PACMAN: a Platform for Automated and Controlled network operations and configuration MANagement

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

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

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

ABSTRACT

The lack of automation associated with network operations in general and network configuration management in particular, is widely recognized as a significant contributing factor to user-impacting network events. In this paper we present our work on the PACMAN system, a Platform for Automated and Controlled network operations and configuration MANagement. PACMAN realizes network operations by executing *active documents*, which systematically capture the dynamics in network management tasks. Active documents not only enable the complete execution of low-level configuration management tasks, but also allow the construction of more sophisticated tasks, while imposing additional reasoning logic to realize network-wide management objectives. We present the design, realization and evaluation of the PACMAN framework and illustrate its utility by presenting the implementation of several sophisticated operational tasks.

Categories and Subject Descriptors

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

General Terms

Design, Management

Keywords

Network Management, Automation, Petri Net

1. INTRODUCTION

Network management plays a fundamental role in the operation and well-being of today's networks. 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, *e.g.*, planned maintenance, performance monitoring, fault management, service realization and capacity planning, are performed.

CoNEXT'09, December 1–4, 2009, Rome, Italy.

The inadequacies of current network management and operational procedures have been widely recognized [1], and many solutions have been proposed. However, a variety of factors conspire to make more automated network operations an elusive goal. One of the biggest challenges in network management automation is to find the right level of abstraction. What is needed, on the one hand, is the ability to abstractly describe operational goals without getting bogged down in the minutiae of how to achieve those goals. On the other hand, because of the sophistication of modern networking equipment, the ability to fine tune the details of how an operational task is performed, is in fact critical to achieving the desired effect. This inherent tension is exacerbated by the fact that network equipment vendors, driven in part by feature requests from operators, have allowed network configuration languages to evolve into arcane sets of low-level commands. Operators therefore have to become accustomed to designing and reasoning about their networks in the same low-level nomenclatures which result in significant resistance to evolving to new network management paradigms.

Indeed the lingua franca of network operations continue to be libraries of so called *method of procedure (MOP)* documents. MOPs describe the procedures to follow in order to realize specific operational tasks, usually containing the following components: (i) configuration changes (or *actions*) that need to be performed, (ii) operational checks (or *conditions*) that have to be satisfied before such actions can be taken and/or after them for the purpose of verification, and (iii) execution or *workflow logic* that ties actions and conditions together. Currently, MOP documents are literally stored as libraries of text descriptions which is not suitable for automation or holistic network-wide reasoning.

In modern network operational environments, parts of MOP-defined procedures are typically automated to limited extent. For example, configuration actions could be performed by scripts that push configlets to network elements [2, 3, 4]. However, for the most part, network operations still require human operators to verify the result of actions and to navigate through the logic involved in operational procedures. This is especially true in terms of being cognizant of possible interactions among different operational procedures and understanding the holistic impact of such actions. For example, sophisticated tools have been developed to help operators understand the possible impact of their actions [5]; however, operators typically consult these tools independently and then use the information they provide to manually "close the loop" to perform operational tasks. In other words, such tools are not fully integrated into the process of network operations.

The ultimate goal of our work, is to create an environment which allows the full automation of this operational control loop. Towards this end, in this paper we present our work on the *PACMAN* system,

a Platform for Automated and Controlled network operations and configuration MANagement. Our work builds on two basic observations related to the elements contained in MOP documents. First, MOP document structure, *i.e.,* actions, conditions and the associated workflow logic, presents a natural way for operators to think and reason about operational activities. Second, the logic embedded in the design of these procedures represent the expert knowledge of MOP designers to ensure that high-level operational goals and conceptual designs are met, while minimizing unwanted side effects of operational actions.

At a high level, PACMAN maintains these desirable properties by allowing experts to define operational procedures as before with one significant difference: The procedures defined in the PACMAN framework are not static documents meant for human consumption, but instead *active* method of procedures, or simply *active documents (AD)*, meant for execution in the PACMAN framework. Active documents formalize the configuration management procedures described in MOP documents. I.e., ADs capture, in a systematic manner, the actions, conditions and logic associated with operational tasks, thus forming a fundamental building block for the automation of network operations. ADs enable the complete, repeated, programmatic and automated execution of low-level management tasks, but more importantly enable the construction of more sophisticated tasks. Specifically, simple active documents can be combined into *composed* ADs whose execution is dictated by *policies* capturing holistic network-wide management objectives. In so doing, PACMAN raises the level of abstraction as the high-level operational goal becomes part of the composed AD, thus being enforced in an automated fashion, eliminating the need for operators to be continually concerned about and actively involved in *how* to carry out a goal amongst multiple tasks. Furthermore, the PACMAN framework allows easy interaction with external tools to enable sophisticated decision making to be naturally integrated into the network operational loop.

In this work, we make the following contributions:

- We analyze method-of-procedure documents from an operational network to extract the *network management primitives* associated with network operations.
- We introduce *active documents* as a concise, composable, and machine-executable representation of the actions, conditions and workflow logic that operators perform during network management tasks.
- We present the design and implementation of the *PACMAN* framework, an execution environment that automates the execution of active documents.
- We bridge the gap between current operational practices and our automated environment with the AD *creation framework*, a set of tools which allow operators to work in their native idiom to easily generate active documents.
- We demonstrate the effectiveness of the framework using several case studies of common configuration tasks such as fault diagnosis, link maintenance, and a more complicated task of IGP migration.

2. RELATED WORK

PACMAN enables automation and abstraction while not requiring operators and network designers to adopt a completely new network management paradigm. At the same time and using the same framework, PACMAN does go beyond simple configuration generation, allowing operators to articulate network-wide operational goals and indeed adopt new network management paradigms. In contrast to PACMAN, most related work does not bridge the gap between current network management approaches and new network management paradigms, but rather fall in one of the two categories. The state-of-the-art in practical network configuration management is exemplified by the PRESTO configuration management system [2]. PRESTO templates lack the necessary execution logic to capture sophisticated operational tasks, or indeed the ability to intelligently create composed tasks from simpler components. Similar to PRESTO, a simple template based approach has been used to automate BGP configuration of new customer links [3]. While also limited to BGP configuration, the work by Bohm *et al.* [4] had a broader scope in that it addressed the creation of configuration files to realize network-wide inter-domain routing policies.

The EDGE architecture [6] had the ambitious goal of moving to automated network configuration by first analyzing existing network configuration data and to then use such network intelligence to create a database to drive future automated configuration changes. This work seemed to have stopped short of actually taking the last step to create an automated configuration system and is therefore more similar to efforts that attempted to analyze the correctness of existing network configurations [7, 8, 9]. Major router vendors have also proposed their own network management automation frameworks [10, 11]. Those management frameworks remain device-centric and are mostly used to handle local failure response, while PACMAN allows a full spectrum of network management tasks with a network-wide perspective.

Several proposals exist that address network management complexity through approaches that are less tethered to current operational practices and device limitations [12, 13, 14]. These works do not cover the full range of network management tasks required in operational networks [12], or attempt to limit the potential negative impact of configuration management without directly addressing its complexity [13, 14].

A number of "autonomic" network architectures are related to PACMAN [15, 16]. Conceptually the FOCALE architecture [15] is perhaps the closest to PACMAN. Specifically, like PACMAN, the FOCALE architecture contains an explicit control loop so that network operations can be cognizant of network conditions. However, unlike PACMAN, which closely models current operational practices and ties in with existing network capabilities, the FOCALE approach requires the adoption of new paradigms and tools.

Finally, PACMAN adopts Petri net [17] to model network management activities. Petri net, developed in the 1960s, employs a compact representation and is convenient to model concurrency, control flow, and system dynamics. Recently Petri nets have been used to model IT automation workflows [18]. Formal verification techniques are applied on Petri net models, *e.g.,* Gadara [19] uses Petri nets to model multi-threaded programs, identifying deadlocks using structural analysis and patching them automatically. In this paper, we took a modest first step of using the Petri net model to allow advanced network management workflow to be built and high-level policy to be constructed and imposed. We leave formal verification of the generated workflows as future work.

3. NETWORK MANAGEMENT PRIMITIVES

To build a useful and practical automation system for network management, we first closely examine the current best practice to extract the fundamental primitives and requirements. As a start, we analyzed a month's worth of method of procedure (MOP) doc-

uments from a tier-1 ISP network, as well as from major router vendors. These documents cover a wide variety of network management tasks, including customer provisioning, backbone link and customer access link migration, software/hardware upgrade, troubleshooting VPN, *etc.*

MOP documents are essentially instruction manuals for performing specific management tasks. They are usually modularized, consisting of multiple sub-tasks or steps. For example, a BGP customer provisioning MOP usually consists three steps of link setup, IP setup and BGP session setup, where each step involves configuration change on a router and running status verification. A fault diagnosis MOP [20] often contains a sequence of network tests to perform, such as `ping`, `show bgp`, and for each test an instruction of how to interpret and act upon the result.

At a micro level, we categorize the fundamental network management primitives that make up the MOPs as follows:

Configuration changing: Most of the management tasks involve configuration modification, which directly leads to network device behavior change. For example, configure a BGP session, change OSPF link metric *etc.*

Status acquiring: Network status information is crucial for the progression of network operations. Two types of status are usually obtained: static information, such as configuration, hardware components; dynamic information, such as BGP session states, routing tables. The acquired information can either be stored for future use or processed immediately.

Status processing: Status information is evaluated in a variety of ways, for example, check router configuration for OSPF-enabled interfaces, verify if a routing table contains a specific route, or even compare the current BGP peer list with previous captured list. Based on the evaluation, different next steps may be taken.

External synchronization: Explicit synchronization with other parties, including field operators, centralized decision engine, *etc.*, is very common. The operator can either notify an external party indicating operational progress or wait on external parties for their progress update, for example, wait for a field operator to finish an on-site physical upgrade.

The lack of automation also manifests as the fact that these primitives need to be *composed* together manually. We identify the following composition mechanisms (or workflow logic):

Sequential: This most basic composition simply perform one subtask after another. It is useful for stitching many stand-alone operations into a complex operation.

If-else split: The purpose of status processing is to choose different subsequent sub-tasks based on an if-else logic. For example, for different OS versions or interface types, the configuration to change could be different.

Parallel split: In some cases, the operator is required to work on multiple devices at the same time. In other cases, a monitoring subtask is spawned on the side. For example, creating a new terminal session to launch a continuous ping to monitor delay and jitter of a potentially impacted path.

Iterative processing: Operators may need to process one element at a time, until there is no such element left. For example, to identify all interfaces with IS-IS configured, and disable them one by one.

Wrapper: Predefined "head" and "tail" sub-tasks can be used to wrap around other sub-tasks. For example, saving the configuration and running status before and after the operation for later verification.

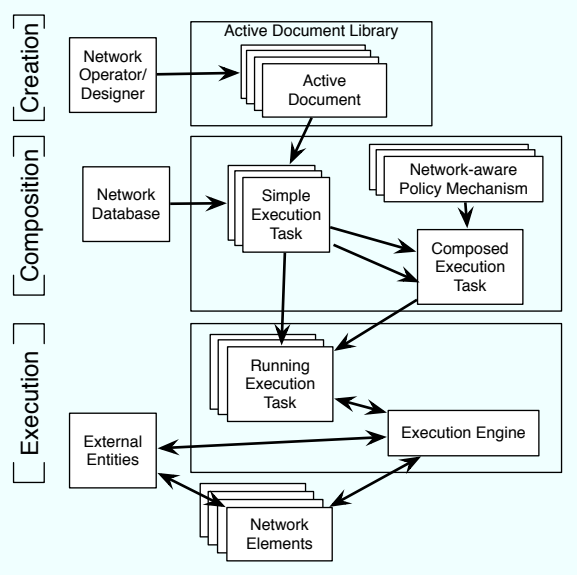

Figure 1: The PACMAN framework

Sequential composition is the easiest to automate, but due to the frequent occurrence of other cases, the majority of network management tasks cannot simply be represented as a sequential flow. Other composition mechanisms are almost always handled by human operators.

At a macro level, the descriptive nature of MOPs dictates that the realization of management tasks have to rely on human operators, who consume the MOPs and carry them out either manually or via limited automation, resulting in a process that is known to be time-consuming and error-prone. Based on our analysis, only configuration changing and network status acquiring are automated, to a limited extent, through automated tools. The decision logic, for example reasoning about network status to determine the proper next step, is usually described in high-level terms and almost always left for human operators to realize. This deficit calls for an automatable representation of workflow logic, which we integrate into our active document design.

MOP documents are by necessity limited in scope, typically focusing on a specific operational task, with little visibility into the network-wide impact of the task or its interaction with other tasks. In the case of multiple concurrent tasks, the burden of avoiding undesired network states based on reasoning about global network status usually falls on human operators, who may not be able to correctly perform such reasoning due to knowledge deficit or resource constraint. This motivates us to design active documents that are composable and capable for network-aware policy enforcement. On the other hand, while tools are available to show how some operational tasks (e.g., costing out a link) might impact the network [5], the interaction with such tools is currently largely left to operations personal.

4. THE PACMAN FRAMEWORK

PACMAN is motivated by the fact that automation is limited in current network management, as discussed in the previous section. PACMAN bases on new abstractions, which incorporate all required operational primitives and compositional mechanisms identified from MOP documents, allowing natural absorption of the expert knowledge and full automation of the network operations. As

a step further, the abstractions allow multiple tasks to be independently specified but automated simultaneously with global awareness seamlessly imposed without additional manual involvement; therefore, they overcome the task-centric nature of MOP documents. As shown in Figure 1, the PACMAN framework is conceptually divided into three components, creation, composition and execution.

In PACMAN, we introduce a new abstraction, named *Active Document*, to describe and further enable the automation of the workflow of network management tasks in a form that is accurate, extensible, and composable, see "Creation" in Figure 1. Unlike MOPs that are used to *guide* human operators, active documents can be *executed* on different networks to fulfill different management tasks. Compared to traditional scripts that at best automate the generation and modification of configuration on network devices, an active document also encapsulates the *logical reasoning* that guides the workflow of a management task. This capability enables full automation of network management tasks, minimizing human involvement. Active document models the primitives and composition mechanisms derived from MOP documents in a straightforward fashion, thus can be created by anyone who understands these documents, enabling our framework to quickly absorb the expert knowledge from existing MOP documents to form our own *active document library*. To illustrate this conversion process, a framework to enable semi-automated active document creation is described in §5.

Active documents model network operations generically, or in the abstract. As shown in the "Composition" component of Figure 1, to realize a specific operation, an active document is instantiated into an *execution task*. Typically, a simple execution task is generated by selecting the AD designed for the corresponding management task from the AD library and assigning proper parameters from external network databases. Furthermore, we allow the creation of composed execution tasks from one or more simple execution tasks to fulfill complex operations. *Network-wide policies*, which take global network conditions into consideration, can be imposed to automatically guarantee that the execution of a collection of task-centric operations would not violate network-wide constraints. Note that such policies are pre-defined and can be selected and automatically imposed during composition. This provides the means by which operators can start small and simple (developing task-centric ADs) yet achieve automation of network-wide coordination. Existing coordination and policy enforcement in management operations is usually either undocumented or written in high-level terms in MOP documents, due to the complexity involved. It is mostly done by skilled human operators, *e.g.*, who can decide when to execute which script so that traffic shift is minimized. PACMAN, on the other hand, provides a fully automated solution, enabled by the flexible and generic active document design.

As depicted in the "Execution" component in Figure 1, simple or composed execution tasks are executed with the support provided by an *execution engine*, which we envision to be available for each network. The execution engine runs the execution tasks in a fully automated fashion, achieving the goal of each task by reproducing the workflow and decision logic. Illustrated in Figure 1, as a result of running the execution tasks, the execution engine interfaces with devices in the network to perform the configuration change specified by the execution task, obtain various types of network status, and carry out the embedded reasoning logic. The execution engine also interacts with entities external to the PACMAN framework. As shown in the figure, these external entities might also interact with the network. Examples might include standalone network monitoring tools, or an on-site operator that is signaled that the network has been readied for the replacement of a router linecard, or some other manual operational task. The execution engine is also responsible for scheduling multiple tasks to run concurrently, providing failure handling support, *etc.*, moving closer to its goal of minimizing human involvement.

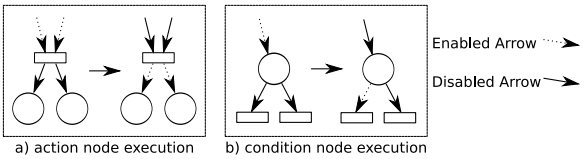

Figure 2: Active document node execution

Finally, we note that the relationship among active documents, the execution engine, and running execution tasks is analogous to that among program binaries, the operating system, and running processes. Similar to what an operating system does, the execution engine provides the running environment to the execution tasks, ensuring the correct and automated task execution according to the AD. We now consider each of the PACMAN components in detail.

4.1 Active Documents

Active documents provide the basis for achieving network management automation under PACMAN. In a nutshell, active document is a graph representation that encodes the required primitives and provides flexible composition mechanisms of network management operations. Like a program binary, an active document can be executed on the network with sufficient input parameters.

Elements: We use Petri nets [17] to model active documents. Petri nets are bipartite directed graphs containing two types of nodes: places, shown as circles, and transitions, shown as bars. Each type of nodes encode a special type of management activity. *Action* activities (corresponding to bars) include configuration or state modification and external notification. *Condition* activities (corresponding to circles) are status acquiring followed by status processing. We abstract receiving information from external parties as a type of status acquiring as well. This functional division keeps our AD model simple without compromising its functionality. The edges between nodes encapsulates the workflow of active documents, as we describe next.

Execution: When executed, a node in the graph effects the corresponding type of activity embedded. For example, an action node may emit a configuration change that add a BGP neighbor setup on a router, while a condition node retrieves the BGP neighbor status and verifies if the session is established. The execution of action and condition nodes may result in calling a set of APIs provided by the execution engine, which will be described later, to interact with devices or external parties, as shown in Table 1.

The progression among these activities is modeled as the arrows between nodes. An arrow from node a to node b represents a happen-after relationship during execution. The basic execution mechanisms of active documents are shown in Figure 2. Each arrow is marked as either *enabled* or *disabled*.[1] An action node is executed only if *all* of its incoming arrows are enabled. After execution, all incoming arrows of the action node are changed to disabled, while all outgoing arrows are marked as enabled (shown in Figure 2-a). A condition node is executed if *one* of its incoming

[1]Petri net executes by passing tokens between places through transitions, which is equivalent to enabling and disabling arrows in AD execution.

Node type	API Call Name	Functionality
Action	`CommitConfigDelta()`	Commit a configuration change to a target device
	`NotifyEntity()`	Send messages to external entities
Condition	`QueryDeviceStatus()`	Obtain physical device status information
	`QueryEntity()`	Obtain information from external entities
	`QueryExecutionState()`	Obtain execution task running status
	`TaskSucceed()`, `TaskFail()`	Notify execution engine that task has succeeded or failed

Table 1: API calls supported by the execution engine

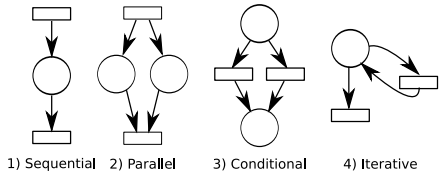

Figure 3: Active document design paradigms

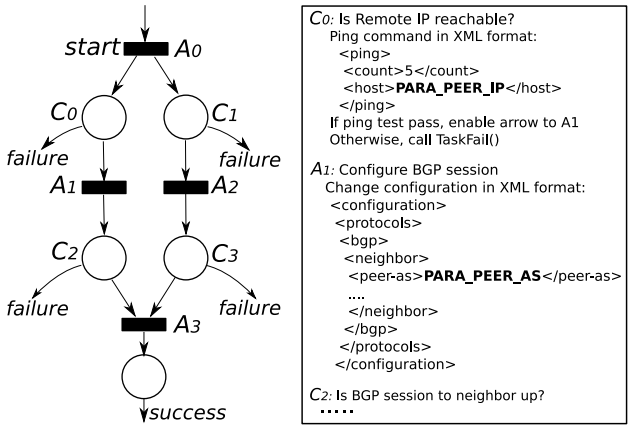

Figure 4: An example active document

arrows are enabled. After executing, one of the enabled incoming arrows is switched to disabled, and *only one* of outgoing arrows is enabled based on the status processing result performed of the condition activity (shown in Figure 2-b). By using the structural elements [18] shown in Figure 3, Petri net is capable of modeling generic and complex workflows, fully covering the compositional mechanisms from MOP documents.

For automation, an active document must be reusable, meaning that it can be executed to handle similar network tasks in different parts of the network or even on different networks. To achieve this, the activities associated with the nodes in an AD are stored as templates. For example, a condition node that checks BGP session establishment would be specified as "Check BGP session to `PARA_PEER_IP` on router `PARA_TARGET_DEVICE`", where the two placeholders are replaced during execution with actual values specified in the execution task.

Besides choosing a follow-up action, an executed condition node can decide that the management task has succeeded or failed. In these cases, the API functions `TaskSucceed()` and `TaskFail()` are called respectively, similar to `exit()` statement in C programs. The execution engine stops the execution task and handles the failure if `TaskFail()` is called.

Example: Figure 4 shows an active document that can be used to set up a BGP session between two routers. Action A_0 is not performing any activities, except to create two parallel branches to

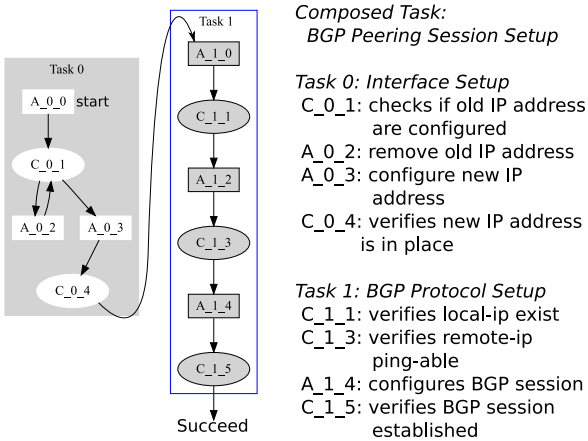

Figure 5: Sequential task composition

operate on two routers. Conditions C_0 and C_1 launch a `ping` on both routers to see if the other end is reachable. The task fails if either `ping` fails. Otherwise, actions A_1 and A_2 are executed to add the actual BGP neighbor configurations on both routers. Then, conditions C_2 and C_3 check if the configured BGP session is up on both ends; only if both condition checks succeed, can the action in A_3 be executed. A dummy action node is used at the head and a dummy condition node at the tail of the active document, if necessary. For example, the bottom condition node in the example does not perform any activities but directly calls `TaskSucceed()`.

4.2 Execution Task Composition

While active documents describe the workflow of management tasks in the abstract, *simple execution tasks* are used to specify specific instantiations of ADs by replacing all template placeholders with appropriate parameter assignments, as shown in Figure 1. For example, an active document to configure an IP address on a particular interface of a router needs the parameters of router IP address, interface name and IP address to set. The parameters are usually generated from external network related databases [6, 2]. A practical concern is that the database could be out-of-sync with the actual network state, which is a problem for existing management methods as well. To alleviate the potential negative impact, ADs can be designed to always perform in-sync checks at the start of execution.

PACMAN takes advantage that ADs are composable to enable building complex tasks from simple tasks. Moreover, it allows generic network-aware policies to be imposed. The combination of these two abilities can support high-level management goals, like "execute these tasks, but *avoid network partition*". We describe three composition mechanisms in detail next.

Sequential: Figure 5 shows the result of applying a simple *sequential ordering* to two tasks (failures are not shown for simplicity), namely the link setup task as Task 0 and the BGP configure

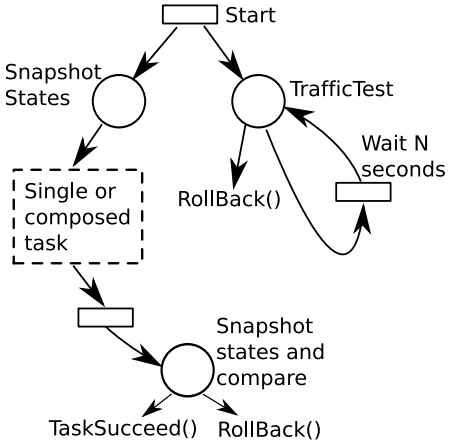

Figure 6: Wrapper construct for concurrent traffic disruption detection and state diffing

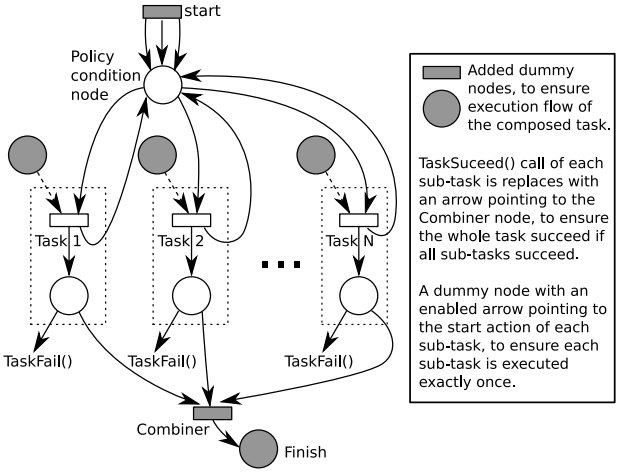

Figure 7: Policy enforcement in parallel composed tasks

task as Task 1, resulting in a "BGP peering session setup" task as the composed task. A strict ordering is enforced: a task is only executed when the previous task succeeds; the composed task succeeds if the last task succeeds. Note that node `C_0_4` originally calls `TaskSucceed()` if task succeeds. This API call is replaced with an arrow pointing to `A_1_0` to stitch the two tasks together.

Meta structure: Figure 6 shows an example meta structure for automating operations before, during and after an execution task. The composed task starts by taking a snapshot of the running status of the target device. At the same time, a loop structure (shown on the right) is used to continuously monitor network running status via `ping` or dedicated traffic generation engine. If a network disruption is detected, `TaskFail()` is called to perform roll back immediately. When the wrapped task finishes, another snapshot of the network status is taken and compared with the previous one. Failure is reported if certain criteria are not met, *e.g.*, some BGP sessions fail to establish. This is particularly useful for supporting software or hardware upgrade tasks.

Parallel with policy enforced: Figure 7 shows how several tasks are composed to execute in parallel but with a network-aware policy enforced. Each dotted box contains the original AD of each task for composition (only one action node and one condition node are shown for simplicity). A *policy condition node* (or policy node) is added to point to every action node in each task. Once imposed, the policy node becomes an additional condition to satisfy for each action node, thus it can embed a network-aware decision logic that goes beyond individual tasks. We show later how generic policies, like "prevent network partitioning" and "prevent link overloading", can be implemented. Policy nodes are usually written by network experts and can be directly used to regulate generic execution tasks. This further lowers the bar for AD creation, as existing policies can be applied to carry out the more complicated decision logic.

The policy node does *not* simply serialize the actions in each tasks. There are multiple arrows pointing from the dummy start action node to the policy node. This effectively adds multiple enabled arrows to the policy node, so that the policy node does not need to wait for an action to finish before enabling another action, allowing multiple action nodes to be executed concurrently, if permitted by the policy. As shown in Figure 7, when the action node is done, it would enable the added arrow pointing back to the policy node, such that the policy node can launch again to select the next action node to run, if there is any.

The active document design together with the sophisticated composition support completes the picture of PACMAN's capability for fulfilling automated network management. It fully satisfies the requirements of automating MOP documents, but also goes beyond that by imposing network awareness without additional manual work.

4.3 Execution Engine

An execution engine runs all execution tasks like separate programs, by providing three main functions:

Provide execution environment: Like an operating system, the execution engine allows each running execution task to interact with physical devices or external entities through a set of API calls. The execution state of an execution task is maintained as a collection of enabled arrows by the execution engine. To start an execution task, a dummy condition node is added with an enabled outgoing arrow pointing to the start action node. This effectively allows the start action node to activate the whole execution task. The enabled arrows for each execution task is updated after a node execution finishes. An execution task finishes by calling `TaskSucceed()`.

Handle API calls: To support the most common `CommitConfigDelta()` and `QueryDeviceStatus()` calls, the configuration delta or status query template is first parametrized, based on the input parameters to the execution task, and then fed into the proper device, which is usually indicated in the input parameters as well. If the configuration change is accepted by the device, the API call is done. A configuration delta may not be accepted by the target device for various reasons, *e.g.*, command syntax error, missing reference links, or device errors. In these cases, `TaskFail()` is called by the execution engine for the execution task. `NotifyEntity()` and `QueryEntity()` are invoked based on the node specification. The message or query should be parametrized as well. `QueryExecutionState()` returns the list of enabled arrows and the current nodes to the calling condition node, mostly used in policy nodes that need to reason about the execution state.

Handle failures: An execution task fails if `TaskFail()` is called. The execution task is stopped immediately, and a snapshot of the execution status is taken, which consists of the result for `QueryExecutingState()` along with the condition node that

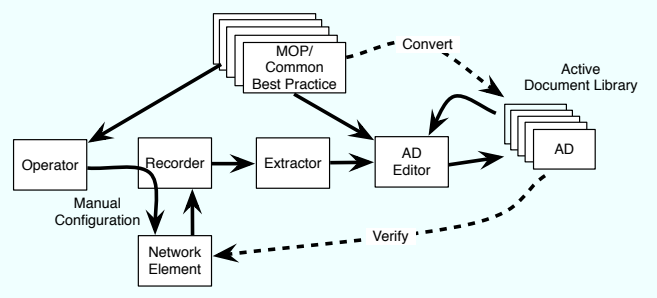

Figure 8: AD Creation Framework

reports the failure. These are recorded for future manual inspection. The execution engine allows different failure mitigation strategies. By default, the effect of the whole execution task is rolled back. To support this, the execution engine maintains an execution history for each task. The rollback action is done by undoing all the configuration changes made based on history information. If external entities are notified in any form, revoke messages are sent. Other mitigation strategies may be also be used, such as no-rollback, partial rollback or redo.

5. CREATING ACTIVE DOCUMENTS

In Section 4 we abstractly described the creation of active documents and their derived execution tasks. We now describe a practical framework to assist the rapid creation of ADs. This AD creation framework is depicted in Figure 8. There are two requirements for building this creation framework: i) high usability, which can lead to quick adoption; ii) high expressiveness, so that the generated ADs describe management tasks accurately. An AD is created via the following three steps, allowing quick transformation from MOPs to ADs, forming an AD library:

Task observation: An operator or AD designer, guided by the MOP documents or with a common best practice in mind, performs a network management task on a set of network elements, typically in a testbed environment. Operators directly access the target devices, *e.g.*, spawn multiple SSH sessions to CLI, while we use a *recorder* to transparently capture the full interaction. We record: i) performed activities through CLI, *e.g.*, modify configuration, change protocol state (such as BGP session reset), acquire network status; ii) device response and internal states, *e.g.*, displayed network status, emitted SNMP trap messages and device log messages. The recordings are tagged with timestamps.

Event extraction: Action and condition activities are extracted from the recordings automatically by an *extractor*. Contiguous configuration changes are grouped together as a single event, as long as there are no condition activities in the middle. Similarly, repetitive condition checks with the same result are combined. When device status is inspected in the CLI, we correlate logs from other sources, such as SNMP or device log message, to augment the condition event. That is, we allow the operator to specify the decision logic based on those information sources as well.

Operator annotation: The events extracted from previous step are presented as action or condition nodes in an *AD editor*, pending operators' annotation to complete the AD generation. The operator has to specify i) the parameters that are specific to tasks, so that we abstract them as placeholders for future re-use, ii) the workflow logic by drawing arrows between nodes, *e.g.*, identify two parallel branches, iii) the information source and decision logic in each

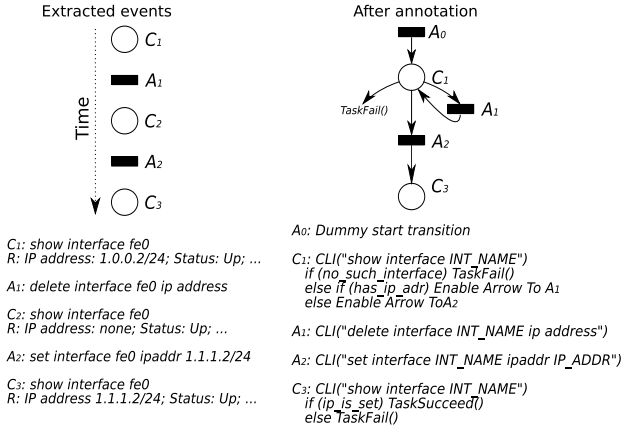

Figure 9: Example of operator annotation

condition node, iv) external synchronization events, since they are not recorded, v) additional events for hypothetical scenarios, *e.g.*, failure detection (condition) and response (action).

Figure 9 shows an example of operator annotation. The left side shows the processed CLI log by events generation. C indicates a condition checking, followed by the response (R) from the device; A indicates an action performed by the user. On the right side, the annotation result is shown: all IP addresses and interface names are replaced by generic placeholders, such as `INT_NAME`; the nodes are connected by arrows, indicating execution flow; for each condition nodes, the actual decision process is formally specified in the form of a sequence of `if-then-else` statements, which process the retrieved status information and determines a follow-up action. The same framework for annotation can be used to directly create or modify ADs, *e.g.*, by a skilled designer. This creation process only needs to be done once, as the generated ADs can be re-used and composed in the future to fulfill similar or even more complex tasks.

One limitation of this creation framework is that the recorded AD reflects the operations on a *fixed* amount of devices. As such, tasks like "to enable IS-IS on *all* routers" cannot be captured by a single AD, because the number of routers in the creation environment does not match that in the production environment. To overcome this problem, it is advised that the operators create ADs that are smaller and more specialized, *e.g.*, "to enable IS-IS on a single router", and use the composition mechanisms to stitch multiple ADs together. For more sophisticated tasks, *e.g.*, "operate on all the routers that meet a certain criterion", the AD designer can design ADs to operate on one device, while encoding the selection criteria into the beginning of the AD, so that the operator can simply compose an execution task that works on all routers. Another solution is to leverage on external databases to determine the set of devices to operate on.

6. CASE STUDIES

In this section, we use several realistic examples to show how active documents are used to perform complex yet automated network operations in the PACMAN framework. Since a quantitative measurement of improvement is hard, we qualitatively evaluate the benefit of PACMAN comparing to existing approaches.

6.1 Fault Diagnosis

Active document is an ideal candidate for automating the fault

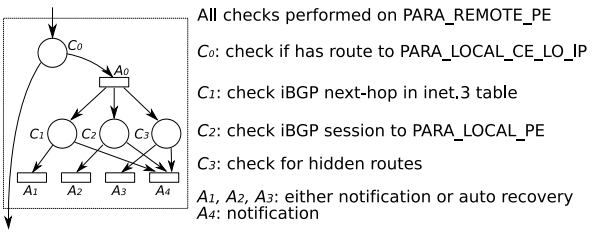

Figure 10: Layer-3 VPN diagnostic AD

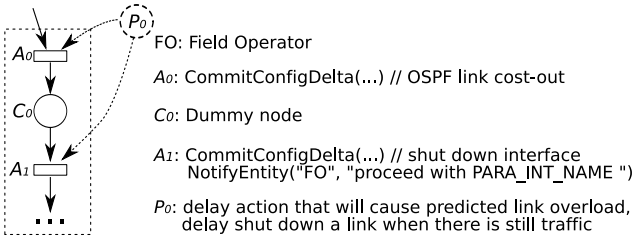

Figure 11: Planned maintenance AD

diagnosis process. Condition nodes can be used to retrieve relevant information from various devices and then reason about the symptom. The outgoing arrows of condition nodes correspond to different diagnosis results and may lead to additional steps.

Figure 10 shows a portion of an active document that is used to diagnose layer-3 VPN connectivity. This AD is converted from a MOP provided by a major router vendor [20]. The whole diagnosis procedure checks multiple routers to see if VPN routes can properly propagate from a local customer edge (CE) router, through the local provider edge (PE) router, and reach the remote PE router and remote CE. The example shows the portion that diagnoses if routes propagate correctly from local PE to remote PE. C_0 logs into the remote PE router to check if the loopback IP address of the local CE router is seen on its layer-3 VPN routing table. If true, it means that routes from the local CE correctly propagate to remote PE, thus this portion of AD can be bypassed. Otherwise, A_0 is executed to spawn multiple tests to further diagnose the problem: *e.g.*, C_2 checks on remote PE if the iBGP session to the local PE is properly established; if not, the problem is found, leading to the execution of A_2 which either starts another sub-task to automatically fix the BGP session or calls `NotifyEntity()` to contact an operator about the diagnosis result.

The flexible composition capability provided by active documents allows network-wide fault detection, fault diagnosis and fault recovery in a closed loop by stitching appropriate ADs, reducing human involvement significantly. The state of the art in automated fault diagnosis relies on router vendor support [10] to execute diagnosis scripts automatically when certain condition is met. This support is limited to a single device, while PACMAN can easily correlate and reason about status from devices across the network.

6.2 Link Maintenance

Figure 11 shows a planned maintenance task with enhancement by applying a network-aware policy. The dashed box contains part of the original active document: action node A_0 increases the OSPF metric of the target link to cost it out; A_1 brings the link down by changing configuration and, at the same time, notifies field operators, signaling them to start the on-site maintenance on related physical interfaces. The link bring-up procedure is similar thus ignored for brevity.

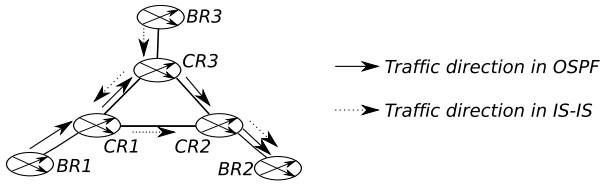

Figure 12: A simplified ISP Backbone

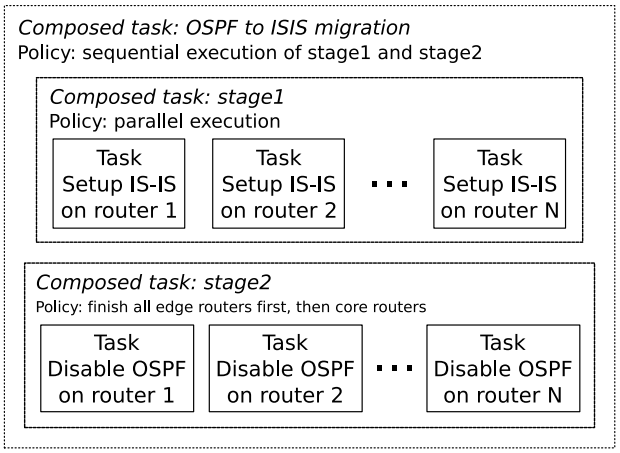

Figure 13: Task design for OSPF to IS-IS migration

This task involves OSPF weight change and interface shut down thus has the potential of negatively impacting live traffic. Current solutions rely on operators to manually predict and avoid negative impact, a usually slow and unreliable process, which is particularly undesired for such tasks with stringent requirements on timing and reliability. In PACMAN, we can impose a policy node, like P_0, to enforce a high-level policy that automates a network-aware decision process for minimizing traffic disruption. P_0 is composed with the original AD with added arrows pointing to all action nodes. In essence, P_0 is a condition node that reasons about network-wide states, such as traffic demand matrix, existing OSPF weights, *etc.*, and makes decisions by enabling the arrows to appropriate action nodes. In effect, P_0 will not allow A_0 (OSPF cost-out) to proceed, unless the estimated traffic shift caused by A_0 would not overload other links; P_0 will not allow A_1 (interface shut down), unless i) the routing has converged and ii) indeed no traffic is flowing through the link. Given the composition capability, P_0 can be used to regulate arbitrary tasks without additional manual work. This is especially useful for carrying out simultaneous maintenance tasks, which are hard to coordinate by operators and may cause significant network downtime, *e.g.*, a partitioned network.

Besides using network-aware policy control, this maintenance job can also take advantage of external reasoning platforms, such as a traffic engineering planner [5]. For example, C_0 can query the planner if it is permitted to shut down the interface. This allows PACMAN to take full advantage of existing infrastructures.

6.3 IGP Migration

Many ISP networks have performed IGP migration for a variety of reasons [21]. IGP migration is a challenging task as IGP is deep down the dependency stack — many other network services and protocols depend on it. Let us consider the task of migrating a network from running OSPF to IS-IS (actually performed by two large ISPs previously [22, 23].)

The migration process first enables IS-IS (with a lower preference) in the network and then disables OSPF. One of the challenges is to prevent transient forwarding loops. Consider a simplified ISP topology in Figure 12. After IS-IS is enabled and running together with OSPF, it is possible that link $CR1 \rightarrow CR2$ has a high weight in OSPF and $CR2 \rightarrow CR3$ has a high weight in IS-IS. The traffic from $BR1$ to $BR2$ goes from $BR1-CR1-CR3-CR2-BR2$, as OSPF is still the preferred IGP. If OSPF is disabled first on $CR3$, $CR1$ still forwards traffic to $CR3$ because $CR1$ still runs and prefers OSPF, and the shutdown of $CR3$'s OSPF will not be detected after a timeout. $CR3$, on the other hand, switches to IS-IS immediately, thus starts to forward traffic via the path $CR3 - CR1 - CR2 - BR2$. As a result, packets would bounce between $CR1$ and $CR3$, until OSPF re-converges. A simple solution to prevent this in common ISP setups is to disable OSPF on all edge routers first and then on all core routers [23]. This enforcement, however, is unreliable and requires much manual effort in existing approach.

PACMAN automates this process using a composed execution task, with two major stages, as shown in Figure 13:

Stage 1: for each router, i) configure `iso` layer on all interfaces; ii) verify `iso` is enabled; iii) configure IS-IS protocol to run with a lower preference than OSPF; iv) verify the IS-IS protocol has learned all the routes as OSPF does.

Stage 2: for each router, i) deactivate OSPF; ii) verify no loss of routes; iii) remove OSPF config, adjust IS-IS preference.

Both stages are also composed tasks, executed in sequential order. For stage1, all sub-tasks are executed in a simple parallel fashion, because they do not interfere with each other. For stage2, all sub-tasks are executed in parallel, with additional policy enforcement (ordering constraint) to avoid forwarding loops. We will illustrate in the §8.1 the effectiveness and correctness of this composition.

7. IMPLEMENTATION

In this section, we briefly describe our implementation of the PACMAN framework. Two major components are the AD creation framework and the execution engine. All implementations were performed in Java, and we mostly focus on Juniper routers due to availability in our test environment, but our methodology extends to other network devices.

7.1 Active Document Creator

Our implementation of AD creator contains several pieces, some of which leverage existing software packages. We customize `screen` and `script` Linux commands such that SSH sessions can be made simultaneously to the same or different devices while each session interaction being recorded with timing information. SNMP messages and device log messages are constantly being monitored and later retrieved to correlate with console commands based on timing. The annotation is done in a Java-based GUI. For each action or condition node, a pop-out window allows the operator to specify the parameters. For condition nodes, a chain of tests is specified to represent an if-then-else decision making. Each test need to specify: an information source, which could be the result of a status-checking command, SNMP or device logs, or previously saved information; a predicate as test body, which can be as simple as string matching, or as complicated as an XML query — Juniper routers support XML-based interaction for retrieving device status; a test result, which can be an arrow to enable or calling an API.

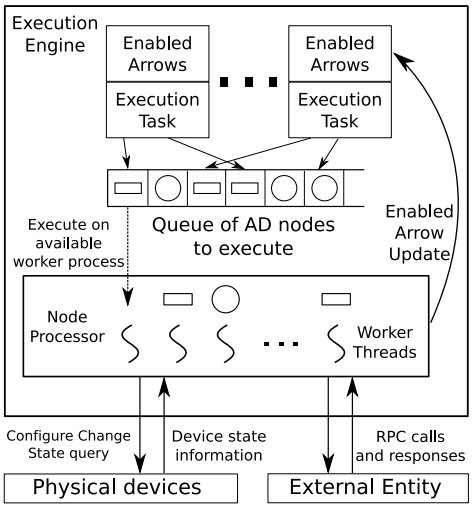

Figure 14: Execution engine architecture

7.2 Execution Tasks and Execution Engine

A simple execution task is created from an AD and a parameter assignment. A quick sanitization process is performed to make sure that enough parameters are specified and the values conform to the parameter types. When composing execution tasks together, the node names and parameter names used in ADs of different subtasks are renamed to avoid confusion. For example, node `M` is renamed as `N_M` where `N_` is a prefix added for all the nodes of the sub-task. (This renaming effect can be seen in Figure 5.)

Figure 14 shows the high-level architecture for our execution engine. Each running execution task is associated with a list of enabled arrows. The execution engine scans all execution tasks periodically. Based on the enabled arrows, the nodes that are ready to execute in each executed task is added into a queue waiting for execution.

A *node processor* is responsible for actual execution of the nodes. Multiple worker threads are spawned to handle concurrency. If a worker thread is available, a node is fetched from the waiting queue. Rather than picking nodes from the head of the queue, a node is randomly selected from the queue, to ensure fairness and avoid potential live lock. To execute a node, parameter values are copied from the execution task to replace the parameter placeholders in the node.

To handle `CommitConfigDelta()` and `QueryDeviceStatus()` in a node, the worker thread contacts physical devices specified via either CLI or NetConf interface. Connections to recently contacted physical devices are cached and reused to reduce connection establishment overhead. Configuration changes made to the same device are serialized to avoid potential conflicts. `QueryEntity()` and `NotifyEntity()` are simple wrappers to external scripts. For example, executing `NotifyEntity('mail','a@b.com','done')` invokes a shell command `./mail.sh a@b.com done`.

7.3 Programming Policy Nodes

Policy nodes can be much more complicated than the regular condition nodes created via the AD creator. In fact, we allow policy nodes to be written in Java and handled using the same execution engine. When executed, a policy node first identifies a set of action nodes that have all other pre-conditions satisfied and are waiting for its permission to proceed. Among these nodes, the policy node

Algorithm 1 Implementation of the prioritization policy node

Require: AdSpec AS, ExecutionState ES, NetworkState NS, DemandMatrix DM

1: $W \leftarrow GenWaitingNodeList(AS, ES)$
2: **for** action n in W **do**
3: $NewState \leftarrow NS$ applies action of n
4: calculate connectivity matrix and traffic on each link based on $NewState$ and DM
5: **if** in $NewState$ network is not partitioned and no overloaded link **then**
6: **return** n
7: **end if**
8: **end for**
9: **return** $null$

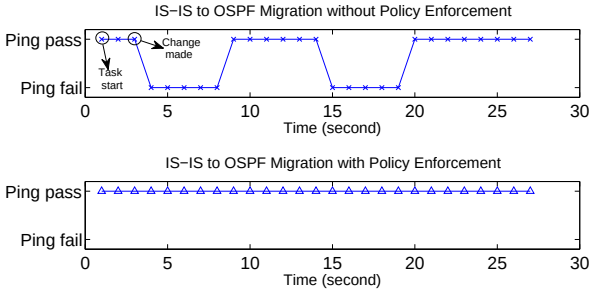

Figure 15: Effectiveness of policy enforcement

can choose one from them to allow its execution by enabling the final arrow. It is possible for the policy node to decide that none of those actions should proceed at the moment. On the other hand, a policy node sometimes need to consider the action nodes that might be executed in the future, because it might be a better choice to execute them rather than currently ready nodes. Determining these two sets of nodes can be done via flow analysis based on the graph structure of the composed AD and execution state. We provide generic helper functions to ease the development process.

Here we describe the sketch of a policy node, which specializes in avoiding network partitioning and traffic overloading caused by arbitrary simultaneous network tasks, shown in Algorithm 1. The first line uses a provided function to generate a list of action nodes that are waiting for permission. Line 2-8 iterate through all such nodes. For each action node being considered, the resulting network state $NewState$, including reachability and routing table, is calculated based on the current network state and the configuration change embedded in the action node. If a partitioned network is detected, the action node will not be permitted. Combining the traffic demand matrix and new routing table, an action is permitted if it does not cause other links to overload or exceed some pre-defined threshold, *e.g.*, 90% utilization ratio.

8. EVALUATION

We evaluated our prototype implementation to demonstrate its effectiveness in preventing operational errors and ensuring efficient configuration management, which scales well with network size.

8.1 Network-awareness Support

To exemplify the effectiveness of policy enforcement, we perform stage 2 of the IGP migration task (disabling OSPF on all routers) in two different ways: i) in multiple individual tasks, each of which disables OSPF on one router, processed by the execution engine concurrently, mimicking the effect of several operators working on different part of the network simultaneously, yet unaware of the potential problem; ii) in one composed task, using the prioritization policy to ensure edge routers are first updated before changing the configuration of any core routers. We used six Juniper routers in a local testbed, connected as shown in Figure 12. The experiments were performed on the network state after stage 1 of IGP migration had finished (IS-IS was configured as the less-preferred IGP, while OSPF was still running as the preferred IGP). One machine connected to $BR1$ was sending `ping` to another machine connected to $BR3$ during the migration period. Link weights of OSPF and IS-IS were intentionally tweaked to create the situation discussed in §6.3.

Figure 15 shows the result. When individual tasks were executed in parallel, repeating this experiment multiple times showed that when the task working on $CR3$ is executes first a forwarding loop was indeed created as shown in the top figure: the connectivity was temporarily lost for a few seconds (the amount of time to commit configuration changes on Juniper routers) after the task started. The connectivity was resumed and lost again before it eventually stabilized, mostly due to the complex interaction of the two IGP protocols. In contrast, the composed task using prioritization policy did not experience any problems, as shown in the bottom figure.

8.2 Automating Network Operations

We again use the IGP migration task in an ISP depicted in Figure 13 to estimate the time saving by using PACMAN to automate network operations. For comparison purposes, one of the authors who is proficient in network management and router configuration performed the migration task. That author performed the task several times beforehand for training purposes and then reported the lower-bound estimate of how long a sub-task would take when executed manually (we expect the actual performance numbers from real operators to be quite similar).

The amount of time to manually perform configuration change on the routers takes less than 2 minutes each, thus ideally 12 minutes to finish six routers. Interestingly, the total amount of time to finish the migration task for all routers takes no less than 25 minutes, due to additional network status verification and inter-device synchronization. In contrast, PACMAN finishes the whole migration task within 2 minutes - 90% of time on effecting configuration change and acquiring status via NetConf and the rest on internal processing. If we extrapolate to a network of 100 routers, the manual operation time is over 400 minutes, exceeding an entire maintenance window, which is typically of a 3 to 4 hour duration. Even worse, when the operated network is considerably larger, the manual operation time is unlikely to scale linearly, despite the potential use of automated scripts, due to more complicated network status verification and additional synchronization between involved human operators. In fact, the IGP migration processes documented online [23] took several maintenance windows across 3-5 days to finish. For PACMAN, since all the verification process are accurately modeled and automatically carried out, it can easily scale with the network size.

8.3 System Constraints

The execution engine directly interacts with physical devices. Out-of-band access, which is standard in ISP environment, provides a more reliable connectivity channel, but the bandwidth is limited, ranging from 9.6kbps serial console, 56Kbps modem line to 1.5Mbps T1 connection. Router configuration in XML format is usually tens or hundreds of kilobytes. Assuming a T1 connection, it may take around hundreds of milliseconds to transfer a complete configuration file. Fortunately, most management activities can be

performed in-band where bandwidth is not an issue.

We performed some micro-benchmarks to investigate resource constraints the server that runs the execution engine. On a server with 2.5G Intel core 2 duo CPU, it takes about 360μs to load a 2KB XML file with 86 lines, describing 10 routes in the routing table. It takes about 950μs to perform an XPath query to count the number of routes described in the XML file. The processing time should be on the order of hundreds of milliseconds to handle 10000s routes. The processing power may become a bottleneck when the reasoning activity becomes significantly more complicated. This can either be mitigated by using multiple execution engines for load-balancing, or offloading some reasoning logic to programmable routers.

9. CONCLUSION

Network management has been an enduring research topic. Despite efforts by both academia and industry, network management remains largely driven by manual efforts, and is thus error-prone and time-consuming. In this paper, we proposed the PACMAN platform, aiming to automate existing network management operations and enabling the adoption of new holistic network-wide operational practices. The key intuition behind our work is to use the right level of abstraction which is both close enough to current management approach, thus enable quick adoption, general enough to capture the complexity of existing approaches, and powerful enough to automate and augment them.

Towards the goal of building automated network management system, PACMAN uses the Active Document abstraction to systematically capture the dynamics of network management tasks. This abstraction allows the composition and execution at task level, thus raising the level of abstraction. The ability to integrate network-wide policies distinguishes PACMAN from device-centric support from vendors and task-oriented nature of MOPs.

We described the design and implementation of the PACMAN framework, and used realistic usage scenarios to show its effectiveness. As future work, we plan to corroborate with network operators for feedback and comments in order to further improve the usability and practicality of PACMAN. In particular, we aim at allowing more flexible creation and more programmable composition of active documents by improving the interaction between human operators and our system.

Acknowledgement

We would like to thank our shepherd Sanjay Rao and the anonymous reviewers for their valuable comments. We are also grateful for the feedback from Yun Mao, Jennifer Yates and Bobby Bailey. Chen is supported in part by US NSF Award CNS-0939707 and DARPA Computer Science Study Panel. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding sources.

10. REFERENCES

[1] Z. Kerravala. Configuration Management Delivers Business Resiliency. The Yankee Group, Novenver 2002.

[2] William Enck, Patrick McDaniel, Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel, Albert Greenberg, Sanjay Rao, and William Aiello. Configuration management at massive scale: system design and experience. In *Proceedings of USENIX Annual Technical Conference*, 2007.

[3] Joel Gottlieb, Albert Greenberg, Jennifer Rexford, and Jia Wang. Automated Provisioning of BGP Customers. *IEEE Network*, 17, 2003.

[4] Hagen Bohm, Anja Feldmann, Olaf Maennel, Christian Reiser, and Rudiger Volk. Network-wide inter-domain routing policies: Design and realization. Presentation at the NANOG34 Meeting.

[5] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, and Jennifer Rexford. NetScope: Traffic engineering for IP networks. IEEE Network Magazine, March/April 2000, pp. 11-19.

[6] Don Caldwell, Anna Gilbert, Joel Gottlieb, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. The cutting EDGE of IP router configuration. In *Proceedings of ACM SIGCOMM HotNets Workshop*, 2003.

[7] Anja Feldmann and Jennifer Rexford. IP network configuration for intradomain traffic engineering. *IEEE Network Magazine*, pages 46–57, September/October 2001.

[8] Nick Feamster and Hari Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *Proceedings of Symposium on Networked Systems Design and Implementation*, May 2005.

[9] Xu Chen, Z. Morley Mao, and Jacobus Van der Merwe. Towards Automated Network Management: Network Operations using Dynamic Views. In *Proceedings of ACM SIGCOMM Workshop on Internet Network Management (INM)*, 2007.

[10] Juniper Networks, Configuration and Diagnostic Automation Guide. `http://www.juniper.net`.

[11] Cisco Active Network abstraction. `http://www.cisco.com`.

[12] Hitesh Ballani and Paul Francis. CONMan: A Step towards Network Manageability. In *Proceedings of ACM SIGCOMM*, 2007.

[13] Richard Alimi, Ye Wang, and Yang Richard Yang. Shadow configuration as a network management primitive. In *Proceedings of ACM SIGCOMM*, 2008.

[14] Yi Wang, Eric Keller, Brian Biskeborn, Jacobus van der Merwe, and Jennifer Rexford. Virtual Routers on the Move: Live Router Migration as a Network-Management Primitive. In *Proceedings of ACM SIGCOMM*, 2008.

[15] John C. Strassner, Nazim Agoulmine, and Elyes Lehtihet. FOCALE - A Novel Autonomic Networking Architecture. Latin American Autonomic Computing Symposium (LAACS), 2006.

[16] Hemant Gogineni, Albert Greenberg, David A. Maltz, T. S. Eugene Ng, Hong Yan, and Hui Zhang. MMS: An Autonomic Network-Layer Foundation for Network Management. Rice University Technical Report TR08-11, December 2008.

[17] Tadao Murata. Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE 77, 4 (1989).

[18] W.M.P. van der Aalst. The application of petri nets to workflow management, 1998.

[19] Yin Wang, Terence Kelly, Manjunath Kudlur, Stephane Lafortune, and Scott Mahlke. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *Proceedings of OSDI*, 2008.

[20] Juniper Networks: Troubleshooting Layer 3 VPNs. `http://www.juniper.net/techpubs/software/junos/junos93/swconfig-vpns/tr%`

oubleshooting-layer-3-vpns-using-ping-\
\and-traceroute.html.

[21] Manav Bhatia *et al.*. IS-IS and OSPF Difference Discussions. http://www.join.uni-muenster.de/Dokumente/drafts/draft-bhatia-manral-di%ff-isis-ospf-01.txt.

[22] Vijay Gill and Jon Mitchell. OSPF to IS-IS. http://www.nanog.org/mtg-0310/pdf/gill.pdf.

[23] Results of the GEANT OSPF to ISIS Migration. http://www.geant.net/eumedconnect/upload/pdf/GEANT-OSPF-to-ISIS-Migrati%on.pdf.