Evolving Software with an Application-Specific Language

Eddie Kohler, Massimiliano Poletto, and David R. Montgomery
{eddietwo, maxp, dmontgom}@lcs.mit.edu
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

Abstract

Software systems can be developed through evolution (gradual change) or revolution (reimplementation from scratch). Both approaches have advantages and disadvantages. An evolutionary approach keeps the system working throughout, allowing early problem detection, but tends to retain ingrained design flaws and can result in complex, ad hoc systems. A revolutionary approach is required to change the basic architecture of a system, but many more resources must be invested before the system can be evaluated. In this paper, we describe how we used a little application-specific language to combine these approaches' advantages.

The context of our work is CTAS [2], the next-generation air traffic control automation system developed originally by NASA. The overall goal was to redesign and reimplement one of the CTAS processes in Java, while retaining its ability to communicate with unmodified processes—a project complicated by CTAS's ad hoc message formats. To address this, we designed a language that combines C code copied from CTAS source, to express the message formats, with new Java code for message actions. A compiler then automatically generates code for marshalling and unmarshalling. The result is a system with both evolutionary and revolutionary properties, exemplified by the use of both old CTAS code and new Java code in the message language.

This paper discusses the language and compiler and evaluates some of the engineering tradeoffs inherent in their design.

1 Background and introduction

CTAS, the Center-TRACON Automation System, is a next-generation air traffic control system designed at the NASA Ames Research Center under the direction of Dr. Heinz Erzberger [2]. The core of CTAS is a sophisticated expert system that optimizes the flow of airplanes into an airport. A CTAS installation consists of tens of Solaris workstations, each running one or more processes: CPU-intensive route analyzers and dynamic planners; graphical user interfaces, which allow air traffic controllers to examine traffic and evaluate CTAS recommendations; input managers, for radar feeds and the like; and one Communications Manager (CM), the central communications hub of CTAS. The CM's main function is to mediate all process-to-process communication in CTAS, relieving processes of the need to keep track of each other.

Although the CM is central to CTAS, NASA research focused on air traffic control algorithms rather than software development methodology. As a result, the original CM became increasingly complicated and difficult to maintain. Core scheduling functionality was tied to a debugging GUI, low-level network code was spread throughout the system, and the CM's high-level message-passing structure was obscured by debugging code and other features. The goal of the work described in this paper (part of a project for MIT course 6.894, "Workshop in Software Engineering," taught by Daniel Jackson in the fall of 1998) was to redesign and reimplement the 70000-line CM in Java, using modern object-oriented design techniques, so as to make it more robust and maintainable.

The redesign of the CM explicitly combined evolutionary and revolutionary approaches to software change. While our approach to the CM was revolutionary—reimplementation from scratch in a very different language environment—our approach to CTAS as a whole was evolutionary: the new CM had to be compatible with unchanged processes from the rest of CTAS. These contrasting requirements caused some of the hardest engineering problems in the redesign. This paper focuses specifically on one of these problems: the CM's message handling code.

The CM must understand every message that any process in the system might send. There are about 350

Max Poletto was supported by an IBM Cooperative Fellowship. Eddie Kohler was supported by a National Science Foundation Graduate Research Fellowship and DARPA.

messages in total, and each is sent in one of around 200 message formats. (Even the minimum functionality implemented in the prototype Java CM required understanding 30 messages in 20 different formats.) Unfortunately, CTAS does not use any existing library, like XDR/Sun RPC, RMI, CORBA, DCOM, or PVM, to implement message passing. Instead, CTAS messages are just C structs directly copied onto the wire, each with a header giving message ID and length. This non-standard format complicated the design by making it impossible to find an existing Java library to facilitate marshalling and unmarshalling. Writing marshalling and unmarshalling code by hand was unacceptable: it would have been tedious, error-prone, and hard to scale up to all 350 messages.

The solution that we present in this paper is a *message language* that abstracts away the marshalling and unmarshalling code. This language includes the subset of C for defining types (structure definitions, typedefs, and enumerations), allowing us to cut-and-paste message structure definitions directly from CTAS header files. It also includes message handlers written in Java, letting the programmer plug messages directly into the new Java CM. Thus, the language neatly bridges the two methodologies for software change—evolutionary and revolutionary—by combining old and new CM. The result is a flexible system, with considerable readability advantages, that let us completely redesign the CTAS CM without worrying about message formats. Encapsulating both old and new in a single application-specific language also makes both kinds of changes easier in the future: an evolutionary change requires less work (the application-specific language handles some of it for you), and for some revolutionary changes, only the compiler needs to be changed.

This paper discusses the language and its compiler in some detail, focusing particularly on how it combines evolutionary and revolutionary approaches to software change. We provide an extended evaluation of the language and some alternatives from several points of view, including engineering tradeoffs, implementation cost, and overall developer productivity. Finally, we discuss how the language evolved and why, and summarize some general lessons distilled from our experience.

2 Related work

This paper mainly concerns two research areas, the specification of message formats for communication among processes and the use of domain-specific languages for software engineering.

2.1 Message formats

Courier [1], part of the Xerox Network Systems (XNS) architecture of the 1970s, was one of the first languages designed to specify a machine-independent encoding of data structures as well as tools to produce those encodings from machine-specific data representations. Several data-encoding standards have been developed since then, among them Sun Microsystems' External Data Representation (XDR) [7] and the ISO Open System Interconnection (OSI) architecture languages. OSI uses a language called ASN.1 [9] to specify abstract objects, and several sets of rules, among them BER [10], to specify their wire encoding. ASN.1 is a powerful and complex notation; like the CTAS message format, and unlike any other popular data-encoding standards, it allows encodings in which the value of a variable can affect the data layout.

The Universal Stub Compiler (USC) [8] is a generic stub compiler that emits code to translate between two user-specified formats, rather than between a user-specified encoding and a fixed wire format such as XDR. Our CM message compiler has a similar philosophy, but translates between C and Java message representations, rather than between two different C representations.

Had we decided to rewrite the entire CTAS system in Java, an alternative would have been to use Java RMI, a standard for transferring arbitrary objects between Java processes. RMI would not have been very practical when rewriting only the CM, because it would have required CTAS-to-Java proxies between the CM and all other processes. Given the possibility to rewrite or significantly modify much of CTAS, one could also replace the custom message system with standard RPC systems such as Sun RPC or OSF/DCE, or with an object-oriented RPC system such as CORBA or DCOM. A group at Lincoln Labs [6] made substantial changes to CTAS and improved its reliability by replacing its message system with the Parallel Virtual Machine [12], a software package generally used to simulate a large parallel computer on a collection of networked computers.

2.2 Domain-specific languages

Spinellis and Guruprasad [11] survey common applications of domain-specific languages (DSLs) in software engineering, provide interesting examples of their use, and suggest implementation techniques. They emphasize that DSLs should make the development process easier and more efficient, and that therefore the effort spent to develop a DSL and an application must be less than that required to build the application without using the DSL.

While we have not investigated it in detail, the FAST (Family-oriented Abstraction, Specification, and Trans-

lation) process developed at Bell Labs appears to exemplify the potential of little languages. Weiss [14] reports three-fold productivity increases when using this process. FAST languages differ from our language in that they are designed for use in an entire domain; we discuss a language targeted to a single application.

Van Deursen and Klint [13] acknowledge the utility of DSLs in software engineering, and cite several examples of their benefits. However, they warn that DSLs may increase the difficulty of making large changes to and maintaining software. In our case, the message language helped us radically redesign the CM, and it would probably be useful when making changes to other CTAS processes. Admittedly, long-term maintenance of a compiler as part of a large software project might be costlier than a more straightforward implementation.

Hook and Walton [3] describe a system superficially similar to CTAS that also uses a domain-specific language to specify message handling. Their language appears to be much more formal and heavyweight than ours.

3 The language

The message language concisely and abstractly expresses the CM's messaging functions, eliminating the low-level network issues, debugging code, and other detritus that cluttered the old CM. As discussed above, the language divides into two sections: a subset of C for specifying message formats, and a superset of Java for specifying message actions. The message format section consists of C statements for defining structures, enumerations, and typedefs, allowing us to use the original message definitions from CTAS header files unchanged. The message action section defines how the CM receives, sends, and forwards messages to and from other CTAS processes. Each message is associated with some *message handlers*—code fragments, written in a Yacc-like extension of Java, that specify computations to be performed when that message is sent or received. The compiler then desugars the message handlers into standard Java, taking care of lower-level networking issues. This design allows the user to focus on the general messaging structure.

The remainder of this section discusses message handlers and an extension for variable-length messages, closing with a description of the code generated by the message language compiler.

3.1 Message handlers

The message handlers for a given message type are grouped into *message blocks*, introduced by '`mesg`', which also associates the message type (generally an enumerated constant) with its corresponding wire format (a C structure). The actual message handlers are introduced by '`send`' or '`recv`', depending on whether the handler should be executed when sending or receiving a message. Different actions can be specified depending on the type of the message's source or destination process.

The code used to specify these actions is Java, slightly extended to make message handling easier. The Java code is not fully parsed by the compiler, which just syntactically desugars the extensions, leaving the main body of each action untouched. This design owes a lot to Yacc actions [4]. The particular extensions are as follows:

- Field names from the wire format structure can be used as variables in an action to refer to those fields in the actual message. Thus, if a message has a field named `x`, code like '`x = 3`' can be used in a `send` block to set that field in an outgoing message, and '`x`' can be used in a `recv` block to refer to that field's value in an incoming message.
- The keywords '`send`' and '`forward`' simplify two frequent operations: sending a new message to other processes and forwarding a received message to other processes.
- The name '`mesg`' can be used to access the current message object. In a `recv` block, the name '`remote`' can be used to access the remote process which sent the message.
- Several special names (`TGUI`, `PGUI`, `ISM`, etc.) represent classes of CTAS processes. For instance, one can say '`forward(PGUI|TGUI)`' to forward the current message to all PGUI and TGUI processes.
- The special expression `ac[`*`id`*`]` denotes the aircraft object in the CM's aircraft database corresponding to a given aircraft ID.

Figure 1 shows how this fits together. The structure declaration defines a simple wire format; the enum defines the IDs for two messages. We'll describe the handler for the first of these messages in detail. The `PINGPONG` message block associates the ID `PINGPONG` with the wire format `wire_st`, and contains one `send` handler and one `recv` handler. The `recv` handler is simple: when the CM receives a `PINGPONG` message from a PGUI process, it forwards it on to all PGUIs. The `send` handler specifies how the CM should react to an explicit request to send a `PINGPONG` message. This one tells the CM to send only to PGUI and TGUI processes; specifies that the `send_PINGPONG` method takes one extra argument, an integer; and tells the CM to set the message's `wireval` field from that argument.

```
typedef struct {
  int wireval;
} wire_st;
enum { PINGPONG, TRIGGER };

mesg PINGPONG wire_st {
  recv PGUI { forward(PGUI); }
  send PGUI|TGUI(int v) { wireval = v; }
}
mesg TRIGGER wire_st {
  recv TGUI {
    send(PINGPONG, PGUI|TGUI, wireval);
    remote.send(mesg);
  }
}
```

Figure 1: Sample message handlers for a fictional CM. When this CM receives a `PINGPONG` message from a PGUI process, it resends that message to all PGUIs. Receiving a `TRIGGER` message from a TGUI starts the pingpong.

3.2 Variable-length messages

The message language actually accepts a superset of C's types in order to support variable-length messages. Some CTAS messages contain variable-length arrays of data, preceded and followed by fixed-length elements: for example, an integer number of aircraft, followed by that number of `aircraft_info` structures, followed by another integer. This is impossible to describe in C structures alone.

The message language uses elegant syntax to express such constraints. The example above would look like this:

```
struct {
  int num_aircraft;
  aircraft_info a[num_aircraft];
  int more_data;
} wire_st2;
```

Here, the number of elements in `a` is explicitly equal to the value of the `num_aircraft` field.

The compiler will require that the `num_aircraft` field be set in every `send` handler using this wire format so that the correct message length can be determined. Furthermore, it correctly calculates the byte offset to `more_data` by including a variable component based on `num_aircraft`. This feature eliminates a difficult marshalling task that was the potential source of many errors.

```
public class MessageProcessor
             extends engine.MessageProcessor {
  public static final int PINGPONG = 0;
  public static final int TRIGGER = 1;
  ...
  private void forward(Message mesg, ClientFilter f) {
    clientGroup.send(mesg, f);
  }
  public void processMessage(Message mesg, Client c) {
    try {
      switch (mesg.getType()) {
      case PINGPONG:
        receive_PINGPONG(mesg, c);
        break;
      case TRIGGER:
        receive_TRIGGER(mesg, c);
        break;
      default: break;
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  private void receive_PINGPONG(Message mesg,
                                Client remote) {
    switch (remote.getProcType()) {
    case ProcType.PROC_PGUI: {
      forward(mesg, PGUI_filter);
      break;
    }
    default: break;
    }
  }
  private void receive_TRIGGER(Message mesg,
                               Client remote) {
    switch (remote.getProcType()) {
    case ProcType.PROC_TGUI: {
      send_PINGPONG(PGUI_TGUI_filter, mesg.getIntAt(0));
      remote.send(mesg);
      break;
    }
    default: break;
    }
  }
  public void send_PINGPONG(ClientFilter filt, int v) {
    try {
      Message mesg = new Message(PINGPONG, 4);
      mesg.putIntAt(v, 0);
      clientGroup.send(mesg,
            ClientFilter.and(filt, PGUI_TGUI_filter));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

Figure 2: Compiler output for the sample message handler.

3.3 Compiler output

The compiler takes the wire format definitions and message handlers and generates a single Java "message processor" class. Generally, two methods are created for each message type: `send_`*message-name* and `receive_`*message-name*. These methods contain desugared versions of each of the message's `send` and `recv` handlers. As part of the desugaring, names of message fields compile to methods that read or write data at appropriate offsets in the byte array representing the message. Other language features, like process identifiers, `mesg`, and `send`, are desugared into more or less simple Java code. Finally, run-time errors are handled elegantly with Java exceptions. Figure 2 shows most of the code generated by the handlers defined in Figure 1. The code makes use of some Java CM abstractions, such as `ClientFilter` and `ClientGroup`, that are beyond the scope of this paper, but it should be relatively easy to read.

In addition, the message compiler can generate Java classes corresponding to C structures. These classes know how to unmarshal themselves from a message object and how to marshal themselves into a message object starting at a given offset, and contain public fields that match all the fields of the corresponding C structure.

4 The compiler

This section describes the compiler for the CM message language. We begin with a quick overview of its structure, then give a detailed description of reparsing, an interesting compiler technique.

4.1 General structure

The compiler consists of about 10,000 lines of C++ code. Much of this code, however, is either unmodified library code or code originally written for another compiler project. Specifically, 2000 lines of code came directly from libraries, and 2400 more were copied, with modifications, from the Prolac compiler [5]. (Some of these modifications were fed back into the original source.) Of the remaining code, a large fraction—at least 1600 lines—is not specific to the CM message language and could be reused in another project.

Compiler processing is divided into three phases: parsing, resolution, and output. In the first phase, the compiler reads and parses all its input files. Unlike C compilers, it does not immediately resolve each object it sees; instead, to allow for order independence, it creates a simple intermediate form. In the second stage, resolution, C declarations are resolved, undefined types are reported, message blocks are merged together, send blocks are checked to make sure they set any required fields, and so forth. In the final stage, the compiler creates its output Java files, including the main message processor class and any exported marshalling classes.

Most of the compiler's work consists of intermediate form transformations completed in the second phase, resolution. In an architecture based on the Visitor pattern and experience with functional languages, each transformation is modeled as a separate transformer class. Various transformers detect and report errors, collect information, create new expression trees, and so forth; there are 13 transformers in all, which together implement a large part of the compiler's functionality.

The compiler understands most C types' size and alignment, and can mimic conventional C structure layout, including any holes required to maintain alignment. Member offset calculations are complicated by variable-length structures: the offset to one field may depend on the value of some other length field. Our solution was simply to return a general expression tree, instead of a simple integer, as the result of any member offset calculation.

4.2 Parsing message handlers

One interesting parsing wrinkle is caused by message handlers. As discussed above, message handlers resemble actions in a Yacc grammar: both consist of uninterpreted code with some strings specially handled by the language tool. The difference is that Yacc's desugaring is much easier; it simply replaces strings like '`$1`' with strings like '`yyvsp[0].str`'. In contrast, the message compiler must desugar field references and field assignments to different method calls, although both are signalled by the same text (a field name). It therefore must consider expression context and even precedence before deciding what replacement text to use: a field name to the left of an equals sign will generate quite different code than the same field name on the right. For example, assume that `f` and `g` are integer fields starting at byte offsets 12 and 16 in the message structure. These message language fragments would then be desugared to the corresponding Java fragments:

Message language	**Java desugaring**
`int x = f + 2;`	`int x = mesg.getIntAt(12) + 2;`
`f = x + 2;`	`mesg.putIntAt(x + 2, 12);`
`f = g = 97;`	`mesg.putIntAt` `(mesg.putIntAt(97, 16), 12);`

It seems clear that we must parse Java expressions to generate correct code. However, a full Java parser certainly seems like overkill.

Our solution involves *reparsing*. When first encoun-

tered in the message language source, a message handler is stored as a stream of bytes (comments, whitespace and all). During the second phase of compilation, each message handler is sent back through the lexer, creating a new stream of tokens; the compiler then reads expressions from this token stream. If a token is not part of any expression, the compiler simply stores the token verbatim. Thus, each Java action has been parsed, without too much extra work, into a list of tokens and expression trees, where the expression trees automatically contain context and precedence. The necessary desugarings can then be implemented just by transforming the expression trees.

5 Evaluation

5.1 The project as a whole

One way to evaluate the message language is to ask whether the whole Java CM project was successful. The redesigned CM was completed on time, just one month after submission of the final design proposal, and it successfully drove a subset of the CTAS system from traces provided by NASA. We believe that the message language played an important part in this success—not only because some message subsystem was necessary, but also because of unique properties of a language-based solution:

- Our language design allowed us to copy code directly from the CTAS headers with at most a couple of changes, reducing human error. In addition, error-prone marshalling and unmarshalling code was machine-generated.

- The Java-related part of the message language was enough like Java that other students could write it directly. We could have designed a simpler input language, which would have made the compiler simpler, but such a language would probably have had less functionality and been harder to understand.

- The compiler's code base was completely separate from the rest of the Java CM. Therefore, we could work on it earlier, in our own development environment, without developer communication overhead. Furthermore, we could test it apart from the Java CM, which, as part of CTAS, required a complex environment to run—and didn't work until far into the project. Having one subsystem tested and working simplified debugging the rest of the system.

5.2 Measuring developer productivity

Spinellis and Guruprasad [11] convincingly argue that a lightweight language is most useful as a software engineering tool when it increases developer productivity, and therefore decreases total development and maintenance costs—where this total includes the cost of developing the language. If one measures cost only in terms of lines of code, the message language is a failure. The compiler and its input file together take about 12000 lines of code, while the compiler's output is about 2000 lines of Java code. This is a factor of 6:1 against the language. Even not considering the compiler, the ratio of message language to output is a disappointing 1:1.1.

However, these numbers are deceiving. First, about 1450 lines of the message language file consist of C definitions lifted from the CTAS sources without change. Only 350 lines were written for this project. Considering this makes the ratio of message language to output a much more attractive 1:5.7.

The compiler itself is not so easily justified, even taking into account that most of it was reused from other projects. It begins to make economic sense only when you consider other costs, like debugging time, and other benefits, like ease of evolution. The compiler output probably contained fewer errors than corresponding hand-written code, since it used the CTAS message structure definitions directly. For the same reason, the language system can immediately respond to changes in the CTAS message structures. Furthermore, the compiler would not need to be rewritten if other CTAS processes were changed along the same lines as the CM—its cost could be amortized. The compiler also makes some kinds of software evolution easier, as discussed above. Finally, its ability to be tested independently was a real help in making the project's tight deadlines.

5.3 How the language evolved

The message language evolved significantly in the few weeks from initial design to final use. This section discusses the evolution, its implications, and two lessons we learned: first, that a language that encapsulates some system knowledge (here, message structure definitions and enumerated constants) should make that knowledge available to the rest of the system; and second, that even an application-specific language should be as general—as *non-application-specific*—as possible.

The C-like subset of the message language encapsulates a lot of information about the original CTAS system (the structure definitions and enumerated constants). We originally isolated this information in the message language, reasoning that the rest of the Java CM should be designed without influence from the old CM. This choice also molded the message language: code that

creates or reads messages must be placed in the message language, where message formats are known, leading to the combined C-and-handlers design we have described.

There were alternatives to this design. One of us (Montgomery) argued from the beginning that the compiler should have a more limited role—that it should simply generate Java classes corresponding to the message structures. We abandoned this option because we worried that it would create too many Java classes, provide a difficult-to-use interface, and maintain the creeping influence of the old CM. However, we eventually added exactly this functionality to the compiler (though we didn't remove message handlers). Now, as described earlier, the compiler automatically generates Java classes corresponding to C structs or enumerated types in response to a '`pragma export`' statement.

We made this change because other programmers needed the structure information. One person, for example, was writing some debugging code independent of the message language. He needed the values of some enumerated constants from the old CM and counted by hand to get them, introducing an off-by-one error which caused a couple hours of debugging. In response, we developed '`pragma export`', forestalling further errors of this type.

We had initially forgotten that the goal of a little language is to get the job done, not to enforce policy decisions that others would have to work around. Our decision to provide access to all the information the compiler had—including C structure layout and enumerated constants—was simple to implement and got the job done, making the message language significantly more useful. This runs counter to conventional wisdom about interfaces and information hiding, and seems to merit an aphorism: Let the user access everything the compiler can figure out.

The other general trend we observed as the language evolved was towards simplicity and away from application specificity. We had originally designed several language features, amounting to application-specific syntactic sugar, which were either unused or used less than we expected.

One feature that wasn't even implemented was the *translate block*. This was an annotation meant to make message handlers even simpler: a programmer could specify that an outgoing message's field should be filled in with a particular value from system memory, and that the memory value should be updated when that message field was received, all with a simple one-line notation. The problem with this notation soon became clear: it wasn't flexible enough to cover some common situations, and it didn't offer enough syntactic advantage over simple assignments to justify its implementation cost.

Some syntactic sugar is still in the compiler. Expressions of the form "`ac[`*x*`]`" can be used to refer to aircraft structures; they are translated to Java expressions like "`AircraftTable.get(`*x*`)`". The language's `send` and `forward` expressions, and the client types (`PGUI`, `ISM`, `DP`, and the like), are also more or less simple syntactic sugar. (We had imagined that each of these expressions would expand to several lines of Java code, but this turned out not to be necessary, often due to high-level Java features like exceptions.)

Syntactic sugar has attendant advantages and disadvantages. Some disadvantages include compiler difficulty, lack of flexibility, and language complexity. Advantages include readability, conciseness, and, surprisingly, maintainability: If the compiler translates a higher-level expression to lower-level code, then many lower-level interface changes can be accommodated simply by changing the compiler. (In the Java CM project, for example, the client type interface frequently changed. Each time, we only needed to make simple compiler changes, since client type expressions were all sugared from simple language forms that didn't need updating.) However, these advantages don't matter if the syntactic sugar isn't used. In the message description for the Java CM, "`ac[`*x*`]`" is used only once. In contrast, `send` is used 16 times, `forward` six times, and the client types on the order of 50 times.

Thus, in our experience, application-specific bells and whistles like translate blocks and `ac[`*x*`]` expressions are not worthwhile: they tend to be too inflexible to be frequently useful. Even a domain-specific language should be made as general as possible by focusing only on the core of the domain.

5.4 Future use in CTAS

The message language can provide a long-term path for evolving CTAS. It is flexible enough to be used on other, currently unmodified CTAS processes as they are redesigned, and simple message format changes can be incorporated in the message language as is. Even large changes—moving to a very different wire format, say—wouldn't require throwing away all of the message language; for instance, only the compiler and message format sections might need updating.

Although we believe the language would be flexible enough to last for some time, it may not be truly useful in the long term. If CTAS were uniformly updated to use a standard communications protocol, like CORBA, DCOM, or Java RMI, existing libraries for marshalling and unmarshalling could be used instead of a proprietary solution. As discussed in Section 5.6, this doesn't eliminate all the arguments for a message language, but it does make one much more difficult to justify on engineering grounds.

5.5 Development beyond CTAS

A message language like the one we have described could be useful for evolving other applications. How much of the infrastructure we have described could be easily adapted to another project? And could it be generalized to handle an entire class of message-handling applications without compiler changes?

The language has already become mostly general, as discussed in Section 5.3. Most remaining CTAS-related extensions are syntactic sugar: "`ac[`*x*`]`" expressions and process names such as `PGUI` and `ISM`. Some of these ("`ac[`*x*`]`") should be removed; others (process names) could easily be made generic, with the compiler reading a list of process names from a configuration file rather than having them hard-coded.

The compiler's output is tied not to CTAS as much as to the architecture of the Java CM. Since the compiler generates Java classes meant to fit into the Java CM, it assumes the presence of several specific abstractions: objects like `Client`, `ClientFilter`, `Message`, and so forth. These abstractions are not CTAS-specific, however; they are a plausible interface for a class of message-passing applications. If a new application could use these abstractions with only minor changes, the compiler would not need to be changed at all.

Unfortunately, not all applications will fit this framework. A better solution would be to provide flexible tools that don't enforce a possibly inappropriate application architecture. The current compiler is somewhat flexible already: the large majority of CTAS-specific code is effectively localized in a string table. Thus, a user could adapt the compiler to a new application architecture with little difficulty. Even the output language is not too hard to change; the only Java feature we rely on is exception handling, which simplifies code generation. We could avoid per-application compiler changes by designing a configuration metalanguage for describing the compiler's output. Some care would be necessary to ensure that every relevant output feature was expressible in the metalanguage. This approach would make the compiler so much more complex, and changing the compiler is so easy, that it does not seem practical.

5.6 Alternatives

In this section we discuss a number of alternatives to the language choices we made. We evaluate each, particularly in terms of developer productivity.

A simpler-to-implement language. We could have designed a significantly simpler compiler, if we had made the language harder—or at least uglier—to write. For instance, we could have removed around 20% of the compiler had we eliminated all syntactic sugar, including message fields, from the language. While the impact on readability and understandability would have been significant, some might prefer a 20% smaller compiler on engineering grounds. Our choice—an elegant language and a more complicated compiler—was dictated by aesthetics.

No message handlers. We also could have focused solely on marshalling and unmarshalling, leaving message handlers to be written entirely by hand. The compiler would then become simply a message format conversion tool, rather than a message handling language system. As a result, its implementation could be shortened by about 30%. The arguments for and against a language, instead of such a tool, are essentially the arguments for and against domain-specific languages in general. On the plus side, the language promises to make message handling code simpler and more elegant, and the possibility of changing the compiler facilitates certain kinds of system evolution. However, the language's unfamiliarity can be a drawback in the long run, and of course it requires writing a compiler.

A very high level language. One might think that the compiler would have been easier to write in a very high level language like Perl or Python. However, this argument does not apply to this compiler: much of it had already been written in C++, and the more complex expression- and type-handling code would have been difficult to write in a language like Perl.

Off-the-shelf message formats. If the original CM had used an off-the-shelf messaging system like PVM, or at least an existing wire format like XDR, message format parsing would not have been an issue. As a result, engineering tradeoffs would have discouraged us from using a little language to evolve the system, even though some of the arguments for a language would still have been valid. In the actual CTAS system, we knew it was necessary to handle the proprietary message format; the message language was a small burden on top of this. If message formats hadn't been a problem, implementing a message language would have seemed proportionately much harder.

No language. The starkest alternative would simply have been to write the message formatting code by hand. After all, the compiler generates only 2000 lines of Java; in hindsight, writing those 2000 lines by hand must be easier than writing a 10000 line compiler and a 2000 line file in the message language! As we've discussed, however, those 2000 lines would have been riddled with errors, extending the prototype to full functionality would require far more code, and we actually wrote far fewer than 12000 lines for this project. Even considering these problems, writing the message formatting code by hand might have been easier for this prototype; but, again, the compiler was partly an aesthetic choice.

6 Conclusions

This paper has described one way to evolve a software system by using a little language. The language was concise (resulting in an effective ratio of input language to output code of 1:5.7), readable, and maintainable. Furthermore, despite its size, the compiler worked on time: the message language was an important factor in the timely implementation of the Java CM. The language is specific to the CTAS application, but seems more generally useful for message-passing systems with proprietary message formats. Some of the lessons we learned, however, are more widely applicable still:

- A language can provide the advantages of two methodologies of software change—evolution and revolution—by combining aspects of both old and new systems.
- A little language should let the user access anything the compiler can figure out.
- Domain-specific languages should be made as general as possible by focusing only on the core of the domain.
- Evolving a system that includes a little language can be easy, since one can make large changes to the system with small changes to the compiler.
- Consider the implementation costs before making a little language elegant.

Acknowledgements

We would like to thank Daniel Jackson, for teaching a great class; the other students of 6.894, especially Ilya Shlyakhter and Phil Sarin; Michelle Eshow and the NASA-Ames CTAS group, for support and the opportunity to work on CTAS; and Rick Lloyd from MIT Lincoln Labs, for perspective on other ways to redesign the system.

References

[1] Xerox Corporation. Courier: the remote procedure call protocol. Technical Report XSIS 038112, December 1981.

[2] H. Erzberger. CTAS: Computer intelligence for air traffic control in the terminal area. TM 103959, NASA Ames Research Center, July 1992.

[3] J. Hook and L. Walton. The design of message specification language. Technical report, Pacific Software Research Center, OGI, June 1997.

[4] S. C. Johnson. Yacc – yet another compiler-compiler. Computer Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, July 1975.

[5] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable TCP in the Prolac protocol language. To appear in ACM SIGCOMM '99, September 1999. See also `http://www.pdos.lcs.mit.edu/~eddietwo/prolac`.

[6] R. Lloyd, December 1998. Lecture to MIT class 6.894.

[7] Sun Microsystems. XDR: External data representation standard. Request for Comments 1014, June 1987.

[8] S. O'Malley, T. Proebsting, and A. Montz. USC: A universal stub compiler. In *Proceedings of the ACM SIGCOMM Conference on Communications Architectures, Protocols and Applications*, London, September 1994.

[9] International Standards Organization. Specification of abstract syntax notation one. Technical Report International Standard 8824.

[10] International Standards Organization. Specification of basic encoding rules for abstract syntax notation one. Technical Report International Standard 8835.

[11] D. Spinellis and V. Guruprasad. Lightweight languages as software engineering tools. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997.

[12] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[13] A. van Deursen and P. Klint. Little languages: little maintenance? SEN Report R9704, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, March 1997.

[14] D. Weiss. Creating domain-specific languages: the FAST process. In S. Kamin, editor, *Proceedings of the First ACM Workshop on Domain-Specific Languages*. Department of Computer Science, University of Illinois, January 1997. See also `http://www-sal.cs.uiuc.edu/~kamin/dsl`.