Adaptive Parsing of Router Configuration Languages

Donald Caldwell
AT&T Labs Research

Seungjoon Lee
AT&T Labs Research

Yitzhak Mandelbaum
AT&T Labs Research

***Abstract*—Network functionality is growing increasingly complex, making network configuration management a steadily growing challenge. Router configurations capture and reflect all levels of network operation, and it is highly challenging to manage the detailed configurations of the potentially huge number of routers that run a network.**

One source of difficulty is the constant evolution of router configuration languages. For some languages, particularly Cisco's IOS command language, and its relatives, these changes demand frequent maintenance of configuration parsers in any configuration management tool. The essential problem is that config parsers understand a statically determined set of inputs, requiring human intervention to modify that set. We propose an alternative design for router configuration parsers: adaptive parsers. Such parsers can infer the configuration language based on real configs and automatically adapt to changes in the config language, all with minimal human involvement. We present the design of such a parser and discuss its prototype implementation for the Cisco IOS configuration language. We have validated our prototype's accuracy and efficiency by running it on the configuration files of Tier-1 ISP networks. Our results show that from only 81 configuration files, we can learn enough IOS to successfully parse all of the 819 IOS configurations in under 10 minutes.

I. INTRODUCTION

Automated configuration management is becoming increasing crucial to providing reliable network service and troubleshooting. Yet, some configuration management tools face a vexing problem: the constant evolution of router configuration languages, demanding frequent modifications to configuration parsers. Specifically, network analysis tools targeting Cisco IOS-based routers, and others with similiar configuration languages, require their parsers to be manually updated to incorporate new commands – a tedious and error-prone process. Additionally, the size of these router configuration languages makes creating a new parser similarly tedious and error-prone.

In this paper, we present an *adaptive* parsing system for Cisco's IOS command language, which simultaneously addresses the problems of configuration language size and frequent evolution. Our system, starting with a small, fixed amount of domain-knowledge about IOS, infers an approximation of the router configuration language as it runs, and remembers the inferred structure for to improve the performance and robustness of future parses. We have divided the implementation of our system into two major components. One leverages IOS domain-knowledge to infer an approximation of the IOS language from input configuration files, and then generates a description of the inferred language in PADS/ML, a human-and-machine readable data description language [1]. The inferred description captures the syntax used by each command to group together its subcommands (if any) and the set of such subcommands[1]. The other component uses the inferred IOS description to parse configuration files and process them (e.g. convert to XML) for downstream use. In this way, our system can both learn the configuration language and adapt to new language elements, with one mechanism. When our system encounters new commands, the inference engine detects their syntax and set of subcommands and adds them into the language description. Then, the system can parse the new configurations based on the modified language.

We have targeted our adaptive parsing system to Cisco IOS– the operating system and configuration comand language of Cisco's routers. While we can apply and intend to expand our adaptive parsing method to other, related, router configuration languages, this paper focuses on Cisco's IOS because (1) Cisco routers are widely deployed and (2) their configuration language is particularly difficult to parse.

This paper makes the following contributions:

- We have developed a system for effectively parsing Cisco IOS configuration files based on a small core of IOS-specific knowledge. In Section III, we discuss the two-phase design of the system, wherein Phase I infers an approximation of the Cisco IOS language from valid router configuration files and generates a formal description of the language in PADS/ML (Section III-A), and Phase II uses the generated description to translate parsed configs into XML suitable for downstream processing (Section III-B).
- We have implemented (in less than 1100 lines of code) the ideas presented in this paper and validated them through experiments using real Cisco router configurations from Tier-1 ISP networks (Section IV). Our results show that after parsing only a small set of configuration files, the parser can accurately parse hundreds of configurations (millions of lines combined) in less than 10 minutes.

Before describing the details of our adaptive parser, we provide some background on the Cisco IOS command language and the PADS/ML data description language in the following section.

II. IOS AND PADS/ML

A. *Cisco* IOS *Configuration Files*

The set of IOS commands and their syntax is referred to as the IOS language, and collections of IOS commands are referred to as IOS configurations (or *configs* for short). IOS is a declarative language and its basic element is a "command". There are *simple commands*, whose influence

[1] The description does *not* include the format of the command parameters

```
interface POS2/0
 description interface description
 ip vrf forwarding 123:123
 ip address 135.12.34.65 255.255.255.252
 ip accounting precedence input
 pvc 0/1919
  service-policy input interfacename_input
 !
class-map match-any classname
  match ip dscp cs6
  match ip dscp cs7
!
router ospf 12
 log-adjacency-changes
 area 0 authentication message-digest
 network 135.12.34.64 0.0.0.3 area 0
 maximum-paths 6
!
router bgp 7018
 neighbor 12.23.34.45 activate
 neighbor 12.23.34.45 send-community
 neighbor 12.23.34.45 route-map InRMName in
 neighbor 12.23.34.45 route-map OutRMName out
 address-family ipv4 multicast
 neighbor 12.23.34.45 peer-group FOO:bar
 no auto-summary
 exit-address-family
!
crypto key pubkey-chain rsa
 addressed-key 12.23.34.45
  address 12.23.34.45
  key-string
   00302017 4A7D385B 1234EF29 335FC973
   2DD50A37 C4F4B0FD 9DADE748 429618D5
   90288A26 DBC64468 7789F76E EE21
  quit
 !
!
banner exec +
============= WARNING =================
This system is intended strictly for use
by authorized users.
+
```

Fig. 1. Sections from an IOS configuration file.

does not extend beyond a line, and *mode commands*, which change the state of the command interpreter. Each mode has a restricted set of allowable commands, which may be simple commands or mode commands. We call the portion of the configuration in which a mode is active a *section*. Commands can have arguments which set particular parameters of the router operating system. For example, the `area` command in the `router ospf` [2] section in Figure 1 specifies the authentication method to use in OSPF area 0.

The above description of IOS captures its logical structure. Unfortunately, that logical structure is not expressed in configs in a syntactically consistent manner across all commands. Rather, a set of several common syntactic structures exist, all expressing the same logical structure. In addition, some commands have their own, unique structure, shared by no other commands. We have identified five unique syntactic structures, each of which are demonstrated in Figure 1.

The `interface` and `class-map` commands demonstrate the most prevalent syntactic structure: a command starts a section and its subcommands are indented at least one more space than the starting command. The section is closed when a command is encountered with equal or less indentation than the starting command. Notice that `interface`-subcommand `pvc` starts its own (sub)section, with one member (`service-policy`). Note that we consider simple commands to be just a special case of this syntactic form, just without any children.

Often, sections are distinguished by the first token in the starting command. However, some commands rely on the command's first parameter (the second token) to determine the set of subcommands for the section. In essence, that first command names a subcommand of the section. For example, notice that the two `router` sections in Figure 1 have distinct sets of subcommands, because "`ospf`" and "`bgp`" are essentially independent subcommands of "`router`". We call such sections *two-dependency sections*, classifying them with there own syntactic form, because they require distinct treatment from ordinary indentation-based sections.

Another structure uses explicit begin and end commands to delimit a section. The `key-string` command is one such example, relying on `quit` to explicitly close the section. This command also demonstrates another form, in which elements of the section are data. For the `key-string` section, the elements are the contents of a public key. The final syntactic form uses user-chosen parameters to delimit the end of the command. An example is the `banner` command, whose purpose is to specify the banner shown to users upon login and other activities. For this command (and others like it), the second parameter indicates the closing delimiter for the contents of the banner.

The above syntactic forms pose a number of challenges. First, a parser must have a way to determine the syntactic form used by a command in order to parse it correctly. The parser can make these determinations based on hard-coded knowledge, or perform them dynamically. The latter case poses a second challenge: these dynamic determinations can adversely impact parser performance. Third, indentation-based commands have no explicit command to end their section. A parser must either use indentation for clues, know the set of commands that implicitly indicate that the section has ended, or know the set of commands allowed in each section. Last, as we can see with the `banner` command in Figure 1, we cannot model IOS using a context-free grammar. Moreover, parsing sections based on their indentation level is also context-dependent. This context-dependency greatly complicates or even precludes processing by many off-the-shelf grammar compilation tools, like YACC, which do not support context-sensitive grammars.

B. PADS/ML*: A Data Description Language*

PADS/ML is a *data description language* designed for the concise and declarative description of legacy data formats, including those for which no context-free grammar exists [1]. PADS/ML's support for legacy data formats goes beyond its description language: from a description, PADS/ML can generate many essential software artifacts, including a data structure for representing parsed data in memory, a recursive-descent parser, a printer and a data validator. All software artifacts are generated as source code in OCAML, a safe, general-purpose programming language, supporting functional, im-

```
ptype section (min_indent : int) = {
  indent : [ i : pstring_ME(``/ */'') | length(i) >= min_indent];
  start_cmd  : command; peol;
  sub_cmds   : section ( length(indent) + 1 )
                  plist(No_sep, Error_term)
}
```

Fig. 2. A description of the indentation-based section form in PADS/ML.

perative and object-oriented programming [3]. In addition to generating software, PADS/ML provides a framework for developing *generic tools* in OCAML– functions that can operate on any description-derived data structure, regardless of its type. PADS/ML does the work of specializing generic tools written in its framework to the particular data structures derived from a description.

The extensive, end-to-end support that PADS/ML provides for processing legacy data formats makes it a natural choice for addressing the challenges of the IOS language. The small codebase size mentioned in the introduction (< 1100 LOC) owes almost entirely to the conciseness with which we can write PADS/ML descriptions and generic tools.

We now briefly describe the PADS/ML language. A more complete description of the language and other features can be found in earlier publications [1], [4]. A PADS/ML description specifies the physical layout and semantic properties of an ad hoc data source. These descriptions are formulated using types. Base types describe atomic data, like ASCII-encoded, 8-bit unsigned integers (`puint8`), dates (`pdate`) and strings (`pstring`). Certain base types take additional OCAML values as parameters. For example, `pstring_ME(re)` describes strings that match the regular expression `re`. Literal values describe exactly themselves.

Structured types describe compound data, and are built using ML-like type constructors. For example, *records* and *tuples* (records with unnamed fields) describe fixed-length sequences of data with elements of different types. *Datatypes* (or *variant* types) describe elements of a format where multiple alternative types of data may appear. *Lists* describe homogeneous sequences of data and *type constraints* describe data satisfying arbitrary programmer-specified semantic conditions – for example, that a string has at least ten characters.

Users can define their own type constructors using *polymorphic* types – types parameterized by other types, and *dependent* types – types parameterized by values. Many base types are examples of dependent types. There are also a variety of built-in polymorphic types; for example, `plist`, a polymorphic, dependent description of lists. It has three parameters: on the left, a type parameter: its element type; on the right, a pair of value parameters: an optional *separator* that delimits list elements, and an optional *terminator*.

As an example, we demonstrate in Figure 2 how to describe the simplest section form: a single command, with zero or more children of greater indentation. Type `section` is parameterized by the minimum required indentation, and its actual indentation is checked against the parameter. The first field, `indent`, describes the indentation preceding the first command of the section. Its string value is constrained to

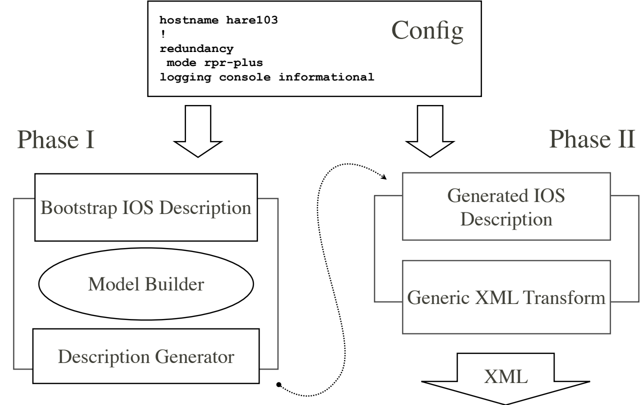

Fig. 3. The adaptive parser phases and their components.

be at least the minimum length supplied as an argument. A lesser indentation would violate the constraint, resulting in a parse error. Field `start_cmd` describes the first command of the section, and field `sub_cmds` describes the list of its subcommands, each of which must have a minimum indentation *at least one greater* than the current indentation. We use a `plist` whose elements have no separator (`No_sep`) and which terminates when it encounters an element with an error (`Error_term`). When a subcommand is encountered without the required minimum indentation, the constraint on the `indent` field will be violated – a parse error, thereby terminating the list (with that command returned to the input). In this way, only commands with the correct minimum indentation are included in the current section. Notice the dependence of the type of `sub_cmds` on the value of field `indent`: this kind of dependency cannot be expressed in a context-free grammar, and is a key feature of PADS/ML.

III. Adaptive Parsing

We have implemented an adaptive parsing system to address the challenges described in Section II-A. Our system has two phases: Phase I, the inference phase, and Phase II, the parsing and processing phase. In Figure 3, we illustrate the relationship between the phases and their composition. The input to both phases are IOS configs. The output of Phase I is a PADS/ML description of IOS, which becomes an input to Phase II. The output of Phase II is an XML representation of the input config(s). For both phases, we layout their components according to the order of the data flow within the phase. Note that the choice of PADS/ML for both IOS descriptions is natural because of its support for context-sensitive parsing, a requirement for parsing IOS.

A. Phase I: Inference

Phase I has three components: a bootstrap IOS description (in PADS/ML), a *model builder*, and an IOS-description generator. At approximately 130 lines, the bootstrap IOS description is a concise, high-level description of pretty-printed IOS

configs, including their five syntactic structures together with other, more minor, syntactic details. It has evolved over time, through trial-and-error and the experience of the authors, and contains the bulk of the IOS-specific domain knowledge in the system. This description is valuable in its own right, because it addresses the first challenge noted in Section II-A [2].

Except for a handful of commands, the description distinguishes commands based on their structure, not their name. Commands mentioned explicitly in the description are those with a specialized section type: two-dependency, character-delimited, and command-delimited sections. Still, though, we infer the details of their structure and their subcommmand sets. For example, the command-name "`ip`" is listed as indicating a two-dependency section, but the details of the "`vrf`" subcommand are inferred by the parser.

From the bootstrap description, the PADS/ML compiler generates a parser, which performs two functions: first, it classifies each command in a config according to its syntactic structure; second, it groups each command together with its subcommands (if any) into a tree-shaped data structure. The *model builder* is responsible for using the output of the parser to create a simple model of IOS: a mapping from commands to their syntactic structure and set of child commands. In essence, creating a map means combining the results of parsing all of the commands in the input configs into a single tree data structure with one entry per command (as opposed to one entry per command *instance*). Note that the system is sensitive to the context of a command – a command appearing as a subcommand of multiple different commands is treated distinctly for each appearance. Therefore, the inferred mapping from commands to their structure and child-set is tree-shaped, rather than flat.

Once the map has been created from the input configs, it is converted into, and output as, a PADS/ML description. Unlike the original description, this description is specialized to the particular commands whose structure was determined during Phase I. This specialization has a number of benefits, which we discuss at the end of this section.

B. Phase II: Parsing and Processing

Once Phase I has created a (specialized) IOS description, Phase II can parse and process configs. The structure of Phase II is similar to that of Phase I, albeit simpler. It consists of an IOS description and an XML generator, embedded in a "driver," which coordinates the components and provides the user with a simple command-line interface. The code for Phase II can be thought of as a harness into which generated IOS descriptions are "dropped" to produce an IOS-processing application. In our case, the processing is a conversion into a simple XML format, which groups commands into XML elements according to their section, while maintaining the order in which they appear in the original config. The original text of each command's arguments (if any) are saved together with the command in the XML. Note that the choice of XML format, and even the choice to produce XML, rather than, for example, run some analysis, are not essential to the system. One could easily modify the XML generator to produce XML with a different schema, or even replace the XML generator with a different component entirely.

Our XML-conversion component is implemented as a *generic* function, meaning that it works on data of any type. It is written using PADS/ML's generic programming toolkit [1]. The use of a generic function is critical to the success of the conversion tool, for a number of reasons. First, the size and complexity of the type of parsed IOS commands are directly proportional to the size and complexity of the IOS format. Therefore, writing a tool that would work directly on data of this type (that is, parsed IOS commands) would be a very tedious process. Second, while each IOS description shares the same metastructure, the in-memory representation of each one is different, with different OCAML types. Therefore, a (non-generic) function written to work on one IOS representation will not work on another IOS representation. This incompatability would require us to modify the conversion function each time a new description is inferred – again, a tedious and error-prone process. The generic tool framework saves us from both of these hassles. We write one, generic, XML converter and PADS/ML adapts it to each inferred description.

C. Handling New and Modified Commands

Under most circumstances, the conversion program will run without modification. Occassionally, however, it will encounter new commands, or commands whose syntax has changed. When a new command is encountered, it is not recognized by the Phase II parser and is therefore reported as an error. Under these circumstances, we rerun Phase I on the offending config to determine the new command's syntax and appropriately update the description used by Phase II. Currently, this automation is not supported within the Phase I or Phase II programs, but by a script. Phase II outputs a list of all configs with unrecognized commands and the script passes this list to Phase I along with the current description used in Phase II. Phase I produces an updated description and then the script recompiles Phase II. At that point, regular config processing resumes.

If a command has changed in structure then the process is somewhat more complex. Again, Phase II's parser will report an error upon encountering the command and the script will rerun Phase I on the offending config. However, at this point, the process diverges. Phase I will recognize a conflict in the previously-recorded syntax of the command and its syntax in the current config. At this point, our system merely reports the error and leaves the previous description in place. We could add flags to change the behaviour of Phase I - for example, to change the default behaviour to replace the syntax, or to specify overrides for particular commands. We leave more complex behaviours, such as conditioning the choice of syntax based on the IOS version reported at the header of the config, for future work. In our experience, we have only seen these

[2]While we don't have space to include it here, we have posted the description on the web [5].

Number of Configs (Increment)	4 (4)	8 (4)	25 (17)	50 (25)	81 (31)
Parsing success ratio	0.32	0.61	0.93	0.96	1.00
No. of sections learned	37	40	43	52	56
Time to learn additional configs (sec)	19.6	7.6	70.4	69.0	106.9

TABLE I
PERFORMANCE WHEN WE VARY THE CONFIGURATION COUNT FOR LEARNING

Phase I		Phase II	
General description	146	Parsing engine	173
Model builder	324	XML formatter	220
Description generator	106	Common types description	65
		Miscellaneous	17
Subtotal	576	Subtotal	475

TABLE II
LINE COUNT FOR DIFFERENT SECTIONS OF CODE (1051 TOTAL)

types of conflicts when trying to use the same parser for IOS and IOS XR.

D. The Benefit of Two Phases

If we have a component which is able to parse IOS well-enough to infer an (approximate) IOS grammar, then why bother with a second component which uses the inferred grammar? Their are multiple motivations for this design. The foremost benefit is that inference is relatively costly, so our system is designed to pay the cost of determining the structure of command once – in Phase I, and then benefit from it repeatedly – in Phase II. Moreover, for indentation-based commands, knowing the set of valid subcommands lets us ignore the indentation entirely. This feature can also be helpful in parsing the occasional config which we encounter whose indentation has been stripped away. Thus, the two-phase approach addresses the second and third challenges from Section II-A.

The second benefit is to increase the robustness of the parser. In our experience with a large network, we occassionally encounter configs with errors in them. Our system flags configs which don't conform to the inferred grammar, allowing network operators to verify their correctness. Even if the error is caused by valid, yet unrecognized, command, a network operator can benefit simply from being informed of the use of a previously unused command.

However, to benefit from the improved robustness, the system must be used somewhat differently. It cannot be run on any config, with the assumption of its being valid (that is, error-free and pretty-printed). Rather, Phase I must be used more judiciously to avoid learning incorrect information about IOS. In this situation, Phase I "trains" on *selected* configs that are known to be valid, and then Phase II can be run on arbitrary configs. The system can still adapt to configs with new commands after the training process, but such configs must be manually vetted to ensure their validity. While the selection and vetting processes requires human effort, it is nearly always limited to validating configs – the system still automatically updates the IOS description if and when the config has been validated. The only situation that requires more than config validation is when a new syntactic form is encountered, or new commands are encountered that belong to one of the specialized syntactic forms: two-dependency, character-delimited, and command-delimited sections. However, this situation seems to be rare.

IV. EVALUATION

We report the performance of our prototype learning parser implementation using hundreds of router configurations for a Tier-1 ISP network.

A. Experimental Set-up

In our experiments, we use 819 Cisco router configurations, which we can divide into multiple groups depending on their role. For example, some routers are at the edge of the ISP network and interact with external network elements (e.g., in Tier-2 customer networks), and some other routers are in the core of the network and mainly provide connectivity among other routers in the network. We use the following simple strategy for training set selection, while we envision the input selection can potentially be further automated in practice. We first use a small set of randomly selected configurations as training input to the inference component. Then, we see if the inferred model can parse the whole set of 819 configurations. If not, we randomly select a subset of the configurations that the model failed to parse, and use them as input to the next iteration of inference. We repeat this procedure until we can parse all the 819 configurations. We perform our experiments on SUN Fire-15000 multi-processor server (900MHz CPU) running Solaris 5.8. The total human time taken to configure and run these experiments was a few hours.

B. Experimental Results

In Table I, we report experimental results for our parser. The first row of Table I denotes the number of configurations used for inference. In fact, since our parser employs incremental inference (Section III-A), the values within parentheses are the numbers of additional configurations used on top of previously inferred models. Specifically, to complete the learning of 819 configs, our experiments needed five iterations, where we first used 4 configurations files, and subsequently added 4, 17, 25, and 31 files for respective iterations.

In Table I, we observe that we need only around 10% of configurations (81 out of 819) to learn the language model and achieve 100% success ratio. When we first use 4 configurations for inference, we achieve only 32% success ratio (i.e., 263 out of 819), while another 4 configurations help us parse 61% of configs successfully. As we increase the size of training set, the parser can handle more top-level commands (e.g., 37 top-level commands from 4 configs vs. 56 from 81 configs). In our experiments, the 56 top-level commands are sufficient to parse all the given configurations.

Suppose that a model learned from 25 configurations works well with the current network configuration, and then a network operator decides to introduce several new features in the network (e.g., new services). Our result illustrates that we can simply select some of the configurations containing the new features and re-train the parser, which in turn will pick up the new command and be able to parse the new configs.

Running Time: In Table I, we present the running time for each iteration. We observe that the total time to learn from the five iterations is 273 seconds. When we performed an experiment with only one iteration with the final set of 81 training configurations, the learning time was around 240 seconds. This illustrates that the iterated learning does not significantly degrade the running time, compared to the ideal scenario where we know which configurations to use for learning. Finally, we report on the performance of actual parsing. There are millions of command lines combined in the 819 configurations, and it takes around 9 minutes to parse all the configurations.

In Table II, we report the breakdown of lines of code for different parts in our implementation.

V. Related Work

Caldwell et. al. [6] propose a configuration management tool called EDGE, which reads all router configurations in a network, extracts network-wide information from them, and creates a database and relevant reports. Although EDGE employs a parser that handles whole configurations, its parser does not learn by itself and does require manual maintenance whenever changes in configuration language or new features are introduced. In contrast, our parser can learn many such changes automatically, and involves minimal human interaction for maintenance. Many network analysis and management tools use static router configuration information [7]–[10]. Our parsing scheme is an essential building block for such static analysis, and can improve the effectiveness of these network management tools. Our work is different from other configuration management research to *generate* configurations for different routers in an automated fashion [11], [12], in that our focus is on parsing existing router configurations, after which independent tools can extract information and perform configuration analysis.

Outside of configuration management, there has been considerable work studying the problem of *grammar induction*. De La Higuera provides a recent survey [13]. However, this body of work is far more general than the system that we present and their results far less applicable to the problem of network configuration management. We are starting with an accurate description of pretty-printed configs (the bootstrap IOS description), which drastically simplifies the problem of inferring a full description. Additionally, we are seeking a very particular structure to the inferred grammar – correctness alone is not enough. The inferred grammar must allow for straightforward translation of the parse tree into XML with a known schema.

The most closely related work from the area of grammar induction is a concurrent project by Fisher et. al., which explores the inference of a format description from ad hoc data without human intervention or domain knowledge [14]. Once again, the generality of their result makes their work largely inapplicable to our problem. In particular, their system cannot recognize the complex functional dependencies between commands and subcommands, and is therefore unable to infer the relationships between commands which are essential to the IOS language. However, their results seem very promising for inferring the format of parameters for particular commands.

VI. Conclusions and Future Work

Network configuration management is a daunting task with excessive manual involvement, and only growing more complex. In this paper, we have presented an important addition to the network configuration manager's toolbox and validated it with positive experimental results: a parser that can infer an effective approximation of Cisco IOS and use it to support efficient parsing of IOS configs, and adapt to changes in the IOS configuration langauge.

To improve their chances for acceptance by customers, some router manufacturers have created configuration languages very much like IOS, resulting in a number of IOS "dialects." We are investigating expanding our work to include support for these languages. Beyond that, we hope to explore whether the design underlying our system could provide benefits to more regular configuration languages as well.

In addition to router configuration files, network service providers often log all the commands executed on their routers, which typically contain audit trails of IOS commands. The language learned by our system should prove useful for analyzing commands extracted from these logs.

References

[1] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernandez, and A. Gleyzer, "PADS/ML: A functional data description language," in *POPL*, 2007.

[2] *Cisco IOS IP Command Reference, Volume 2 of 3: Routing Protocols, Release 12.2*.

[3] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon, "The objective caml system, release 3.10, documentation and user's manual," 2007.

[4] M. Fernández, K. Fisher, J. N. Foster, M. Greenberg, and Y. Mandelbaum, "A generic programming toolkit for PADS/ML: First-class upgrades for third-party developers," in *PADL*, 2008.

[5] "Cisco IOS description in PADS/ML." [Online]. Available: http://www.research.att.com/~yitzhak/pml/cisco-ios.pml

[6] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford, "The cutting EDGE of ip router configuration," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 1, pp. 21–26, 2004.

[7] N. Feamster and H. Balakrishnan, "Detecting BGP Configuration Faults with Static Analysis," in *2nd Symp. on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[8] G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, "On static reachability analysis of IP networks," in *Proc. IEEE INFOCOM*, Mar. 2005.

[9] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, and J. Rexford, "NetScope: Traffic engineering for IP networks," *IEEE Network*, vol. 14, no. 2, pp. 11–19, 2000.

[10] A. Shaikh and A. Greenberg, "OSPF Monitoring: Architecture, Design and Deployment Experience," in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[11] J. Gottlieb, A. Greenberg, J. Rexford, and J. Wang, "Automated provisioning of BGP customers," *IEEE Network*, vol. 17, no. 6, 2003.

[12] 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 *USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[13] C. D. L. Higuera, "Current trends in grammatical inference," *Lecture Notes in Computer Science*, vol. 1876, pp. 28–31, 2001.

[14] K. Fisher, D. Walker, K. Q. Zhu, and P. White, "From dirt to shovels: Fully automatic tool generation from ad hoc data," in *POPL*, 2008.