Experiences "Unwrapping" Two Legacy Systems for Reuse

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

Abstract—
Many organizations depend on software systems that represent large capital investments. Changes in the economics of the surrounding hardware and operating systems continually reduce the value of such systems. For example, the economics of software tools for engineering changed dramatically when personal computers running the Windows operating systems overtook Unix workstations running Unix and mainframe as preferred hardware and operating systems platforms. Many critical tool capabilities remain tied to these older platforms. In this paper we discuss several approaches to investing in the migration of such systems to preserve or enhance their value in the face of such evolution. We focus on the approach we call *unwrapping.* We contribute two results. First, we define unwrapping and we characterize it in terms of its position in a spectrum of related approaches, with unrestricted reengineering at one end and wrapping at the other. Second, on the basis of our experience with two case studies in the domain of software tools, we identify a set of technical problems that appear to arise commonly in unwrapping projects. Awareness of such problems can help design decision-makers to anticipate technical risks and thus to make better decisions about how to manage legacy migration; and it builds the case for research in particular dimensions.

I. Introduction

Existing software systems often represent major capital investments. As with nearly any machinery, the value of a software system tends to decline as the economic and technology context changes [2], [5], [29]. Changes in hardware and operating systems have an especially great impact because these platforms are the means by which the functions of the software are delivered to users. For example, the ascension of personal computers and associated operating systems and user interface styles signiŢcantly diminished the value of many software tools for engineering modeling and analysis on Unix workstations and mainframes.

Reimplementing such tool functions on new platforms can have obvious beneŢts. First, valuable functions can be accessed through the most economical means. Second, the capabilities of the new technologies add value to the existing tool function. Sophisticated graphical interfaces and interoperability features can enhance the value of earlier bare-bones Unix implementations of engineering modeling and analysis functions, for example.

Such beneŢts are unfortunately not free. The value added by a new implementation must be calculated net of cost. The problem is that the cost of an entirely new implementations is often prohibitive. A more cost-effective approach in many cases involves use of existing code as raw material in the production of a new version for a new operating environment. This aspect of software evolution—often called legacy migration—is the topic of this paper.

We discuss a particular legacy reuse approach that we call *unwrapping*. In particular, we discuss unwrapping in the context of the migration of software tools for engineering applications. The approach is not new: it has probably been used for almost as long as software has been built. Our contributions are to identify it as a distinct approach, to name it, to put it in context with related approaches, to discuss tradeoffs in selecting it over related alternatives, to document some of the technical difŢculties designers face in using the approach, and to discuss implications for future research and for the development of tools to support it. We discuss two case studies that we undertook in developing a new tool, and from salient points of our experience we discuss the factors that contributed to our decision making process.

The unwrapping approach is based on three main assumptions. First, many tools are organized as two roughly identiŢable bodies of code without a sharp modular interface separating them—i.e., as proto-modules. One proto-module comprises what we will call the *core code;* the second, the *superstructure code,* or what Shaw called packaging. The core implements the primary computational function of a tool. The superstructure embeds the core within a technology-speciŢc operating environment. For example, a fault tree analysis tool [38] will contain certain core algorithms and data structures for solving fault trees. That core code will be integrated with superstructure code providing the interface to the environment: to Ţle systems, system calls, input and output devices, etc.

The second assumption is that core code has often evolved into a state of considerable complexity: it mostly works but it is poorly documented, poorly understood, complex, brittle, and critical. By critical we mean that the consequences of compromising the correct functioning of the code could be signiŢcant. Breaking the code that performs fault tree analysis, for example, could inţ uence key design decisions that lead to the production of critical systems having less than the required levels of reliability. The key point is that these properties place a demanding premium on using such core code without change—or, more precisely, without any change whose correctness is not obvious.

The third assumption is that there are signiŢcant costs associated with the retention of legacy elements of the old superstructure in a new technology context. One of the primary drivers of efforts to implement existing functions in new technology contexts is to break completely from dependence on obsolescent technology. Any remaining dependencies of the core on old technology would preclude separation, defeating the purpose of the reimplementation exercise. More subtly, including old superstructure elements in the new context incurs costs in design inconsistency and associated development, maintenance, risk, and perhaps runtime costs. This is true even if this is done

without dragging in the old technology itself, such as the hardware on which the old operating environment runs. It is always ideal from a technical perspective to retain only core code and no legacy superstructure.

The problem is that even if the legacy superstructure can be separated from the old technology, the intertwining of core and superstructure makes it hard to isolate the core for use in the new environment. The designer thus has to make a tradeoff of an economic nature. *The tradeoff is driven by the presence of dependencies of core code on legacy superstructure that are hard to break without changes that risk compromising the correctness of the core code.* The designer is forced to decide how much of the legacy superstructure to include in the new implementation in order to avoid having to make such risky changes, while also avoiding burdensome costs of including legacy code and technology in the new system.

A tradeoff spectrum emerges. At one end, it is easy to reuse a core without change if the entire legacy superstructure is included in the new system. A new superstructure suitable for the new environment is superimposed upon and encapsulates the old superstructure. This is the familiar *wrapping* approach [1], [30]. A Windows program could wrap a Unix tool, for example, by presenting an interface whose functions make remote procedure calls to the legacy tool running on the legacy platform. No dependencies of the core on the legacy superstructure need to be broken or rebound, nor are dependencies of the legacy superstructure on its legacy environment disrupted; so few or no changes have to be made to the existing system. The problem is that this solution fails to satisfy the common requirement that the core function work in the new environment alone—that the legacy environment be left behind entirely.

At the other end of the spectrum, dependencies of the core on the legacy superstructure and of the legacy superstructure on the legacy environment can be eliminated entirely by the unrestricted *reengineering* of the core code. Chikovsky and Cross deŢne reengineering as “examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [13].” Reengineering involves reverse engineering the existing code to produce a speciŢcation, followed by development of a new implementation. The problem is that this approach involves unrestricted changes to the legacy core, which, as we have observed, can carry unacceptable costs and risks, not to mention development intervals.

In general, neither approach is economically optimal. Unwrapping seeks solutions in compromise positions. By *unwrapping* we mean an approach whose fundamental operation is the identiŢcation of a boundary along which a legacy system is separated, in the simplest case, into two bodies of code in an economically optimal or suitable manner. One part contains the core, but also, in general, some legacy superstructure elements. The other part contains only legacy superstructure that is to be discarded. The Ţrst part, the *extended legacy core*, is then wrapped or otherwise modiŢed, consistent with the stricture on changes that are hard-to-verify, to sever any remaining dependencies on the legacy environment and to make it suitable for integration into a new system.

In general there are many boundaries along which such a separation can be made. Each boundary has an associated economic payoff corresponding to the beneŢts of having the core function in the new context, net of costs over the lifetime of the resulting system. The costs depend on the factors that we have discussed: an up-front design cost that depends on the complexity of the task of separating the core; the cost of any delays incurred in producing code that functions in the new environment; any downstream maintenance and evolution costs incurred by incorporation of any superţ uous legacy elements or by the use of a legacy core; as well as any performance overheads incurred by retention of legacy structures, including both runtime overhead and continued dependence on legacy technology.

Speaking informally, the potential value of unwrapping in a particular case is the maximum payoff over the set of boundaries that can be selected. At one end of the spectrum, as we have noted, is the wrapping of the entire legacy system, including hardware technology. Close to that extreme is the wrapping of the entire legacy application but with a separation from the legacy hardware, typically by the provision of a virtualizing layer, such as a legacy environment emulator, running on the new platform. More interesting cases occur as the boundary is moved inward, cleaving off parts of the legacy superstructure and virtualizing some parts of the cleaved environment as necessary to enable the newly isolated core to work. Other legacy superstructure elements are incorporated into the new core because separating the core from them would be too costly or risky. We call these unwanted but hard-to-remove elements “warts.”

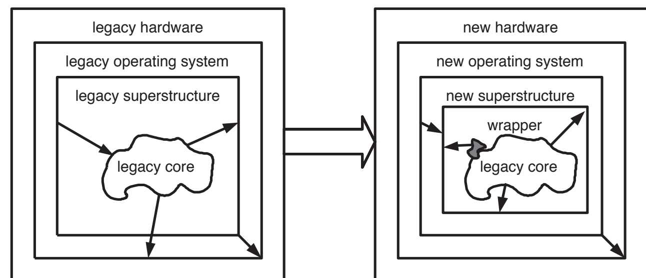

Fig. 1. The reuse of a system using unwrapping.

Figure 1 illustrates the unwrapping concept. Figure 1a depicts one possible extracted core embedded in a legacy superstructure running on a legacy platform. It also represents dependencies of the core on the legacy superstructure, and of that superstructure on the core. Dependencies are illustrated by lines with circles on the ends. The circles indicate the elements depended upon. Figure 1b depicts the unwrapped core, with both warts and unbound dependencies included. Figure 1c depicts the integration of the unwrapped core into a new system: only the unwrapped code, along with any unremoved warts, is wrapped for integration into the new system. The wrapper servers two purposes. First, it binds the dependencies of the legacy-core-with-warts required for the core to operate, through the provision of an interface that virtualizes those parts of the legacy superstructure on which the core continues to depend. Second, it provides a clean interface to the core for use by the new system.

In the next section we make the idea of unwrapping concrete by discussing the integration of embedded legacy tool code into our tool, Galileo. In particular, we have integrated into Galileo computational cores from two Unix-based legacy fault tree anal-

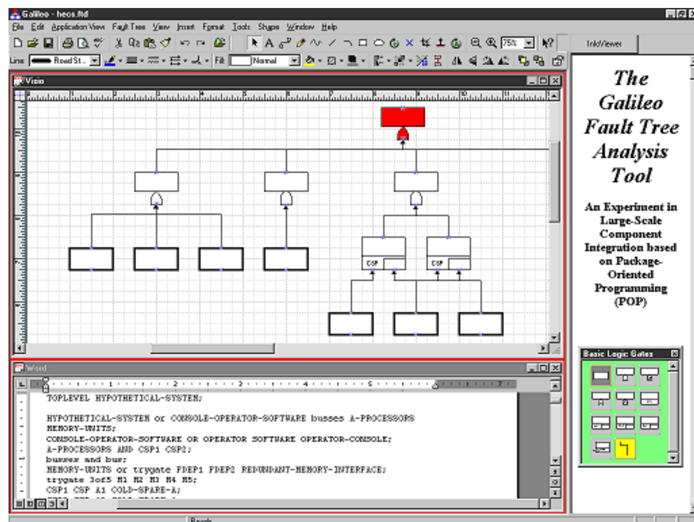

Fig. 2. Screen shot of Galileo

ysis tools: DIFTree [22] and MCI-HARP [6]. Sections III and IV present details of the unwrapping of the legacy cores. Section V presents our evaluation of the unwrapping concept based on work done to date. Section VI discusses research related to unwrapping. Section VII concludes with a discussion of options for future work.

II. Overview of the Galileo Project

The Galileo project is a case study in the construction of software tools using an approach we call *package-oriented programming* (POP) [35], [36]. It is also a project involved in producing novel tools for reliability engineering. In particular, the project is building a novel fault tree analysis tool that is intended for use in the reliability engineering of aircraft. POP is an approach in which multiple shrink-wrapped packages are used as large-scale components to be integrated into systems. Our goal was to use POP to build a fault tree analysis tool initially supporting the computational core of an earlier Unix-based tool called DIFTree III. Galileo was to have significantly enhanced usability features and was to run on a Windows platform.

Fault tree analysis [38] is a reliability engineering modeling technique. A fault tree model of a system expresses how component-level failures combine to produce system-level failures. See Figure 2 for an example. The interior nodes are called gates. They express how lower-level failures combine to produce higher-level failures. Traditional fault tree models employ combinatorial "and" and "or" gates. The failure that a gate represents is assumed to occur if the specified combination of failures corresponding to the children of the gate has occurred. The top-level gate in a tree represents the condition under which a particular system-level failure occurs. At the leaves of a tree are "basic events," whose stochastic failure characteristics are specified as inputs. Fault tree analysis computes the stochastic failure characteristics of the system-level failure based on the characteristics of the basic events and the structure of the fault tree. DIFTree is distinguished in its support of special gates that express infeasible combinations of failures (useful in keeping the required computation manageable), and that express temporal relations required for failures to occur and causal relations that describe how failures can cascade through a system. The technical details are not critical in this paper.

The original implementation of the core analysis functions of DIFTree was hosted in a superstructure that implemented graphical and textual manipulation using a complex combination of Unix mechanisms, including the pipe-and-filter composition of separate executable programs, the use of programming tools such as Tcl/Tk [28], Python [37], Perl, and operating system calls. We wanted to reuse the core without change in Galileo while severing all ties to Unix. We were faced with removing the Unix-based superstructure, including editing and storage functions, the use of Unix scripts, and the pipe-and-filter structures that were used in integrating aspects of the core analysis algorithm.

Upon inspecting the existing computational core we found it to be extremely complex and brittle and also intertwined with elements of the Unix-based superstructure. The complexity presented daunting barriers to rewriting the code and to changing it. In the complexity of the existing code were embedded many subtle decisions that were critical to the correctness of the computation, but which were documented nowhere outside of the code itself.

We ruled out complete reengineering for two reasons. First, we were afraid that we would get a new implementation wrong for not knowing all of the special-case logic in the existing, complex code. Second, for similar reasons, we judged that we could not change the existing core code in any substantial way without running a serious risk of compromising its proper functioning. The code was extremely complex, having been designed, implemented, and maintained by less-than-expert programmers. Third, we were not sure that the Galileo tool would be well received "by the market," so we were loath to make the substantial up-front investment needed to produce a completely reimplemented core in the face of such uncertainty about the ultimate payoff. Rather, we sought an iterative, prototyping approach in which we would produce a system at a low up-front cost, with the option to make substantial changes in the future contingent on the success of the prototype [12]. We eliminated comprehensive reengineering as an alternative on both technical and economic grounds.

Similarly, complete wrapping of the existing tool including the legacy Unix environment was judged not workable. The point was to provide a version of the tool at low cost on desktop PC machines. Wrapping the whole implementation while breaking its dependences on Unix was also judged not acceptable. There would have been many problems with such an approach. The legacy user interface and process structures were wholly incompatible with Windows. We would have had to emulate Unix system calls. We would have ended up with an extremely complex tool that had parts written in both compiled C++ and interpreted Perl and Python languages. It was clearly not an optimal solution, even in light of our need for an inexpensive, evolutionary prototype of a Windows-based tool. We were left with unwrapping as apparently the most attractive approach.

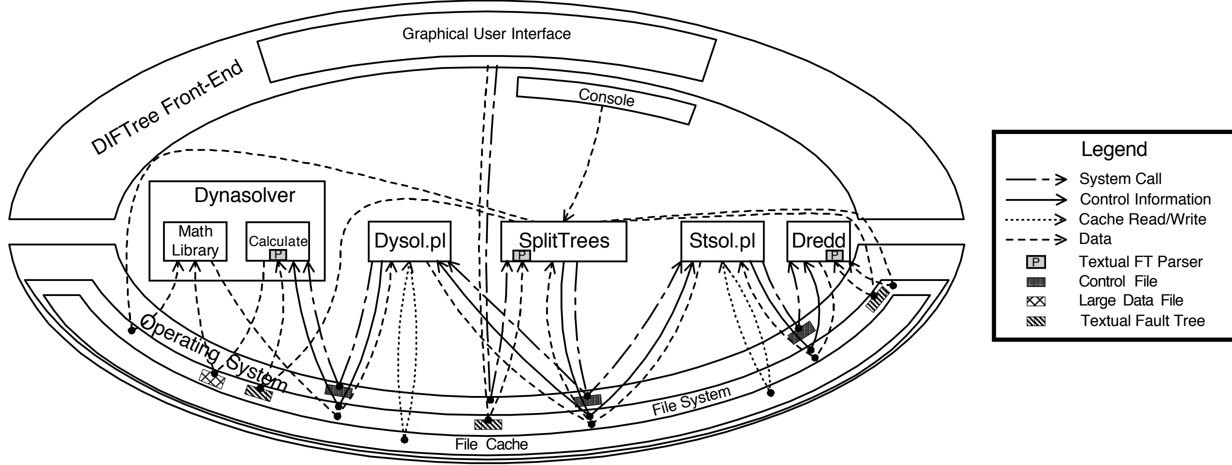

Fig. 3. File system and console usage in DIFTree.

III. Unwrapping DIFTree

The design and implementation of the DIFTree tool, including the legacy core, comprised 32,159 lines of C++, C, Perl, Tcl/Tk, and Python. The program was quite poorly designed to begin with (though quite clever in its core data structures and algorithms), and its structure was further compromised by uncoordinated changes made by four graduate students over several years. The core source code was extremely hard to understand, and almost completely undocumented. Developed for use on Unix, the DIFTree tool as a whole was structured as several applications communicating with each other, with the user, and with the operating system through a variety of interfaces, as illustrated in Figure 3.

In Section III-F, we discuss the results of the unwrapping, rebinding, and integration of the DIFTree core into the new environment. However, it is useful at this point to view Figure 4 in comparison to Figure 3. As one can see, the dependence on the Unix file system for inter-module communication has been replaced with memory-based communication, and the dependence on the console has been replaced with a graphical interface. Furthermore, the Dysol.pl and Stsol.pl modules have been removed, and their use of the file cache replaced by a hash in memory.

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 DIFTree 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 that are output as text files for solution 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. We now describe the unwrapping process and its application to DIFTree in more detail.

In the following five sections, we describe the steps of the unwrapping process in the context of the DIFTree example, namely: finding an internal interface, identifying dependencies, extraction of the core, wrapping the exposed core, and integrating the wrapped core into the new superstructure. In Section III-F, we discuss the results of this work.

A. 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 (or at least value-producing) tradeoffs under uncertainty and limited resources. Examples of factors to consider while choosing a suitable interface are: the time available; the difficulty of isolating the interface; the labor required in removing the superstructure at that interface; the en-

gineer'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 interface "close to the surface" of the legacy tool might realize short-term gains more quickly, while incurring higher long-term costs, performance degradation, and possibly architectural incoherence. On the other hand, an approach that utilizes an internal interface that circumscribes the desired core code tightly might result in a cleaner integration, but requires a higher up-front investment of time and other resources.

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 would have 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 of the core on the legacy superstructure that we had to rebind, including the dependence on console I/O. Concern regarding performance also led us to choose an internal interface that removed the file system.

B. Identify Dependencies

The next step in unwrapping is to identify dependencies of the remaining 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 Unix shell redirection. DIFTree also depended on the Gnu C++ library [24], 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 C++ language 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.

C. Physically Extract the Core

The next step is to physically separate the source code implementing the core from that implementing the superstructure. 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. Fortunately, no files contained both superstructure and core code. If they had, we would have split the files in order to isolate the core code.

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 Obstack was not used, we decided to port it—as a wart—to the new environment.

Other dependencies on the legacy environment manifested themselves not as explicit source code references, but rather as subtle system dependencies. For example, we found that the core depended upon an outdated compiler, which required that we modernize the code in order to remove incompatibilities resulting from changes in the source language's definition.

D. 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 encapsulated within the core's new wrapper, as do any necessary functions 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 by substituting standard date routines in the C++ library that provide equivalent functionality. 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 the graphical window 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. In the case of the large complex data file used within DynaSolver to store temporary values, we identified two possible means of removing the dependence on the file system: either to reverse-engineer the data file's format to a high-level abstract representation, or to keep the complex data format, but store it in main memory instead of a file. Because we were not comfortable with the risk that we would misunderstand the file format, we decided to replace the file stream with a stream to memory by rebinding the C++ `ostream` file interface to a `strstream` memory-based one. Since the semantics of `ostream` and `strstream` were comparable, we were confident that this change did not adversely affect the code.

The file system is also used by DIFTree to store three different fault tree representations, which are input into SplitTrees, DynaSolver, and Dredd. Unlike the large DynaSolver file, these files were easier to understand, which allowed us to identify the syntactic elements that corresponded to specific fault tree structures. Using this correspondence as a guide, we unwrapped SplitTrees to remove its file output, and then created a wrapper that built an equivalent data structure in memory. In this way, we were able to replace the non-standard file communication with the transfer of a canonical fault tree data structure in memory.

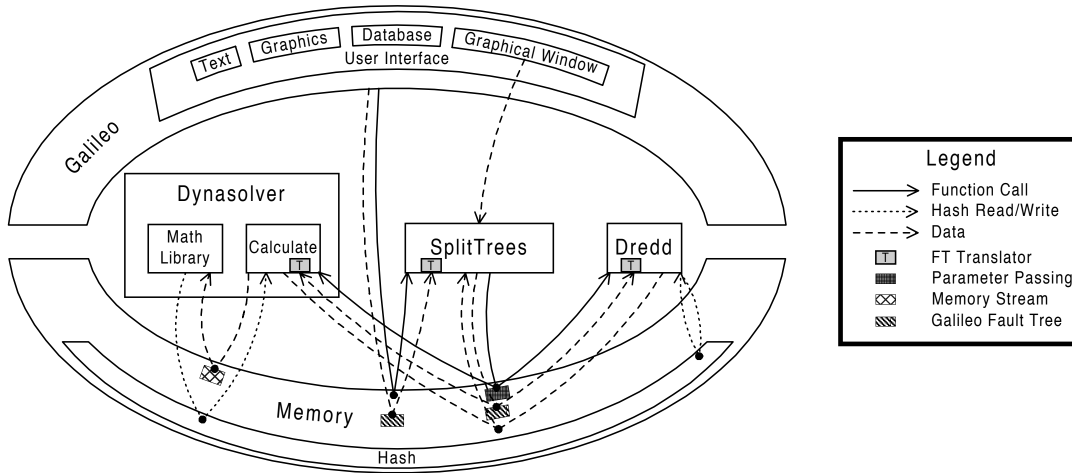

Fig. 4. Reengineered DIFTree structure.

Next we adapted the input routines of SplitTrees and DynaSolver to use the new data structure. Because these modules used LEX and YACC [23], [25] to parse their input files, we were able to use the semantic action rules of the parsers to determine how to populate SplitTrees and DynaSolver's internal data structures for a given fault tree data structure. 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).

E. 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. We found that wrapping and integration occur simultaneously to some extent because one must be cognizant of the future environment when constructing the wrapper.

The strategy we chose for integrating the core into the new system was to set a series of milestones to ease the integration process. The first milestone was to compile the modernized source code (including part of the superstructure) on the Unix platform, but with a new compiler. The second milestone was to compile the same source code on the Windows platform. Both of these milestones helped isolate any remaining dependencies on the legacy system.

During the next milestone, we removed the remaining superstructure from the core code, and compiled the core as a library to be used by Galileo. It was at this step that we encountered unanticipated interactions between the newly integrated core and the new environment. In particular, we found that DynaSolver and SplitTrees referenced different global variables by the same name, a problem referred to as "name clash". To solve this problem, we encapsulated the two modules to limit the scope of the variables.

The final milestone was to modify the Galileo superstructure to invoke the DIFTree core. At this point we discovered 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 responsible of exiting the program to derive preconditions strong enough to prevent that code from executing unsafe statements.

F. 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 400 people have downloaded it over a two-year 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 rebound, 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 canonical fault tree data structure is now the common format exchanged between SplitTrees and the subtree solvers, with object translators replacing the parsers in the legacy system.

IV. 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) [6]. Using Monte Carlo techniques, this program simulates stochastic failures of the basic elements of a fault tree to determine the probability of 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 a fault tree evaluation package called HARP, developed at Duke and Clemson universities [17]. About seven years later HARP was used in a Monte Carlo simulator (MCI-HARP) built at Northwestern University [32]. 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.

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. The costs would have been higher in this case in part because, unlike the DIFTree case, we lacked a convenient way to rebind them using a `strstream` object. Because of the modular nature of the code, we found it easy to identify a reasonable interface to the computational core in the form of a simple function call, two input files, and a report in a text file for output. In choosing this interface, we bypassed MCI-HARP's superstructure for editing of fault trees, thereby rendering it "dead code". Because this was a prototype integration, we did not perform the step of physically separating the core code from the superstructure.

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 to our implementation in this manner, 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 only in a way that was easy to understand and verify.

V. 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 possible need to implement a "screen scraper" [39], 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 a Galileo tool that would run using multiple processes dependent on Perl and Python
5. a poorly constructed component within the well-structured Galileo system

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 file 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 an I/O-bound, non-running state. We infer that most of this overhead resulted from the use of pipes and the file system to communicate data between the Unix processes that constituted the primary modules of DIFtree.

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 superstruc-

ture. Unwrapping MCI-HARP to remove its support for fault tree creation simpliŢed 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 modiŢcations. 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.

Using unwrapping followed by rebinding and wrapping, we were able to integrate two legacy systems into a modern application while removing undesirable dependencies on the Unix operating system, graphical and console interfaces, and the 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 deŢned, 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.

VI. Related Work

Unwrapping is clearly related to a great deal of work on reverse and re-engineering, especially that on reusable component extraction. We also address work on wrapping and repackaging.

A. Reusable Component Extraction

Much work has been done on the extraction of reusable components from legacy source code [4], [9], [10], [11], [18], [26], [14], [15]. 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 not immediately obvious, our work differs substantially 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 [18]." Similarly, the *reuse re-engineering* work of Cimitile, Canfora, et al. [9], [10], [11], 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 reuse. 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 task, then, is to determine the most suitable interface between the core and superstructure.

There is little doubt that the use of code analysis tools, whether semantically rich or not, could assist in this 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 reţ exion model techniques [26], promises to be especially valuable as an aid in unwrapping. Automated approaches to user interface reengineering such as that used by Csaba [15] and Claßen, et al. [14] 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. [4], also offers a good Ţrst step in separating core code from superstructure.

B. 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 [27], OLE [7] and similar developments enable encapsulation of legacy systems behind distributed object-oriented interfaces. The theory of wrapping was addressed by Parodi [30], who identiŢed four major types of wrappers, and Aronica and Rimel [1] who examined implementation issues.

Wiederhold's CHAIMS [31] deŢnes 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++ [3], and Van Camp used wrappers to improve library portability [8]. Flint [20] wrapped legacy COBOL applications using an object-oriented wrapper. Reznick wrapped Unix applications to enhance their functionality [33]. The HP Encapsulator provides a wrapper-based framework for integrating Unix tools into the HP SoftBench environment [21]. Many of these techniques might be useful in wrapping core code once unwrapped.

C. Flexible Packaging

DeLine's work on Flexible Packaging [16] allows the developer to defer some design decisions about the external interactions of a core component until integration time. The core component, called a "ware" is integrated with a wrapper, called a "packager", which provides an external interface suitable for a given environment. By substituting different packagers a speciŢc ware can be made usable in a number of contexts.

Flexible Packaging is a natural complement to unwrapping. Unwrapping deals with the isolation of a core component and the removal of the superstructure. By modifying the core's external interface, it can be made into a ware that supports a number of packagers. Unwrapping extends the notion of Flexible Packaging beyond newly developed code, to include valuable core code that already exists but is integrated into existing systems.

D. Systems Reengineering Patterns

Systems reengineering patterns is a nascent Ţeld of research, the goal of which is to codify and document the processes by which systems are reengineered. The Framework-based Approach for Mastering Object-Oriented Software Evolution (FAMOOS) project [19] is currently constructing a handbook consisting of patterns describing how to perform reengineering

tasks. Stevens [34] provides a survey of current work in the Ţeld, as well as several example patterns.

This paper presents a candidate strategy that can be further elaborated into a reengineering pattern for inclusion into the catalog of standard practice. By making the process explicit in a pattern, we can create a discrete unit of knowledge that can be learned and applied in relevant contexts.

VII. Conclusion and Future Work

Unwrapping as we have deŢned it in this paper refers to a style of reengineering legacy code for reuse in a new technology context when valuable core code is too complex and brittle to be changed signiŢcantly with conŢdence. Unwrapping leaves the core code unchanged but for changes whose correctness is easily veriŢed. The key question in an unwrapping activity is where to draw the boundary at which point dependences of the part inside are severed from the parts outside by virtualization of the relevant parts of the outer part.

We have characterized this decision as being fundamentally an economic decision. We want to emphasize here that in particular it is generally also a decision made in the face of uncertainty: about the immediate and downstream costs, beneŢts, and technical risks of any given choice. In one case we wrapped a legacy core without change but were surprised to Ţnd that our system failed when the core code took an action (exit) that was appropriate in the Unix environment but not in the Windows environment. The cost to wrap the core thus increased unexpectedly, in that we had to enhance the wrapper to prevent such inputs from reaching the core, and doing that required understanding the core in signiŢcantly greater detail than we had anticipated.

One area for future work is of course in tool support for unwrapping. But we would like to suggest that tool support for decision-making might also be useful. Modeling the up-front and downstream costs, beneŢts, and technical risks the uncertainties involved, and the designer's tolerance for risk (e.g., in changing core code, or in wrapping it without fully understanding it), we hypothesize, could help engineers to make better choices in how to get value from complex code embedded in and intertwined with increasingly obsolescent hardware and software environments.

In addition to tool support, process guidance might be developed and evaluated. In our case studies, for example, we addressed uncertainty by building a series of prototypes that resolved speciŢc issues, and analyzing the technical details of our options.

The contributions of this paper are the identiŢcation and characterization of unwrapping as a distinct approach in the context of other related approaches, and the examination of relevant tradeoffs and technical difŢculties involved in using the approach in the context of a case study in the domain of software tools. We also framed unwrapping as involving a process of decision making under uncertainty with an economic (utility maximization) objective. We did not show how to make such decision problems precise.

VIII. 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):83–88, June 1996.

[2] Howard Baetjer. *Software As Capital: An Economic Perspective on Software Engineering*. IEEE Computer Society Press, 1998.

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

[4] 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.

[5] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, March 1976.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] Prasad Chalasani, Somesh Jha, and Kevin Sullivan. An options approach to software prototyping. Computer Science Technical Report CMU-CS-TR-97-xxx, Carnegie Mellon University, June 1997.

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

[14] 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.

[15] 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.

[16] Robert DeLine. Avoiding packaging mismatch with Flexible Packaging. In *Proceedings of the 21st International Conference on Software Engineering*, pages 97–106, Los Angeles, California, 16–22 May 1999. IEEE.

[17] 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.

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

[19] SCG/FAMOOS, 1999. URL: `http://iamwww.unibe.ch/~famoos/`.

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

[21] 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.

[22] 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, 13–16 January 1997.

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

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

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

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

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

[28] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, Reading Massachusetts, 4 edition, 1994.

[29] David Lorge Parnas. Software aging. In *Proceedings of 16th International Conference on Software Engineering*, pages 279–87, Sorrento, Italy, 16–21 May 1994. IEEE.

[30] John Parodi. Building "wrappers" for legacy software application, 1997.

[31] 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.

[32] 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.

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

[34] Perdita Stevens and Rob Pooley. Software reengineering patterns. Technical Report ECS-CSG-40-98, Department of Computer Science, Edinburgh University, Edinburgh, United Kingdom, March 1998. URL: `http://iamwww.unibe.ch/~famoos/`.

[35] Kevin J. Sullivan, Joanne Bechta Dugan, John Knight, et al. Galileo: An advanced fault tree analysis tool, 1997. URL: `http://www.cs.virginia.edu/~ftree/index.html`.

[36] 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.

[37] Guido van Rossum. *Python Reference Manual*. Stichting Mathematisch Centrum, Amsterdam, 1996.

[38] 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.

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