# Generating Wrappers for Command Line Programs: The Cal-Aggie Wrap-O-Matic Project

**Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbu**
Department of Computer Science,
University of California,
1, Shields Way
Davis, CA 95616 USA
+1 530-752-7004
{wohlstad|jacksoni|devanbu}@cs.ucdavis.edu

**ABSTRACT**
Software developers writing new software have strong incentives to make their products compliant to standards such as CORBA, COM, and JavaBeans. Standards-compliance facilitates inter-operability, component-based software assembly, and software reuse, thus leading to improved quality and productivity. Legacy software, on the other hand, is usually monolithic, and hard to maintain and adapt. Many organizations, saddled with entrenched legacy software, are confronted with the need to integrate legacy assets into more modern, distributed, componentized systems that provide critical business services. Thus wrapping legacy systems for inter-operability has been an area of considerable interest. Wrappers are usually constructed by hand, which can be costly and error-prone. In this paper, we specifically target command-line oriented legacy systems, and describe a tool framework that automates away some of the drudgery of constructing wrappers for these systems. We describe the Cal-Aggie Wrap-O-Matic system (CAWOM), and illustrate its use to create CORBA wrappers for a) the JDB debugger, thus supporting distributed debugging using other CORBA components, and b) the Apache web-server, thus allowing remote web-server administration, potentially mediated by CORBA-compliant security services. While CAWOM has some limitations, in several relatively common settings it can produce better wrappers at lower cost.

**Keywords**
Wrappers, CORBA, legacy systems.



## 1 INTRODUCTION

Legacy systems ($\mathcal{LS}$) are ubiquitous, and provide many useful functions. Organizations often reluctantly endure a continuing dependency on these systems for critical business functions. Efforts to maintain, re-engineer and evolve such systems are hindered by poor documentation, lack of source code, lack of appropriate (perhaps outdated) skills, and brittle architectures. On the other hand, there are strong incentives to update these systems to support inter-operation with the rest of an organization's information technology infrastructure. Systems that can inter-operate using standards such as CORBA, COM and Javabeans can exploit a large and growing body of components (e.g., COM GUI components), services (e.g., CORBA Event and Security Services), and architectures. Thus, organizations using legacy systems face an unpleasant dilemma: bear the cost and risk of updating legacy systems to inter-operate, or live with fractured information technology.

*Wrappers* offer a potential way out: a surrounding software layer shields a $\mathcal{LS}$ from the burden of inter-operability. Nestled within a wrapper, a $\mathcal{LS}$ can remain unchanged, and live comfortably in the past; the wrapper assumes the burden of mediating the $\mathcal{LS}$'s interaction with more modern, standards-conformant systems. Traditionally, wrappers are constructed by hand[1] Building a wrapper for a specific $\mathcal{LS}$, for a particular inter-operability standard, would require expertise both in the $\mathcal{LS}$ and in the standard. Standards are complex, and legacy systems often are, as well; thus, manual development of wrappers is likely to be time-consuming, expensive and error-prone.

### The Problem

We focus on a specific, but fairly common situation—wrapping command-line oriented $\mathcal{LS}$s. Such systems are ubiquitous both in Unix and in older, legacy operating systems. Such systems can certainly be used directly by a user sitting at a terminal, typing commands and

[1]There has been work on automating wrapper generation for heterogenous, distributed information systems, e.g., answering queries by extracting information from web pages. Our work relates more behavourial aspects of wrapping rather than data re-structuring; more on this when we discuss related work.

observing the reply. They might also be driven by batch scripts that mimic interactive users, or by user-interface wrappers that hide the command-line $\mathcal{LS}$ behind a nice GUI that provides a more pleasant user-experience (e.g., the DDD [32] wrapper for the popular GNU debugger JDB. Wrapping command-line $\mathcal{LS}$s for standards-based inter-operability provides several advantages:

1. At a basic level, the wrapper can enable remote access to the $\mathcal{LS}$'s services, and thus allows for better integration of the $\mathcal{LS}$.

2. The type system in the relevant interface definition language (IDL), together with the applicable programming language type system, impose a certain level of programming discipline on the development of client systems that use the legacy system via the wrapper. This avoids run-time errors due to type errors. Systems that directly interact with the legacy systems do not provide this discipline.

3. The leverage of other COTS standards-based components can allow additional functionality. For example, we wrap the JDB Java debugger and demonstrate how the CORBA Event Service can be leveraged to support distributed debugging sessions in a natural way.

4. Standards such as CORBA provide the opportunity for finer-grained control over access to command-line systems. The traditional approach of providing a restricted shell is fairly coarse-grained. CORBA security mechanisms [28], for example, can be used to implement far more intricate access control policies.

Our research goal is to find ways to simplify the task of wrapping command-line oriented systems for inter-operability. In this paper, we describe a tool, the Cal-Aggie Wrap-O'Matic, (CAWOM) which generates wrappers to enable command-line systems to be accessed through the OMG CORBA inter-operability standard. Although CAWOM works with the CORBA standard, the problems we confronted, and the solutions we have developed, transcend the details of CORBA standard itself, and are applicable in other settings. Specifically, we wrap the GDB-like debugger for Java (JDB) to allow CORBA-compliant clients to access JDB's services. By simply adding a CORBA-compliant Event Service, which effectively provides an off-the-shelf multicast channel for debugging-related events, we can easily support distributed debugging sessions with one controller and many observers.

We assume that the reader has some familiarity with the basic CORBA framework. The outline of the rest of the paper is as follows. In Section 2, we present an overview of our approach, and delve in more detail into the issues that arise. In Section 3, we describe the design choices we have made, along with rationale. In Section 4, we describe our experience with wrapping two different systems: the Java Debugger JDB, and the Apache web server, and finally we conclude with future directions.

## 2 APPROACH OVERVIEW

The overall scheme is shown in figure 1[2]. The goal is

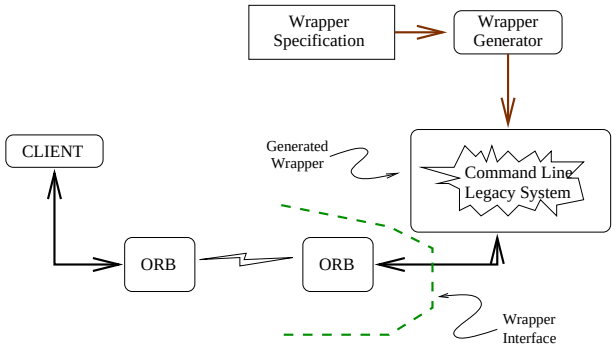


Figure 1: Overview of wrapper-generation framework. The legacy command line system is shown within a jagged boundary. The wrapper that surrounds it allows it to inter-operate with a CORBA ORB. The wrapper itself is generated from a specification in a high-level language. The main contribution of our work is the development of this language, and an appropriate architecture and run-time environment for its implementation. Some CORBA-specific details are omitted.

to place a wrapper around a command-line oriented $\mathcal{LS}$ (shown with the jagged boundary) and make it available for inter-operation as a CORBA object-implementation. This wrapper interacts with the CORBA ORB, accepting CORBA method calls, and forwarding them to the $\mathcal{LS}$ as ASCII commands. The ASCII stream responses from the $\mathcal{LS}$ are parsed, and the appropriate CORBA-compliant response is forwarded back to the ORB.

A wrapper for a command-line oriented system must deal with the following aspects:

1. the interactions between the wrapper and the other parts of the (standards-compliant) system (things that happen across the dotted line in Figure 1), including both the data-type aspects of the interaction (number and type of arguments in method invocations) and the pragmatics of the interaction, *e.g.,* the synchronization and direction of the interactions;

2. the "grammatical structure" of the strings accepted and generated by the $\mathcal{LS}$ *i.e.,* the strings exchanged between the wrapper and the $\mathcal{LS}$ at the jagged boundary.

We re-emphasize this point: *these issues pertain to any wrapper for a command-line $\mathcal{LS}$, and transcend any spe-*

[2]Certain CORBA-specific details, such as object adapters, are omitted here for simplicity

*cific inter-operability standard*, such as CORBA, Java RMI, or COM. Developing and implementing conceptual approaches to these issues is of central concern to our research. The wrapper specification language, and code-generation must deal with each of these aspects; we now discuss these issues, using the example of the Java debuggger JDB.

**Interface Description and Pragmatics**

The wrapper interface must support all and just those features of the $\mathcal{LS}$ that are to be made visible on the wrapper interface.

First, the number and types of arguments must be specified. For example, a `stop at` command in the JDB debugger must include a class name and line number to set break point. Likewise, a break-point hit message reported by the debugger will include the line number and class name. These aspects of the interactions are described using features in the interface description language (IDL) supported by the inter-operability standard. For example, CORBA IDL includes features for specifying the number and type of arguments to a method call.

Second, a command-line program includes several modes, or pragmatics, of interactions with users. Some commands, such as `classes` immediately print out the list of active classes in the run-time. These interactions can be naturally abstracted as *synchronous* procedure calls *inwards*, to the wrapper interface. In other interactions, the command-line program may ask the user for information; these might reasonably be abstracted as synchronous calls *outwards* through the wrapper interface. Other interactions are *asynchronous*: trace messages are freely generated by a command-line program without need for response. These must be accommodated by sending *asynchronous*, non-blocking messages *outwards* through the wrapper interface. *Deferred Synchronous* interactions are also possible— the command line program might accept a command from a user and respond much later; additional information in the response might provide the user with disambiguating context. All these types of interactions are allowed within CORBA, and must be supported by the wrapper interface, and the wrapper specification language must be able to describe them. Currently, CAWOM can handle inwards synchronous, asynchronous and deferred synchronous interactions, and outwards asynchronous messages on a designated multi-cast channel (CORBA event channel). In principle, given the generic architecture of CAWOM, we believe there is no hurdle to handling more general types of synchronization primitives, including the ones in the ADA language or in SR [1].

**Interaction Syntax**

The wrapper must send commands to the legacy systems, and also process the messages it generates.

The wrapper must generate the commands sent to the $\mathcal{LS}$, based on the data it receives through the wrapper interface. Generating strings from data can be viewed in general as unparsing, or pretty-printing [13, 29]. The wrapper specification must include high-level unparsing specifications to determine the generation of commands based on received data.

The wrapper must also process the messages generated by the $\mathcal{LS}$ and extract the semantics of the message and the contained data. This is certainly akin to parsing; however, in theory, an $\mathcal{LS}$, being an arbitrary program, can generate arbitrary strings. In general, an unrestricted grammar (*i.e.,* Type 0 grammar) would be required to specify this set of strings; the parsing problem would therefore be undecidable. However, from a pragmatic point of view, given a specific $\mathcal{LS}$ it may well be possible and desirable to implement a wrapper to process most (if not all) of the strings it generates. For greater flexibility, it would thus be undesirable for the parser to be limited to just context-free languages; we use definite-clause grammars [17] for additional expressiveness.

In the following section, we describe in greater detail how the design of CAWOM handles these issues.

## 3 DESIGN RATIONALE

In this section, we describe the design of the wrapper architecture, as well as the wrapper-specification language.

**Wrapper Architecture**

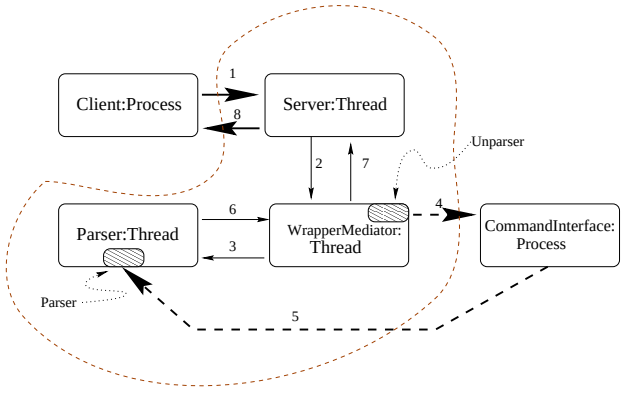


Figure 2: Details of the wrapper architecture, shown in the UML interaction diagram style. The "thread" within the dotted line belong to the wrapper. The "command interface process" box is the legacy system. The Client process is the CORBA client that accesses the $\mathcal{LS}$ through the wrapper. The interactions outside the wrapper are shown in heavy lines: the CORBA interactions are solid lines, and the command-line interactions are dashed. The *Server* thread interacts with the CORBA environment. The *Parser* thread listens to the $\mathcal{LS}$ systems output messages, parses and them and generates response events. The *Wrapper mediator* thread co-ordinates between the client requests received from the server thread, and the responses from the parser thread. The Parsing and unparsing components are shown in shaded rectangles. The numbers indicate the interaction sequence, described in more detail in the text.

A UML-style interaction diagram (Figure 2) describes the architecture of our generated wrappers. The wrapper consists of three Java threads, the Server, the Mediator and the Parser, all shown within the dotted line. The Client process, which interacts with the wrapper via the CORBA infra-structure (omitted in this diagram for simplicity) is shown outside the dotted lines. The Server thread handles interactions with the CORBA environment. The parser thread handles the grammatical complexities of analyzing and categorizing the responses from the $\mathcal{LS}$. The CORBA invocations that arrive at the server thread, and the responses that come back to the parser thread, can overlap in an arbitrary, concurrent fashion. The mediator thread co-ordinates between these two threads, and provides a cleaner separation of concerns between the parser and the server threads. The resulting reduction of coupling makes the entire wrapper structure easier to evolve, to handle other types of legacy systems and inter-operability standards. Sullivan *et al* [26] have described this type of mediation. This powerful modularity-enhancing technique has a natural application in CAWOMwrappers that must handle a variety of interactions between the wrapped system and the outside environment.

Specifically, our mediator thread has two main responsibilities. First, it handles the relatively simple task of generating command-line strings from the CORBA invocations. Second, before it sends out these command strings to the $\mathcal{LS}$, it also arranges with the parser thread (using states encapsulated in so-called *command objects*) to be notified when the parser receives and recognizes the associated responses from the command-line programs. CAWOM wrappers allow several requests to be outstanding, and allow overlap between synchronous, asynchronous, and deferred synchronous calls. The three threads in the wrapper manage multiple pending calls and events using synchronization objects that encapsulate the state of each request.

We clarify these functions with a scenario. The server thread gets a method invocation from the client through the CORBA environment (Sequence number 1). In response, the server thread creates a command object and forwards (2) it to the mediator thread. The mediator thread pretty-prints the command into an equivalent ASCII command-line. This command-line, together with a particular response from the $\mathcal{LS}$, may constitute a synchronous method call. Here, the mediator forwards a command object with information needed to find the matching response, and forwards it (3) to the parser, thus alerting the parser to be on the lookout for this response. It then sends (4) the command-line to the $\mathcal{LS}$. When the $\mathcal{LS}$ responds (5), and the parser recognizes this response as one associated with the command object in step 4, it first extracts required values from the response string (*e.g.* small`return`ed or `out` values, fields of exception structures, etc.). It then alerts the mediator thread (6), when then signals (7) the server thread that a response to the particular invocation initiated in step (1) is ready; the server then sends (8) it back to the client process. If there are multiple pending command objects, the parser matches the responses with the right command object, using the enscapsulated state information.

**Wrapper Implementation**

Our implementation exploits several off-the-shelf software tools. However, all the tools we use only require that the Java 1.2 run-time is available on the platform on which the wrapper is to run. Again, we emphasize that the principal concepts in CAWOM transcend language (Java) and inter-operability standard (CORBA). The wrapper-generator itself (shown in figure 1) is implemented with the Java version of the ANTLR parser-generator [16, 22]; GJ [4] is used for the parse-tree representation. We built a simple template-style macro language to facilitate programming the code-generation; future versions will use a more powerful hygienic macro system, such as JTS [3]. The generated code in the wrapper is all custom, except for the component that parses the the responses from the command-line $\mathcal{LS}$.

Parsing responses from the $\mathcal{LS}$ is the most complex and difficult aspect of wrapping. Indeed, the most complex code in the DDD [32] GUI-wrapper for legacy debuggers can be found in the part that parses responses from the debuggers; this code is large, intricate, and difficult to understand and maintain. . Our goal is to provide a response-parsing facility in CAWOM that is both simple and powerful. As discussed earlier, a "vanilla" context-free grammar cannot in general always manage the arbitrary strings that can be generated by a command-line program. So we use a definite-clause grammar (DCG) system, built around PrologCafe [2], a Prolog-to-Java translator. DCGs are powerful enough to express arbitrary grammars. Despite their expressive power, DCGs offer simple, intuitive style for writing many types of context-sensitive grammars, and have a proven track-record with complicated grammars, for example in natural-language processing. A key aspect of DCGs is an efficient implementation that exploits the built-in unification and back-tracking in Prolog. The PrologCafe system accepts a DCG-style grammar describing the responses from the $\mathcal{LS}$ and produces a parser in Java that incorporates the needed Prolog mechanisms. The presence of the Prolog run-time increases the footprint of our wrapper ; however, our wrappers demonstrate acceptable run-time performance in the examples we describe below. The use of DCG grammars trades off greater usability for better performance. However, a wrapper-builder has the option of building a custom

response-parser if so desired; she can still exploit the rest of the CAWOM infra-structure.

The infra-structure we have described above constitute the run-time environment of the generated wrappers in CAWOM. We now turn to the specification language that users would use to write the wrappers.

**Wrapper Specification Language**

There are two parts to a wrapper specification: the command-line interface specification and the command-line response grammar. The interface specification is written in `cIDL`[3], while the response grammar is written in `cRGL`[4]. The interface specification primarily describes the interface between the wrapper and the standard-compliant systems (again the dotted line in figure 1). An example `cIDL` specification can be found in figure 3. It's also used to define the unparsing or pretty-printing to be performed by the Mediator component of the wrapper (figure 2). The `cRGL` specification (See the example in figure 4) specifies just the response grammar $\mathcal{LS}$ for parsing responses from the $\mathcal{LS}$. We now describe these languages in more detail.

From figure 3, one can see that most of `cIDL`'s syntax and semantics are the same those of CORBA IDL; we clarify only the differences below (more information on CORBA IDL can be found [23]).

The first notable difference is the key word `command` preceding the otherwise typical interface definition on line 1. This indicates that the interface that is being defined is a wrapper for a command-line interface. Next, we move to the pairs of curly braces and the expressions they enclose (lines 5-10, 15, 19, 26 and 32). These specify the syntax of the command to be issued to the wrapped application whenever the corresponding operation is invoked. Finally, we note the `deferred` keyword on line 24 that qualifies the `gracefulRestart` operation. This qualifier specifies that `gracefulRestart` is an "deferred synchronous" call. With this type of CORBA call, a client invoking `gracefulRestart` does not wait for the server to finish and returns before continuing. The client may later poll to see if the server has completed the request and can retrieve any results if the server has completed.

The example does not show `push` operation qualifier. A push operation is essentially a call back the server can use to post information that the client did not request, but may be interested in. Such operations are typically used for servers to asynchronously notify a client when an interesting or unusual event has occurred.

[3] `cIDL` is CAWOM's IDL, and is an extension of CORBA's IDL. More details will be presented shortly.

[4] `cRGL` (CAWOM response grammar) is CAWOM's definite-clause grammar language based on PrologCafe [2].

```
1 command interface Server {
2   void start( in string serverRoot,
3.              in int mineSeervers )
4.              in int minServers )
5       {
6         "apachectl start " +
7         "-c \"MaxSpareServers " + maxServers + "\" " +
8         "-c \"MinSpareServers " + minServers + "\" " +
9         "-c \"ServerRoot " + serverRoot + "\"\n";
10      }
11      raises( CouldNotStartException,
12              ConfigurationSyntaxException ) ;
13
14  void stop()
15      { "apachectl stop\n"; }
15      raises( CouldNotStopException ) ;
17
18  void restart()
19      { "apachectl restart\n"; }
20      raises( CouldNotRestartException,
21              CouldNotStartException,
22              ConfigurationSyntaxException ) ;
23
24  deferred
25  void gracefulRestart()
26      { "apachectl graceful\n"; }
27      raises( CouldNotRestartException,
28              CouldNotStartException,
29              ConfigurationSyntaxException ) ;
30
31  boolean configTest()
32          { "apachectl configtest\n"; }
33          raises( ConfigurationSyntaxException ) ;
34  exception CouldNotStartServer {string msg};
35  exception CouldNotRestartServer {string msg};
36  exception ConfigSyntaxError {string msg};
37  exception CouldNotStopServer {string msg};
38};
```

Figure 3: A sample specification written in the CAWOM language ("`cIDL`"). Note the similarity to CORBA IDL, except for the added keywords such as `command`, and the additional syntax describing how to generate commands to the legacy system. The rest of the CAWOM language, for specifying how to parse the responses from the $\mathcal{LS}$ is shown in figure 4

Now we turn to figure 4, which shows an example of the `cRGL` response grammar specification which characterizes the response strings from the `apachectl` utility. Specifications in `cRGL` contain a grammar that prescribes how to parse the responses from the $\mathcal{LS}$, and also how to relate the responses to pending interactions with the CORBA world. For brevity, we only describe here the `cRGL` specification for ordinary (synchronous) CORBA calls that go in the wrapper which result in a command being executed synchronously by the $\mathcal{LS}$.

In response to an incoming synchronous request, the wrapper creates a "command" object encapsulating the state of the request. The wrapper then prepares a pending "command" object with the information required to process the expected response, and then unparses the request to generate a command to the $\mathcal{LS}$. The information in the pending command objects are then used by the parser thread in the wrapper to identify the matching response from the $\mathcal{LS}$. The `cRGL` grammar-bases parsing in allied with a command object; when certain non-terminals in this grammar are recognized, a

```
1 start → response,start.

2 response → "apachectl start:  ", startRule.
3 response → "apachectl stop:  ", stopRule.
4 response → "apachectl restart:  ", restartRule.
5 response → any, response.

6 startRule [command start] →
7           startedResponse.
8 startRule [command start] →
9           raise = notStartedResponse,
10

11 startedResponse →
12 "httpd started".
13 startedResponse →
14          "httpd (pid ",integer,") already running".

15 notStartedResponse [exception CouldNotStartServer] →
16          msg = couldNotStartServerMsg.
17 couldNotStartServerMsg [string] →
18          "httpd could not be started".

19 stopRule [command stop] → stoppedResponse.
20 stopRule [command stop] →
21          raise = notStoppedResponse.

22 stoppedResponse → "httpd stopped".
23 stoppedResponse →
24          "httpd (no pid file) not running".
25 stoppedResponse →
            "httpd (pid ",integer,"?)  not running".

26 notStoppedResponse [exception CouldNotStopServer] →
27          msg = couldNotStopServerMsg.
28 couldNotStopServerMsg [string] →
29          "httpd could not be stopped".

30 restartRule [command restart] → restartedResponse.
31 restartRule [command restart] → startedResponse.
32 restartRule [command restart] →
33          raise = notRestartedResponse.
34 restartRule [command restart] →
35          raise = notStartedResponse.
36 restartRule [command restart] →
37          raise = badSyntaxResponse.

38 restartedResponse → "httpd restarted".
39 notRestartedResponse [exception CouldNotRestartServer] →
40          msg = couldNotRestartServerMsg.
41 couldNotRestartServerMsg [string] →
42          "httpd could not be restarted".

43 badSyntaxResponse [exception ConfigSyntaxError] →
44          msg = configSyntaxErrorMsg.
45 configSyntaxErrorMsg [string] →
46          "configuration broken, ignoring restart"
```

Figure 4: Example of a CAWOM response-parsing specification. The specification relates the grammar for the responses expected from the legacy system to the returned values and exceptions in the methods of the interface defined for the wrapper; an example interface specification is given in figure 3

corresponding command-object gets return values and exceptions filled in from the elements of the recognized response. The parser thread then notifies the mediator and then the server thread, which then initiate the appropriate CORBA response. For example, on line 6 and 8, we have we have two possible rules for command objects relating to the `start` method. The rule on line 8 generates an exception, whereas the other one does not. These should be compared with the `start` method defined (lines 2-11) in figure 3.

The (simplified) grammar of a `cRGL` rule is as follows:

| | |
|---|---|
| *rule* | *:- nonterminal object-decl "→" body* |
| *object-decl* | *:- "[" type identifier "]"* |
| *type* | *:- "command"* or *"exception"* or *'struct"*or *"sequence"* or *primitive-types* |
| *primitive-types* | *:- long* or *boolean* or *string* *double* or *octet* or etc |
| *body* | *:- RHS body* |
| *RHS* | *:- nonterminal* or *Assignment* |
| *Assignment* | *:- field "=" value* |

The head of a rule identifies a non-terminal, and an affected object. The affected object may be a command (as on lines 6 and 8 in figure 4). The body has a list of non-terminals that have to be recognized for this rule, or some assignments that assign the recognized values of non-terminals to the fields in the affected object. For example, on line 9, the body of the rule for the `startRule` non-terminal attempts to recognize the non-terminal `nonStartedResponse` (see rule on line 15), and assigns the recognized string to the field `raise` on the `start` command object. This is a special reserved field name that indicates that an exception has been raised. The exception object itself is associated with the rule on line 15: the nonterminal `notStartedResponse` creates the exception object, which has a defined `msg` field (see line 34 in figure 4 for the definition of this exception in the interface specification) that is recognized by rule 17 as a `string`.

Methods can have `out` parameters specified in the interface, or have returned values. In such cases, the corresponding command object would be specified at the head of a rule; the body would assign values to the `out` parameters. If a returned value is being recognized in a response, then the reserved field name `returns` would be used to assign that value from the response. CORBA IDL also allows structures. In this case, the structure name would be associated with the `struct` keyword in the head of a corresponding rule, and the fields of the structures could be assigned in the body.

The description of `cIDL` and `cRGL` given above has been abbreviated, due to space constraints. From the `cIDL` and `cRGL` specifications, the CAWOM compiler generates

the code for the components of the wrapper.

## 4 EVALUATION

In this section, we describe the application of the CAWOM framework to generating wrappers for the Apache server, and also for JDB. We also place our work in the context of the existing literature.

### WRAPPING APACHE

In one experiment, we specified and generated a wrapper for Apache [30] that enables certain administrative functions of the Apache server to be accessed programmatically from a CORBA client. In addition to the normal benefits of CORBA componentization (remote access, type safety, leverage of other CORBA assets) it is also important to note that the full power of the CORBA security framework can be used for fine-grained access-control over these administrative functions.

`apachectl` is a UNIX shell script that enables an administrator to start, stop, and restart a web-server, and perform several status checks. It also supports a "graceful" restart which (unlike a a normal restart) allows web servers to complete outstanding requests before restarting them. The graceful restart is a deferred synchronous call. We describe here the implementation of a CAWOM wrapper for `apachectl`.

First, we define the `apachectl`'s command-line interface in `cIDL`. This specification is shown in figures 3. From this interface description, CAWOM will be able to derive a CORBA IDL and the part of the wrapper that maps incoming CORBA requests into commands for the shell.

The string expression in quotations following an operation's signature is the command issued to the shell when the operation is invoked. Thus, when the `start()` method is invoked, the command "`apachectl start\n`" will be issued to the shell (see lines 2 through 11) The erroneous response than can occur with `apachectl` are shown with exceptions. Second, we write a grammar for the command-line responses in `cRGL`; this is shown in figure 4. Using the grammar, CAWOM will produce the part of the wrapper that interprets the command-line results and maps these results into the output parameters of the CORBA request.

Once defined, the interface description and response grammar are fed to CAWOM, which produces the CORBA wrapper. The wrapper consists of CORBA stubs for clients, and a fully implemented CORBA server. The server can be started and ready to serve immediately. Clients can then be implemented using the stubs to issue commands to `apachectl` via CORBA. It's interesting to note that the entire wrapper specification is only about 80 lines long. CAWOM generates about 1,000 lines of Java code that implements all the details of the wrapper. The performance was satisfactory. We measured the end-to-end performance of the wrapper: this includes the time from when the wrapper receives a CORBA invocation to the time the corresponding command-line string is sent to the $\mathcal{LS}$, and the time from when the corresponding response is received from the $\mathcal{LS}$ and the response is initiated back to the CORBA environment. In all cases, for all methods, we found the end-to-end times to be less than 4 milliseconds. However, since the resolution of the clock itself is only a millisecond, we can only say that the wrapper provides an acceptable overhead in most situations where distributed objects are used, and networking delays are involved. Measurements were performed on a 366 MHz Sun Solaris Ultra-10 machine, with 256MB of main memory and 1 MB cache.

### WRAPPING JDB

We have also implemented a CAWOM wrapper for JDB. This was a more complex wrapper than `apachectl` example discussed above, and used more of the features in the specification languages. In this section, we describe some of the highlights of this wrapper specification, and evaluate its effectiveness. We assume some basic familiarity with the Java language and run-time.

The JDB debugger supports many useful commands, some of which are simple and synchronous, like the `print` command which prints values of variables. Other commands, like `stop at`, which sets break-points in a Java class source file, are more complex. If the class named in the `stop at` command is loaded, JDB returns promptly with a confirmation. If the class is not yet loaded, the JDB returns promptly, but with a message confirming that the break-point will be set when the class is loaded. When the class *is* finally loaded, JDB pipes up with a message saying it has complied with the earlier `stop at` request. This command is handled in our wrapper with a pair of methods in the `cIDL` interface specification. We show a relevant excerpt below:

```
1 booleanstopat( in string cls, in long line )
2        {
3          "stop at "+cls+":"+line+"\n";
4        } raises( StopUsageException );

5 push void setStopAt(in string where);
```

The first method (line 1 above) sets the breakpoint. The two `in` arguments supply the class name and the line number; the unparsing instructions on line 3 specify how to generate the $\mathcal{LS}$ command. An exception is possible, in case the class name is improper; in this case, the `StopUsageException` might be raised. The second method uses the `push` keyword which we have not yet discussed. This keyword marks a method which is *outbound*, and

signifies an asynchronous event that is sent out from the wrapper to a designated event channel (using the CORBA Event Service [15]) that can be monitored by a CORBA client.

The description of the response-parsing associated with the synchronous `stopat` method is handled in the `cRGL` specification is similar to the other methods described in `apachectl` wrapper specification. The `cRGL` rule for the `push` method (`setStopat`) is given below:

```
6 setStopAt [command setStopAt]
7           → "Set deferred breakpoint ",
            where=atTail.
8 atTail [string]
9           → qualified,":",integer.
```

When a class targeted by a `stopat` command is finally loaded, the JDB sets the break-point and generates a message of the form,

```
Set deferred breakpoint late.coming.class:23.
```

This response is parsed by the rule at line 6 above, and the asynchronous event push is generated.

Other interesting methods in the `cIDL` specification (not shown) include `locals`, corresponding to the `locals` command in JDB that prints out a list of values of local variables, and a list of values of arguments. These lists get parsed by the parser thread in the wrapper and returned as two separate CORBA `sequences` (CORBA IDL supports a `sequence` abstract datatype) of type `Variable`, which is defined in the `cIDL` specification as a `struct` consisting of 3 elements, representing the name, type and value of each variable (all CORBA `strings`). The `cIDL` specification handles these in similar way to normal CORBA IDL. The `cRGL` specification has features to parse lists and fill them into CORBA sequences; it also includes ways of recognizing fields of `structs` and filling them in during parsing.

For most simple commands such as setting breakpoints, the wrapper end-to-end overhead was on the order of 20 ms; more complex commands, like `print` which require more extensive parsing, are on the order of 40 ms. We expect these times (and the ones for `apachectl` to be considerably faster with better-tuned commercial implementations such as BinProlog[5].

It is reasonable to compare our wrapper implementation to that of DDD [32], which wraps JDB to provide a more pleasant user experience via a GUI. The code in DDD pertaining to wrapping JDB is about an order of magnitude larger that the `cIDL` and `cRGL` specifications for the CAWOM wrapper. The ad-hoc response-parsing code in DDD is also much more complex, intricate and difficult to understand. The performance of this wrapper is also quite reasonable. Of course, a CORBA-compliant wrapper, which can inter-operate with other CORBA components and services, is arguably a far better vehicle for adding value to JDB than a custom wrapper for a specific purpose (*i.e.,* a user interface). The JPDA API [14] enables programmatic control of the Java debugger; however, it only allows access to JDB functionality from other Java programs; a CAWOM wrapper, enables access via the open CORBA standard to any command-line system[6]

[5]Please see `http://www.binnet.com`

## RELATED WORK

In this section, we survey related work in the area of wrappers[7]. The fundamental goal of a wrapper is to intervene between two different systems, hiding the details of one from the other. The structural role a wrapper plays is in the same spirit as the ADAPTER or FACADE patterns in object-oriented programming [10] although the implementation details vary: our work can be described as generating wrappers for command-line systems from a specification in a high-level language (based on enhanced IDL). We describe work that relates to ours, first in terms of the work's goals (adaptation) and then in terms of the implementation technique used (generation from specifications, specially based on enhanced IDLs).

Of most immediate relevance, Javamatic [18] is a wrapper that makes the services of command-line $\mathcal{LS}$ systems available via a Java applet. No details of the architecture or the specification language are available in [18], and the authors have stated that the work has been abandoned. Rather providing GUI access, CAWOM wrappers allow inter-operation with the $\mathcal{LS}$. Previous work in software engineering has attacked the problem of adaptation. Purtilo and Atlee [19], described a technique to adapt interfaces into a more suitable form. Yellin and Strom [31] describe a technique for mediating between two subsystems that conform to different protocols of interaction.

Wrapper technology has frequently been employed in the domain of security. Wrappers in the security domain are used to either monitor or restrict the behavior of applications. Naccio [8] uses wrappers to enforce security policies on a particular platform. Fraser [9] defines an architecture for wrappers to either enforce or monitor applications on a given platform. Macoridis [24] describes wrappers that mediate between different security architectures (*e.g.* Unix and CORBA).

There has also been considerable work in the database community on wrapper generation. Garlic [20] and

[6]We are currently working on wrapping GDB using CAWOM.

[7]Due to space considerations, our survey of related work is representative, rather than comprehensive.

TSIMMIS [6] are systems that integrate heterogeneous data sources using a mediator based approach. Wrappers (called translators here) are used to provide a common query interface on top of the existing source interfaces. In this way, sources are adapted to a global query environment. Gruser et. al. [11] similarly describe a toolkit for refitting, via wrappers, web-based data sources with JDBC compliant interfaces. Sahuguet and Azavant [21] developed W4F, a GUI for semi-automated creation of wrappers for web-based data sources. PrismTech in their OpenSynergy software suite offer OpenMigrator [27], a wrapper based technology for supporting data migration. The goal in databases, has been translating data between different data models, query interfaces, or views. CAWOM-generated wrappers actually mediate between different execution environments—one based on CORBA, and the other on ASCII command streams.

There has been quite a bit of work on enhancing IDLs to introduce additional functionality. Flick [7] is a modular IDL compiler supporting multiple IDLs and language mappings. Flick's modular design lends it for use in custom IDL languages and code generation; thus allowing wrapper code to be generated as part of the product of an IDL compiler. Sterne *et al* [25] embed DTEL++ (domain type enforcement language for the object oriented model) into CORBA IDL, thusly providing access control for CORBA objects. Brose [5] describes how access control for CORBA objects can be achieved by introducing the notion of views into CORBA IDL. Hence, Koch and Kramer [12] demonstrate how concurrency controls for CORBA objects can be achieved by making synchronization assertions in an IDL. Our work adopts the same general approach but with the specific goal of wrapping command-line systems.

**Limitations**

CAWOM has several limitations. Some of these are the subject of ongoing work; however, others are inherent to the problem setting and/or our design approach.

Currently, CAWOM assumes that the $\mathcal{LS}$ reads commands from and writes responses to the same stream channel. It's fairly straighforward to generalize this so that the wrapper can feed commands to one of several streams and also read responses from several channels. CAWOM wrappers can now handle all three types of interactions (synchronous, asynchronous, and deferred synchronous) going *into* the wrapper; however, CAWOM wrappers can only generate asynchronous events, and only to fixed event channel, going outward. Again, extending CAWOM to handle all three types of interactions into and out from the wrapper would be fairly straightforward within our general framework. The above two limitations are being addressed in current work.

Other limitations of CAWOM are problems inherent to the particular legacy setting (command-line systems) and also the design choices we have made. Thus, some command-line systems may have response languages that are too complex to parse even with DCGs. Still other languages may be parseable, but may require much expensive backtracking before a successful parse can be found. In such cases, the most viable choice may be to hand-craft a custom, heuristic (but perhaps incomplete) parser that works efficiently most of the time.

Finally, there are many legacy systems that are not command-line programs. We would like to provide wrapper-generators that help in more settings; but this remains an open problem.

## 5 CONCLUSION

Legacy systems continue to provide critical business functions in many contexts, but do not inter-operate well with more modern and standards-compliant systems. The difficulty of re-engineering these systems has encouraged developers to wrap them for inter-operability. Our goal is to simplify the construction of wrappers. We focus specifically on command-line oriented legacy systems, which are very common, and try to automate away some of the drudgery of building wrappers for such systems. CAWOM is a tool that generates wrappers for command-line systems from a high-level specification. We describe its design and implementation, and evaluate it with two examples, JDB and Apache.

We continue to refine CAWOM's features, and are testing it on other legacy applications. Up-to-date information on CAWOM can be found at `http://castle.cs.ucdavis.edu`. Our long term goal is to build a modular suite of wrapping tools for a variety of legacy systems and inter-operability standards.

## ACKNOWLEDGEMENTS

We gratefully acknowledge support from the National Science Foundation (SGER Grant #9985560), without whose generosity this work would not have been possible.

## REFERENCES

[1] G. Andrews and R. Olsson. *The SR Programming Language—Concurrency in Practice*. Benjamin-Cummings, 1993.

[2] M. Banbara and N. Tamura. Translating a linear logic programming language into java. In *Proceedings of ICLP'99 Workshop*, 1999.

[3] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages.

In *Proceedings, 5th International Conference on Software Reuse*, June 1998.

[4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *OOPSLA*, 1998.

[5] G. Brose. Towards an access control policy specification language for corba. In *ECOOP EWDOS*, 1998.

[6] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of Heterogeneous Information sources. In *IPSJ Conference*, 1994.

[7] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A flexible, optimizing idl compiler. In *ACM SIGPLAN PLDI*, 1997.

[8] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, 1999.

[9] T. Fraser, L. Badger, and M. Feldman. Hardening cots software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, 1999.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[11] J. Gruser, L. Raschid, M. Vidal, and L. Bright. Wrapper generation for web accessible data sources. In *CoopIS*, 1998.

[12] G. Henze, T. Koch, and B. Kramer. Annotations for synchronization constraints in corba idl. In *IEEE SDNE*, 1996.

[13] J. Hughes. The Design of a pretty-printer Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer-Verlag, 1995.

[14] Javasoft Inc. JPDA API for Java debugging, 2000. `http://java.sun.com/products/jpda/faq.html`.

[15] OMG. The SECURITY service `http://www.omg.org/technology/documents-formal/event-service.htm`, 1995.

[16] T. Parr and R. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, July 1995.

[17] F. Pereira. Definite-Clause Grammars. *Artificial Intelligence*, 1980.

[18] C. Phanouriou and M. Abrams. Transforming command-line driven programs into web applications. In *Proceedings, Sixth WWW Conference*, 1997.

[19] J. M. Purtilo and J. M. Atlee. Module reuse by interface adaptation. *Software Practice and Experience*, 21(6), 1991.

[20] M. T. Roth, M. Arya, L. M. Haas, M. J. Carey, W. Cody, R. Fagin, P. M. Schwarz, J. Thomas, and E. L. Wimmers. The Garlic Project. *ACM SIGMOD Record*, 1996.

[21] A. Sahuguet and F. Azavant. WysiWyg Web Wrapper Factory (w4f). In *WWW Conference*, 1999.

[22] G. L. Schaps. Compiler construction with antlr and java. *Dr. Dobb's Journal*, March 1999.

[23] J. Siegel. *CORBA-3 Fundamentals and Programming*. John Wiley and Sons, 2000.

[24] T. S. Souder and S. Mancoridis. A tool for securely integrating legacy systems into a distributed environment. In *Working Conference on Reverse Engineering (WCRE)*, Atlanta, GA, October 1999.

[25] D. F. Sterne, G. W. Tally, C. D. McDonell, D. L. Sherman, D. L. Sames, P. X. Pasturel, and E. J. Sebes. Scalable access control for distributed object systems. In *USENIX Security Symposium*, 1999.

[26] K. J. Sullivan, I. Kalet, and D. Notkin. Evaluating the mediator method: Prism as a case study. *IEEE Transactions on Software Engineering*, 22(8), August 1996.

[27] S. Trythall. Solving the data migration problem: Openmigrator, September 1999. PrismTech: www.prisimtechnologies.com.

[28] V. Varadharajan and T. Hardjono. Security model for distributed object framework and its applicability to CORBA. In *Proceedings of the 12th International Information Security Conference IFIP SEC'96*, May 1996.

[29] P. Wadler. A Prettier Printer. *Journal of Functional Programming*, 1999.

[30] *The*. APACHE. *website*. `http://www.apache.org`.

[31] D. M. Yellin and R. E. Storm. Protocol specifications and component adaptors. *ACM TOPLAS*, 19(2), 1997.

[32] A. Zeller and D. Luetkehaus. DDD - a Free Graphical Front-End for UNIX Debuggers. *ACM SIGPLAN Notices*, January 1996.