Unwrapping

David W. Coppit and Kevin J. Sullivan
University of Virginia Department of Computer Science
Thornton Hall, Charlottesville, VA 22903
Email: {sullivan,coppit}@cs.virginia.edu

Technical Report CS-98-08
May 4, 1998

Abstract

A key driver of software obsolescence is change in hardware and system software standards. In the area of software tools, there is now great pressure to host on Intel/Windows platforms tool functions that in the past were typically found in Unix or mainframe environments. In such cases, there can be value in reusing core parts of such legacy systems. Two standard methods for doing this are reengineering and wrapping. Reengineering, being unrestricted in the changes allowed, permits the removal of obsolete parts of a system but creates the risk that changes will break the complex and brittle reused code. Wrapping involves the reuse of existing code without change, but at the cost of including obsolete elements into the new environment. In this paper we propose *unwrapping* as a new synthesis of these two approaches. To unwrap is to remove unwanted design elements, but with a strong emphasis on leaving core code unchanged. We discuss our preliminary use of this approach to reuse core elements of two Unix-based reliability engineering tools in a modern tool based on package-oriented programming.

1 Introduction

Existing software systems often represent major capital investments. Unfortunately, even well designed software systems tend to depreciate over time, creating an increasing incentive to be able to reuse the still-valuable core functions in new contexts. There are two traditional approaches to the exploitation of code embedded in depreciating systems. The first is *reengineering*, which Chikovsky and Cross define as "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [10]." The second is *wrapping* [1, 33], in which an existing system is wholly incorporated into a new one behind an information hiding interface.

A strength and weakness of reengineering is that it tacitly sanctions arbitrary changes to existing code. The benefit is that suboptimal elements can be removed and beneficial architectures can be imposed. On the other hand, complex legacy codes that have stabilized over a long time are brittle and often poorly understood. Reimplementing them can be very costly, and changing them can create significant risks. A key benefit of the wrapping approach is that it avoids the cost of reimplementation and the risk of breakage involved in changing such code. However, it incurs the cost of incorporating what can be significantly suboptimal design elements of the old system into the new environment. For example, wrapping a Unix-based tool to implement a Windows-based tool would involve the need to maintain the whole legacy Unix infrastructure in order to host the now wrapped legacy tool, which would typically be invoked over a network. That is clearly not what most users of a Windows-based tool want or expect.

In this paper, we contribute *unwrapping* as a new strategy for legacy integration that promises to reap many of the benefits of each of these approaches without incurring all of their disadvantages. Our approach can achieve much of the elimination of unwanted code, as is possible with general re-engineering. But like wrapping, unwrapping acknowledges the delicacy of legacy code, and so emphasizes the need for no disruption of existing code, with the possible exception of changes whose

correctness is easily verified.

1.1 The Basic Strategy

Unwrapping is the removal of unwanted code by identifying an interface that partitions a system into a core providing the desired functionality of a system and a superstructure containing unwanted elements. Unwrapping involves the discarding of superstructure elements surrounding valuable code. It is often complicated by the complex and poorly understood coupling between the two. Thus, in general, the first step in unwrapping is to analyze the structure of the system in order to identify an interface — though not necessarily a "clean" one — at which the cleaving off of the superstructure will occur. In some cases unwrapping is partial, with unwanted but hard to remove old code carried into the new system, e.g., to avoid the need to disrupt brittle core code. We call such unwanted code elements "warts."

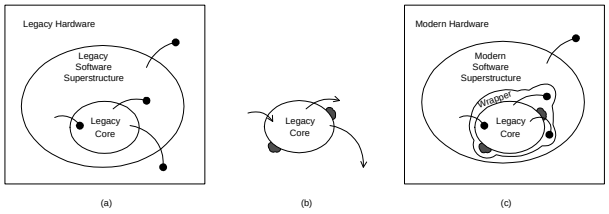

Figure 1: The reuse of a system using unwrapping.

Figure 1 illustrates the idea. Figure 1a depicts valuable core code embedded in the legacy superstructure hosted on a legacy platform. It also depicts dependencies of the core code on the legacy superstructure, and of the superstructure on the core. Figure 1b depicts the partially unwrapped core, with warts and now unresolved dependencies included. Figure 1c depicts the integration of the unwrapped core into a new system. Instead of the entire old system, only the unwrapped code, along with any unremoved warts, is wrapped for integration into the new system. The new wrapper resolves the dependencies required for the core to operate, and provides an interface to the core for use by the new system.

1.2 The Objective

The goal in unwrapping is to maximize the value added through the reuse of core code. The value added in a particular case depends on several factors: the value of the core code isolated for reuse; the cost of isolating the core code; the cost of unwrapping and then wrapping the code; the cost of any disruption of the old core code; and the short and long term cost of integrating warts into the new system.

Various tradeoffs are possible along an unwrapping design spectrum. For example, time-to-market might be improved at the expense of long term evolution costs by wrapping more of the existing system. The challenge to the designer, then, is to pick a point along this spectrum that maximizes value added at a particular point in time in light of available resources, tolerance for risk, uncertainties, time-to-market constraints, etc. One consulting company has noted that, "Picking the right application system layer and product is critical... Wrapping large scale, monolithic applications rarely makes economic or technical sense [31]." At one end of our spectrum, then, we have complete reuse of the old core code with no unwrapping. At the other end is unconstrained reengineering, in which the desired functionality is re-constituted, possibly from scratch.

1.3 Application to Tools

In this paper, we focus on the specific domain of software tools for engineering modeling and analysis. This domain is broad enough to be interesting as a subject of software engineering research, and it is one of considerable practical importance; but it is also narrow enough to be characterized by certain common structures. In particular, we claim that many engineering modeling and analysis tools are organized roughly as computational cores surrounded by what we call superstructures for input, output, graphics, and so on.

Many existing tools were designed to operate on Unix or mainframe platforms. These tools are now losing significant value because of the heavy premium now placed on tools that run on personal computers. We are exploring an approach to modernizing these existing tools by integrating their computational cores into new superstructures built from multiple shrink-wrapped software packages [28].

In the next section we make the idea of unwrapping concrete by discussing the integration of legacy tool code into our fault tree analysis tool called Galileo. Galileo is an engineering tool into which we have integrated computational cores from two Unix-based legacy fault tree analysis tools: DIFTree [18] and MCI-HARP [4]. Next, Sections 3 and 4 present details of the unwrapping of the legacy cores. Section 5 presents our evaluation of the unwrapping concept based on work done to date. Section 6 discusses research related to unwrapping. Section 7 concludes with a discussion of options for future work.

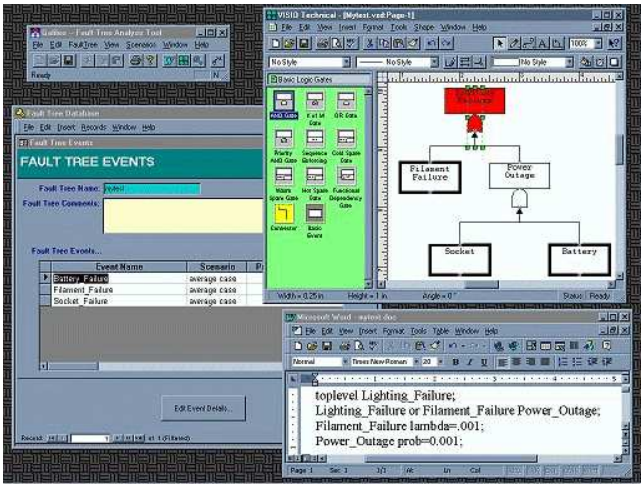

Figure 2: Screen shot of Galileo.

2 Case Study: Galileo

The concept of unwrapping evolved during our work on the Galileo project [29], an experiment in the use of *package-oriented programming* (POP) techniques in the development of software tools. Package-oriented programming is an approach to software development in which multiple shrink-wrapped packages, used as large-scale components, are integrated into systems.

Galileo is a tool for reliability engineers that provides the ability to edit, store, and solve fault trees [32]. See Figure 2. Fault trees are models of system failure, where the top-level event represents the failure of the whole system. The failure of the system is dependent on some combination of subsystems, which themselves can be broken down into subsystems until the "basic events" of the system are reached, whose failure characteristics are specified. Our original hope was to be able to quickly design and implement the editing and storage capabilities using POP, and then to be able to "just plug-in" different computational cores from existing systems [30].

Unfortunately we found the core elements of the existing systems to be complex and brittle. The complexity made us wary of rewriting the code (or even of changing it) for fear of getting a new implementation wrong or of breaking the existing one. Nor did we want to commit to investing the time and resources necessary to reverse engineer and then reimplement the existing code given the uncertainty and risk that there would be insufficient demand for our tool. Moreover, the useful parts of the systems were embedded in complex, legacy superstructures. For example, the core of the DIFTree tool (which we discuss next) was hosted in an implementation of graphical and textual manipulation functions that were tied closely to Unix and its pipe-and-filter programming model.

We wanted to reuse the core code without change, and without being tied to a Unix environment, which would have been wholly inconsistent with our goal of providing a PC-based tool. We also wished to remove the the original DIFTree editing and storage capabilities, which were already provided by the Galileo tool. We now discuss the process of unwrapping that allowed us to meet our objectives.

3 Unwrapping DIFTree

The goal of the Galileo project was to provide a tool supporting the same function as DIFTree. DIFTree solves fault trees with static and dynamic elements by splitting these trees into static and dynamic subtrees, which are passed to engines that use different solution algorithms. The technical details are irrelevant here.

3.1 DIFTree's Original Implementation

The design and implementation of DIFTree, comprising 32,159 lines of C++, C, Perl, Tcl/Tk, and Python, mirrors an earlier Fortran system. It was not well modularized to begin with, and its structure was further compromised by the loosely coordinated modifications made by four graduate students over several years. The system has degraded structurally, is not well documented, and is hard to understand. Developed for use on Unix, it is structured as several applications that communicate with each other, with the user, and with the operating system through a variety of interfaces, as illustrated in Figure 3.

The rectangles in the middle of the figure represent the different Unix programs constituting the tool. The arcs represent data flows between these programs and their supporting environment. The outer oval shape is meant to suggest that the central computational elements of the tool are embedded in and to some extent intertwined with a superstructure consisting of a graphical front-end and the Unix operating system: its console interface; file system; and command shell, as invoked by `system` (program invocation) calls. The shapes in the lower oval represent aspects of Unix that the tool uses. For example, the tool maintains a cache of partial results within the file system.

The user creates fault trees with the graphical front-end implemented in Python and Tcl/Tk. When the analysis is begun, the front-end outputs the fault tree to a file using a text-based representation, and invokes SplitTrees. The user then enters analysis parameters through the SplitTrees console interface. SplitTrees reads the fault tree file and partitions it into subtrees

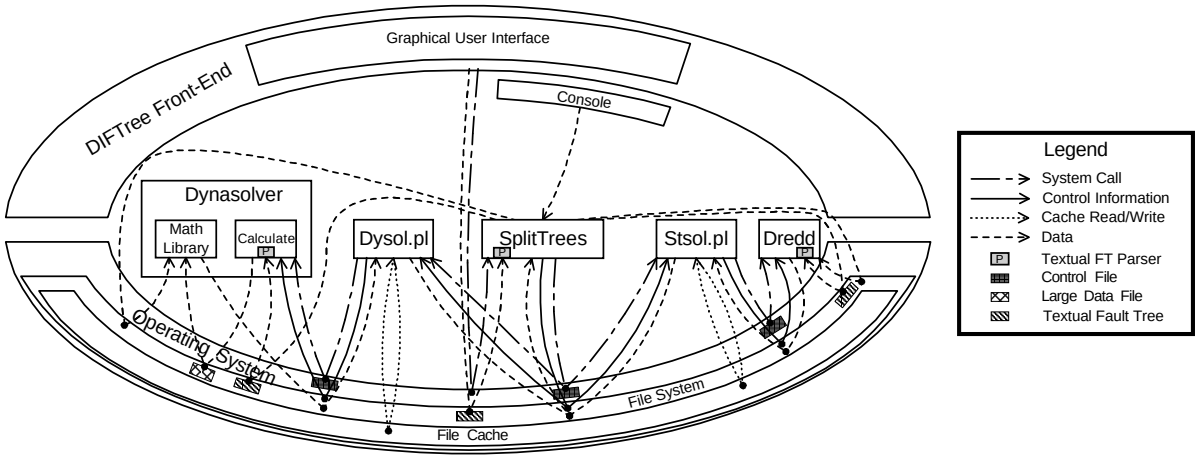

Figure 3: File system and console usage in DIFTree.

that are output as text files for use by DynaSolver and Dredd. SplitTrees generates control files containing simulated user input for Dysol.pl and Stsol.pl (Perl programs), which are invoked with Unix `system` calls.

Dysol.pl and Stsol.pl consult the file-system-based cache of results of previous computations. If a cached file matching the current inputs is found, Dysol.pl or Stsol.pl copies the cached data to make it appear as the expected result. In case of a cache miss, Dysol.pl or Stsol.pl generates another control file containing simulated user input to drive either DynaSolver or Dredd. Dysol.pl and Stsol.pl capture and cache the output files generated by DynaSolver and Dredd and then return the results and control to SplitTrees.

As an added complication, DynaSolver uses the file system internally to store temporary values involved in producing the matrices that are ultimately passed to the mathematical library. This kind of implementation detail, which results in embedded dependencies on the legacy environment, complicates the process of legacy code reuse. A key part of the unwrapping approach is to locate and then to "rebind" these kinds of dependencies. By rebind we mean to identify and create surrogate implementations to replace functionality not provided by the new environment.

3.2 Unwrapping the DIFTree Computational Engine

We now describe the unwrapping process and its application to DIFTree.

3.2.1 Find a Suitable Internal Interface

The first step in unwrapping a legacy system is to identify an interface that separates the core from the superstructure, consistent with value maximizing tradeoffs under uncertainty and limited resources. For example, the best interface might be at the outermost boundary of the entire system. That is, a traditional wrapping approach might be best, in which one communicates via remote procedure call to a legacy application hosted on obsolete hardware. Examples of factors to consider while choosing a suitable interface are: the time available; the labor required in removing the superstructure at that interface; the engineer's knowledge of the system; the warts that will remain after unwrapping is finished; and the superstructure dependencies that will have to be reimplemented in the new environment.

A common tradeoff, in our experience, is the speed of integration versus the number of warts remaining in the unwrapped core. An unwrapping approach using an external interface may realize short-term gains more quickly, while incurring higher long-term costs, performance degradation, and possibly externally visible architectural incoherence. On the other hand, an approach that utilizes an internal interface that bounds the desired core code tightly may result in a cleaner integration, but requires a higher up-front investment.

Our requirements made the option of wrapping DIFTree in its entirety unattractive. One alternative that we considered briefly was to unwrap it to enough to make it independent of the Unix operating system, but to leave it structured as a set of separate processes. We discarded this alternative because it required the use of Perl and yielded a poorly structured design that would have been costly to manage, even in the short term. Instead, we decided to unwrap more aggressively, so that the core code would be integrated into Galileo as a statically linked library. That degree of unwrapping exposed several dependencies that we had to rebind, including on the console. Concern regarding performance

also led us to cleave off the Unix file system.

3.2.2 Identify Dependencies

The next step in unwrapping is to identify dependencies of the core on parts of the superstructure being removed. In some sense the consideration of dependencies plays a role during the selection of a suitable unwrapping interface, since tight coupling increases the cost of separation. Likewise, the difficulty of resolving dependencies between the core and its new environment can affect the location of the unwrapping interface.

In addition to the file system and console dependencies mentioned earlier, DIFTree relied upon other aspects of its environment. The simplest case was a system call within SplitTrees to the Unix `date` program, which, on the Windows operating system sets the date instead of prints it. In invoking Dysol.pl and Stsol.pl, DIFTree depended on the availability of Perl, and on the meaning of shell redirection. DIFTree also depended on the Gnu C++ library [20], since it relied on particular definitions and on a memory allocation component called Obstack. Furthermore, the code assumed a *specific version* of the compiler that allowed the use of constructs that later versions did not. Lastly, DIFTree assumed the the console to be a part of the user interface, an incorrect assumption for the Windows-based Galileo.

3.2.3 Physically Extract the Core

The source files we removed entirely were those related to the graphical front-end and the parsing of the textual representation of a fault tree, as well as Dysol.pl and Stsol.pl. During the removal of the superstructure code, we realized that we had a second objective: the backward compatibility of the newly unwrapped code with the Unix operating system. We therefore decided to enable compilation in either environment by separating the remaining, shared, source files using conditional compilation, a change that was easy to reason about.

A particularly difficult aspect of the legacy environment was Obstack, a memory management class, from the Gnu library mentioned earlier. At first we intended to simply remove the dependence on this component, but our lack of understanding of its precise behavior and its widespread use in the desired core code suggested that this would be risky. Rather than change the core so that it was not used, we decided to port it — as a wart — to the new environment.

3.2.4 Rebind Dependencies and Wrap Interface

After core code has been unwrapped, exposed dependencies must be rebound. A wrapper around the extracted core can serve this purpose as well as to provide a better interface for use by the new host environment. Warts from the legacy system become part of the core's wrapper, as does any necessary functionality that must be reimplemented as a result of being removed during the unwrapping process.

For example, we rebound DIFTree's use of the Unix `date` program to standard date routines in the C++ library, and we ported Obstack from the Gnu libraries. Similarly, the wrapper implemented a graphical window consistent with the user interface of the Galileo tool, and redefined the C++ `cout` and `cin` within the core to use it instead of the Unix console.

We used a variety of approaches to rebind dependencies on the Unix file system. We removed the user interfaces to DynaSolver and Dredd and replaced them with direct function calls, passing simulated user input contained in the control files as arguments. We replaced the large complex data file used within DynaSolver to store temporary values with a stream to memory by rebinding the C++ `ostream` file interface to a `strstream` memory-based one. In this case we we achieved significant performance improvement by avoiding the file system, even in the face of a tradeoff in which we sacrificed a possibly cleaner non-stream-based method of communication. Since the semantics of `ostream` and `strstream` were nearly identical, we were confident that we did not adversely affect the code.

DIFTree uses three different fault tree file representations as input to SplitTrees, DynaSolver, and Dredd. Unlike the large DynaSolver file, these files were easier to understand, which allowed us to identify syntactic elements that corresponded to structures within the Galileo abstract data type (i.e. object-oriented) representation of a fault tree. Using this correspondence as a guide, we took the opportunity to convert the non-standard file communication into parameter passing of a common Galileo fault tree representation. We accomplished this by unwrapping SplitTrees to remove its file output, and then creating a wrapper that populated the data structures of a Galileo fault tree object.

Since the input routines of SplitTrees and DynaSolver used LEX and YACC [19, 21] we were able to leverage the semantic action rules of the parsers to determine how to populate the legacy data structures based on a given Galileo fault tree object. For example, when the original system parsed a basic event and its parameters, the `add_params()` function in the legacy core was called. Similarly, our wrapper acquires the analogous parameters from the Galileo abstract data type and calls the same function. Thus, we engaged in a re-engineering task within the overall unwrapping exercise.

Our approach to removing and rebinding the file

caching implemented by Dysol.pl and Stsol.pl was to create a hash data structure in memory in which the keys are the inputs to DynaSolver or Dredd and the values are the resulting outputs. Since the result returned by the solvers is small in size, the amount of memory required to hold the hash values is correspondingly small (about 6 bytes for each subtree solved). If the output had been large or the output code not well encapsulated, the stream-to-memory approach discussed earlier could have been used.

3.2.5 Integrate Core Into New System

Once the core has been wrapped to resolve exposed dependencies and to provide a suitable interface for use by the new system, it is integrated into the new environment. (In fact, wrapping and integration occur simultaneously to some extent.) The first milestone in integrating DIFTree was the modernization of the code to remove its dependence on the outdated Gnu compiler. The next was the removal of the dependencies on Unix, followed by the verification that DynaSolver could compile and run as a stand-alone application on an Intel machine.

During integration, we found that unanticipated interactions between the newly integrated core and the new environment had to be resolved. In particular, at the next milestone of integrating DIFTree as a statically linked library, we found that DynaSolver and SplitTrees referenced different global variables by the same name. As a result, we had to encapsulate the two to scope the variables correctly. Later, we found a more serious integration problem in that all of the legacy components were designed to simply abort execution on detecting an error. This approach was workable in the context of loosely coupled Unix applications, but it was untenable for tightly coupled components within a Windows-based program. To resolve this issue, we modified Galileo to validate the input before the core code was invoked in order to ensure that none of these error conditions could occur; and we provided error-handling code that would gracefully exit Galileo in the case of catastrophic error. In other words, we had to reverse engineer the core code to derive preconditions strong enough to prevent that code from reaching certain points.

3.2.6 Results

The careful unwrapping and integration of DIFTree into the Galileo project resulted in the merger of a rich user superstructure and a state-of-the-art analysis functions. By making DIFTree available on the Windows operating system we have expanded the market for the tool, so that over 230 people have downloaded it over a ten month period. Our work also created a close collaboration with engineers at Lockheed-Martin, which has helped in the development of both Galileo and DIFTree. Having resolved the uncertainty over the demand for our tool while hedging our risks on the cost front, we are now undertaking a more costly and comprehensive reengineering of the legacy core to resolve long term maintenance and correctness problems.

Referring back to figure 3, the unwrapping interface we chose included the core elements of SplitTrees, DynaSolver, and Dredd, except for the textual fault tree parsers, textual output routines, and `system` calls. Figure 4 shows the structure of the resulting implementation, where DIFTree's core elements have been unwrapped, their legacy dependencies removed, and then wrapped. We rebound dependencies on the file system to memory-based methods, the console user interface to a graphical one, and operating-specific `system` calls to function calls. The Galileo fault tree objects are now the common format exchanged between SplitTrees and the subtree solvers, with object translators replacing the parsers in the legacy system.

4 The MCI-HARP Case Study

Our second application of unwrapping was the integration, into Galileo, of another fault tree solving engine called the Monte Carlo Integrated Hybrid Automated Reliability Predictor (MCI-HARP) [4]. Using Monte Carlo techniques, the program simulates stochastic failures of the basic elements of a fault tree to determine what ratio lead to overall system failure. Unlike our careful integration of DIFTree, our goal for MCI-HARP was to produce a prototype integration quickly as a proof-of-concept for purposes of demonstrating the potential of package-oriented-programming to collaborators at NASA. The tradeoffs made during the work described in this section were in favor of time instead of long-term system structure.

MCI-HARP has had a long history, beginning in 1981 as simply HARP, developed at Duke and Clemson universities [14]. About seven years later HARP was used in a Monte Carlo simulator (MCI-HARP) built at Northwestern University [26]. For about the past five years, the system has undergone several enhancements at NASA, and is now called MCI-HARP. All told, about 20 people at four institutions have worked on it.

Unlike DIFTree, MCI-HARP integrates computation and fault tree editing into one executable program. Its console-based interface is menu-driven, and it uses files as input and output and to store data between program executions. At about 38,000 lines of Fortran code, MCI-HARP is 18% larger than DIFTree, but is better modularized.

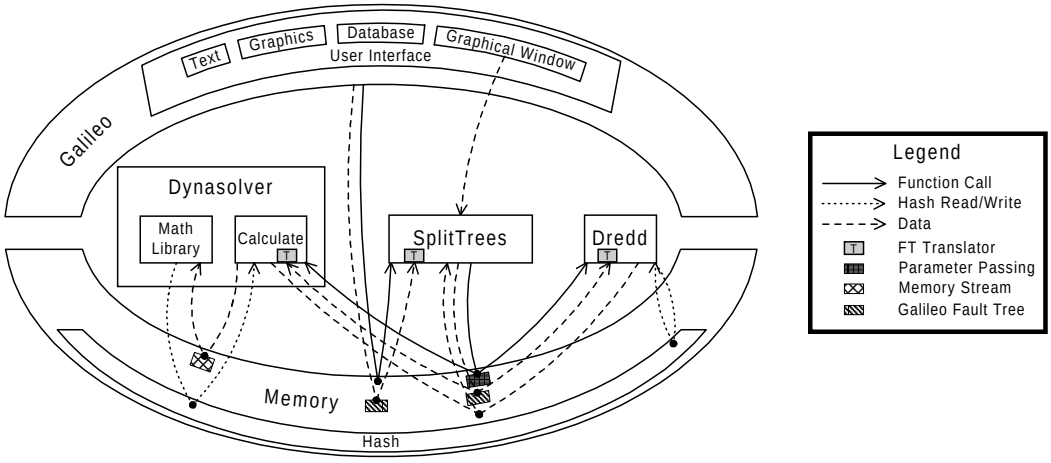

Figure 4: Reengineered DIFTree structure.

In contrast to the decision to remove dependencies on the files system made in the DIFTree project, we chose to retain the files used by MCI-HARP in order to avoid the costs associated with removing them. Because of the modular nature of the code, the interface to the computational core was easily identified as a simple function call, two input files, and a report in a text file for output. In choosing this interface, we excluded MCI-HARP's superstructure for editing of fault trees, although we chose not to remove the "dead code" resulting from our use of an internal interface.

Because MCI-HARP had been ported to several platforms (including Intel-based computers), there were no adverse dependencies upon the operating system, and no compiler-specific constructs in the code. However, the computational core did rely upon the input text files generated by the superstructure. We determined the extent of core's dependence on the input files by examining the contents of the files themselves and referring to the software's extensive documentation. We determined the formats of the data in the files by talking to the original developers of the software at NASA and the University of Virginia. This task was somewhat more difficult than with DIFTree because the files did not have an easily identified correspondence with the Galileo fault tree representation.

There were two major issues during the rebinding and wrapping phase. The first was the need to implement a wrapper that fabricated MCI-HARP's input files from a Galileo fault tree object and which then called the computation engine. This was straightforward given our previous reverse engineering of the file formats. Similarly, we wrote a simple routine to read the result of the computation from the output file and return the desired value.

The second issue was the legacy console interface. Here we encountered a significant difference between Fortran and C or C++: the `PRINT` statement in Fortran is part of the language and can not be redefined. This forced us to modify the code at every output statement. We wrote a small script that replaced each call to `PRINT` with a call to an external C++ procedure that we wrote. After rebinding the Fortran `PRINT` statement, we were able to use the same technique that we used in DIFTree to replace the standard `cout` with a GUI-based implementation. Thus, in this unwrapping exercise, we did change core code, but in a way that was easy to understand and verify.

5 Evaluation

Naïve wrapping of DIFTree in its entirety would have saved us the trouble of having to modify the original code, but it would have caused unacceptable problems:

1. inefficiencies of file-based communication
2. the need to implement a "screen scraper", a wrapper that captures console output and presents the data to an external program
3. the need to port part of the Unix operating system or remain on Unix and communicate with Galileo over a network
4. the need to distribute Galileo as multiple processes dependent on Perl and Python
5. a poorly constructed component within the well-structured Galileo system

Not including code generated by LEX and YACC, the original DIFTree system contained 32,157 lines of code, 16,415 of which was devoted entirely to a graphical front-end written in Python. Of the original code, 19,095 lines were removed as superstructure, and our wrapper code consisted of 1,428 lines.

Wrapping MCI-HARP without modification would have required us to write a screen scraper to capture console output, and to simulate extensive user interaction to create fault trees using its built-in support. Furthermore, following this course would have incurred extra overhead and redundant superstructure. Unwrapping MCI-HARP to remove its support for fault tree creation simplified the functional interface substantially.

As a prototype integration, we did not physically remove MCI-HARP's code resulting from unwrapping, however we estimate that about 1,700 lines of code were rendered unreachable by our modifications. The wrapper we constructed consisted of 283 lines of code, which does not include the GUI code that we reused from SplitTrees. In addition, 1283 lines of code were automatically added by the transformation that rebound the `PRINT` operation.

While it is not quite fair to compare across computer architectures and operating systems, a DIFTree solution that previously took about 45 seconds to compute on an unloaded 167 Mhz UltraSparc 1 on a network files system took only 6 seconds to compute on a single-user 133 Mhz Pentium. Informal timing of the original code using the Unix `time` command shows that of the 45 seconds, about 36 seconds (80%) were spent in a non-running state, most likely being I/O bound.

Using unwrapping followed by rebinding and wrapping, we were able to integrate two legacy systems into a modern application while simultaneously removing undesirable dependencies on the operating system, graphical and console interfaces, and foreign language processors. Part of the success of these projects is because the operating system and console interfaces represent levels of functionality that are well defined, narrow, and usually easily accessible. For tools, the separation between the core and superstructure appears likely to hold. It is not yet clear how successful this technique would be for other legacy systems.

6 Related Work

Our approach is clearly related to a great deal of work on reverse and re-engineering in general, and on reusable component extraction in particular. We also address work on wrapping and repackaging, as well as tool support.

6.1 Reusable Component Extraction

Much work has been done on the extraction of reusable components from legacy source code. Space prevents us from discussing many useful results. Our work could use many of the tools and techniques that have been developed. However, what is important is that, though perhaps not obvious to the casual observer, our work differs fundamentally from previous efforts in this area.

Previous work starts with the assumption that legacy systems *might* contain code with reuse value. The work seeks tools and techniques to identify and then to extract candidate reusable code, which is then evaluated and reengineered for purposes of populating a reuse library. Etzkorn and Davis, for example, seek natural-language and structure-analysis-based tools for "identifying reusable subroutines or code fragments in legacy systems [15]." Similarly, the *reuse re-engineering* work of Cimitle, Canfora, et al. [7, 8, 9], is founded on the basic notion of *candidature,* which involves the application of code analysis, often using semantically powerful tools, to identify potentially reusable elements in poorly understood legacy code.

Our work differs from these previous efforts in its basic assumption: we *know* that our legacy system contains code that we want to leverage. The difference is basic but profound. Our problem is not to ascertain the presence of potentially valuable code within legacy systems, but to extract valuable code that we already know is there, without breaking it. The concept of unwrapping appears to be a novel contribution to our set of intellectual tools for dealing with the migration of legacy systems.

There is little doubt that the use of code analysis tools, whether semantically rich or not, could assist in the unwrapping process. We have not used semantically rich tools to date, owing to the real-world messiness of our legacy systems, which are written in multiple languages, which use Unix-based composition mechanisms, and so forth. The use of semantically light-weight approaches, such as reflexion model techniques [22], promises to be especially valuable as an aid in unwrapping. Automated approaches to user interface reengineering such as that used by Csaba [12] and Claßen, et al. [11] might be used to help unwrap and rebind complex text-based user interfaces utilizing menus and windows. Tool support for automatically identifying system dependencies, such as the work of Baratta-Perez et al. [3], also offers a good first step in separating core code from superstructure.

6.2 Wrapping

Unlike unwrapping, wrapping is now a widely known approach to legacy code integration. Much work has been done in this area. Much of the interest comes from the business sector, where CORBA[23], OLE [5] and similar developments enable encapsulation of legacy systems behind distributed object-oriented interfaces. The theory of wrapping was addressed by Parodi [24],

who identified four major types of wrappers, and Aronica and Rimel [1] who examined implementation issues.

Wiederhold's CHAIMS [25] defines a high-level language for composing large modules, often wrapped versions of legacy systems running on legacy platforms and invoked by remote procedure call. Baker described procedures for wrapping C-language libraries using C++ [2], and Van Camp used wrappers to improve library portability [6]. Flint [16] wrapped legacy COBOL applications using an object-oriented wrapper. Reznick wrapped Unix applications to enhance their functionality [27]. The HP Encapsulator provides a wrapper-based framework for integrating Unix tools into the HP SoftBench environment [17]. Many of these techniques might be useful in wrapping core code once unwrapped.

6.3 Repackaging

DeLine et al. describe lessons that they learned from a case study in converting batch systems to support interaction [13]. The conversion of the UNICON system required them to "break open" the computation to modify the original assumptions made by the system with regard to partial processing, error prevention, control paradigm, etc. Our work differs from theirs in that they explored the problems associated with modifying the deep-cutting design decisions of a legacy system, while we retain those design decisions that impact the core in an effort to avoid the cost of comprehension and to mitigate the risk of breaking the code.

7 Conclusion and Future Work

As a design technique, unwrapping represents a promising point in the design space of techniques for legacy code integration. We have found that for two analysis systems designed as pipe-and-filter systems, unwrapping was a workable and successful approach.

It is our belief that in both projects the amount of effort required to use the unwrapping approach discussed in this paper was less than what would have been required had we simply wrapped the original system whole with only slight modification. In both case studies, the amount of code in the wrappers is far less than the external superstructure that was removed, and we believe that it was also less than what would have been generated by wrapping alone.

Choosing the right level of abstraction at which to intercept functionality is important. The engineer must weigh the relative costs of the possible interfaces, choosing one that achieves the desired functionality consistent with the given constraints.

Future work includes tool support for discovering levels of functionality within a legacy system. We also believe that there may be some benefit in more rigorously specifying the iterative unwrapping process to help reason about the cost of unwrapping at a particular interface.

8 Acknowledgements

This work was performed under NSF grant numbers CCR-9502029 and CCR-9506779. We thank Jimmy Ramsden and Tom Herald of Lockheed-Martin for their suggestions during the development of Galileo. We also acknowledge the many useful conversations with, and the work of, Gail Murphy and David Notkin.

References

[1] Ronald C. Aronica and Donald E. Rimel Jr. Wrapper your legacy systems. *Datamation*, 42(12), June 1996.

[2] Larry E. Baker Jr. C++ interfaces for C–language libraries. *Dr. Dobb's Journal*, 22(8):34, 36–7, 90–1, August 1997.

[3] Grace Baratta-Perez, Richard L. Conn, Charles A. Finnell, and Thomas J. Walsh. Ada system dependency analyzer tool. *IEEE Computer*, 27(2):49–55, February 1994.

[4] Mark A. Boyd and Salvatore J. Bavuso. Simulation modeling for long duration spacecraft control systems. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 106–13, Atlanta, Georgia, 26–28 January 1993. IEEE.

[5] K. Brockschmidt. *Inside OLE*. Microsoft Press, Redmond WA, second edition, 1995.

[6] Kenneth E. Van Camp. Using wrappers to improve portability of commercial libraries. *C Users Journal*, 11(1):35–37, 40, January 1993.

[7] G. Canfora, Aniello Cimitile, and M. Munro. A reverse engineering method for identifying reusable abstract data types. In *Proceedings of Working Conference on Reverse Engineering*, pages 73–82, Baltimore, MD, USA, 21–23 May 1993. IEEE.

[8] G. Canfora, Aniello Cimitile, and M. Munro. Re^2: Reverse-engineering and reuse re-engineering. *Journal of Software Maintenance: Research and Practice*, pages 53–72, March–April 1994.

[9] G. Canfora, Aniello Cimitile, M. Munro, and C.J. Taylor. Extracting abstract data types from C programs: A case study. In *1993 Conference on Software Maintenance*, pages 200–9, Quebec, Canada, 27–30 September 1993. IEEE.

[10] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[11] Ingo Claßen, Klaus Hennig, Ingo Mohr, and Michael Schulz. CUI to GUI migration: Static analysis of character-based panels. In *Proceedings. First Euromicro Conference on Software Maintenance and Reengineering*, pages 144–9, Berlin, Germany, 17–19 March 1997. IEEE.

[12] László Csaba. Experience with user interface reengineering: Transferring DOS panels to Windows. In *Proceedings. First Euromicro Conference on Software Maintenance and Reengineering*, pages 150–5, Berlin, Germany, 17–19 March 1997. IEEE.

[13] Robert DeLine, Gregory Zelesnik, and Mary Shaw. Lessons on converting batch systems to support interaction. In *Proceedings of the 19th International Conference on Software Engineering*, pages 195–204, Boston, Massachusetts, 17–23 May 1997. IEEE.

[14] Joanne Bechta Dugan, Kishor S. Trivedi, Mark K. Smotherman, and Robert M. Geist. The hybrid automated reliability predictor. *Journal of Guidance, Control, and Dynamics*, 9(3):319–31, June 1986.

[15] Letha H. Etzkorn and Carl G. Davis. Automatically identifying reusable OO legacy code. *IEEE Computer*, 30(10):66–71, 1997.

[16] E. S. Flint. The COBOL jigsaw puzzle: Fitting object–oriented and legacy applications together. *IBM Systems Journal*, 36(1):49–65, January 1997.

[17] Brian D. Fromme. HP Encapsulator: Bridging the generation gap. *Hewlett-Packard Journal: Technical Information from the Labratories of Hewlett-Packard Company*, 41(3):59–68, June 1990.

[18] Rohit Gulati and Joanne Bechta Dugan. A modular approach for analyzing static and dynamic fault trees. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 57–63, Philadelphia, Pennsylvania, January 1997. IEEE.

[19] S. C. Johnson. YACC — Yet another compiler-compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, N.J., 1975.

[20] Douglas Lea. libg++, the GNU C++ library. In *USENIX proceedings: C++ Conference*, pages 243–56, Denver, Colorado, 17–21 October 1988. USENIX Association.

[21] M. E. Lesk and E. Schmidt. Lex — A lexical analyzer generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, N.J., 1975.

[22] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. *SIGSOFT Software Engineering Notes*, 20(4):18–28, October 1995.

[23] *CORBA: Architecture and Specification.* Object Management Group, Inc., 1995.

[24] John Parodi. Building "wrappers" for legacy software application. URL: `http://www.digital.com/objectbroker/product/obwp_wrap.htm`.

[25] Louis Perrochon, Gio Wiederhold, and Ron Burback. A compiler for composition: CHAIMS. In E. Nahouraii, editor, *Proceedings Fifth International Symposium on Assessment of Software Tools and Technologies*, pages 44–51, Pittsburgh, PA, 2–5 June 1997. IEEE.

[26] M. E. Platt, E. E. Lewis, and F. Boehm. General monte carlo reliability simulation code including common mode failures and HARP fault/error-handling. Technical Report Contractor Report 187587, NASA, Langley Research Center, January 1991.

[27] Larry Reznick. Hiding UNIX applications in utility wrappers. *Sys Admin: The Journal for UNIX Systems Administrators*, 4(5):68–82, September/October 1995.

[28] Kevin J. Sullivan, Jake Cockrell, Shengtong Zhang, and David Coppit. Package-oriented programming of engineering tools. In *Proceedings of the 19th International Conference on Software Engineering*, pages 616–617, Boston, Massachusetts, 17–23 May 1997. IEEE.

[29] Kevin J. Sullivan, Joanne Bechta Dugan, John Knight, et al. Galileo: An advanced fault tree analysis tool. URL: `http://www.cs.virginia.edu/~ftree/index.html`.

[30] K.J. Sullivan and J.C. Knight. Experience assessing an architectural approach to large-scale systematic reuse. In *Proceedings of the 18th International Conference on Software Engineering*, pages 220–229, Berlin, Germany, 25–30 March 1996. IEEE.

[31] Systems Techniques, Inc. Wrapping legacy systems for reuse: Repackaging vs. rebuilding. URL: `http://www.systecinc.com/white/whitewrp.htm`.

[32] W. E. Veseley, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. U. S. Nuclear Regulatory Commission, NUREG-0492, Washington DC, 1981.

[33] Paul Winsberg. Legacy code: Don't bag it, wrap it. *Datamation*, 41(9), May 1995.