Rehabilitation of Legacy Systems by Automatic Methods

by Steven Popkes

Introduction

Few programmers in the late sixties and seventies, or even in the early eighties, expected the applications they developed to last for decades. They fashioned solutions to their assigned tasks with the best technology and techniques they knew. In the passage of time, different programmers enhanced or rewrote portions of the older product in unexpected ways, using different techniques and styles, and with a different understanding of the product from that of the original authors. Given the limited documentation that is the custom in the M community, knowledge was lost in the transitions between programmers.

The process of knowledge attrition and unanticipated changes culminates in systems operating in obscure and arcane ways, presenting information that, though perhaps accurate, is not certifiable. Such systems do not inspire confidence in their users.

The systems often cannot be easily replaced, however, because their idiosyncrasies have over time come to mirror the idiosyncrasies of the organization. As these systems mature, they ultimately embody the rules and structure by which the organization operates. System and organization have become an organic whole. Software developed over twenty years of effort has become, by definition, a legacy. Though now these cranky, clunky systems are viewed with contempt, organizations heavily depend on them. Businesses put up with the gradual calcification of the architecture because the prospect of rewriting the system is viewed as painful or impossible.

In fact, a legacy system is a mine of knowledge. The model of the organization is embodied in it. The flow of transactions is described by it. And the information required from and for users is encoded in it. In a given organization, the logical structure of the legacy system may be the only reliable or unambiguous source for knowledge of the organization's business.

The effort of replacing a legacy system can take several forms:

- The legacy system can be rewritten. That is, the effort is made to change the system by rebuilding on the existing code. This requires a thorough understanding of the systems operation.
- A new system can be written in-house to replace it. New code and/or a new platform is used to replace the old in its entirety. A new system can be written only if the rules by which the organization operates (embodied in the legacy system) is understood.
- A modern system written by some third party can be purchased to replace it. The proper replacement system can only be chosen if the business rules are understood.
- A wrapper can be built that puts a modern face on the old code. Wrappers must be written with a thorough knowledge of the underlying target code.

All of these methodologies have at their heart an understanding of the legacy system as it currently operates and a means by which the legacy system can be modified or replaced.

Understanding the Legacy System

The goal of understanding the legacy system has two components: finding ancillary information outside the system by interviews, by reading documentation, or by watching the system in operation and by studying the legacy system itself. It is the latter that will be discussed here: studying the internal dynamics of the legacy system.

There is a human component to any legacy system study—no automated means can replace it. However, while code *apprehension*, defined here to be an understanding of code operation from the reading of static code, is a human trait; code *auditing*, defined here to be the accumulation of code facts, is the province of software. This is why code analysis tools were originally developed even decades ago.

M is particularly amenable to code auditing. It is a clearly defined language. While the complexity of a given system is potentially quite large, the components of the language are quite simple. Indirection and XECUTEs are exceptions to

this rule. However, most systems use indirection rather sparingly and in easily characterized ways that, once discovered, can usually be incorporated into rules and processed like any other part of an M system.

This is not, of course, universally true. Indirection in system level code can often be esoteric and not at all amenable to either automatic analysis or automatic regeneration. It is also said that all M systems can be simulated by the line of code "READ X XECUTE X"—an analogy to the way that all systems can be modeled by an appropriate use of Turing machines. It is equally true though, that systems that make extensive use of idiosyncratic indirection or overuse XECUTEs, are also those systems that are typically rewritten by hand or replaced.

Most code apprehension techniques can quantify uses of variables, calling trees, parameter linkages, etc., of an M system. These are the individual bones of the M system's skeleton. Auditing an M system is more complex and difficult. The limits of auditing even comparatively small M systems are more computational than anything else.

The full examination of code auditing is beyond the limits of a single article. We can, however, illuminate some of the problem by viewing some of the issues of variables and the data they contain.

M makes liberal usage of variables. Variables in any given M process have a lifespan from the time their scope is created to the time when it ceases. Following a variable's computational trail can be quite useful in determining the possible meanings of naked references, the resolution of code module definition, or the nature of data that ultimately is placed in a global.

The general lifespan of an M variable can be shown in the following flow chart:

variable name allocation	NEW or start of process. (no allocation at start of process)
↓	
variable binding	SET, READ or MERGE command
↓	
variable reading	use of var in expression
↓	
variable destruction	end of context or KILL command.

The concept of a "module" is not clearly defined in M. The definition of a module is essential to any discussion of M code. Modules are the means by which the functional analysis of code must proceed. For our purposes, we can consider a module a discrete unit of M commands and expressions that has a defined entry point callable by a DO or GOTO and a defined exit point using a QUIT.

Once the concept of module is established, it is useful to have the concept of "arc" defined. An arc can be described as the code processing of M variables or data. It is separate from but related to scope since scope is tightly coupled to the behavior of the NEW command. The arc of a variable is often independent of the data the variable contains. The arc of data can span modules and variables, has the ability to fork, be modified, and other properties beyond the purpose of this article.

The arc of any M variable is not restricted to a given module as it is in many languages. The relationship between the variable arc and the module is one of varying independence. Variables can be set in one module, read in a second, and killed in a third, all without a parameter being passed. Coupled with M's embedded concept of late binding, this can make the analysis of a given M module difficult.

Legacy systems were commonly developed before parameter passing was part of the language. This makes analysis difficult because the variable requirements of a subroutine, module, or routine are implied rather than explicitly defined. Some legacy system designs can have the effect of making the auditing of any single module trigger the auditing of the entire system. There are systems currently in operation in the M community where this is the case.

For example, consider the dependence of variables in the following code:

```
MOD(A,C)    ;
            S B=A+C
            S D=A*C
            I D>B S B=10
            W !,B
            Q
```

The values of A and C are, of course, dependent on the outside invocation of the subroutine MOD. They can therefore be functionally described as:

```
A    ==   f(ext(A))
C    ==   f(ext(C))
```

where "ext(x)" is defined to be the external binding of the variable value "x".

The first iteration of B, B.1, is defined within the context of the MOD and is dependent on the values of A and C. D is similarly dependent but based on a different expression:

```
B.1  ==   f(ext(A))+f(ext(C))
D    ==   f(ext(A))*f(ext(C))
```

B.2, the second iteration of B, is *conditionally* dependent on the value of D:

```
B.2  ==  f(D)>f(B.1):10 OR f(B.1)
```

The matrix of what B.2 can be, therefore, is the following:

```
B.2  ==  |f(D)>f(B.1):10|
         |f(B.1)        |
```

Assume that there are two values each of A and C on entry into MOD:

```
A    |1|    C    |1|
     |2|         |3|
```

The possible combinations of A and C then become the following:

```
MOD(A,C)     |1   1|
             |1   3|
             |2   1|
             |2   3|
```

B.1 and D then become:

```
B.1    |2|    D   |1|
       |4|        |3|
       |3|        |2|
       |5|        |6|
```

B.2 then can be equal to:

```
B.2   |1>2:10 OR B.1:2 ==>  2|
      |3>4:10 OR B.1:4 ==>  4|
      |2>3:10 OR B.1:3 ==>  3|
      |6>5:10 OR B.1:5 ==> 10|
```

This is an extremely simple example. Yet, the logic that must be used to describe it is rather complex. Even this relatively simple result is not straightforward. The range of possible values of B.2 is only four because we know the actual values. More often, the values passed to a given routine are not known but only describable functionally. In that case, the number of possibilities for a given variable are the products of the number of possibilities for its constituent parts. The example variables can then be described as:

```
f(B.1)   ==   |f(ext(A(1)))+f(ext(C(1)))|
              |f(ext(A(2)))+f(ext(C(1)))|
              |f(ext(A(1)))+f(ext(C(2)))|
              |f(ext(A(2)))+f(ext(C(2)))|
```

4 possibilities

```
f(D)     ==   |f(ext(A(1)))*f(ext(C(1)))|
              |f(ext(A(2)))*f(ext(C(1)))|
              |f(ext(A(1)))*f(ext(C(2)))|
              |f(ext(A(2)))*f(ext(C(2)))|
```

4 possibilities

```
B.2      ==   |f(D(1))>f(B.1(1))=>cnst   f(B.1(1))|
              |f(D(2))>f(B.1(1))=>cnst   f(B.1(1))|
              |f(D(3))>f(B.1(1))=>cnst   f(B.1(1))|
              |f(D(4))>f(B.1(1))=>cnst   f(B.1(1))|
              |f(D(1))>f(B.1(2))=>cnst   f(B.1(2))|
              |f(D(2))>f(B.1(2))=>cnst   f(B.1(2))|
              |f(D(3))>f(B.1(2))=>cnst   f(B.1(2))|
              |f(D(4))>f(B.1(2))=>cnst   f(B.1(2))|
              ...etc.
```

where "cnst" denotes a constant.

It is obvious that the computational complexity of the analysis increases exponentially. Complete variable auditing of a particular system may not be practical due to computational constraints. However, the variable auditing of a given module or selected variable can often be done. Successive analyses will generate more information with which to tailor the analysis. Thus, tracing the lifespan of a given variable for specific reasons is often possible and useful.

Variables, as important as they are to code, are merely vehicles for data. The passage of data through a system—the data *arc*—is more significant. Consider the following code:

```
S A="TEST"
S B=A
S C=B
```

In this most trivial of examples, the important information is not the list of variables containing the string "TEST". Rather, the path of "TEST" and its ultimate destination and form is much more interesting. It should also be noted that data can "fork". Consider if the value in A was not "TEST", but a bank balance. B is its local repository. The value in C also contains the same bank balance. In effect, the data originally in A has now forked into the variables B and C, as well as being retained in the variable A. Each of these three variables can now be manipulated *physically* independently of the other two, though they could each be logically related to the original data in A. The *arc* of the data in A is continued through the variables A, B, and C as they are manipulated or until the information is destroyed or permanently stored.

If the code were changed in the following way:

```
S A="TEST"
S A="bank balance"
S B=A
```

The arc of the data "TEST" extends no more than one command since "TEST" is destroyed in the following set. In the above example, if the initial value of A is significant, resetting it—destroying the data—can have potential side effects. Finding such occurrences is a good result from finding out the data arcs.

Simply put, M systems take in data, process it according to some rules as expressed in code and store it in M globals and the reverse. The variables that contain the data on its way to the global are not significant. Only the data, its operations and its ultimate destination are significant.

Consider the following code:

```
            D MOD(1,10)
            Q
            ;
MOD(S,A)    ;
            S A=A*5
            S ^G(S)=A
            Q
```

From the above example, we can define the variable A and S in the following way:

```
A.1  ==  f(ext(A))
S    ==  f(ext(S))
A.2  ==  f(A.1)        [A.1*5]
```

G(S) then can be described in this way:

```
G(f(ext(S)) ==   f(A.1)
```

The data residing at node f(ext(S)) is defined in a general form of f(ext(A))*5. In effect, this analysis has yielded a definition of the required operations for that particular node. This act has two profound side effects: 1) it is the first step in defining a data dictionary for the global in question and 2) it describes the global transaction.

M data handling is far, far more complex than this of course. At first glance, tracing the variable or data lifespan in a given M system appears to be an intractable problem. The method of solving any problem that appears intractable is invariably the same: solve what can be solved and then analyze the remainder. Such an approach can achieve significant results. We have found that a surprising amount of transactional information can be obtained with automated methods. These methods are limited by the constraints described here, but they are useful nonetheless.

Only variables have been covered here. Other components of M code are amenable to similar patterns of analysis. The analysis of a legacy system using these methods could yield dataflow diagrams, modular calling trees based on the flow of information through the code and the basic outlines of possible transactions.

Rewriting the Legacy System

Once a good understanding of the operation of the legacy system is established, a methodology by which the system can be adapted to the organization's needs must be selected.

The four kinds of approach were mentioned in the introduction:

- The legacy system can be rewritten.
- A new system can be written in-house to replace it.
- A modern system written by some third party can be purchased to replace it.
- A wrapper can be built that puts a modern face on the old code.

The four approaches are not mutually exclusive. Some components of a given system could be replaced while other components could be rewritten. If the system is just replaced or written anew from scratch, then rewriting is not necessary.

However, sometimes the operation of a legacy system can suit an organization so well that alternatives are inadequate. If this is so, then the legacy system must be adapted somehow to the modern computing world. This can be done by hand. For simple changes, where the operation of the system is thoroughly understood, or where the cost of the adaptation is no object, changing the system by hand could be the best decision.

In other cases, the amount of information that is needed or the scale of the changes to be executed can make automated means the method of choice.

An automatic method can be as simple as using a routine to change the occurrence of every string or as complex as an AI tool that attempts to analyze the intent in the mind of the original programmer. The best solution is usually a compromise between the two.

We have found that most of the time spent building the tools supporting a recoding effort fall into two basic areas:

- analyzing the code in sufficient detail to determine unambiguously the execution of the code
- describing the code in an abstract manner.

The abstracted code then becomes the base for manipulations, not the original code itself. After the abstracted code is modified, it is then reconstructed; code rewriting occurred only on code assembly:

base code
↓
abstracted
code model
↓
manipulation
of abstracted
model
↓
code reassembly

The goal is to be able to take any block of M code and reconstruct *functionally equivalent code*. No attempt is made to bring original code forward. Manipulation of the abstracted model would result in functionally superior code. It is also possible that the resulting code could be formally provable,

though this is not a current avenue of our research. Certainly, a system regenerated in this manner would be better able to take advantage of automated QA tools already present in the market.

Regenerating code has many beneficial side effects even when no manipulation of the abstracted code occurs. For example, long lines can be broken down into constituent parts. Dependent clauses, the arguments of FORs and IFs, can be turned into argumentless DOs, if that is desired. Direct references to specific globals could be transformed into data dictionary references or processed into calls to a common filing or retrieving program.

Conclusion

The main advantage of automatically analyzed and reconstructed code is one of thoroughness and reproducibility. Machines do not get tired of doing the same thing over and over again. If they do a thing once correctly, they will do the same thing correctly a thousand times. This is their virtue and why we use them. We have developed tools that can automatically analyze M code and derive an abstract model of the code. We are currently using these tools to derive other programs to both analyze M code and derive information regarding variables, global transactions and data lifespan, as well as building new code based on a modified abstract code model.

We have also found the automated analysis of code reveals issues and problems that could only have been foreseen by the original authors or after great effort. The information so gleaned becomes grist for the mill of further analysis as well as the generation of other enabling technology such as data dictionaries, common calling tools, and the building of administrative tables.

Techniques and algorithms such as those we have developed should be considered enabling technology, to be used to bring forward legacy systems and their inherent knowledge into the modern computing community. M

Steven Popkes is VP in charge of Research and Development of Jacquard Systems Research. He has been using M since 1978. JSR specializes in tools and consulting to regenerate systems by automatic methods.

Calendar

October 15–19, 1995

ACM's 10th OOPSLA (Object-Oriented Programming, Systems, Languages, and Applications), Austin, Texas. For more information on the meeting contact the OOPSLA '95 Office, 7585 SW Mohawk Street, Tualatin, Oregon 97062. Phone: 503-691-0890; fax: 503-691-1821; email: oopsla95@acm.org.

October 28–November 1, 1995

Symposium on Computer Applications in Medical Care (SCAMC), New Orleans, LA. For information contact AMIA, 4915 St. Elmo Ave., Suite 401, Bethesda, MD 20814. Ph: 301-657-1291, Fax: 301-657-1296.

October 29–31, 1995

Healthcare Information Management Systems Society (HIMSS)'s NETCON '95, Keystone, CO. "The Evolving Healthcare Network: Growth Technologies and Applications for the Reformed Health System." For information call 312-664-4467, ext. 116.

November 6–10, 1995

MTA-Europe Annual Meeting, Barcelona, Spain. For more information, contact the MTA-Europe Office, Avenue Mounier 83, B-1200, Brussels, Belgium. Phone: 32-2-772-9247; fax: 32-2-772-7237.

March 24–28, 1996

MTA Annual Conference with Database and Client/Server World Conference and Exposition, Boston, MA. Registration booklet will be mailed early November.

Advertiser Index

We appreciate these sponsors of the September issue and all the companies who support the M community through their commitment to excellence.

This index appears as a service to our readers. The publisher does not assume any liability for errors or omissions.