to be populated to the network as fast as possible. The synchronization process usually takes time, *e.g.,* tens of seconds to effect configuration changes. The challenge is how to handle the access to those tables containing data entries that are "in transition".

On the other hand, network status should be reflected by the tables in a timely fashion. Instead of querying the network devices at a fast rate to get a close-to-realtime view of the network, a more scalable solution is to rely on router programmability such that notifications can be sent out when relevant events occur on the devices, *e.g.,* routing table update, interface status change.

**Failure handling support:** There is already the notion of transactional group commit, roll-back support, *etc.* in database literature. The roll-back support in traditional databases can be done by reverting a set of changed table entries. In DECOR, however, updating table entries has direct or indirect impact on the actual networks, thus additional care must be taken to to prevent transient bad network states.

We consider two types of failure handling in DECOR:

*Implicit handling* can be done by reversing the list of rules executed, based on execution history information. For example, `R1` in listing 1 is the start of a sequence of operations. We can simply modify the rule to add a marking, so that whenever the DECOR system executes this rule, it takes a checkpoint of the related table entries and start recording the rules fired afterward. If failure occurs, we can undo the rules fired one by one in the reverse order.

*Explicit handling* makes use of an additional failure state. If failure is detected (*e.g.,* a failure detector rule fired), the management operation transit explicitly into the failure state, after which a sequence of well-defined rules are fired to handle the failure like another network operation. This might be desired if a special routine should be carried out.

**Constraint enforcement:** Formally, a constraint is a predicate that is based on network status and always should be evaluated to be true. These predicates are usually associated with the basic well-beings of the network, *e.g.,* "network is not partitioned", which if violated could potentially have disastrous effects. The constraints are usually written by network experts that are familiar with the overall network design. DECOR can easily facilitate the support of setting up rules to *detect* predicate violations and generate alarm messages. However, at the time of the alarm, the network has already entered an undesirable running state. As such, we want to design DECOR so that the execution planner can intelligently *cancel or delay* a rule to fire, if one of the predicates may no longer hold if the rule fires at the current network state.

**Prioritization in rule execution:** In existing declarative systems, rules fire whenever the conditions are met. DECOR, on the other hand, takes priority into consideration when it schedules and executes rules. The rationale here is that the computation capability of the rule processing engine is always limited, and it is important to prioritize more important rules. When comparing the importance of rule firings, both the rule body and the rule firing parameters are considered. For example, routing protocol rules should be processed with the highest priority, because the timeliness directly translates to faster convergence time. On the other hand, for the same maintenance rule fires with different parameters, it would be better to prioritize the task for core routers than edge routers.

**Function refactoring for distributed rule processing:** In the first instance we envision the DECOR system architecture as a centralized database which is populated with network states and interacts with devices from the whole network. To deal with the scalability issue when the networks become larger, DECOR can trade-off the centralized processing overhead with distributed communication overhead by deciding to offload portion of the database tables and rule processing to the distributed devices, taking advantage of the increasingly more available programmability support.

# 5. RELATED WORK

Applying declarative approaches to system and networking problems has gained considerable attentions in recent years. The declarative networking project proposes a distributed recursive query language to concisely specify and implement traditional routing protocols at control plane [7]. Since then, the declarative approach is taken by numerous projects, *e.g.,* to implement overlay [9] and sensor network protocols [10], data and control plane composition [11], and distributed storage policies [12]. Compared with those work, DECOR focuses on a different application domain—network management. DECOR is also unique in terms of exploiting database transaction semantics and global consistency constraints in network operation task executions.

In network management, NetDB [13] is probably the closest work to our vision. However, it provides a read-only database abstraction for router configurations, where one can write queries to audit and analyze existing configurations in offline fashion. In contrast, DECOR allows writes to the database to change the network configurations, as well as queries of network status as part of the operation specifications. Combining these two capabilities with policy constraint enforcement, DECOR tries to detect and prevent policy violations at the operation stage, as opposed to offline auditing to find already-happened damages.

Much work has been done for automating network configuration changes [4, 3]. DECOR takes a step further trying to automate the holistic network management operations. Other management automation frameworks feature different abstractions. For example, CONMan [14] takes the approach in which devices and network functionality is captured by modules. Many management tasks, tunneling connectivity in particular, can be accomplished via manipulation of the abstracted modules.

# 6. CONCLUSION

In this paper, we introduce DECOR, a new paradigm for network management and operations. The fundamental idea is that

by abstracting a network as a database, we have the potential of providing the different levels of abstraction for carrying out different network management tasks. The declarative feature of DECOR enables a unified framework to allow both management operations and network-wide constraints be expressed and carried out. The future work includes a complete design, implementation, and evaluation of DECOR's capability of automating network management and operations.

# 7. REFERENCES


[1] X. Chen, Z. M. Mao, and J. Van der Merwe, "Towards Automated Network Management: Network Operations using Dynamic Views," in *Proceedings of ACM SIGCOMM Workshop on Internet Network Management (INM)*, 2007.

[2] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, and J. Rexford, "NetScope: Traffic engineering for IP networks." IEEE Network Magazine, March/April 2000, pp. 11-19.

[3] W. Enck, P. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. Greenberg, S. Rao, and W. Aiello, "Configuration management at massive scale: system design and experience," in *Proceedings of the USENIX'07*.

[4] J. Gottlieb, A. Greenberg, J. Rexford, and J. Wang, "Automated Provisioning of BGP Customers," *IEEE Network*, vol. 17, 2003.

[5] "The Mosaic Project." https://mosaic.maoy.net.

[6] R. Ramakrishnan and J. D. Ullman, "A Survey of Research on Deductive Database Systems," *Journal of Logic Programming*, vol. 23, no. 2, pp. 125–149, 1993.

[7] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan, "Declarative Routing: Extensible Routing with Declarative Queries," in *Proc. of SIGCOMM*, (Philadelphia, PA), 2005.

[8] "Juniper Networks: Troubleshooting Layer 3 VPNs." `http://www.juniper.net/`.

[9] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing Declarative Overlays," in *Proc. of SOSP*, 2005.

[10] 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 *Proc. of SenSys*, (Sydney, Australia), November 2007.

[11] Y. Mao, B. T. Loo, Z. G. Ives, and J. M. Smith, "MOSAIC: Unified Declarative Platform for Dynamic Overlay Composition," in *Proc. of CoNEXT*, (Madrid, Spain), Dec 2008.

[12] N. Belaramani, J. Zheng, A. Nayte, M. Dahlin, and R. Grimm, "PADS: A Policy Architecture for building Distributed Storage systems," in *Proc. of NSDI*, April 2009.

[13] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford, "The cutting EDGE of IP router configuration," in *Proceedings of ACM SIGCOMM HotNets Workshop*, November 2003.

[14] H. Ballani and P. Francis, "CONMan: A Step Towards Network Manageability," in *Proc. of SIGCOMM*, 2007.