Argus

Functional Specification
Draft: For Review Purposes Only
28 June 1989 at 11:42

Wendy E. Mackay
Brian R. Gardner
Toben H. Mintz
Richard A. Pito
Jeffrey B. Siegal

1. Introduction

The original inspiration for Argus was the Information Lens, developed under the supervision of Professor Thomas Malone at the MIT Sloan School of Management. Lens was written in InterLISP-D and designed to work with the Lafite mail system on Xerox LISP workstations.

Argus was created for MIT's Project Athena, a Unix environment, and is written in C and X Windows. The design was based on the results a 2-year user study of the original Information Lens, conducted by Mackay, and a significantly revised version of Lens, called Common Lens, that was designed by Ramana Rao at Xerox PARC. The Argus programmers did not see either the Information Lens or Common Lens. Features considered successful in the original versions of Lens were kept in Argus. Less successful features were omitted or changed, and Argus has several concepts that differ significantly from the original work.

Argus, V.1 was written during the summer of 1988. A working prototype was delivered in September, 1988. During the Fall, three of the programmers continued working (part-time) on the project, reorganizing the code to increase maintainability and providing user support. In the Spring, Argus, V.2 was released and a user study was initiated with Project Athena staff members. New functionality included the Rule Server[1] and the Message Type Editor. The plan for the summer of 1989 is to convert Argus to Motif (the Open Software Foundation's version of the X Window System), incorporating new ideas and results from the user study. We also hope to provide at least one specially-tailored application for Project Athena. In September, 1989, the code, documentation, technical specification, and a demonstration will be presented to Digital, the Open Software Foundation and Project Athena.

Argus is currently available in test release form at Project Athena on VAXstation-II's, VAXstation-2000's, and IBM PC/RT's. In addition, components have been successfully tested on the IBM PC/AT. Please refer to the References section for additional information about the Information Lens and Project Athena.

Please note that this is a working document. Some sections (the Rule Server section in particular) have not yet been added. The user interface will be significantly revised in the conversion to Motif and some of the underlying structures will change as well. Thus, some sections of this document will change significantly over the next few months.

2

[1]A generalized version of the "anyone" server associated with the Information Lens.

[2]The first year of funding for Argus was provided by Digital's BOSE group. Tom Malone also contributed 10 weeks of funding, based on funding not used by Mackay, for a summer student. The Open Software Foundation is providing funding for the summer of 1989.

This project is not intended to become a product for Digital, Project Athena or the Open Software Foundation. Digital has full access to the code and design process and may choose to create products based on this effort. The Open Software Foundation may provide Argus to its members. Project Athena has full access to Argus for internal use. MIT's Committee on the Use of Humans as Experimental Subjects has carefully reviewed and approved the user study.

2. The Capabilities of Argus

Electronic mail messages contain a set of structured fields (the header) and unstructured sections of text (the body). One of the key ideas in the Information Lens was that users should be able to take advantage of some of the inherent structure in a mail message. Argus is able to compare the contents of fields in actual messages with fields in a user-defined rule and perform specified actions. While Argus can also compare words in the body of the message, most of the power stems from ability to take advantage of the limited domain in a field.

The Information Lens was designed to handle electronic mail specifically. Argus has been designed to manipulate any kind of semi-structured message, including mail, notes files, and databases. Argus uses a generic interface, called a folder, which converts messages into a form that Argus can handle. New formats can be added by adding a new "folder object". Argus currently supports two folders: an interface to the mh mail handler and an interface to Emacs' RMAIL (Babble format mail). Project Athena's staff has requested folders to handle Zephyr messages, OLC messages, network traffic messages, and system software maintenance messages.

2.1. Why Not Just Use a Database Program?

Database programs categorize and index large amounts of information for quick retrieval. This approach is useful in situations in which the form and lifetime of the data is relatively stable and the information is fully structured and known in advance. In situations in which a stream of information needs to be processed, but not necessarily saved, another solution is necessary.[3] The Information Lens was designed to process a stream of information and filter it according to a set of rules. The data is "semi-structured", in that messages can contain both structured and unstructured sections. A system of "message types", in which additional fields are pre-specified by an administrative organization, allows users to choose different kinds of messages.

Argus provides a greater level of freedom and a broader definition of "semi-structured" messages. Senders of messages can add individual fields to messages, independent of the message type. They can also change their minds about which message type is being created at any point during the composition process. Recipients of messages can write rules based on either the fields, the formally-defined message types or both. In addition, arbitrary groups of users can create and use private message types, in addition to formally sanctioned ones. This flexibility was added to address concerns expressed by users of the original Lens. Please refer to ***Semi-Structure in Messages*** Mackay, 1988, for further discussion of these issues.

Both Lens and Argus allow users to create simple, graphically-based filters for incoming messages. In both, users create filtering rules by filling in the blanks of a message and specifying a corresponding action. Argus permits users to choose fields of different types (matching of text, date, booleans, and numeric arguments). Argus also goes beyond the actions normally associated with handling mail (move,

---

[3] An example of such a high capacity stream of information would be the monitoring of automated network statistics coming from workstations, servers, and other devices on a network. Another applications is the dynamic monitoring of source trees, a time-consuming part of Project Athena's release engineering process.

copy, delete, forward), and provides a full set of actions, including the ability to run any arbitrary Unix program. Additional actions include the ability to run other sets of rules (or stop processing of those rules) and to create or modify user-defined properties of a message. Any rule can run additional rules (or stop them).

Argus processes messages sequentially. Thus, given a stream of messages in a folder and a set of rules, Argus will begin with the first message, apply each rule in turn, and then go to the next message, until each of the messages has been processed.

Argus rules contain fields that are grouped into boxes and a list of actions to be performed if the rule's predicate matches a message. The value of the field in the rule is compared to the value of the corresponding field in a message and evaluated as either true or false. Groups of fields can be evaluated together, using the set operators 'and', 'or' and 'not'. (For example, a rule might check if the TO: field contains "Smith" and if the DATE: field is less than yesterday.)

The original Information Lens provided a single ruleset, to be run prior to the reading of mail. Argus allows users to create a variety of different rulesets, for different purposes, and to run them at different times. Thus, a user might have an everyday ruleset to be run prior to reading mail, an "after-vacation" ruleset to be run upon return from an extended time away, and a separate ruleset for cleaning up old files and organizing them for future reference. Argus allows users to nest rulesets and rulesets can be run from other Unix applications.

Note: A table should be added that lists the differences
between Information Lens, Common Lens, and Argus

2.2. Abilities, Limitations and Extensions

Argus rules contain an *IF* (predicate) and a *THEN* (action). Technical aspects of predicates and actions are described in section RULELANGUAGE. This section describes the general capabilities and constraints.

2.2.1. Overview of Functionality

Argus allows a user to categorize a message and perform actions based on its contents. Rules are organized into sets called rule sets. Data is organized into streams called folders. Argus is designed to support extensions to both the sources of information (folders), categorization of that information (predicates), and actions to manipulate the information (actions). Argus's extensibility is one of its greatest strengths.

Information enters Argus through a generic interface object called a folder. In order to extend Argus to manipulate information from a new source, a new folder type object must be written. The folder object gives the internals of Argus a generic data type. This generic data type is simply called a message. This data type is a frame with one optional distinct field called 'body'. The body field refers to the unstructured part of that piece of information. For email this would simply refer to the body of a message. Although it may not be the case that a type of information has only one unstructured field, Argus assumes that if there is at least one it is called the 'body'.

The members of the frame that is a message are called fields. Although the folder object itself does not ascribe types to these fields, the fields will be typed and classed by the rule language. The class of a field refers to where the field came from. Their are 4 field classes: **message**, **user property**, **folder specific**, and **computed**. Fields of class message come from the message explicitly, i.e. the 'to:' field of an email message. Fields of class user property are fields created by the user and attached to the message. Field of folder specific class come from the folder type object itself. For instance, several mailer formats keep track of weather a message has been seen or not, forwarded, or replied to. Fields of class computed are computed about the message by the folder object. Computed fields may also be referred to as **states**. Some states are 'number of lines in body', 'are their unprintable characters in the body?', or 'has this message been moved yet?'. All these fields are available to the rest of Argus. The user property fields may be explicitly altered by the actions of a rule, the rest cannot.

The information supplied by the folder object is fed to a module called ***runrules***. ***Runrules*** takes a source of information (a folder) and a set of rules to apply that source of information (a rule set). ***Runrules*** will evaluate each rule on each message in sequence. More explicitly, each message in turn is processed by each rule in the ruleset.

The predicate of a rule has fields which mimic the structure of a message. These fields are named, classed, typed, and have an operator and a value. The name of a field refers to the name of the field in the frame supplied by the folder object. The class of the field determines the source of that field as described above. The type of a field determines how the values in the message and rule fields are to be interpreted. The type of the field is called the *field type*. Examples of some field types include 'date', 'integer', 'boolean', and 'any case text'. The operator determines how the value of the field in the rule is compared to the value of the field in the message. Examples of some operators for the Integer field type are '=', '>', and '<'.

Field types can be created by a user and linked into Argus much like folder types. A field type has a parser and set of *operators* associated with it. When a field in a rule is to be compared to a field in a message, both fields are first parsed by the field types parser. The output of both parsers is then passed to the field operator for comparison. The operator returns TRUE or FALSE and the comparison of the rule's predicate and the message continues until either the predicate must be true (by the rules of logic) or the predicate has not evaluated to false and all the fields have been tested in which case we say the rule fires.

When a rule fires all the actions in the THEN part of the rule are executed in turn. We say that an action is a verb with arguments. So the action "move-to garbage" has a verb 'move-to' and one argument 'garbage'. Like folder types and field types, verbs too can be created by a user and linked into Argus.

The predicates of a rule may group fields by connectives such as 'AND' and 'OR'. These 'boxes' may be nested. So for example, a rule may group the 'to:' and 'cc:' fields inside an OR box and have that OR box and a 'date:' field inside an AND box. There are three types if boxes: boolean, inter-field, and user defined. The boolean boxes are those described above like 'AND'. Inter-field boxes operate among fields and include such things as '='. This is a useful way to check, for instance, if the contents of the 'to:' field is equal to the contents of the 'from:' field without having to actually specify what those contents are. The third type of field is user defined. The protocol used to create user defined fields has not yet been defined but the general idea is that a user defined function could be passed information from the message and rule and make a decision about their equivalence in some way. User defined fields could also be used to communicate information to user defined applications. None of these boxes has actually been implemented. The boolean boxes are currently being implemented and should be completed soon. The implementation dates of the other two kinds of boxes is still undetermined.

2.2.2. Issues in Design

This subsection presents the major issues in the current design and future extensions to the design. Each topic or problem is presented below with possible/planned solutions.

MAJOR EXTENSIONS AND SYSTEM DESIGN ISSUES

- Each rule in a rule set has access to each message in turn. Therefore, the only information available to the predicate of a rule is the information available from the current message. There is no inter-rule communication. This presents problems when one wishes to compare two different messages like in a sorting algorithm. To address this problem, the Argus team is proposing adding variables associated with a rule set. This would allow a rule to set a 'global' variable to be used by another rule in the same rule set.
- Argus does not yet support the ability to access the relative position of a message in a stream. This would be useful for cleanup operations when the last message in a stream is being processed. This would also be useful in order to keep track of the number of messages which have been processed so far. This would be useful, for example, to keep track of the amount of storage space used so far. The general problem of accessing the relative position of a message in a stream leads us to consider the possibilities of pre and post processing of rule sets/folders. For example, when filtering has been completed on a folder the user may then want to send the contents of the folder to a printer for a hardcopy. The Argus team has listed the addition of pre and post processing actions as a future addition to the rule set editor.
- There is still no robust mechanism to handle message types within and across organizational boundaries. A message typing scheme, though, does exist but does not address these issues and is designed to allow an individual to keep track of outgoing message types only. The outgoing message types can be created by the user or accessed from a set of message types endorsed by the users local organization. Although the Argus team has spent considerable time discussing solutions to the intra/inter-organizational message typing problem, no acceptable solution has been developed. Because this issue is so complex it will not be discussed here. The reader is referred to the many publications on the subject.

USEFUL EXTENSIONS

- When a rule set is run on a folder the user does not have an indication of the actions taken by that rule set. That is, the user does not currently receive a summary of actions telling her that, for example, 10 messages were moved to folder x. Summarizing the actions taken by rules is seen as very important to the current users of Argus and this extension is high on our list of priorities.
- The predicates of Argus do not allow the comparison of fields to other fields. That is, the user cannot create a predicate which determines if the CC: field is equivalent to the FROM field. This problem has been addressed by the Argus team as the concept of the inter-field operators (described above). There are no current plans on implementing this.

ENVIRONMENT DEPENDENCY ISSUES

- Since the Argus Rule Server is capable of sending mail messages in rapid succession, it was important to avoid overloading the Athena mail. This is an important scaling issue: the ability to automatically send copies of messages may cause both space and network problems.

Refer to the Pygmalion specification for a proposed solution to this problem, based on sending pointers rather than mail messages.

2.2.3. Proposed Applications/Extensions

This section outlines some of the proposed applications for Argus. These proposals have originated from a wide variety of sources.

- **Network Management**: It has been proposed to extend Argus to filter automated messages coming from workstations in the Athena Environment. These messages contain information about the state of a workstation and are used to monitor and prevent hardware/software breakdowns. The Athena staff would use Argus to filter these messages to summary programs based on the contents of them. The summary programs would keep track of the changing states of the machines and signal when their is a breakdown, malfunction, or a possibility of breakdown. To implement this extension a new folder type object would have to be created as well as the summary programs. New field types may have to be created as well.

- **System Software Management**: As the size of the source code directories at Athena grows (they are currently over 800 meg in size) maintaining the sources becomes more and more difficult. It has been proposed to extend Argus so that it could traverse the source trees in search of situations which should be brought to the attention of a systems administrator. Such situations would include files with incorrect protections or out of date binaries. Although other anomalys which Argus would search for are complex combinations of events (not worth diving into here), Argus could handle all of them. To implement this extension a new folder type object would have to be created as well as new field types.

- **On Line Teaching Assistant (OLTA)**: The OLTA would be an extension of the Argus rule server. The OLTA would keep a database of answers to questions most frequently asked about a particular subject (or in a particular class). Questions and answers could be posted to the rule server as messages. TAs would search for questions and then post answers. These answers would consist of a header with fields describing the information contained in the body. A user would create a set of rules describing what information he/she needed. For example, if the user were in a neuroanatomy class, she might create a rule which looked like this:

```
brain-area:       corpus collosum
subject-area:     histology
tissue-type:      meninge
information-type: text
```

Note that the OLTA is an excellent application for multi-media messages which could incorporate not only still images but video and sound as well. For example, in the above request the student could have requested an actual color picture of the cells of the meninge surrounding the corpus collosum. To implement the OLTA the rule server must be made more robust, a database program must be incorporated into lens to maintain the large databases one would expect, and a new folder object must be written to interface with the database.

3. The Design of Argus

This chapter discusses the design of the existing Argus prototype at the level of implementation modules.

3.1. Benefits of Object-Oriented Programming

The design of Argus benefits from an Object-Oriented Programming methodology implemented within Draft Proposed ANSI C [details described elsewhere]. Because Argus is built from a number of well-defined Objects, it was easy to include an extension mechanism for supporting new environments and applications.

These Object-Oriented Programming techniques greatly reduced the complexity of the development task by allowing the components to be constructed and tested separately. Several "building-block" objects were made available to all the implementors, simplifying their work.

3.2. Argus Organization

At the highest level, the Argus implementation can be viewed as having three major components: the Message object, the rule system, and the folder interface. User interaction with each of these components is supported by an X-Toolkit application (the "message editor" is under development).

The Message object manages the actual information examined and processed by Argus. It provides for access to a typed, structured component of each "message" and an untyped, unstructured component. When used in support of traditional electronic mail systems, the structured component represents the "headers", while the unstructured component represents the "body".

Several Argus-specific data elements are attached to each "message". These include user-properties (named variables containing arbitrary string or numeric contents), states (status flags), and folder-specific values (variables whose names and contents depend on the message context).

The Rule System supports multiple named rule sets. Each rule set contains rules, each of which has a predicate and one-or-more actions. Predicates support typed operations on structured message components. The operations permitted depend on the type, and are supported by a separate FieldType object. Arbitrary combinations using AND, OR, and NOT are supported. Actions include "delete", "move", "set-property", etc.

The Rule System provides an external representation for rule sets and rules.

The Folder Interface links Argus to a mail system or other source of information. Several different types of folders can be used simultaneously. The details of input/output from/to the folder are hidden by the Folder Interface, which always supplies and accepts Message objects.

Additional support for almost any mail file format or simple database can be added transparently to the rest of Argus. Currently, only sequential access is permitted, although design work is being done to support direct access.

3.3. General Purpose Support Objects

A number of objects were developed which have a wide range of uses within Argus, and would probably be useful outside Argus as well. These are the Memory, Darray, Registry, and Biregistry objects and perhaps a few others of lesser importance.

The Memory object provides for dynamic memory allocation very similar to the standard C routines malloc, realloc, and free. However, the Memory object supports optional pointer checking and storage leak tracking. When enabled, these additional features assist in debugging these extremely common software defects.

The Darray object supports a dynamically sized array containing arbitrary objects. This object is used in dozens of different ways, often encapsulated within a more specific type of object (e.g. the Rule Set object uses a Darray to store the rules within the rule set).

The most important use of the Darray object is as an interface convention between other objects. Wherever a variable number of objects is passed around, the Darray object stores the number of objects being passed, and it manages the storage.

The Registry object supports one-way mappings--it stores keys and values so that when later given the key, it returns the value. It is almost always used within another, special-purpose, object to simplify its implementation.

Most often the Registry is used to map names to values (e.g. field names within the Message object, field type names to FieldType objects within the Field Type Registry), but it can also be used for other purposes like mapping X-Toolkit widgets to other objects.

Two important implementation details of the Registry object are that it does not maintain any order of its contents and that it does not permit multiple entries with matching "key" components.

The Biregistry is an extension of the Registry object which supports using both components as keys or values. Given either component of an entry, it returns the other. Accordingly, multiple entries with either component matching are not permitted.

3.4. Argus Support Objects

A number of support objects are used widely within Argus. In general, they establish a common representation for an often or widely used item or set of items. The most important of these are the FieldType, FieldCore objects and the Lens Profile.

The FieldType object supports the typed data elements (structured component) in the Message object. Each FieldType specifies which operators the rule system may apply to data elements of that type, and includes implementation routines for each operator.

The range of FieldTypes is extensible. A document is a available which describes the conventions used to define a FieldType. Once defined, a FieldType provides additional operators for the rule system.

The FieldCore object stores typed data elements both within the Message object and within the rule system. A FieldCore object is an instance of a FieldType, to which the operators provided by the FieldType can be applied.

The Lens Profile supports user configuration in an environment-independent way. Rather than having the overall Argus design depend of a configuration-file, command line options, or environment variables, the Lens Profile supports a common interface to Argus options. Wherever the value of a customization variable is needed within Argus (e.g. the location of rule sets, the default mail system type, etc.), the Lens Profile provides the variable.

By replacing or modifying the Object which implements the Lens Profile, different methods of supporting user customization could be supported throughout Argus. For example, a control-panel interface to the Lens Profile is being considered.

3.5. Feature Objects

Several Features which are necessary or useful for an Information Lens are provided by several objects which do not fit in any convenient category. These are Message Types, Message Type Registries, Folder Types, Verbs, and Actions.

Message Types are forms or templates with the variation that in the Argus, the user can add or omit fields from the form. They are used to simplify the creation of messages (of the given MessageType) and rules (intended to process messages of the given type). Examples of important Message Types are "Meeting Announcement" and "Phone Message."

To support cooperative use of message types between members of groups a message type registry (MTrgy) object is provided. This object allows users to subscribe to a list of message types (Message Type Registries), giving them shared access to the message types in the list. Currently, these lists are stored in files, but other implementations are possible (e.g. a network server).

Folder Types support using Argus to access mail files or, in general, simple databases. The access method is sequential, but design work is underway to add direct access.

A document is available which describes the conventions used to define a Folder Type. Folder Types implemented in the Argus test release are for mh, xmh, and Emacs' RMAIL. Also, applications which exploit the ability of Argus to access databases are being considered.

Verbs allow a rule to move, copy, or delete messages; invoke rule sets; set or modify user properties; etc. Verb can be added to Argus similar to the way Field Types and Folder Types can be added.

Actions package a verb with its arguments. They, in turn, are packaged inside rules.

3.6. The Argus Core

The easiest way to describe the core Argus components (the Message object, the rule system, and the folder interface) is by describing how they function together.

When the user invokes a rule set on a folder, the rule system locates the rule set from the rule set name provided by the user using the Rule Set Registry. Similarly, the folder system locates the folder from the folder name.

Then the rule system gets the first Message object from the folder using the folder interface and the gets first rule from the rule set using the Rule Set object. The rule system then tests the rule against the message.

The rule testing process involves retrieving FieldCore objects from the Message object and the Rule object. These FieldCore objects are compared using the appropriate operator from the FieldType object.

If the rule fires, the Action objects contained in the rule are executed.

After a rule is tested against a message, regardless of whether it fires, the next rule from the rule set is attempted. This stepping through the rules is performed by the rule system.

After all the rules have been tested, the next Message object from the folder is processed in the same way.

Destructive operations such as "delete" are deferred until all the rules have been tested. Thus, each rule in a rule set has access to each message yet the order of rules in a rule set does matter. Each rule in a rule set has access to the original message excluding user and folder-specific properties. When user and folder specific properties are set by a rule, succeeding rules in a rule set may refer to these properties of the message. In this sense, the order of rules in a rule set is significant.

4. The Implementation of Argus

This chapter provides a programmer's view of Argus. The audience consists of the members of the Argus group, Digital engineers and the Project Athena systems development staff. It is very much a working document, based on the implementation of the existing prototype. It is designed to identify what's been done and why and explain how things are organized. Another goal is to help Digital with technology transfer, by specifying both what worked and what the major problems have been.[4] We describe the basic structure of Argus and we attempt to point out the important concepts.

4.1. Environment

Argus is written in Draft Proposed ANSI C on the Unix operating system, using the X Toolkit and the Athena and Hewlett Packard widget sets. Current work is involved in porting Argus to use OSF's Motif toolkit and widgets.

Argus currently accommodates three mail handlers: mh, xmh and RMAIL.

4.2. Modules, Objects and Resources

The early design discussions created three categories of work: modules, objects, and resources. Modules include the various Athena Lens editors (rule editor, rule set editor, message type editor, etc.), the rule runner command, parsers, and data collection. Objects are handled by modules and include messages, rules, rule sets, message types etc. Resources are used by everyone and include registries, folders, files, message syntax, and user interface guidelines.

The rule editor, rule runner, registries for verbs, simple message types, fields, message objects, the rule language, predicates and actions were all identified as top priority. Extensibility was an important part of the design. Thus, many extra functions can be added easily, even though they were not included in version 1.0 of the code.

Several of the more frequently used objects are described below. More in depth explanations can be found in the appendix.

- **Registries**: Argus uses registries to keep track of different objects and provide a consistent way to refer to and store them. Registries can be used in a hierarchical way, e.g. the message type registry. Registries are usually passed a key and return a value. Biregistry are similar, except they can also be passed a value and return a key. The is useful when a program retrieves a value from a registry, operates on it, and returns it to the registry.

- **Darrays**: Darrays are dynamic arrays which provide an easy way to keep track of a list of things. Darrays provide useful data access capabilities and storage that everyone can use. Individuals did not have to reinvent tools for these tasks for their own internal code and were able to save time and increase overall efficiency. A more difficult problem, which has not been entirely solved, has been to create conventions for deciding which modules create, modify and destroy shared blocks of memory. These cross-module conventions are needed to allow these individual program components to work together.

[4] The User's Guide focuses on how users use Argus to accomplish various tasks.

4.3. Error Handling Proposal

Athena Lens should use a global "errcode" variable and a global "errstring" variable. When a function encounters an error it should set the "errcode" to a globally known error-code which is defined in a header file)]. It should then optionally set the "errstring" to something that can be displayed to the user or printed to a errlog file, or just for debugging. If the "errstring" is not set, there should be a default mapping of error-codes to error-strings.

The function should then return an error indicator to the caller. The caller should then judge if it can take any action itself (such as retrying the failed call in a different manner), or it can send the error upward to it's caller.[5] If the error is ever reported to the user it can be displayed with a specific "errstring" or use the default errstring associated with the error-code. An error-logging function should also be included.

Perhaps there should be a hierarchy of errors, so a caller can know in detail or just in general what the error was; there might be a few different actions that a caller might take depending on the type of error, or what class it belongs to, but not on the specific error (e.g., any kind of error dealing with file-access might be handled one way, whether it's because the file doesn't exist or because the program does not have permission to get access to it).

It is important to have a dialogue among developers on what error information will be useful for their modules. Adding detailed errorcodes does no good if no one pays attention to them, and can be dangerous if no one knows what to do with them. I think it is almost always the case that errors should only be printed by user interface level functions. However, it is probably the case that errors can be logged by lower level procedures, especially for debugging purposes.

The predefined errorcodes in errno.h (and their related strings that perror() uses) can be used for system errors (e.g. file-related errors). The errcode can be set to what the system-related error was (e.g. what the error to a fopen() call might be). This would mean that codes defined by Athena Lens will have to start after 75, which is where pre-defined errorcodes end. To allow for the possibility that unix-errorcodes could be added in later versions of Unix we might want to start at 100 or higher.

There should be a pseudo-errorcode that means "print what's in the errorstring, but don't worry, it's really not an error but something happened that the user should know about."

4.4. Data Collection

4.4.1. Overview

Motivation for the Data Collection module is that designers, researchers, and Argus users need to record certain information about their use of Argus. For example, how often users create new rules or rulesets, how often they delete or change them, how often a given ruleset was invoked, and how often a given rule fired on a message. Other types of relevant data include information on the different *types* of messages being processed.

---

[5]Maybe error strings should be set up in such a way that functions can add to them?

The design of the data collection module allows different categories of data to be collected independently of each other. For example, one can collect data about rule creation and deletion without collecting information about rule firing activity. There are three types of data collected by the data module: rule activity, rule life, and message activity. Rule activity data logs testing and firing information about rules. Rule life data logs creation, deletion, or changing of rules and rulesets. Message activity data logs information about the processing of messages.

Data for a given user is stored (by default) in files in the *data* subdirectory of that user's *lens* directory. The default names for the three files are as follows:

- *lens.rl.log*: stores data about creation, changing, and deletion of rules and rulesets.
- *lens.ra.log*: stores information for every rule in each rule set; the information shows how many messages a rule was tested on and, of those, how many the rule fired on.
- *msg.log*: stores data about messages that are processed by Argus, such as the *type* of the message and the time at which it was processed.

4.4.2. Rule Life Data

Each time a rule or ruleset is created, deleted, or modified, the *lens.rl.log* file records the event. A ruleset is considered to be modified when a rule is added to or removed from the ruleset, or when a rule in the ruleset is modified. The log file refers to rules and rulesets by the date at which they were last changed. Rulesets are also identified by their name, whereas for rules the names are optional.

When a rule is created the rule itself is recorded in the log file, along with its date identifier.

4.4.3. Message Activity

Each time Argus handles a message the *msg.log* records the time of the event and the type of message processed.

4.4.4. Rule Activity Data

Unlike with rule life and message activity logging, information about rule activity is not immediately recorded in a log file. The data collection module keeps track of rule activity over a certain period of time (which the user modify). Then, at the end of the time period, statistics are recorded in *lens.ra.log* for each rule in every ruleset for that period. The statistics show how many messages the rule was tested on and how many the rule fired on.

4.4.5. Formats of the Data Collection Files

There are four files associated with data collection. The three files listed above are in a human readable format. The fourth file, *lens.data*, is read from and written to by the data collection module and should not be modified. It is, however, an ascii file with the following format:

```
format_version <num>        /* format version number */
**RULE-LIFE:
<enabled>                   /* indicated whether specific collection ok */
<in-time>                   /* used internally for computation */
```

```
<start-time>                   /* format: seconds since Jan 1, 1970 */
<stop-time>                     /* same format */
<period>                         /* collection period length */
**RULE-ACTV:
<enabled>                        /* indicated whether specific collection ok */
<in-time>                       /* used internally for computation */
<start-time>                   /* format: seconds since Jan 1, 1970 */
<stop-time>                     /* same format */
<period>                         /* collection period length */
**MSG:
<enabled>                        /* indicated whether specific collection ok */
<in-time>                       /* used internally for computation */
<start-time>                   /* format: seconds since Jan 1, 1970 */
<stop-time>                     /* same format */
<period>                         /* collection period length */

/* the following occur only if rule-activity statistics are still being
   collected at the time of termination of rule running
*/
<rule-date-id> #<num>   /* same format; number of rule sets following */
<rule-set-name> #<num>  /* number of message type following */
<message-type-name> <num> <num> <num> <num> <num> <num>
        .
        .
        .
/* number of tests, then fires for shuffled, incoming,
   and outgoing messages
*/
```

So a sample *lens.data* file might look like this:

```
format_version 5
**RULE-LIFE:
1
1
614620707
630172707
15552000
**RULE-ACTV:
1
1
614829469
614833069
3600
**MSG:
1
1
614620716
630172716
15552000

614554829 #1
rset #2
phone 4 0 30 5 0 0
meeting 2 2 0 0 0 0
```

The block of four lines starting at the line break are temporary. They contain information about the statistics of rule testing and firing being collected in the current rule activity collection period. When the collection period has ended the block of data is deleted here and put in a more human-readable form in the file *lens.ra.log*. The above block records that within the current collection period, the rule created on Thursday, June 22 1989, at 17:40:29, was invoked by one ruleset, "rset". The rule was tested on two different message types, "phone", and "meeting". For the "phone" messages, the rule was tested on 4 and fired on none while shuffling messages; it was tested on 30 and fired on 5 incoming messages; it was tested on no outgoing messages.

4.5. Rule Language

The rule editor presents a number of difficulties, both technically and from a user interface standpoint. Development in a Unix/C environment introduces problems, because it is not clear whether or not an intermediate rule language is necessary and what it should look like. From a user's perspective, it is important that that rules be easy to understand and it should be quick and easy to create the most commonly used rules. A visual rule editor that is similar to creating messages is an effective way to address this. However, it is not clear how to enable people to create more complex rules as well.

At the initial review meeting, we discussed a number of options. The output of the visual editor has to be saved in some form. The question was whether there should be an intermediate form, from which a compiled version could be created or whether the output should be handled directly by the rule runner. Another question was whether the final version should be readable by people (implying that they could also edit the rules) or not. We separated the issue of creating a file that could be easily reparsed by the visual editor and in creating a file that could easily be converted into an optimized compiled form. After much discussion, it was decided to create an intermediate rule language, which is human-readable. This turned out to be a useful decision. The technical problems associated with using the X toolkit and the user interface issues involved in permitting users to create complex rules (i.e. those with AND, OR and NOT) significantly delayed the development of the Rule Editor. By having an intermediate, human-readable rule language, we were able to create rules and test the rest of the software even without the rule editor.

The rule predicates and actions were expanded beyond the original proposal and include a wide range of options. A major difference between the original versions of the Information Lens and Argus is that users have the ability to create and register their own message types simply by adding new fields. The rule language can operate on those new message types and has a sophisticated set of user properties and access to information about the state of the message. The message fields can be one of three possible types: strings, lists of strings (such as an implicit OR between two names separated by a comma), and dates. The date parser is very flexible and can handle a wide range of date specifications.

Rules can handle a range of actions, including standard mail activities (e.g. move and delete messages), setting and manipulating user-specified properties, reacting to the state of the message and calling any Unix shell command. Rules can have multiple actions, including running other rule sets, which permits complex rule actions and the equivalent of IF-THEN-ELSE constructs. Rule actions are registered and different rule sets can have different sets of actions enabled. This will permit the flexible development of services such as the rule server. The latter is simply a special user that accepts messages and rule sets addressed to "anyone". The rule sets have only the "send" action enabled, to prevent unauthorized deletion of other people's messages.

4.5.1. IF Conditions

The Rule Editor provides a template for editing the conditions under which the rule will fire. Incoming messages with fields filled out in the same way as the rule will cause the rule to fire. If you fill in the **TO** field of a rule with a particular string, such as "Smith", messages that contain that string in the **TO:** field will cause the rule to fire.

You can specify what will happen if an incoming message meets the specified conditions. For example, you can specify that messages addressed to "Smith" are to be moved to the ***Urgent*** folder, by filling in the **MOVE TO:** field with the string "urgent".

4.5.2. THEN actions

Athena Lens allows you to specify one or more actions if a rule fires. In addtion, you can create your own actions by using ***system***, which calls a Unix shell command. The action routines will operate on the current message and the current folder, unless otherwise specified.

The following actions are possible:

Action	Arguments	Definition
Move:	1	Move the message to a specified folder and set the state to "MOVED".
Copy:	1	Copy the message to a specified folder and set the state to "COPIED".
Delete:	0	Delete the message and set the state to "DELETED".
Runrules:	2	Run the specified ruleset on the specified folder.
Add:	2	Add an integer to the specified user property.
Subtract:	2	Subtract an integer from the specified user property.
Set:	1	Set a user property.
Clear:	1	Clear a user property.
Stop:	0	Stop processing of this ruleset
System:	1	Execute the specified Unix shell command on the message.
Permanent:	1	If "on", then save the user properties on disk. Otherwise, delete them after the session.

4.6. Rule Sets

The results of previous lens user studies indicated that users want flexible control over the creation and running of rule sets. In Argus, users can create as many rule sets as they like. Rules must exist within a rule set and are run sequentially over the messages in a folder. All rule sets may be run manually at any time. In addition, rule sets may be run as rule actions or called from other programs. The Argus team is also considering applications which would run a rule set in response to an event. That event could be, for example, a certain time or the existence of new mail.

Names of rules are optional while rule sets must be named. Rule Sets are generally stored in a special directory (*~/lens/rule sets*) where they are easily accessible to the rule set registry. Users can store rule sets in other locations and specify the appropriate paths. These rule sets will be registered when the user calls them with the ***runrules*** command.

Development of the rule set editor has been postponed, due to ongoing problems with the Athena widget set. The current version of the rule editor specifies both the name of the rule set and the number of the rule. When the rule set editor is completed, users will be able to look at and manipulate a visual representation of each rule as well.

4.7. Running Rules

4.7.1. The runrules Command

The format of the ***runrules*** command is:

runrules ruleset [folder] [-Mmailer_type] [-n] [-S[on/off]] [-U[on/off]] [-w] [-D[option]] [-Ttime] [-help]

One of the two interesting features of ***runrules*** treatment of folders is that a folder may be read from standard input (stdin). This allows ***runrules*** to be used as a filter, pipe, or forked processes in the standard Unix environment. While this in not an option for all mailer types, it is particularly useful for processing the folders of popular mailers, such as "rmail". To read a folder from stdin, simply omit the folder name in the command line.

The other feature of a folder in Lens relates to the "-n" option, which allows the nesting of folders within each other. In the case of MH, this means that a rule can move or copy a message into a subdirectory of an existing folder. If the subdirectory does not exist, it will be created under the same laws that govern the Unix "mkdir" and MH "rcvstore" commands (i.e. - a new sub-folder called "foldername/subfolder" will be created as "subfolder", under "foldername", only if "foldername" already exists). Once the sub-folder has been created, subsequent executions of ***runrules*** can reference the sub-folder(s) if the "-n" option is used. If the "-n" option is not used, then the rule set will not be applied to the sub-folders. Note that the "-n" option refers to the reading of sub-folders only, and is not required for a rule set to create or write to a sub-folder.

A folder can be specified as either a file with a full pathname or as just the filename. If just the filename is specified, the path will be resolved in the same way as it would be by the type mailer specified (e.g. - for MH, the path returned by "mhpath" is used; for most other types, the current directory, or Unix environment search path is used). A ruleset can be specified as a file with a full pathname or as just the filename. Unresolved paths default to either the path set in the user configuration file or to the current directory. (See Folders and Rule sets further details.)

Certain types of additional information can be optionally be saved with a processed message. During the execution of a rule set by ***runrules***, the user's rules may create user properties which the user may wish to remain associated to the message after ***runrules*** completes. Such user properties are similar to standard programming variables, and may represent importance values, classification codes, a date the message should be deleted by, a flag that the message had already been read, or perhaps a date by which the message should re-surface as a reminder. Additionally, states of a message are changed when running rules. Some examples of states are: the number of rules that have fired on this message, the number of times it has been moved, and the last folder from which it was moved. The user can optionally save or disregard this information after executing the rules on a message. User properties are created by rules as temporary variables which dissappear when the rule set finishes firing on a message, unless specifically commanding that a user property become "permanent" as an associated property of the message. The user can override this by specifying the "-Uoff" option, which will un-associate all user properties from the message when ***runrules*** is done with it. Inversely, all state information about a message is not saved by default; this can be overriden using the "-Son" option to ***runrules***, which will save the state information with the message on completion.

The parameters for ***runrules*** follows:

-Mmailer_type	Specifies the type of folder (how it should be interpreted). The default is: -Mmh
-n	Specifies that nested sub-folders are to be read. The default is to NOT do sub-folders (sub-directories).
-S[on/off]	Saving output of the state information. The default is to not save state information (i.e. - read state info in when processing a message, if it exists, but do not write out when done processing the message). The -S option will cause all state info read or created to be saved with the message, it can be immediately followed by 'on' or 'off' as is appropriate.
-Ttime	Flags the time that the runrule runner is being fired as a string 'incoming', 'outgoing', etc..
-U[on/off]	Save output of the 'permanent' user property information. The default is to save user property information with the message (i.e. - read any existing user properties information in when processing a message, if it exists, and write out all 'permanent' user properties when done processing the message). Note that only user properties which have been set 'permanent' can be saved. The -U option can be immediately followed by 'on' or 'off', as is appropriate, to make the intent explicit.
-D[option]	Choose Data Collection option desired. If \"-d\" is used, with no option specified, then all off is used. The default is all on, if \"-d\" is not specified. Some valid options are: on off -d option can be used more than once on command line.
-w	Suppress top level warning msgs. Warnings are when msg file is not accessible (no read access), etc.
--	Last '-' option. This allows a ruleset name or folder name to be a filename that begins with a dash ('-'). (Either "--" or "- " is valid.)

Example1: ***runrules*** test.rul inbox -n

Example2: ***runrules*** -Mmh -Soff -Uon -Tincoming -Doff -w -n test.rul inbox

Example3: ***runrules*** -Mmh -Soff -Uon -Tincoming -Doff -w -n -- -r.rul inbox

The ruleset MUST occur before the folder in the command line. If the folder is omitted, standard input is used (when possible). The '-' parameters may occur in any order (except "--", which must be the last '-' parameter, if used). Parameters taking arguments are a single capital letter, directly followed by the argument (with no spaces).

By default, Athena Information Lens uses the MH mailer folders; however, other types mailer folders may be specified by using the "-M" option. MH users, who wish to run a certain set of rules when they "inc" their mail, can achieve this by creating a ruleset named "incrules" and using "alinc" in place of "inc".

4.7.2. The alinc Command

The ***alinc*** command takes the same arguments as ***runrules*** and then runs the specified rule set on the specified folder. ***alinc*** is a script which simply ***inc***s the users's mh mail into her inbox and then runs runrules on her inbox. The Argus team is considering an extension to this model which would address the problem of limited personal storage for incoming mail. If a user does not have enough personal storage for her mail, a variant of alinc would inc the mail to a temporary place, run the user's rules on the new mail, and finally move any left over mail to the users Mail folder.

Example: alinc newmailrules +inbox

4.7.3. The alcomp Command

The ***alcomp*** command takes the same arguments as the mh command ***comp*** as well as the arguments of the Argus ***runrules*** command, and a message type. It creates an emacs window with the fields specified in the message type and then runs the specified rule set when the message is sent.

Example: alcomp -messagetype message|meeting|alens sendmailrules

4.8. Module Descriptions

Argus consists of a large number of separate program modules. The following section outlines each of the major programs, provides some of the philosophy as to why the code was written in a particular way, what the major benefits and problems have been and what would have been added if more resources were available.

4.8.1. Rules

Users create rules to manage their incoming messages. They can create rules from scratch with a rule editor or copy an existing rule. The rule editor provides a visual interface for creating simple rules. These are translated into a text rule language (which permits more complex operations) and is stored in a rule set. Rule Sets may be run at different times, according to the user's preferences. The components of the rule system include:

- **Rule Language:** Created in a human-readable form that can be parsed by the rule editor and stored on disk. (Note that this code is very fast, but appears slow because of mh.)

- **Rule Editor:** Written in C, using the Hewlett Packard and Athena Widget Sets for the X toolkit. Users can fill in blank fields, create new fields, and change the type of the field (e.g. from string to date). Users can specify one or more actions if the rule fires. The rule editor is a complex editor and is currently under redesign due to user feedback and ported to use the Motif Widget Set and Toolkit.
- **Rule Set Editor:** Enable users to reorder rules within a rule set, as well as add, change, delete and save them. The current rule set editor is being redesigned and ported to use the Motif Widget Set and Toolkit.
- **Rule Firing Data:** Collected by the data collection module. Saves rule firing history, rule creation history, and snapshots of rules at specifiable periods of time.
- **Running Rules:** Run by the user or from a program. Users can manually run rule sets at any time. They can also use special programs, such as ***alinc***, to run rules on their new mail.

4.8.2. Folders

Argus will handles folders from three different mailers: mh, xmh, RMAIL. Rules can specify the type of folder a message will be moved to. This allows users to convert messages easily from mh to RMAIL formats and vice versa.

Folders can be arranged hierarchically into subfolders. The RMAIL folder supports standard input, which allows users to pipe a folder into ***runrules*** or redirect input and output.

4.8.3. Message Types

Message Types: Organizations may have sets of standard message types, such as phone or meeting announcement. Individuals and small groups can also create their own message types. The organization, storage, retrieval, and naming of message types has largely not been addressed by the Argus team. Although the system is designed to accommodate message types, the message typing system is primative. The system involves a file of global message types kept in a place accessible to all users. The global message types are alterable only by members of the Argus team. Individuals may create and store their own message types in files. Argus does not support special directories or naming conventions for such files yet.

- **Message type editor:** This editor permits the creation of personal message types. It allows the user to create a set of named, classed, and typed fields. Example of the name, class, and type of a field are 'CC:', 'from the message', and 'same case text' respectively.
- **Message type registry:** Stores the names of the message type registries and paths to their locations. Can accommodate message types from multiple files.
- **Message sender:** (In development) It is not clear that this is necessary, since all of the mail handlers use text editors (usually emacs) for creation of messages. The message sender will permit additional functions, such as the ability to designate a particular message as an example of a new message type, enabling "messages-by-example".

4.8.4. Data Collection

Data Collection: Although initially listed as optional, this became a high priority item. Note that because Argus is not tied to a particular mail system, data collection is based only upon Lens-related activities. Thus, we cannot directly measure the numbers of messages sent and received by an individual, only the numbers of messages that Argus processes in some way.

Users have the option of turning data collection completely on or off or specifying that only some parameters are saved. Data is collected in two ways. In some situations, a complete record of every event is important. For example, every rule is stored and each change is recorded. In other situations, the rate of activity is of interest. In these situations, a starting time must be specified and the length of the interval during which information will be recorded. For example, every time a rule fires within a specified time period, an counter is incremented. The firing rate per specified interval is calculated at the end of the period. Argus allows users to specify any length period and either provide a specific starting time (e.g. record data at 2 am on Sundays) or from the time of initialization of the data collection file.

Argus has system-wide default settings which can be overridden by the user's *lens.data* file. In addition, users can provide a temporary override during a particular call of the ***runrules*** command. The location of all the data collection files are specified in the user's lens profile. The *lens.state* file saves information relevant to data collection, including which data settings are on or off, the start and stop times for data collection, the interval for data collection, and the intermediate storage location. Unlike the lens profile, this file is normally updated with a special program.

The types of data collected are:

1. **Message history:** Saves how many messages of each type have been processed by rules over a specified period of time.
2. **Rule activity:** Saves the firing history of each rule over a specified period of time.
3. **Rule life:** Saves the history of how rules are edited and modified over time.
4. **Snapshots:** Saves the state of the user's rules and message types at the time specified.

4.8.5. User Configuration

Argus allows specification of default values in several places. All settings have a system-wide default. Users can specify defaults in their personal directories which override the system defaults. Finally, particular commands can temporarily override these default values for the duration of the command.

The lens profile specifies default values for user-specifiable settings, such as the type of mail handler and the location of lens files. It also specifies whether user properties and states that have been specified as "permanent" will be saved. User properties can be set and deleted with rules. Sometimes, the property is only useful for the life of that particular rule editing process. At other times, the property is required for rules that are run at some point in the future. In the latter case, the user should include a rule action that specifies that the "permanent" flag is on. A user can later decide that none of these user properties be stored and over-ride this setting in the Lens profile.

Users can copy the lens profile shown in the next section and modify it to meet their needs. The system defaults are shown with a comment (#) in front of the line. The user can activate the line by removing the comment and editing the line as necessary.

4.9. Appendices

4.9.1. Registry Object

Athena Lens uses registries to keep track of different objects and provide a consistent way to refer to and store them. Registries can be used in a hierarchical way, e.g. the message type registry. Registries are usually passed a key and return a value. Biregistery are similar, except they can also be passed a value and return a key. The is useful when a program retrieves a value from a registry, operates on it, and returns it to the registry.

The following registries are defined in the Registry specification.

Field Type	Fields within messages can be strings, dates, lists of strings, etc. This registry maintains a list of the possible fields for a message.
Message Type	Message types are distinguished by the different fields they contain. This registry is has two levels. The first level contains a list of all the message type registries. The second level contains the paths to each message type file.
Rule Set	Rule sets contain collections of rules and are usually stored in a special directory. This registry keeps track of the different rulesets in this directory and any others that are registered when a rule is created.
Verb	Rules can perform different actions which consist of a verb and an optional number of parameters. This registry keeps track of which verbs are available and allowed.

The registry functions include:

1. **Registry_create(compare_func, hash_func)**

   Creates and returns an empty registry. compare_func is used to compare items in the registry. It should return 0 if its arguments are to be considered equal. hash_func should return a number between 0 and its second argument, and should attempt an even distribution. If compare_func would return 0 for a pair of objects, hash_fuct should return the same value for those objects. For registries of abstract objects (pointers), Registry_ptrcmp() and Registry_ptrhash() should be passed as the compare_func and hash_func. Registry_strcmp and Registry_strhash() may be used for strings. Registry_strcasecmp and Registry_strcasehash() may be used for strings where case is not significant (case-insensitive).

2. **Registry_size_hint(registry, size_hint_value)**

   The registry may operate more efficiently if this operator is called and size_hint is close to the maximum number of elements to be in the Registry, at the possible cost of additional memory use. Likely to be effective only on an empty registry.

3. **Registry_add(registry, key, value)**

   Adds the association between key and value to the registry. Neither key nor value are copied, and neither may be freed before being removed from the registry. The key should not be modified in way that would change the value of the compare_func or the hash_func until this association is removed from the registry. Will return Bool_FALSE if an association with the same key is already in the registry (in which case the add will not be performed), Bool_TRUE otherwise (on successful completion).

4. **Registry_remove(registry, key)**

   Removes the association with key from the registry. Returns Bool_FALSE if no such association exists, Bool_TRUE otherwise (on successful completion)

5. **Registry_get(registry, key)**

   Returns the value associated with key in the registry. Returns NULL if there is no such association.

6. **Registry_traverse(registry, action_proc, private_pointer)**

   Calls action_proc once for each entry in the registry. private_pointer is a VOIDP which is passed to the action_proc, but not otherwise used. action_proc should not modify the registry in any way. action_proc takes three arguments, the key, the value, and private_pointer.

7. **Registry_entry_count(registry)**

   Returns the number of associations in the registry.

8. **Registry_fetch_contents(registry, key_darray, value_darray)**

   Stores the contents of the registry as follows: In no particular order, each association is processed in turn by storing (using Darray_addh) the key into key_darray and the value into value_darray. Either or both key_darray and/or value_darray may be NULL, in which case the corresponding data will not be processed. Actual Darrays passed (not NULL) must be empty. Any objects added to key_darray must be treated as read-only as long as they remain in the registry.

9. **Registry_destroy(registry)**

   Deallocates all resources needed by the registry. Should be the last operation performed on the registry. Does not deallocate the objects (keys and values) contained in the registry (this should be done after the registry is destroyed). Implicitly removes all associations from the registry.

The biregistry function includes all of the above, plus the following two, which pass a value instead of a key.

1. **Registry_remove_rev(registry, value)**

   Removes the association with value from the registry. Returns Bool_FALSE if no such association exists, Bool_TRUE otherwise (on successful completion)

2. **Registry_get_rev(registry, value)**

   Returns the value associated with key in the registry. Returns NULL if there is no such association.

4.9.2. Lens Profile

```
#             Default personal lens profile
#                   25 August 1988

#            Based on: /mit/lens/lens.profile.0.2

#  To change a value, delete the "#" and edit the line.

#  For example, to change from using mh to Rmail:

#          old:   #MailUserAgent:     mh
#          new:   MailUserAgent:      Rmail


###########################################################################
#                                                 #
#  The mail user agent you use determines which folder format      #
#  Lens should use by default.  The current default is mh.  Other  #
#  options are xmh and rmail.                            #
#                                                 #
###########################################################################


#MailUserAgent:              mh



###########################################################################
#                                                 #
#  State information about individual messages can be saved for    #
#  use by your rules.  (For example, a rule could fire only if a   #
#  message has not already been moved or deleted by another rule.) #
#  The default is to not save state information.               #
#                                                 #
#  User properties allow you to specify information about certain  #
#  messages.  (For example, a rule could fire on messages that     #
#  been given the property "important".  The default is to save    #
#  user properties.                                      #
#                                                 #
###########################################################################


#SaveStates:                False

#SaveProperties:             True


###########################################################################
#                                                 #
#  Default message types when creating rules and sending messages  #
#  can be specified.  The order in which action objects appear in  #
#  the Rule Editor action menu can be specified, as can the order  #
#  of field types.                                      #
#                                                 #
###########################################################################
```

```
#DefaultRuleMessageType:        MSG

#DefaultSendMessageType:        MSG

#VerbPresentationOrder:         move,copy,delete,set

#FieldTypePresentationOrder:    SubString,Date

#####################################################################
#                                                                   #
#  Locations of files for storing personal sets of rules, action   #
#  programs, and the message type registry can all be specified    #
#  individually.  The default is to create a ~/lens directory      #
#  with additional files and subdirectories.                       #
#                                                                   #
#####################################################################


#LensDirectory:                 $(HOME)/lens

#RulesetDirectory:              $(HOME)/lens/rules

#ActionDirectory:               $(HOME)/lens/actions

#MessageTypeRegistryInit:       $(HOME)/lens/mtr.init


#####################################################################
#                                                                   #
#  Lens can save information about your use of electronic mail,    #
#  including numbers of messages processed by rules over a period  #
#  of time (MsgDataOutput), the firing history of your rules       #
#  (RuleActivityOutput), how your rules have changed over time     #
#  (RuleLifeOutput) and a snapshot of your rules and message types #
#  at a specified point in time.  The locations of the data files  #
#  can be specified individually.  The default is to create a      #
#  directory called ~/lens/data directory with data files.  You    #
#  can update the default data collection settings by typing:      #
#   lensdc.                                                         #
#                                                                   #
#####################################################################


#DataDirectory:                 $(HOME)/lens/data

#DataStateFile:                 $(HOME)/lens/data/lens.data

#MsgDataOutput:                 $(HOME)/lens/data/lens.msg.log

#RuleActivityOutput:            $(HOME)/lens/data/lens.ra.log

#RuleLifeOutput:                $(HOME)/lens/data/lens.rl.log

#SnapShotOutput:                $(HOME)/lens/data/lens.data.snap
```

4.9.3. Rule Examples

This section provides a series of examples of how to use the major Athena Lens features. The examples below are for someone named "Smith"; you should use your username instead.

---

1. Identify messages that are addressed to you personally, rather than to a distribution list, and place them in a folder called: **ToMe**. (This is a useful rule for your incoming-mail ruleset.)

If	TO:	smith
Then	MOVE:	ToMe

2. Enhance the previous rule to include messages that are copied to you directly:

If	**OR**	
	TO:	smith
	CC:	smith
Then	MOVE:	ToMe

3. A more complex version of a rule would send you a zephyr message whenever you get a phone call. (Need to write a program that automatically pops up a zephyr message on the screen. This rule also saves a copy of the mail message in the **Phone** folder.

If	**OR**	
	MESSAGE TYPE:	phone
	SUBJECT:	phone
Then	**AND**	
	MOVE:	phone
	SYSTEM ZWRITE	

4. Save meeting announcements in a **Meeting** folder and add them to your calendar.

If	**OR**	
	MESSAGE TYPE:	meeting
	SUBJECT:	meeting
Then	**AND**	
	MOVE:	**Meeting**
	SYSTEM CALENDAR	

5. Save copies of messages that you've sent in a folder called: **messagessent**. (This is a useful rule for your outgoing-mail ruleset.)

If	FROM:	smith
Then	MOVE:	**messagessent**

6. Organize messages from different distribution lists into their appropriate folders. Here, put messages addressed to the Xpert mailing list into your **Xpert** mail folder. (This is a useful rule for your cleanup ruleset.)

If	TO:	Xpert
Then	MOVE:	**Xpert**

7. Delete messages sent to the ADMIN distribution list if they are older than two weeks.

If	**AND**	
	TO:	admin
	DATE:	>2 weeks
Then	DELETE:	

8. Look at messages sent to the anyone server that are of interest to you. (This is a useful rule for your anyone ruleset.)

If	SUBJECT:	AI, artificial intelligence
Then	SEND:	

5. Creating New Argus Interfaces

5.0.1. Creating a Folder Type Object

A folder type allows Argus to manipulate any information which can be represented as a frame. A folder type must support only a small number of basic operations. A new folder type is registered with the Argus system by a call to the function AlFolderT_register(..).

```
NORET AlFolderT_register(Name, CreateFromFile, CreateFromStdin,
                         GetNextMsg, RemoveCurrentMsg, AddMsg,
                         PrintSelf, Destroy);

STRING         Name;
VOIDP_FUNCPTR CreateFromFile;
VOIDP_FUNCPTR CreateFromStdin;
NORET_FUNCPTR GetNextMsg;
BOOL_FUNCPTR  RemoveCurrentMsg;
BOOL_FUNCPTR  AddMsg;
NORET_FUNCPTR PrintSelf;
NORET_FUNCPTR Destroy;
VOIDP         UserDataHook;
```

Where each of the generic function types is described below.

```
typedef VOIDP (*VOIDP_FUNCPTR) ();
typedef Msg (*MSG_FUNCPTR) ();
typedef NORET (*NORET_FUNCPTR) ();
typedef Bool (*BOOL_FUNCPTR) ();
```

An example of this function call can be found in the file *rm_folder.c* in the *src* directory. It looks like:

```
  AlFolderT_register("RMAIL",
                     AlRMFolder_create_from_file,
                     AlRMFolder_create_from_stdin,
                     AlRMFolder_get_next_msg,
                     AlRMFolder_remove_cur_msg,
                     AlRMFolder_add_msg,
                     AlRMFolder_print_self,
                     AlRMFolder_destroy,
                     (VOIDP) NULL);
```

In order for Argus to call a particular AlFolderT_register(..) function, that function call must be placed inside a function with a name of the form.

Al<*foldert-type*>folder_init()

Where <*folder-type*> is a capitalized two character acronym for the folder you are creating. For example, the RMAIL initialization routine is called AlRMfolder_init(). The initialization routine of a folder type performs any initialization required by that folder type and then calls AlFolderT_register(..). Finally, a call to the initialization routine of the folder type is placed in the function Al_init() which is found in the file *al.c*. The programmer then places all the .c and P.h files of the new folder type in the *src* directory and the .h files in the *include* directory and issues the following sequence of commands to remake the Argus applications.

make depend make all

The semantics of the various folder type functions registered with Argus is described below.

- **`VOIDP` CreateFromFile(file_name)**
  `char* file_name;`

  This function creates a folder type specific representation of the folder and returns it as an opaque data type. The opaque data type is passed to all of the functions used to manipulate the folder. What this function actually does is folder type dependent; the only thing that it is required to do is create a unique folder type ID (the returned opaque data type) associated with the file name for this folder type.

- **`VOIDP` CreateFromStdin()**

  This function creates a folder type specific representation of the folder and returns it as an opaque data type. The semantics of this function are identical to that of the *CreateFromFile* function except that input comes from the special Unix file *stdin*.

- **`NORET` GetNextMsg(opaque_data, msg_buffer, state_buffer, user_properties_buffer, folder_specific_registry, do_nested_folders, no_external_state_usage, no_external_user_properties_usage, previous_message);**

```
VOIDP     opaque_data;
char      **msg_buffer, **state_buffer;
char      **user_properties_buffer;
Registry  folder_specific_registry;
Bool      do_nested_folders, no_external_state_usage;
Bool      no_external_user_properties_usage;
Msg       previous_message;
```

  This function is how Argus reads each message from the folder type. Each message is read in succession. The opaque_data is the folder type specific data returned by either the **CreateFromFile** or **CreateFromStdin** functions. The buffers (msg_buffer, state_buffer and user_properties_buffer) are arrays of *generalized fields* (described in the file *msg.h* in the *include* directory). The first thing to do is to write out any states or user properties to the previous message that must be saved. If the flag no_external_state_usage is Bool_FALSE then all the states available from the previous_message should be written out to the previous message read from this folder. This information can be gotten from the function AlMsg_write_states_to_buffer(previous_message) described in *msg.h*. Writing out the user properties depends similarly on the flag no_external_user_properties_usage and the function AlMsg_write_uprops_to_buffer(previous_message). Next, the three buffers must be filled with information from the next available message. The message_buffer, state_buffer, and user_properties_buffer should be filed with fields from the message, previously saved states (attached to the message), and previously saved user properties (attached to the message) respectively. How states and user properties are 'attached to a message' is left up to the folder type. It suffices that the ability to attach these two types of information to a message exist if they are going to be used by the rule language. If no next message is available the three buffers should be set to NULL. Also, any folder specific states should be placed into the folder_specific_registry.

- **`VOIDP` RemoveCurrentMsg(opaque_data,previous_message)**
  `VOIDP opaque_data;`
  `Msg previous_message;`

  This function should delete the last message read with the GetNextMessage function. It is recommended that the folder simply mark the message as deleted and do the physical deletion when the folder is closed.

- **`VOIDP` AddMsg(opaque_data,message)**
  `VOIDP opaque_data;`
  `Msg   message;`

  This function should append the message to the end of the folder. All the information needed can be gotten from the Msg object through various routines described in *msg.h*.

- **`VOIDP` PrintSelf(opaque_data)**
  `VOIDP opaque_data;`

  This function should print a representation of itself to the Unix file *stdout*.

- **`VOIDP` Destroy(opaque_data)**
  `VOIDP opaque_data;`

  This function should destroy the folder identified by the opaque_data. The function should make sure that all deleted messages are actually deleted. This function will always be called.

5.0.2. Creating a Field Type

This subsection describes how to create a new field type to be used in Argus rules. The purpose of a field type is to define the type of information which will appear in the field of a message and ways to compare two fields of the same type. Some familliar field types are *integer* and *date* and some less fammiliar field types are *internet-address* and *host-name*. Two steps are required to compare two fields (one from the rule and one from the message). In the first step, both fields are parsed into an opaque data type unique to the field type. Then the opaque data types are passed to one of the field type's comparison functions which returns either a TRUE or a FALSE. Note that the opque data type derrived from the rule field is passed to the comparison procedure before the one from the message. Order matters. All the relevent information about a field type is organized into a AlFieldTypeRegistry_str and then registered with Argus through predefined function calls described later. The AlFieldTyperegistry_str is shown below and appears in the file *FTreg.h*.

struct AlFieldTypeRegistry_str { const char * name; FieldParserProc *parser; FieldUnParserProc *un_parser; FieldOpArg *fieldOpList; IndexOpArg *indexOpList; InterFieldOpArg *interFieldOpList; FieldFreeParsedDataProc *free_parsed_data; };

The name is a character string which is the name of the field type. The other members of the structure are described according to their data types below. They appear in the file *FieldType.h*.

- typedef char *(**FieldParserProc**)(const char *);

  This function transforms the value of a field into an opaque data type called a FieldValueObject. The FieldValueObject is used to represent the value of the field as interpreted by this field type. It is passed to the other functions of the field type.

- typedef const char *(**FieldUnParserProc**)(const char *);

  This function is used to transform a FieldValueObject back into the sequence of characters passed to the FieldParserProc. It is used mainly for debugging purposes and is not necessary if the transformation done by the FieldParserProc is not symmetric.

- typedef int (FieldOperatorProc)(const char *,const char *);

  This function is used to compare two fields. It is passed two FieldValueObjects and must return a zero (FALSE) if they are not related by the function or non-zero (TRUE) if they are. An example of this function would be the '>' function of the Integer field type.

- typedef struct FieldOpArg_str { const char *name; FieldOperatorProc *value; } **FieldOpArg**;

  This structure is used to name a FieldOperatorProc. fieldOpList is a NULL terminated list of FieldOpArgs.

- typedef char *(**IndexOperatorProc**)(char *,char *);

  This function is designed to take two FieldValueObjects and output a single FieldValueObject. A function of this type is used by the msg object in order to condense, according to some rule, a series of fields with the same name. For example, a mail message may contain multiple TO: fields. A user could request that the last 3 fields be concatenated and then checked to see if they contain the word 'rap'. Condensing the 3 fields is performed by an IndexOperatorProc. The name was derrived because this procedure is seen as condensing a series of indexed fields to one field. If more than two fields are to be condensed the output of one application of the function becomes one of the inputs to the next application.

- typedef struct IndexOpArg_str { char *name; IndexOperatorProc *value; } **IndexOpArg**;

  This structure is used to name a IndexOperatorProc. indexOpList is a NULL terminated list of IndexOpArgs.

- typedef int (InterFieldOperatorProc)(char **);

  This function is designed to accept a NULL terminated list of FieldValueObjects from *different* fields and determine if they are related in some way. The functionality of this type of comparison is different from the other comparison functions in that the user does not specify a value for a field in a rule. Instead, the values of different fields are compared to each other. This type of function would be represented by a box surrounding several fields in the rule editor. An example of an InterFieldOperatorProc is '=' for the any-case-text field type. With this interfieldoperator procedure a user could tell, for instance, if the CC: field is equivalent to the FROM: field without ever having to know what the contents of those fields are.

- typedef struct InterFieldOpArg_str { char *name; InterFieldOperatorProc *value; } **InterFieldOpArg**;

  This structure is used to name an InterFieldOperatorProc. interFieldOpList is a NULL terminated list of InterFieldOpArgs.

- typedef void (**FieldFreeParsedDataProc)**(char *);

  This function is used to free any storrage allocated by a FieldValueObject. It is called with a FieldValueObject.

Once a field type is created it must be registered with the Argus system. This is accomplished through two functions: SystemRegisterRequests and UserResgisterRequests. Field Types and verbs are registered through this mechanism. This mechanism is designed to allow new field types to be introduced into Argus with only a relink. SystemRegisterRequests registers field types and verbs which come as part of Argus. UserRegisterRequests registers a user's new field types or verbs. Inside of UserRegisterRequests you issue a call to the following function to register a field type with Argus.

```
NORET    AlFTreg_register_field_type(FieldtypeRegisterStruct)
struct AlFieldTypeRegistry_str *FieldtypeRegisterStruct;
```

To add the new function and the field types or verbs it registers to Argus simply compile it and relink it into the Argus library (lens.a). The executables will have to be relinked using the new lens.a.

5.0.3. Creating a Verb

5.0.4. Creating a Box

Acknowledgements

The Athena Lens group would like to thank Howard Webber and Tom Malone for their financial and moral support. We would also like to thank Geoff Bock, Bob Fleischer and George Champine from Digital and Dan Geer from Project Athena for their encouragement and help. Finally, we'd like to thank Win Treese, Tony Della Fera, John Kohl, Henry Mensch, Kevin Crowston, Chaing Yu, Jin Lee and Maya Bernstein for their ideas and support.

Table of Contents