The following paper was originally published in the
Proceedings of the Sixth Annual Tcl/Tk Workshop
San Diego, California, September 14–18, 1998

Using Tcl to Rapidly Develop a Scalable Engine for Processing Dynamic Application Logic

Greg Barish
Healtheon Corporation

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

Using Tcl to Rapidly Develop a Scalable Engine for Processing Dynamic Application Logic

Greg Barish
Healtheon Corporation
barish@isi.edu

Abstract

At Healtheon, we used Tcl to rapidly develop a scalable, high performance rule engine for processing dynamic application logic. The nature of our application requirements, plus the challenge of delivering robust software in a timely manner made Tcl an optimal overall choice in our deployment. We were able to improve rule processing performance by careful language construction and support for concurrent execution. We developed a mechanism for implementing data-driven language extensions, called rule concepts, which allowed us to present a customized language for each client, and encouraged rule reusability. Our experience with using Tcl in our application system was also representative of software engineering choices that small companies often make in pursuit of rapidly developing a well-balanced system solution.

1 Introduction

At Healtheon, we used Tcl as the basis for the implementation of a *rule engine*, a mechanism for processing dynamic application logic. Our design, implementation, and deployment were notable because these phases revealed techniques for integrating Tcl in environments which must meet the scalability and performance demands of an on-line service.

In this paper, I initially describe the problem domain and present how Tcl addressed the basic application and system requirements. Next, I present the evolution of our extensions (our rule language). This serves to illustrate techniques for addressing the performance of run-time interpreted languages and suggests a data-driven solution for implementing dynamic extensions. Moreover, the evolutionary description represents a case study for why the adaptability of an extensible third-party interpreter resulted in a more economical solution than having implemented a custom rule processor ourselves. Finally, I discuss various system integration and deployment approaches towards improving the availability and concurrency of a Tcl-based application service, outside of modifying the interpreter itself.

An important sub-theme of this paper is one of software engineering. In particular, I refer to the "rapid rate" of development and deployment of our Tcl-based rule engine as a feature. Since we are a small company, quickly deploying reliable software is a constant demand, and I imply several times why Tcl was uniquely qualified to address not only the basic unit requirements, but was also the best overall solution in terms of system integration and robust operation.

All too often, software are solutions are heralded only for their elegance at handling a single complex problem. Less commonly rewarded are software solutions which successfully meet several software engineering demands, in addition to achieving basic unit functionality. These demands include the ease of integration, method of maintenance, and time and resources required for development. They can be referred to as economical features of software development. A novel aspect of our use of Tcl was how well we found it at addressing these broader issues and thus providing us with the best overall solution.

1.1 The Healtheon Benefit Manager

At Healtheon, one of our key applications is a on-line benefit management system, called Benefit Manager, which employers can use to process employee benefits. Administrators use the system to choose and customize benefit packages and plans for their employees. Subscribers use the application to make and update their benefit choices. These bulk of these selections are usually done once a year, during a series of weeks known as Open Enrollment.

Since Benefit Manager exists as an on-line Internet service, with concurrent sessions potentially numbering in the tens to hundreds of thousands, it must also wrestle with requirements for availability, scalability, high performance, and fault tolerance. These demands are also true of other applications employing the on-line service

model, such as America On-Line, Yahoo, and Amazon. In this sense, Benefit Manager is representative of a model of application deployment becoming increasingly common in commercial, transaction-based systems.

1.2 Benefit Rules

One of the key application requirements of the Benefit Manager application was the need to support the encoding and processing of *benefit rules*. Benefit administrators at a given company specify these rules and associate them with benefits, plans, and selections for various types of employees. Rules figure such things as benefit and plan eligibility, plan costs and credits, plan dates of effectivity, validation of selection combinations, and several other such determinations.

Rules are run on behalf of a given subscriber and are frequently a function of some demographic, employment, or existing benefit attribute associated with that subscriber. For example, a certain medical plan geared towards retirees might have an eligibility rule based on the employee age. Another example would be a case where a legacy benefit plan is only available to subscribers who are already enrolled in that same plan for the current benefit period (a rule based on existing selection information).

However, there are also rules which are partially based on the particulars of a specific transaction. For example, an employee may be eligible for medical plan *M* and dental plan *D*, but not eligible for the selection of both of them. Thus, such rules are run based on the session data associated with the transaction.

Processing rules always involves the generation of *rule results*, which vary based on the type of rule. For example, eligibility rules return true/false results, cost rules return dollar amounts, and effective date rules return calendar dates. While most rules typically return a single result, some do return multiple results, and some even return an unbalanced number of results (i.e., one result on success, two results on failure - the second result being the error message).

At Healtheon, rules are expressed in a rule language. They are associated with the appropriate benefit, plan, or other relevant object in our data model. Therefore, a many-to-many relationship between employees and benefit information exists and thus most rules are shared by large groups of employees.

Typically, making benefit selections during Open Enrollment results in between 100 and 150 rules being processed per subscriber. However, although rules are shared, employers will still have a high number of total rules on the system, to provide coverage for all types of employees and support all possible benefit and plan combinations. It is not uncommon for a typical employer of 10,000 employees to have over 500 rules on the system. Rule population varies by company and is more of a function of their existing benefits complexity rather than the company size.

1.2.1 Pre-Computation

One might be tempted to ask why rules could not be pre-computed, to remove the burden of run-time interpretation. Generally, it is not practical to pre-compute rules. For one, rules are often date-sensitive. Consider a plan eligibility rule such as: "*a subscriber is eligible for plan if that subscriber has been in the same plan for at least two years*". If that rule is pre-computed a month before a subscriber actually tries to choose that plan again, it might obviously return different results.

It could be suggested that rules be pre-computed on a daily basis, perhaps during times when the system is relative idle and better able to engage in such CPU-intensive activity. This is still impractical when considering the workflow of a benefits enrollment. Frequently, subscribers will use Open Enrollment to first change demographic data *before* selecting benefits. Thus, the same session is often used for both tasks. Sudden changes in personal information would therefore often cause the pre-computed rules to be invalid.

It is conceivable that a dependency graph could be constructed such that only those rules which need to be re-computed after such scenarios actually are, but leads to drastic increases in solution complexity. It would have required a sophisticated parsing (actually, compilation) technique to be employed whenever rules were updated in the system. Moreover, this effort implies that a solution for detecting indirect requests for data retrieval be implemented. This is analogous to the problems associated with detecting pointer-based references, such as that which is done during data flow analysis for purposes of compilation.

2 Choosing Tcl

We decided to use Tcl as the basis for our rule language for two major reasons: (a) the effectiveness with which it fulfilled our basic application and system requirements and (b) that it was, from a software engineering perspective, the most economical of the solutions available.

During this discussion, it is useful to also relate observations which led up to our decision, since it illustrates why a language like Tcl becomes attractive to a small company which needs to rapidly deliver reliable software.

2.1 Application and System Requirements

As described earlier, the most serious application unit requirements we had on the interpreter was that it needed be easy to extend and would contain support for moderately complex data structures and their associated data manipulation functions. Obviously, Tcl met both of these demands easily.

System-level requirements included those of attaining high performance, scalability, and availability. Although the choice of interpreter and language design did have some impact on these challenges, it later turned out that it was better to deal with these issues at a higher level of software - the rule engine itself. I describe these approaches in later sections of this paper.

Still, an important additional system-level requirement to address at this point had to do with the ease of integration. As a small software company, we were concerned about how development and runtime system would be affected by incorporating Tcl. While using third party software can be attractive for how much it can enhance an existing system, difficulty related to integration often results in subtracting away any profit which was made.

For example, we were immediately concerned about performance issues associated with parsing. We were also questioning how we our use of multithreading might impact Tcl. Other issues included: integration with C++, integration issues with CORBA (our choice for distributed application development), API portability, effect on system security, and the time required for integration.

2.1.1 Code Maturity

Many of our concerns about integration had to do with how comfortable we felt trusting the code. We did not want to build and deploy a system only to find out that we were plagued by problems with a third-party API.

In addition to being one of the most prolific, freely available interpreted languages, Tcl had been around for several years. We knew that it was heavily used and extended by others in the industry. We felt that there was an excellent chance that the code would be stable and efficient.

Even more attractive was the fact that we were not simply integrating with a code library - we had access to the source code itself. Thus, during debugging, we were able to ensure that various bugs were definitely not Tcl-related.

It should be noted the opposite is nearly always true when dealing with third-party code which has been purchased. In those scenarios, developers have to deal with support teams which look at various potential bugs on a case-by-case basis. In terms of efficient software engineering, this is highly undesirable in terms of both time and cost. Even worse, though the developers typically have already paid large amounts of money to use commercial third-party software, they often then have to pay additional fees for support!

This is not to suggest that the universe of industry software engineering problems would be resolved if companies just gave away their code. Tcl is unique in this respect, and the above observation is merely a testament to this specific case.

2.1.2 Simplicity of Extensibility

There was a minimal learning curve associated with figuring out how to extend Tcl: only one API call was necessary. Understanding the related data structures and requirements involved in writing an extension were also very simple. We successfully tested an extension the same day we downloaded the software.

Later, we also found great value in the ability to remove commands from the language (see *Security*, later in this section).

2.1.3 Portability

At the time we needed to choose a rule language, it was unclear whether our system would be running under Sun Solaris or Windows NT. That Tcl had been ported to several platforms made this concern a low-priority issue during consideration and allowed us to continue development on the rule engine in parallel, without delay.

Furthermore, in looking at the code organization, we could see that the operating system dependent code was well distinguished from the generic code, so we felt that any modifications or platform-specific enhancements we might have to make on our own would be an easier task than usual.

For example, our platform team had come up with a lightweight, portable thread library [Kougiouris97] which we thought we might need to incorporate somehow in the interpreter itself. Given the distinction of operating system dependent code, we felt it would be easier than usual to know what parts of Tcl might be affected and where to find the related code.

2.1.4 Security

Initially, it was unclear who would be coding the rules we needed and how it would be done.

Would they be coded from clients through a Web interface? If so, then what kind of impact could forged rules have on our system, in addition to destroying the correctness of the rules themselves? Would clients be composing rules through a graphical editor (which would more or less guarantee correct syntax – at least there would be a finite bound on the range on input we would be receiving) or did we have to worry about the horrors of handcoded rules (which might bring along things like infinite loops)?

Most of these questions would have become serious issues had we not been able to disable parts of the existing API. The fact that we could take out some of the control flow commands, like `for` and `while` - which we didn't need for our rules – allowed us the best of both worlds. Thus, in addition to our own command set, we could still harness Tcl for parsing, variable support, and mathematical functionality, without having to worry about the risk of

commands with potentially serious consequences, commands we didn't need.

2.2 Software Economics

It is important to emphasize that we did not view Tcl as necessarily an optimal rule language solution. Certainly, a more attractive scenario would be one in which we built our own custom parser, or even more attractively - our own rule virtual machine. Then, we could pre-compile the rules and remove the parsing element altogether. However, such tasks require significant investments in additional time and development staff. As a small company, we could not afford to make that kind of investment.

Furthermore, there was additional risk involved at investing in an a more optimal solution. More time spent on developing a complex solution would result in less time for system integration, meaning that overall system stability and robustness would be sacrificed. Even more disturbing was the fact that, as mentioned earlier, rule language requirements were volatile. Building extensibility into our own parser or virtual machine would be more complex, require an addition investment in time and resources, and thus increased risk.

What was very obvious to us was that using Tcl gave as a near-optimal language solution at a fraction of the cost. We predicted (and were later proved correct) that building a rule engine based on Tcl would require minimal investment in time and development staff, and would lead to more time to be spent on system integration, thereby increasing our ability to address system robustness. At Healtheon, we felt that even the most sophisticated and cutting edge unit technology looked poor if the entire system it is integrated into does not hold together well. The whole was not simply the sum of its parts.

3 The Rule Engine

Before discussing the rule language further, it is worthwhile to first understand the approach we took towards designing the rule engine, the mechanism which housed the Tcl-based rule interpreter.

The Benefit Manager rule engine provides a simple, lightweight API for processing benefit rules. The engine exists as a C++ class library which Healtheon CORBA-based application servers can link with at compile time. Applications merely make a single call to the `interpret()` function when they want to process a rule. The rule engine eventually calls `Tcl_Eval()` to process the logic and then returns a list of zero or more rule results back to the application.

One of the novel aspects regarding integration of the rule engine with Tcl was how easy it was to gain access to important rule engine data structures during the course of interpretation. It is easiest to understand this profit by way of example. Recall that one of our rule processing requirements was the ability to send back rule results after an interpret. One of the extensions we made for this turned out to be the command `EligibleWhen`, which would return a boolean result of true if the list of arguments passed to it were all non-zero. A sample rule which used this command would be:

```
EligibleWhen [IsMarried];
```

Figures 3a, 3b, and 3c show key parts of the Rule Engine related to this example extension. They reflect three of the important integration stages: (i) runtime initialization of commands (in the Tcl interpreter) based on data structures automatically generated from our data model, (ii) when the call was made to `Tcl_Eval()`, in the course of rule processing, and (iii) the implementation of the extension itself - the code which actually performs the logic and stores the rule results.

```
TCM TclCmdMap[] =
{
   ..
   {"EligibleWhen",
    RuleLanguage_EligibleWhen}
   ..
}

int
RuleMotor::initialize()
{
   ...
   ...
   for (int i=0; TclCmdMap[i].fn!=NULL; i++) {
       Tcl_CreateCommand(
           m_interp,
           TclCmdMap[i].name,
           TclCmdMap[i].fn,
           (ClientData)this,
           (Tcl_CmdDeleteProc*)NULL);
    }
   ...
   ...
 }
```

Figure 3a: Initialization

In particular, Figure 3a shows the simple loop used for creating commands upon Rule Engine initialization. The important part of this phase is to note that a subset of the commands (those tied to the data model) were automatically generated by the data model DDL itself , which allowed us the flexibility to change our model without having to, say, rewrite lexer rules every time. This process is described further later in this paper, in section 5.1.

```
RuleResultList*
RuleMotor::interpret(
   char* a_rule)
{
   m_ruleResults->clear();
   ...
   ...
   int code = Tcl_Eval(m_interp, a_rule);
   ...
   ... During eval, results will accumulate
   ...
   return m_ruleResults;
}
```

Figure 3b: Access

Figure 3b shows the integration point with Tcl during rule interpretation. Rule Engine worker threads, called Rule Motors (described later, in section 6.2), make the actual call to `Tcl_Eval()` themselves and keep private data structures for rule result aggregation.

```
int
RuleLanguage_Tcl_EligibleWhen(
   Tcl_Callback* a_ptr,
   int argc,
   char** argv)
{
   RuleMotor *theMotor = (RuleMotor*)a_ptr;
                  ..
                  ..
      (analyze arguments, determine T or F)
                  ..
                  ..
   if (noneAreFalse) {
      theMotor->appendBooleanResult(TRUE);
   }
   else {
      theMotor->appendBooleanResult(FALSE);
   }
}
```

Figure 3c: Processing

Finally, Figure 3c shows how the example command acquires the pointer to the worker thread which is running the rule (the particular Rule Motor) , performs its necessary logic, and then associates the result of the rule with the motor responsible for rule invocation.

Notice that a pointer to a rule engine data structure is specified when making the `Tcl_CreateCommand()` call in part (i). During runtime, (ii) is invoked to perform the rule processing. Then, during interpretation, the data structure from (i) can be accessed by casting the callback pointer upon entry to extension, as shown in (iii). Upon return in (ii), the modified data structure can be analyzed. This was a major asset at runtime, since it prevented us from having to marshal data between steps (ii) and (iii).

There are several other important issues related to improving performance, availability, and scalability of the rule engine in terms of deployment. However, before discussing these issues, it is first useful to understand the nature of the Tcl extensions we made, in other words, the rule language.

4 Rule Language Design

It took us two iterations to arrive at an effective rule language. Both are worth describing because they illustrate how Tcl became so valuable to us while our application system matured. The first attempt saw us make only a few extensions to Tcl, enough so that we could retrieve information from our database and process rule results. The resulting language was sufficient but problematic.

The second attempt was far more effective because, by then, we had a much clearer picture of the rule requirements. It gave us time to look at how clients wanted to encode rules and adapt the language accordingly. Again, it should be noted that had we not used Tcl, the prospect of iterating the rule language as we did would have been far more risky and less likely to succeed.

4.1 Take One: The Minimalistic Approach

The initial version of the Healtheon rule language included less than 10 commands, each of which was essentially a generic mechanism for addressing the language requirements.

For example, we had one command (`DbGet`) for retrieving subscriber information from the database. This extension took a table and attribute name as arguments, issued a dynamic SQL query to our database, and would return the associated values for that subscriber.

We found numerous problems with this approach:

- **data model exposure**: In order to specify the arguments of a `DbGet`, the user was obviously required to understand our data model. Exposing a data model is generally not good practice - the system should present a consistent interface to all users for the long term. Exposing the low level details of the model hampers the ability to change the model.
- **command usage ambiguity**: If the table name could be any one of our tables, how could we enforce that the author specified the correct number and type of table keys in order to resolve a unique row? Again, this would clumsily expose the underlying data model. Even worse was that it made the `DbGet` command more ambiguous - it was unclear how many arguments it needed for a given call.
- **greater parsing demands**: We had to assume that clients might enter table names and attributes in

varying styles - wrong case, slight misspelling, etc - and we needed to determine how to handle such input. Furthermore, there was simply more to parse: having a generic data retrieval command such as `DbGet` implies that additional parsing will need to be done to figure out exactly what to get.

- **manual data typing**: Tcl is typeless, so after getting the data back, the value essentially became a string. Any other function which took the result of a `DbGet` as an argument would have no idea what kind of actual type it was dealing with, unless it was overtly specified by the author.

  For example, consider the problems with comparing two dates - the subscriber's hire date at the company and his birth date. Although we would want to say:

```
if [Compare LessThan
     [DbGet Subscriber BirthDate]
     [DbGet Subscriber HireDate] ]
then
   ...
endif
```

  this would be problematic unless the type information was included:

```
if [Compare date LessThan
     [DbGet Subscriber BirthDate]
     [DbGet Subscriber HireDate] ]
then
   ...
endif
```

These issues are representative of those associated with implementing the other generic extensions we did in our initial language definition. We found that they placed a heavy burden on the rule author, they are far more prone to error, they incurred greater parsing demands, and they were overly verbose. Perhaps the greatest crime was that rule author, the most valued client of all, was presented with an ambiguous, cumbersome interface.

Despite all of these problems, there was one important benefit to our initial version of the language: the speed at which we could implement a functioning rule processor. The rule engine became one of the first modules of the alpha portion of the application software to actually work. This milestone allowed the engineering team to focus more on system integration issues and re-assign developers to other projects which were not yet completed. Additionally, it allowed us to begin coding rules for some of our initial clients.

4.2 Take Two: The Final Cut

With proof of concept under our belt, the second version of Benefit Manager saw us revisit the design of the rule language. By this point in our development, we were now made clearly aware of the problems associated with implementing generic commands. For the second release, we wanted less verbose rules, less parsing, decreased burden on the rule author to understand various uses of the same command, and a syntax which generally left fewer opportunities for user error.

4.2.1 Improving Data Access

Early in the process, we attained a major milestone by developing a new paradigm for data retrieval. Our approach was to create a Tcl extension for every possible attribute of every table in our retrieval domain. The name of these commands was the concatenation of the table and attribute for that item.

Implementing each potential retrieval as an extension also gave us the opportunity to hide the data typing associated with that item. For every database command, our rule language supported two styles of execution. In the first, no arguments were specified and the string (typeless) value of the attribute was resolved. The second form required comparison arguments and returned either true or false, depending on the result of the comparison. It was this second form of data access and comparison which gave us the opportunity to implement *automatic datatyping*.

Figure 4a shows the how a rule based on subscriber salary would be authored in the first and second versions of the rule language. Obviously, the second version is more compact, less cluttered, incurs less runtime parsing, and makes the data typing automatic and transparent.

Version 1:

```
if [Compare double >
      [DbGet Employee Salary] 50000 ]
then
   ...
```

Version 2:

```
if [EmpSalary > 50000] then ..
```

Figure 4a: Improving Data Access

4.2.2 Improving Rule Result Reporting

Another major language improvement was to make rule result reporting easier. As was the case with data access, the first version of the rule language placed the burden of declaring the rule result type in the hands of the author. Even more concerning was that authors might forget to return a result (leaving the application confused about the status of rule satisfaction) and the high potential that certain control flows might not lead to any results being reported.

To address these issues, we devised specific styles of commands which would encourage full reporting. The heart of the problem was the reliance on `IF-THEN` logic. Such logic was characteristic of most rules and we attempted to implicitly capture that logic in our new commands, improving code compactness and removing some of the potential for coding errors.

Figure 4b shows how an benefit eligibility rule appears in both versions of the language. Obviously, the first version contains more code and places a higher degree of responsibility on the author in terms of control flow checking. The second version makes the `IF-THEN` logic implicit in the `EligibleWhen` command. This command simply takes a space delimited list of values and returns false (i.e., "not eligible") if any of these values are zero.

Version 1:

```
if [Compare double >
      [DbGet Employee Salary] 50000]
then
   ReturnRuleResult boolean true;
else
   ReturnRuleResult boolean false;
```

Version 2:

```
EligibleWhen [EmpSalary > 50000];
```

Figure 4b: Improving Rule Results

There are some other worthwhile observations to make about the effect of the language metamorphosis. These included the number of commands in the language: it did require more work on the author's part to know what the categories of commands were. However, we felt that this was not unusual for a scripting language. Also, many of the command names were predictable.

Another related, but subtler, aspect was the level of redundancy in the new language. Recall that we had to process several types of rules: eligibility, costs, credits, dates of effectivity, etc. Now, while there were close to 10 types of rules to deal with, we were always returning a single result, the type of which was either boolean, double, or date. This implies that we really only needed three distinct commands for returning rule results.

While critics might lobby for one mechanism for returning a type of double, there was an obvious subtle benefit to having distinct commands on a rule type basis: increased language transparency. For example, we were freer to update the semantics of eligibility instead of forever remaining tied to the notion that it only represented either True or False.

By making these changes to the rule language, we improved the performance of rule interpretation, decreased the potential for error, and hid the ugliness of data typing. Most importantly, from the authoring point of view, the rules were much easier to read and understand. This, in turn, made them easier to author.

4.2.3 Rule Concepts: Dynamic Extensions

It is often desirable to customize a language to best meet the specific needs of a client. Each company has its own way of doing business, its own *business logic concepts*, which frequently play a role in rule processing. Our goal was to do what was possible to support the expression of rules in these simple, familiar terms, to improve the usability of the rule language, as well as to promote rule compactness.

Our solution was to support the declaration of *rule concepts*, company-specific extensions to the rule language. In this sense, the rule language was thus a union between the base language and the concepts for a given company. In practice, rule concepts were simply macros to facilitate the simple expression of a awkward or complex computation.

Suppose a company is based in California and frequently uses the state tax rate in their benefit plan cost rules. They might want to express a cost rule as something on the order of: "*the cost of this plan is $200 plus 1% of the employee's salary multiplied by the state tax rate*". We might want to capture "tax rate" as a concept. To do this, they would use our administration interface to name a new language command called `TaxRate` and define its meaning. Optionally, if the phrase "*one percent of <some number> times the tax rate*" was a frequently used computation, the company could even define that as a concept (i.e., `OnePercentAndTaxOf`),

The benefits of rule concepts are three-fold: they provide a key level of functional abstraction in the language, they increase the re-usability of rule code in the system, and they also allow us to personalize Tcl to best address the specific business language of our clients.

5 Language Integration

The new version of our rule language was also notable in terms of the issues it raised related to system integration. Some of these had to do with the management of having over 150 Tcl extensions, specifically in terms of proper maintenance and namespace clashing. Other issues involved the relationship of the language to the application data model and how useful application-level data objects might be represented .

5.1 Automating Generation of Tcl Extensions

With over 100 commands for data access alone, there was concern about the ability to manage the development of these extensions without assigning more programming staff to the task. Since the nature of all of these extensions

were the same (retrieve data, optionally provide comparison logic), we decided to spend time developing a mechanism for automatically generating the code for these extensions. Our process for this is shown in Figure 5a. The typical cycle of development was to first update the data model as required, export the associated data definition language (DDL) to a file, use the DDL to the automatically generate source code for the Tcl extensions, and finally rebuild the extensions library. This level of automation allowed us to efficiently adapt the data model as required. In fact, DDL revisions became a far more problematic issue for the applications themselves (which often referred to attributes and tables literally) than it was for the rule engine.

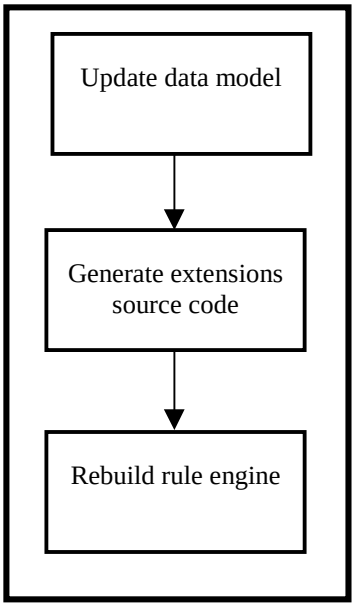

Figure 5a: Code Generation Methodology

5.2 Naming Issues

One additional issue we encountered was that of the command namespace management. For example, all of the tables which could be data access candidates in rules contained attributes which were named the same, but included in different tables for various reasons (they were foreign keys or just deliberate duplicates of information for purposes of security via data partitioning). What this meant, for example, was that we could have a field in the demographic table called "`Birth_Date`" and a field in the employment information table called "`Birth_Date`". We needed to mechanism to establish closure [Neuman89], to ensure name uniqueness.

We took the common route of prepending a unique (table-based) prefix to each command. Thus, a rule for comparing birth dates would appear as:

```
if [EmpAge == [BenAge]]
then
  ...
  ...
```

This would compare the age of the subscriber as listed in the employment (`Emp`) information with the age of the subscriber in the benefits (`Ben`) table.

There is additional naming complexity when considering how to deal with implementing rule concepts. For example, if two companies want to create a concept called `ComputeTax`, how should this be resolved? Should the company who first requested this concept be awarded the right to name it? More alarming was the privacy aspect of concepts: since the same rule interpreter was used by all parties, did this mean that a concept designed by one company would be accessible by all other companies? Namespace collisions are common problems when using Tcl, mostly due to the lack of scoping in the language [Libes95].

Our solution to this problem was to implement *run-time dynamic scoping*. The basic idea was that the Tcl interpreter would consist of the normal extensions of the rule language, plus the union of all concepts (company-based extensions). This meant that, even though two different companies declared their own version of a concept like `ComputeTax`, there was only one extension made. At runtime, a hash table of concepts was maintained in memory. Thus, for `ComputeTax`, the hash bucket associated with that entry would contain a linked list of two elements, representing both versions of the concept.

The methodology for concept resolution - or closure - was simple: since rules are always run on behalf of a subscriber, simply determine which company providing benefits for that subscriber and use that company identifier to choose which concept to execute.

5.3 Data Model Aliasing

As described in 5.1, we generated our extensions based on the data model itself. However, we remained concerned about the potential for legacy rules - ones which were no longer valid after a data model update. For example, if a client had written a rule like that in Figure 4b (version 2) and we suddenly decided to rename that attribute to "`AnnualSalary`", we would indirectly invalidate an existing rule. The lack of a mediator between the rule language and the data model remained problematic.

To address this issue, we devised a map-based mechanism for ensuring data model transparency. The methodology consisted of a scheme in which multiple names, known as *aliases*, could refer to the same attribute. Thus, if an attribute name was changed, we could simply amend the map to include support for the old name by providing an alias, without modifying the actual company rules

themselves. Multiple Tcl extensions would therefore be created for the same attribute, but only one extension would actually be implemented. The rest were simply pointers to this code. The notion of aliases is not new, it is simply a variation on an implementation of symbolic links [Lampson85].

Obviously, while aliases succeeded in resolving the issue of attribute name changing, they did not help out when an attribute was removed from the data model. This is a more complex problem and one where a resolution may not be possible.

6 System Integration and Deployment

The underlying platform software at Healtheon is a scalable distributed object system. Among other features, it contains support for high availability, concurrency, fault tolerance, security, and naming. In this part of the paper, I discuss techniques for the integration and deployment of the rule engine such that these system features could be exploited in an attempt to improve scalability.

In particular, I show describe our experiences with improving rule engine availability and performance. While the former was not successful, the latter was very successful, and led to increased scalability of the resulting software upon deployment. While these experiences only marginally related to Tcl itself, they represent ways in which Tcl can be integrated into systems to both address general reliability as well as to counter performance problems associated with run-time interpretation.

6.1 Improving Rule Engine Availability

As described earlier, building an application in our system which used the rule engine requires linking with the associated library.

Initially, we had wanted to improve the availability of the rule interpreter by wrapping the rule engine in a CORBA shell and allowing an object request broker (ORB) or separate load balancing tool to launch the number of rule engine instances necessary to deal with given client demands. This would have prevented a misbehaving, frequently crashing application from also destroying access to the rule engine for other servers.

While this would have improved availability and fault tolerance, it would have also severely hampered performance. Recall that processing a set of selections for a given subscriber results in over 100 rules being processed. If the rule engine existed as a distinct server, this would lead to an unacceptable number of network calls.

An alternative, slightly more optimized approach would have been to simply combine a CORBA module for the rule engine with the other modules for the Benefit Manager application in the same server. When multiple modules exist in the same server, a form of optimized, lightweight RPC [Bershad90] is often used. This results is what is essentially a local function call.

While this reduces the data copying and network overhead, it does not address the needless marshalling of data between module boundaries. Since the workflow of rule processing is such that one segment of C++ code (the application) is calling another (the rule engine), the arguments passed to the rule engine must thus be marshaled into CORBA types, even though these arguments are never even transmitted with a networking protocol.

In the final analysis, we stuck with our original idea of using a C++ class library as the basis for the rule engine, as the sacrifices to be made for improved availability were not worth the price is decreased performance.

6.2 Concurrent Rule Execution

On a more positive note, we did have far more success with improving performance by increasing rule processing concurrency. Specifically, we made improvements to the rule engine itself, in terms of thread safety, without needing to investigate a Tcl-based solution for multithreading.

To achieve acceptable concurrent processing, we designed the rule engine to support pools of *rule motors* - each essentially a wrapper around a rule interpreter data structure - several of which would be available when rules needed to be processed. Thus, each motor supported its own `Tcl_Interp` data structure, as depicted in Figure 6a.

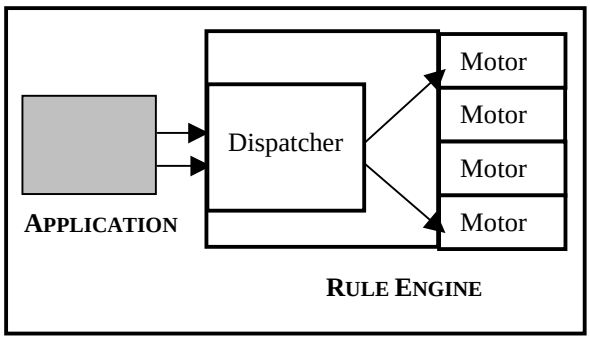

Figure 6a: Concurrent Rule Processing Via Rule Dispatcher
*Each Rule Motor contains a Tcl_Interp**

When handling requests for rule interpretation, the rule engine motor dispatcher would grab the process mutex, search for an available motor, hand off the request to that motor, and mark it as taken. In the event that all motors were occupied, the rule engine would simply wait for a broadcast signal (sent when a motor was done with processing a request) and then choose that motor as the destination for the incoming request.

It should be noted that there was no need to support multithreading for reasons of correctness in processing, since the internal Tcl data structures which cannot support

concurrency (the global variables) are only related to those Tcl commands for file and process management, commands which we did not support with our rule language.

However, multithreading was important for purposes of supporting high concurrency and thus improved system throughput. Simply using a thread-safe version of Tcl would not have been an optimal solution, since what was really needed was for the entire engine to support concurrent access. Therefore, maintaining pools of interpreters was a more reasonable solution to the problem.

7 Discussion

Employing Tcl as the foundation for our dynamic logic engine ultimately proved to be successful. We were able to rapidly construct a powerful rule language, through a combination of our own extensions as well as leveraging existing Tcl functionality. The ease at adding and removing language commands allowed us to write code generators and easily adapt to rapidly changing design requirements with minimal integration.

The rate at which we were able to develop the rule engine allowed us to reallocate our resources and spend more time improving system performance and overall integration. If we had simply devoted several months to the development of our own parser or pre-compiler, we would have not been able to adapt to the moving target of volatile design requirements, all too common at a small company.

There were some disadvantages to using Tcl, but most of them were expected. We ran into some performance problems, some due to our own parsing and some related to the Tcl parser. We almost certainly could have achieved better performance through a more lightweight, application specific parser or even rule virtual machine which would process pre-compiled rules. However, these would have been impractical solutions and would have forced us to sacrifice crucial aspects of overall system integration.

At a larger company, development teams have more time to approach milestones, and can afford to spend long periods of time in the design and prototyping phase. Also, larger companies tend to deliver a tool they think addresses a market demand and then tackle customer requirements in future releases. In short, they can easily push products into the market channel.

In contrast, a small company cannot afford to simply throw software out onto the market. It needs to be highly sensitive to the requirements of its core customers, as much of a moving target as that can be, while still delivering a well-balanced product: it thus becomes essential to deploy something quickly, but which still ensures correctness and robustness, despite changing requirements. Using Tcl allowed us to meet the necessary requirements, quickly deploy advanced functionality, and to spend more time improving system integration.

8 Acknowledgements

I wish to thank the following people at Healtheon for their continued input and support related to the care and feeding of the rule engine: Mohammad Alagebandan, Giamma Clerici, Kittu Kolluri, Shankar Srinivasan, and Theron Tock.

9 References

[Lampson85] Lampson, B., "Designing a global name service". In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, August 1985.

[Libes95] Libes, D., "Managing Tcl's Namespaces Collaboratively", *Proceedings of the Fifth Annual Tcl/Tk Workshop '97*, Boston, MA, July 14-7, 1997.

[Neuman89] Neuman, C., "The Need for Closure in Large Distributed Systems". *Operating Systems Review*, 23(4):28--30, October 1989.

[Kougiouris97] Kougiouris, P., Framba, M., "A Portable Multithreading Framework". *C/C++ User's Journal*. August 1997.

[Ousterhout94] Ousterhout, J., *The Tcl/Tk Toolkit*, Addison-Wesley, 1994.

[Bershad90] Bershad, B.N., Anderson, T.E.., Lazowska, E.D., Levy, H.M., "Lightweight Remote Procedure Call". *ACM Transactions on Computer Systems*, 8(1):37--55, February 1990.