A User-Extensible and Adaptable Parser Architecture

John Tobin and Carl Vogel

Abstract Some parsers need to be very precise and strict when parsing, yet must allow users to easily adapt or extend the parser to parse new inputs, without requiring that the user have an in-depth knowledge and understanding of the parser's internal workings. This paper presents a novel parsing architecture, designed for parsing Postfix log files, that aims to make the process of parsing new inputs as simple as possible, enabling users to trivially add new rules (to parse variants of existing inputs) and relatively easily add new actions (to process a previously unknown category of input). The architecture scales linearly or better as the number of rules and size of input increases, making it suitable for parsing large corpora or months of accumulated data.

1 Introduction

The architecture described herein was developed as part of a larger project to improve anti-spam defences by analysing the performance of the set of filters currently in use, optimising the order and membership of the set based on that analysis, and developing supplemental filters where deficiencies are identified. Most anti-spam techniques are content-based (e.g. [3, 7, 11]) and require the mail to be accepted before determining if it is spam, but rejecting mail during the delivery attempt is preferable: senders of non-spam mail that is mistakenly rejected will receive an immediate non-delivery notice; resource usage is reduced on the accepting mail server (allowing more intensive content-based techniques to be used on the mail that is accepted); users have less spam mail to wade through. Improving the performance of anti-spam techniques that are applied when mail is being transferred via Simple Mail Transfer Protocol (SMTP)[1] is the goal of this project, by providing a platform

---

John Tobin e-mail: tobinjt@cs.tcd.ie · Carl Vogel e-mail: vogel@cs.tcd.ie
School of Computer Science and Statistics, Trinity College, Dublin 2, Ireland.
Supported by Science Foundation Ireland RFP 05/RF/CMS002.

for reasoning about anti-spam filters. The approach chosen to measure performance is to analyse the log files produced by the SMTP server in use, Postfix [13], rather than modifying it to generate statistics: this approach improves the chances of other sites testing and using the software. The need arose for a parser capable of dealing with the great number and variety of log lines produced by Postfix: the parser must be designed so that adding support for parsing new inputs is a simple task, because the log lines to be parsed will change over time. The variety in log lines occurs for several reasons:

- Log lines differ amongst versions of Postfix.
- The mail administrator can define custom rejection messages.
- External resources utilised by Postfix (e.g. DNS Black List (DNSBL) or policy servers [14]) can change their messages without warning.

It was hoped to reuse an existing parser rather than writing one from scratch, but the existing parsers considered were rejected for one or more of the following reasons: they parsed too small a fraction of the log files; their parsing was too inexact; they did not extract sufficient data. The effort required to adapt and improve an existing parser was judged to be greater than the effort to write a new one, because the techniques used by the existing parsers severely limited their potential: some ignored the majority of log lines, parsing specific log lines accurately, but without any provision for parsing new or similar log lines; others sloppily parsed the majority of log lines, but were incapable of distinguishing between log lines of the same category, e.g. rejecting a mail delivery attempt. The only prior published work on the subject of parsing Postfix log files that the authors are aware of is *Log Mail Analyser: Architecture and Practical Utilizations* [4], which aims to extract data from log files, correlate it, and present it in a form suitable for a systems administrator to search using the myriad of standard Unix text processing utilities already available. A full state of the art review is outside the scope of this paper but will be included in the thesis resulting from this work.

The solution developed is conceptually simple: provide a few generic functions (*actions*), each capable of dealing with an entire category of inputs (e.g. rejecting a mail delivery attempt), accompanied by a multitude of precise patterns (*rules*), each of which matches all inputs of a specific type and only that type (e.g. rejection by a specific DNSBL). It is an accepted standard to separate the parsing procedure from the declarative grammar it operates with; part of the novelty here is in the way that the grammar is itself partially procedural (each action is a separate procedure). This architecture is ideally suited to parsing inputs where the input is not fully understood or does not conform to a fixed grammar: the architecture warns about unparsed inputs and other errors, but continues parsing as best it can, allowing the developer of a new parser to decide which deficiencies are most important and require attention first, rather than being forced to fix the first error that arises.

---

[1] Simple Mail Transfer Protocol transfers mail across the Internet from the sender to one or more recipients. It is a simple, human readable, plain text protocol, making it quite easy to test and debug problems with it. The original protocol definition is RFC 821 [10], updated in RFC 2821 [9].

2 Architecture

The architecture is split into three sections: framework, actions and rules. Each will be discussed separately, but first an overview:

Framework	The framework is the structure that actions and rules plug into. It provides the parsing loop, shared data storage, loading and validation of rules, storage of results, and other support functions.
Actions	Each action performs the work required to deal with a single category of inputs, e.g. processing data from rejections.
Rules	The rules are responsible for classifying inputs, specifying the action to invoke and the regex that matches the inputs and extracts data.

For each input the framework tries each rule in turn until it finds a rule that matches the input, then invokes the action specified by that rule.

Decoupling the parsing rules from their associated actions allows new rules to be written and tested without requiring modifications to the parser source code, significantly lowering the barrier to entry for casual users who need to parse new inputs, e.g. part-time systems administrators attempting to combat and reduce spam; it also allows companies to develop user-extensible parsers without divulging their source code. Decoupling the actions from the framework simplifies both framework and actions: the framework provides services to the actions, but does not need to perform any tasks specific to the input being parsed; actions benefit from having services provided by the framework, freeing them to concentrate on the task of accurately and correctly processing the information provided by rules.

Decoupling also creates a clear separation of functionality: rules handle low level details of identifying inputs and extracting data; actions handle the higher level tasks of assembling the required data, dealing with the intricacies of the input being parsed, complications arising, etc.; the framework provides services to actions and manages the parsing process.

Some similarity exists between this architecture and William Wood's Augmented Transition Networks (ATN) [6, 16], used in Computational Linguistics for creating grammars to parse or generate sentences. The resemblance between the two (shown in table 1) is accidental, but it is obvious that the two approaches share a similar division of responsibilities, despite having different semantics.

Table 1 Similarities with ATN

ATN	Parser Architecture	Similarity
Networks	Framework	Determines the sequence of transitions or actions that constitutes a valid input.
Transitions	Actions	Assembles data and imposes conditions the input must meet to be accepted as valid.
Abbreviations	Rules	Responsible for classifying input.

2.1 Framework

The framework takes care of miscellaneous support functions and low level details of parsing, freeing the programmers writing actions to concentrate on writing productive code. It links actions and rules, allowing either to be improved independently of the other. It provides shared storage to pass data between actions, loads and validates rules, manages parsing, invokes actions, tracks how often each rule matches to optimise rule ordering (§3.2 [p. 9]), and stores results in the database. Most parsers will require the same basic functionality from the framework, plus some specialised support functions. The framework is the core of the architecture and is deliberately quite simple: the rules deal with the variation in inputs, and the actions deal with the intricacies and complications encountered when parsing.

The function that finds the rule matching the input and invokes the requested action can be expressed in pseudo-code as:

```
for each input:
    for each rule defined by the user:
        if this rule matches the input:
            perform the action specified by the rule
            skip the remaining rules
            process the next input
    warn the user that the input was not parsed
```

2.2 Actions

Each action is a separate procedure written to deal with a particular category of input, e.g. rejections. The actions are parser-specific: each parser author will need to write the required actions from scratch unless extending an existing parser. It is anticipated that parsers based on this architecture will have a high ratio of rules to actions, with the aim of having simpler rules and clearer distinctions between the inputs parsed by different rules. In the Postfix log parser developed for this project there are 18 actions and 169 rules, with an uneven distribution of rules to actions as shown in fig. 1 on the facing page. Unsurprisingly, the action with the most associated rules is `DELIVERY_REJECTED`, the action that handles Postfix rejecting a mail delivery attempt; it is followed by `SAVE_DATA`, the action responsible for handling informative log lines, supplementing the data gathered from other log lines. The third most common action is, perhaps surprisingly, `UNINTERESTING`: this action does nothing when executed, allowing uninteresting log lines to be parsed without causing any effects (it does not imply that the input is ungrammatical or unparsed). Generally rules specifying the `UNINTERESTING` action parse log lines that are not associated with a specific mail, e.g. notices about configuration files changing. The remaining actions have only one or two associated rules: some actions are required to address a deficiency in the log files, or a complication that

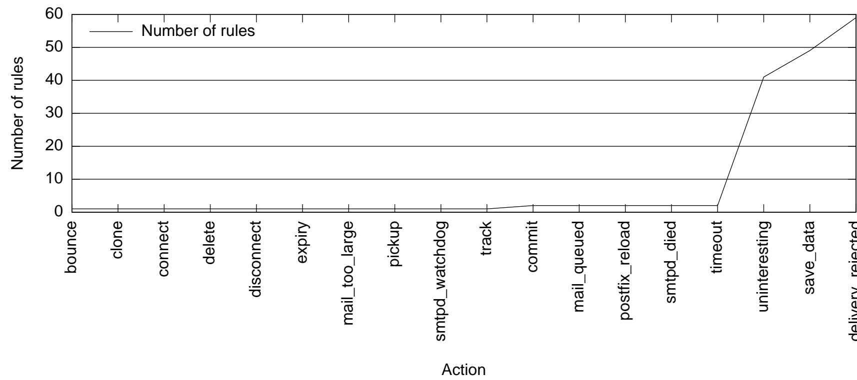

Fig. 1 Distribution of rules per action

arises during parsing; other actions will only ever have one log line variant, e.g. all log lines showing that a remote client has connected are matched by a single rule and handled by the `CONNECT` action.

Using the `CONNECT` action as an example: it creates a new data structure in memory for the new client connection, saving the data extracted by the rule into it; this data will be entered into the database when the mail delivery attempt is complete. If a data structure already exists for the new connection it is treated as a symptom of a bug, and the action issues a warning containing the full contents of the existing data structure, plus the log line that has just been parsed.

The ability to add special purpose actions to deal with difficulties and new requirements that are discovered during parser development is one of the strengths of this architecture. Instead of writing a single monolithic function that must be modified to support new behaviour, with all the attendant risks of adversely affecting the existing parser, when a new requirement arises an independent action can be written to satisfy it. Sometimes the new action will require the cooperation of other actions, e.g. to set or check a flag. There is a possibility of introducing failure when modifying existing actions in this way, but the modifications will be smaller and occur less frequently than with a monolithic architecture, thus failures will be less likely to occur and will be easier to test for and diagnose. The architecture can be implemented in an object oriented style, allowing sub-classes to extend or override actions in addition to adding new actions; because each action is an independent procedure, the sub-class need only modify the action it is overriding, rather than reproducing large chunks of functionality.

During development of the Postfix log parser it became apparent that in addition to the obvious variety in log lines there were many complications to be overcome. Some were the result of deficiencies in Postfix's logging (some of which were rectified by later versions of Postfix); others were due to the vagaries of process scheduling, client behaviour, and administrative actions. All were successfully accommodated in the Postfix log parser: adding new actions was enough to overcome several of the complications; others required modifications to a single existing action to

work around the difficulties; the remainder were resolved by adapting existing actions to cooperate and exchange extra data, changing their behaviour as appropriate based on that extra data.

Actions may return a modified input line that will be parsed as if read from the input stream, allowing for a simplified version of cascaded parsing [1]. This powerful facility allows several rules and actions to parse a single input, potentially simplifying both rules and actions.

2.3 Rules

Rules categorise inputs, specifying both the regex to match against each input and the action to invoke when the match is successful. The Postfix log parser stores the rules it uses in the same SQL database the results are stored in, removing any doubt about which set of rules was used to produce a set of results; other implementations are free to store their rules in whatever fashion suits their needs. The framework warns about every unparsed input, to alert the user that they need to alter or extend their ruleset; the Postfix log parser successfully parses every log line in the 522 log files it is currently being tested with. The framework requires each rule to have `action` and `regex` attributes; each implementation is free to add any additional attributes it requires. The Postfix log parser adds attributes for several reasons: optimising rule ordering (§3.2 [p. 9]); restricting which log lines each rule can be matched against; and to describe each rule. Keywords in the rule's regex specify the data to be extracted from the input, but the Postfix log parser also provides a mechanism for rules to specify extra data to be saved.

Parsing new inputs is generally achieved by creating a new rule that pairs an existing action with a new regex. The Postfix log parser supplies a utility based on Simple Logfile Clustering Tool [12] to aid in producing regexes from unparsed log lines. Decoupling the rules from the actions and framework enables other rule management approaches to be used, e.g. instead of manually adding new rules, machine learning techniques could be used to automatically generate new rules. If this approach was taken the choice of machine learning technique would be constrained by the size of typical data sets (see §3 [p. 8]). Techniques requiring the full data set when training would be impractical; Instance Based Learning [2] techniques that automatically determine which inputs from the training set are valuable and which inputs can be discarded might reduce the data required to a manageable size. A parser might also dynamically create new rules in response to certain inputs, e.g. diagnostic messages indicating the source of the inputs has read a new configuration file. These avenues of research and development has not been pursued by the authors, but could easily be undertaken independently.

The architecture does not try to detect overlapping rules: that responsibility is left to the author of the rules. Unintentionally overlapping rules lead to inconsistent parsing and data extraction because the first matching rule wins, and the order in which rules are tried against each input might change between parser invocations.

Overlapping rules are frequently a requirement, allowing a more specific rule to match some inputs and a more general rule to match the remainder, e.g. separating SMTP delivery to specific sites from SMTP delivery to the rest of the world. Allowing overlapping rules simplifies both the general rule and the more specific rule; additionally rules from different sources can be combined with a minimum of prior cooperation or modification required. Overlapping rules should have a priority attribute to specify their relative ordering; negative priorities may be useful for catchall rules.

Decoupling the rules from the actions allows external tools to be written to detect overlapping rules. Traditional regexes are equivalent in computational power to Finite Automata (FA) and can be converted to FA, so regex overlap can be detected by finding a non-empty intersection of two FA. The standard equation for FA intersection (given for example in [17]) is: $FA1 \cap FA2 = \overline{(\overline{FA1} \cup \overline{FA2})}$, which has considerable computation complexity. Perl 5.10 regexes are more powerful than traditional regexes: it is possible to match correctly balanced brackets nested to an arbitrary depth, e.g. /^[^<>]*(<(?:(?>[^<>]+)|(?1))*>)[^<>]*$/ matches z<123<pq<>rs>j<r>ml>s. Perl 5.10 regexes can maintain an arbitrary state stack and are thus equivalent in computational power to Pushdown Automata (PDA) or Context-Free Languages (CFL), so detecting overlap may require calculating the intersection of two PDA or CFL. The intersection of two CFL is not closed, i.e. the resulting language cannot always be parsed by a CFL, so intersection may be intractable in some cases e.g.: $a^*b^nc^n \ \cap \ a^nb^nc^* \ \rightarrow \ a^nb^nc^n$. Detecting overlap amongst n regexes requires calculating $n(n-1)/2$ intersections, resulting in $O(n^2x)$ complexity, where $O(x)$ is the complexity of calculating intersection. This is certainly not a task to be performed every time the parser is used: detecting overlap amongst the Postfix log parser's 169 rules would require calculating 14196 intersections.

It is possible to define pathological regexes which fall into two main categories: regexes that match every input, and regexes that consume excessive amounts of CPU time during matching. Defining a regex to match all inputs is trivial: /^/ matches the start of every input. Usually excessive CPU time is consumed when a regex with a lot of alteration and variable quantifiers fails to match, but successful matching is generally quite fast (see [5] for in-depth discussion).

The example rule in table 2 matches the following sample log line logged by Postfix when a remote client connects to deliver mail:

```
connect from client.example.com[192.0.2.3]
```

Table 2 Example rule

Attribute	Value
regex	^connect from (__CLIENT_HOSTNAME__)\[(__CLIENT_IP__)\]$
action	CONNECT (described in §2.2 [p. 4])

2.4 Architecture Characteristics

Matching rules against inputs is simple: The first matching rule determines the action that will be invoked: there is no backtracking to try alternate rules, no attempt is made to pick a *best* rule.

Line oriented: The architecture is line oriented at present: there is no facility for rules to consume more input or push unused input back onto the input stream. This was not a deliberate design decision, rather a consequence of the line oriented nature of Postfix log files; more flexible approaches could be pursued.

Context-free rules: Rules can not take into account past or future inputs. In context-free grammar terms the parser rules could be described as:
$<$input$> \mapsto$ rule-1|rule-2|rule-3|…|rule-n.

Context-aware actions: Actions can consult the results (or lack of results) of previous actions during execution, providing some context sensitivity.

Cascaded parsing: Actions can return a modified input to be parsed as if read from the input stream, allowing for a simplified version of cascaded parsing [1].

Transduction: The architecture can be thought of as implementing transduction: it takes data in one form (log files) and transforms it to another form (a database); other formats may be more suitable for other implementations.

Closer to Natural Language Processing than using a fixed grammar: Unlike traditional parsers such as those used when compiling a programming language, this architecture does not require a fixed grammar specification that inputs must adhere to. The architecture is capable of dealing with interleaved inputs, out of order inputs, and ambiguous inputs where heuristics must be applied — all have arisen and been successfully accommodated in the Postfix log parser.

3 Results

Parsing efficiency is an obvious concern when the Postfix log parser routinely needs to parse large log files. The mail server which generated the log files used in testing the Postfix log parser accepts approximately 10,000 mails for 700 users per day; median log file size is 50 MB, containing 285,000 log lines — large scale mail servers would have much larger log files. When generating the timing data used in this section, 93 log files (totaling 10.08 GB, 60.72 million log lines) were each parsed 10 times and the parsing times averaged. Saving results to the database was disabled for the test runs, because the tests are aimed at measuring the speed of the Postfix log parser rather than the speed of the database. The computer used for test runs is a Dell Optiplex 745 described in table 3 on the next page, dedicated to the task of gathering statistics from test runs. Parsing all 93 log files in one run took 2 hours, 19 minutes and 17.293 seconds, mean throughput is 68.994 MB (435,942.882 log lines) parsed per minute; median throughput when parsing files separately was 80.854 MB (480,569.173 log lines) parsed per minute.

Table 3 Computer used to generate statistics

Component	Component in use
CPU	1 dual core 2.40GHz Intel® Core™2 CPU, with 32KB L1 and 4MB L2 cache.
RAM	2GB 667 MHz DDR RAM.
Hard disk	1 Seagate Barracuda 7200 RPM 250GB SATA hard disk.

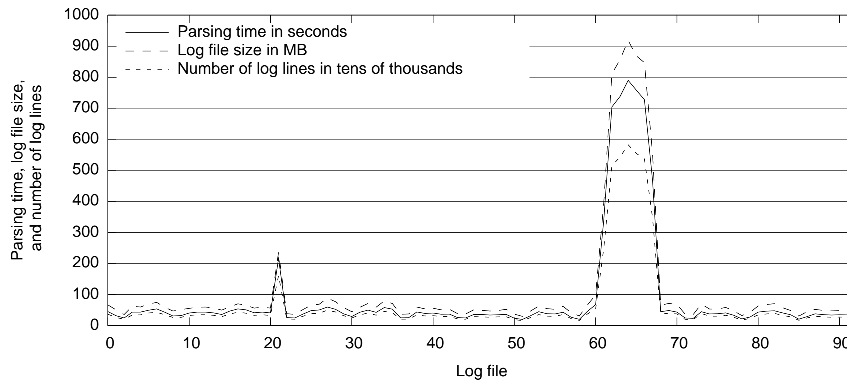

Fig. 2 Parsing time, log file size, and number of log lines

3.1 Architecture Scalability: Input Size

An important property of a parser is how parsing time scales relative to input size: linearly, polynomially, or exponentially? Figure 2 shows the parsing time in seconds, log file size in MB, and number of log lines in tens of thousands, for each of the 93 log files. The lines on the graph run roughly in parallel, giving the impression that the algorithm scales linearly with input size. This impression is borne out by fig. 3 on the next page: the ratios are tightly banded across the graph, showing that the algorithm scales linearly. The ratios increase (i.e. improve) for log files 22 and 62–68 despite their large size; that unusually large size is due to mail forwarding loops resulting in a greatly increased number of mails delivered and log lines generated.

3.2 Rule Ordering

Figure 4 on the following page shows the number of log lines in the 93 log files matched by each of the Postfix log parser's 169 rules. The top ten rules match 85.036% of the log lines, with the remainder tailing off similar to a Power Law distribution. Assuming that the distribution of log lines is reasonably consistent over time, parser efficiency should benefit from trying more frequently matching rules

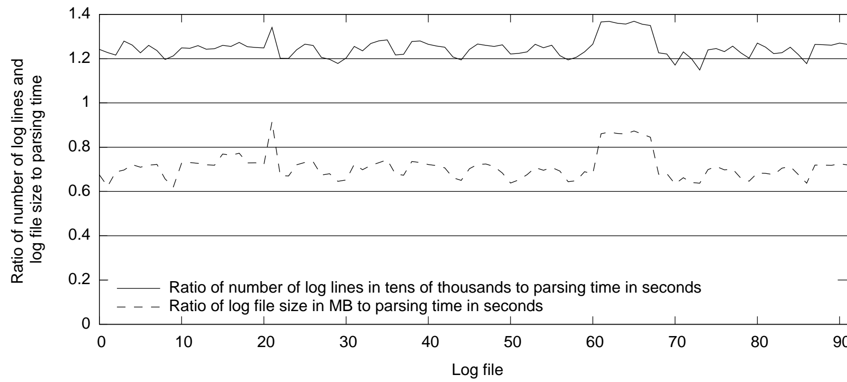

Fig. 3 Ratio of number of log lines and log file size to parsing time

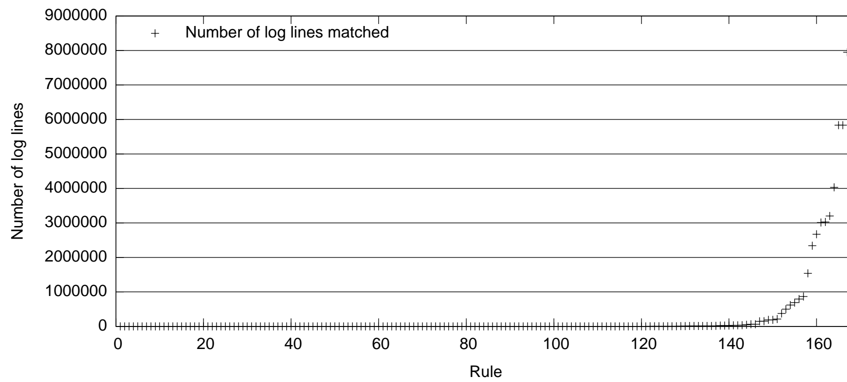

Fig. 4 Number of log lines matched by each rule

before those which match less frequently. To test this hypothesis three full test runs were performed with different rule orderings:

optimal — Hypothetically the best order: rules which match most often will be tried first.

shuffled — Random ordering, intended to represent an unsorted rule set. Note that the ordering will change every time the parser is executed, so 10 different orderings will be generated for each log file in the test run.

reverse — Hypothetically the worst order: the most frequently matching rules will be tried last.

Figure 5 on the next page shows the parsing times of optimal and reverse orderings as a percentage of shuffled ordering parsing time. This optimisation provides a mean reduction in parsing time of 14.785% with normal log files, 5.102% when a mail loop occurs and the distribution of log lines is unusual. Optimal rule ordering has other benefits, described in §3.3 on the facing page.

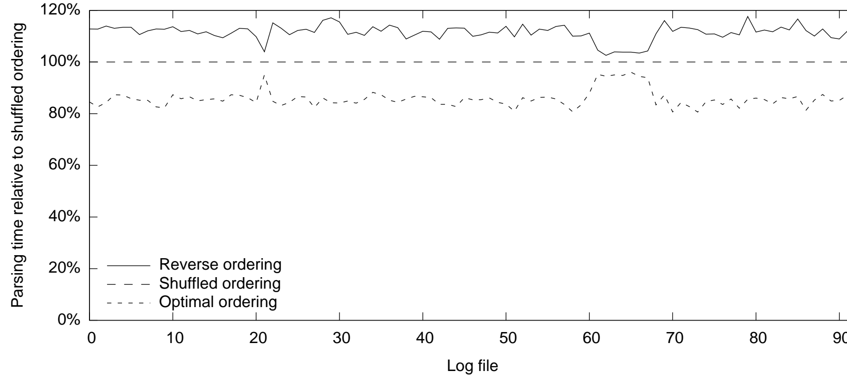

Fig. 5 Parsing time relative to shuffled ordering

3.3 Architecture Scalability: Number of Rules

How any architecture scales as the number of rules increases is important, but it is particularly important for this architecture because it is expected that typical parsers will have a large number of rules. There are 169 rules in the full Postfix log parser ruleset (parsing 522 log files), but the minimum number of rules required to parse the 93 log files is 115, 68.04% of the full ruleset. A second set of test runs was performed using the minimum ruleset, and the parsing times compared to those generated using the full ruleset: the percentage parsing time increase when using the full ruleset instead of the minimal ruleset for optimal, shuffled and reverse orderings is shown in fig. 6 on the next page. Clearly the increased number of rules has a noticeable performance impact with reverse ordering, and a lesser impact with shuffled ordering. The optimal ordering shows a mean increase of 0.63% in parsing time for a 46.95% increase in the number of rules. These results show that the architecture scales extremely well as the number of rules increases, and that optimally ordering the rules enables this.

3.4 Coverage

The Postfix log parser has two different types of coverage to be measured: log lines correctly parsed, and mail delivery attempts correctly understood (the former is a requirement for the latter to be achieved). Improving the former is less difficult, as usually it just requires new rules to be written; improving the latter is more difficult and intrusive as it requires adding or changing actions, and it can be much harder to notice that a deficiency exists.

Correct parsing of log lines must be measured first. Warnings are issued for any log lines that are not parsed; no such warnings are issued while parsing the 93 log

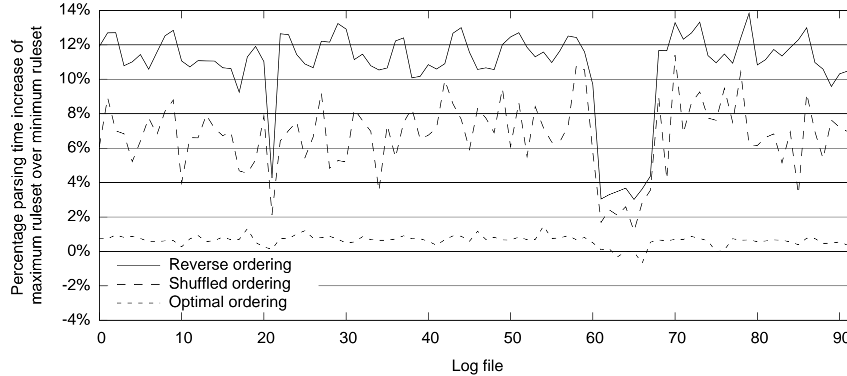

Fig. 6 Percentage parsing time increase of maximum ruleset over minimum ruleset

files, therefore there are zero false negatives. False positives are harder to quantify: manually verifying that the correct rule parsed each of the 60,721,709 log lines is infeasible. A random sample of 6,039 log lines (0.00994% of 60,721,709) was parsed and the results manually verified by inspection to ensure that the correct rule's regex matched each log line. The sample results contained zero false positives, and this check has been automated to ensure continued accuracy. The authors are confident that zero false positives occur when parsing the 93 log files.

The proportion of mail delivery attempts correctly understood is much more difficult to determine accurately than the proportion of log lines correctly parsed. The implementation can dump its state tables in a human readable form; examining these tables with reference to the log files and database is the best way to detect misunderstood mail delivery attempts. The Postfix log parser issues warnings when it detects any errors or discrepancies, alerting the user to the problem. There should be few or no warnings during parsing, and when parsing is finished the state table should only contain entries for mail delivery attempts starting before or ending after the log file. A second sample of 6000 log lines was parsed with all debugging options enabled, resulting in 167,448 lines of output. All 167,448 lines were examined in conjunction with the log segment and a dump of the resulting database, verifying that for each of the log lines the Postfix log parser performed correctly. The implementation produced 4 warnings about deficiencies in the log segment, 10 mails correctly remaining in the state tables, and 1625 correct entries in the database: it produced 0 false positives. No error or warning messages were produced, therefore there were no false negatives. Given the evidence detailed above, the authors are confident that zero false positives or negatives occur when parsing the 93 log files.

Experience implementing the Postfix log parser shows that full input coverage is relatively easy to achieve with this architecture, and that with enough time and effort full understanding of the input is possible. Postfix log files would require substantial time and effort to correctly parse regardless of the architecture used; this architecture enables an iterative approach to be used (similar to Stepwise Refinement [15]), as is practiced in many other software engineering disciplines.

4 Conclusion

This architecture's greatest strength is the ease with which it can be adapted to deal with new requirements and inputs. Parsing a variation of an existing input is a trivial task: simply modify an existing rule or add a new rule with an appropriate regex and the task is complete. Parsing a new category of input is achieved by writing a new action and appropriate rules; quite often the new action will not need to interact with existing actions, but when interaction is required the framework provides the necessary facilities. The decoupling of rules from actions allows different sets of rules to be used with the same actions, e.g. a parser might have actions to process versions one and two of a file format; by choosing the appropriate ruleset the parser will parse version one, or version two, or both versions. Decoupling also allows other approaches to rule management, as discussed in §2.3 [p. 6]. The architecture makes it possible to apply commonly used programming techniques (such as object orientation, inheritance, composition, delegation, roles, modularisation, or closures) when designing and implementing a parser, simplifying the process of working within a team or when developing and testing additional functionality. This architecture is ideally suited to parsing inputs where the input is not fully understood or does not follow a fixed grammar: the architecture warns about unparsed inputs and other errors, but continues parsing as best it can, allowing the developer of a new parser to decide which deficiencies are most important and require attention first, rather than being forced to fix the first error that arises.

The data gathered by the Postfix log parser provides the foundation for the future of this project: applying machine-learning algorithms to the data to analyse and optimise the set of anti-spam defences in use, followed by identifying patterns in the data that could be used to write new filters to recognise and reject spam rather than accepting it. The parser provides the data in a normalised form that is far easier to use as input to new or existing algorithm implementations than trying to adapt each algorithm to extract data directly from the log files. The current focus is on clustering and decision trees to optimise the order in which rules are applied; future efforts will involve using data gathered by the parser to train and test new filters. This task is similar to analysing a black box application based on its inputs and outputs, and this approach could be applied to analyse the behaviour of any system given sufficient log messages to parse. An alternate approach to black box optimisation that uses application profiling in conjunction with the application's error messages to improve the error messages shown to users is described in [8]; profiling data may be useful in supplementing systems that fail to provide adequate log messages.

The Postfix log file parser based on this architecture provides a basis for systems administrators to monitor the effectiveness of their anti-spam measures and adapt their defences to combat new techniques used by those sending spam. This parser is a fully usable application, built to address a genuine need, rather than a proof of concept whose sole purpose is to illustrate a new idea; it deals with the oddities and difficulties that occur in the real world, rather than a clean, idealised scenario developed to showcase the best features of a new approach.

References

1. Abney, S.: Partial parsing via finite-state cascades. Nat. Lang. Eng. **2**(4), 337–344 (1996). DOI http://dx.doi.org/10.1017/S1351324997001599. URL http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=974705. Last checked 2008/08/20
2. Aha, D.W., Kibler, D., Albert, M.K.: Instance-based learning algorithms. Mach. Learn. **6**(1), 37–66 (1991). DOI http://dx.doi.org/10.1023/A:1022689900470. URL http://portal.acm.org/citation.cfm?id=104717. Last checked 2008/07/29
3. Ahmed, S., Mithun, F.: Word stemming to enhance spam filtering. First Conference on Email and Anti-Spam CEAS 2004 (2004). URL http://www.ceas.cc/papers-2004/167.pdf. Last checked 2008/08/20
4. Aiello, M., Avanzini, D., Chiarella, D., Papaleo, G.: Log mail analyzer: Architecture and practical utilizations. Trans-European Research and Education Networking Association (2006). URL http://www.terena.nl/events/tnc2006/core/getfile.php?file_id=770. Last checked 2008/08/20
5. Friedl, J.E.F.: Crafting an efficient expression. Mastering regular expressions pp. 185–222 (2006). ISBN-10: 0596528124
6. Gazdar, G., Mellish, C.: Natural language processing in prolog. An Introduction to Computational Linguistics pp. 63–98 (1989). ISBN-10: 0201180537
7. Graham, P.: A plan for spam. Hackers & Painters pp. 121–129 (2004). ISBN-10: 0–596–00662–4
8. Ha, J., Rossbach, C.J., Davis, J.V., Roy, I., Ramadan, H.E., Porter, D.E., Chen, D.L., Witchel, E.: Improved error reporting for software that uses black-box components. SIGPLAN Not. **42**(6), 101–111 (2007). DOI http://doi.acm.org/10.1145/1273442.1250747. URL http://portal.acm.org/citation.cfm?id=1250747. Last checked 2008/08/20
9. Klensin, J.C.: Rfc 2821 — simple mail transfer protocol. The Internet Society Requests for Comment (2001). URL http://www.faqs.org/rfcs/rfc2821.html. Last checked 2008/08/20
10. Postel, J.B.: Rfc 821 — simple mail transfer protocol. The Internet Society Requests for Comment (1982). URL http://www.faqs.org/rfcs/rfc821.html. Last checked 2008/08/20
11. Sculley, D., Wachman, G.M.: Relaxed online svms in the trec spam filtering track. Text REtrieval Conference (TREC) (2007). URL http://trec.nist.gov/pubs/trec16/papers/tuftsu.spam.final.pdf. Last checked 2008/08/20
12. Vaarandi, R.: A data clustering algorithm for mining patterns from event logs. IP Operations and Management, 2003. (IPOM 2003). 3rd IEEE Workshop on pp. 119–126 (2003). URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1251233. Last checked 2008/08/20
13. Venema, W.: Postfix home page. Postfix documentation (2008). URL http://www.postfix.org/. Last checked 2008/08/20
14. Venema, W.: Postfix smtp access policy delegation. Postfix documentation (2008). URL http://www.postfix.org/SMTPD_POLICY_README.html. Last checked 2008/08/20
15. Wirth, N.: Program development by stepwise refinement. Commun. ACM **14**(4), 221–227 (1971). DOI http://doi.acm.org/10.1145/362575.362577. URL http://portal.acm.org/citation.cfm?doid=362575.362577. Last checked 2008/08/20
16. Woods, W.A.: Transition network grammars for natural language analysis. Commun. ACM **13**(10), 591–606 (1970). URL http://portal.acm.org/citation.cfm?id=362773. Last checked 2008/08/20
17. Zafar, N.A., Sabir, N., Ali, A.: Construction of intersection of nondeterministic finite automata using z notation. Proceedings of World Academy of Science, Engineering and Technology **30**, 591–596 (2008)