TOWARDS AUTOMATING SOFTWARE MAINTENANCE

Stanislaw Jarzabek
CSA Research
Block 12 Science Park Drive #03-01
The Mendel, Singapore 0511

Kelly Tham
Department of Information Systems and Computer Science (DISCS)
National University of Singapore
Singapore 0511

ABSTRACT

This paper describes an ongoing project on automating program maintenance. We present a methodological framework for program maintenance and discuss detailed techniques for specific maintenance tasks. Our research on a methodology for program maintenance covers the whole maintenance life-cycle. The objective of this research is to delimit boundaries for long-term evolution of the project and to define a framework for integrating specific techniques and tools we develop.

In this paper, we focus on the methodological aspects of program maintenance. First, we review techniques and tools used during the various phases of program maintenance. We analyze relations among these techniques to see how they can work together within a maintenance methodology. Next, we observe that there can be no universal maintenance methodology, because an optimal maintenance process depends on many factors which are specific to a maintenance problem in hand. We discuss the concept of a framework for the customization of program maintenance methodologies and for supporting them with integrated tools.

I. INTRODUCTION

A. Project Background

So far, some of the experiences with transferring CASE technology into the industry have been disappointing and disillusioning. CASE tools have been applied in pilot projects, while the mainstream of programming is still mostly done without automated support. One reason for this is the fact that the approach adopted by CASEs does not address the most vital problems faced by programmers. CASE tools concentrate on developing systems from scratch. But in reality, not too many systems are developed from scratch; rather, existing programs are modified and extended to meet new requirements. Often, programs are the only available definition of an organization's data and procedures. Those existing programs are a valuable resource which cannot be just thrown away and replaced by new software developed using CASE technology. Programs have to be maintained, i.e. analyzed for understanding, sometimes restructured, so that they can be adapted, extended and modified in all ways. Unfortunately, current undisciplined maintenance practices are known to gradually degrade programs by introducing poor structure and complexity into them. Over years, programs become more and more expensive to maintain.

Our company has been involved in the development of CASE tools since 1986. As most of the CASE developers, so far we have been paying much attention to front-end activities (i.e. analysis and design). In a way, it is easier to provide integrated and automated support for development activities than for software maintenance tasks. Methodologies for developing software have been intensively researched in both the academic and industrial environments for the last few decades. These methodologies offer notations and techniques which, at least partially, can be automated. On the contrary, software maintenance problems have not been receiving much attention and appropriate methodologies hardly exist. Ironically, while the research community was still busy working on solutions for software development, the DP departments were already spending the majority of their efforts and money on maintenance of existing programs rather than on development of new ones. Not surprisingly, by addressing only a part of the problem, CASE tools, which is a technology built on the results of the software engineering research, do not offer a comprehensive solution to the "software crisis".

We believe that CASE developers must correctly identify the reasons for the "software crisis" and come up with a technology which will address these reasons directly. It seems obvious that this cannot be done without shifting the attention from design problems to those of maintenance. Automation of system adaptation, extension and upgrades is the challenge for CASE technology of 1990s.

How can CASE tools help programmers in their real problems? In our approach, program maintenance issues will be addressed in an integrated way, within a common methodological framework. A methodology is a key factor, as we cannot effectively automate ad hoc procedures. (Current CASE tools would not be possible without development methodologies such as Structured Analysis and Design.) "Evolution rather than revolution" is the second element of our approach. We start by studying the ways maintenance is currently carried out, and work towards a technology which, on the one hand improves current practices, and on the other hand, can be naturally accepted by programmers. This study of the "real world" will lead to identifying techniques, tools and, finally, a methodology which closes the cycle between design and maintenance. Customer field is the first important driving force for our project. The second driving force is research, which is done jointly by CSA Research and the DISCS at the NUS. The research stream offers ideas and proposals which are validated in the customer field before they are accepted.

B. Rationale Behind Our Approach to Program Maintenance

The ultimate goal of the automation of software production is to reduce costs involved in system development and maintenance. Although the above statement sounds like a truism, we observe that it is too often overlooked by the researchers and tool designers. Programmers get methodologies and tools which implement elegant concepts, but are useless for practical purposes.

It is not obvious which techniques or tools actually help programmers. To be able to judge with confidence whether a given method really saves programmers' time, we would have to measure programmer productivity before and after a method is applied. Although software metrics exist to measure program complexity [5], to estimate maintenance effort, and to measure the quality of the software process [14], only few companies use them. (For the same reason, it is difficult to estimate the impact of CASE tools on productivity.) Research studies on the impact of various programming methods and tools on program maintainability are rare. What makes measuring productivity improvements even more difficult is that some of the techniques bring benefits only in the long run. System restructuring at the architectural level and design recovery are examples of such capital-intensive techniques. Despite these problems, there are some good examples which show how measuring can help in evaluating and improving a software process [14].

Another problem with the usage of tools and methods in maintenance is the variability of the maintenance environment. The environment could be constrained by the type of maintenance work to be done (fixing errors, small modifications, extending program functionality, converting batch program into on-line, etc.), the type of a program (batch, on-line, database-intensive, computation-intensive, etc.), program quality, and objectives

of the maintenance work (e.g. to ensure a longer system life span). The question is how a project manager can come up with an optimal maintenance methodology for a problem in hand. Our approach to this problem is to provide a flexible methodological framework for program maintenance within which we can compose an optimal maintenance methodology from available methods and tools, based on the analysis of the maintenance environment. We plan to use a metrics-based approach to evaluate the maintenance methods and tools.

The rest of the paper is organized in the following way. In the next section, we review approaches to program maintenance and specific techniques and tools related to various aspects of program maintenance. In section III, we present the concept of a framework for maintenance methodologies. Concluding remarks end the paper.

II. REVIEW OF TECHNIQUES AND TOOLS FOR PROGRAM MAINTENANCE

A. Support for Managing Program Maintenance

Before initiating the maintenance process, a project manager does portfolio analysis. He formulates the objectives of the maintenance task, estimates the cost, and identifies system modules which need particular attention. Based on the analysis of the objectives, system quality, available resources, etc., a manager can estimate expected returns in both the short-term and the long-term, allocate resources, and recommend techniques and tools to be used for system maintenance.

Tools can be used to help a manager in the portfolio analysis. Combining a software quality/complexity metrics evaluator with a system that defines the relations between maintenance objectives, software quality attributes, improvements required, maintenance cost and expected returns would be useful. A "goal/question/metrics" paradigm [1] provides a useful model for supporting project management decisions and for continuous quality control over a company's software processes.

MONSTR [6] is an example of a protocol-driven personnel communication tool that helps monitor maintenance activities. In particular, the following tasks are supported: personnel management, resource management, subprocess scheduling, walk-throughs, quality audits, planning, communication between project members, and change control.

B. The Use of Software Metrics in Maintenance

Software maintainability depends on many software quality attributes, such as program control flow and data complexity, modularization, complexity of module interfaces,

flexibility, testability and stability. Software metrics help determine the presence or absence of certain quality attributes in a program.

Maintainability is directly related to understandability. Maintainability metrics must measure how easily a non-author can understand the program logic, identify where to make a change, and make the change without breaking another part of the system. Some of the quality attributes and metrics related to maintainability [5] are the following:

- Span of Control: the maximum number of lines a programmer can effectively maintain,
- Mean Time to Repair (MTTR),
- McCabe's Cyclomatic Complexity Metric,
- consistent style used across program components,
- average length of identifiers,
- up-to-date internal documentation, comment blocks for module files, procedures,
- traceability links across requirements, design specs and code,
- consistency of data naming conventions across program boundaries,
- complexity of module interfaces,
- average length of program modules, etc.

As no single metric can reflect the level of program maintainability, a weighted approach seems appropriate. Weights can be assigned to various software elements in order to compute a Maintainability Index (MI). Actual weights and software elements may depend on objectives of a particular maintenance task. The MI, when formally related to the Span of Control, also helps in the estimation of manpower needs for maintenance tasks.

C. Some Software Maintenance Methods

In this section, we discuss some techniques and tools related to the main phases of a program maintenance life-cycle.

1. Program Restructuring

Program restructuring is defined as software transformations at the same level of abstraction, e.g. code-to-code or design-to-design [8]. The purpose of program restructuring is to inject maintainability into a program without changing its functionality.

Program restructuring methods can be classified into two categories: **code level** and **system architecture level** restructuring methods. Examples of code level restructuring include reformatting of code, consistent renaming variables across programs, improving the program control structure (i.e. converting spaghetti code into structured code) [10], and

improving on the data structure such as converting the database schema into third normal form. Automatic program restructurizers [17] is an example of a code level restructuring tool. They analyze both the control graph and abstract syntax of a program to determine transformations which convert unstructured code into structured code. These tools are very expensive to implement and it is still uncertain the extend to which automatically restructured programs are easier to maintain. Firstly, after a few years of maintaining an unstructured old program, maintenance programmers can finally understand it. Upon automatic restructuring of these programs, programmers would again have difficulty understanding them. Secondly, there are problems with the quality of the restructured code. Mechanical restructuring ignores the logic of the application, and does not break the program into logical blocks the way a programmer who understood the application would do. Some experts claim that maintainability cannot be retrofitted into a poorly designed program - if the input to the structuring facility is unstructured garbage, the output will be structured garbage.

System architecture level restructuring changes the modular structure of a program. Examples include breaking huge, monolithic programs into manageable modules, and identifying and isolating program functionalities. In breaking a program into modules in a meaningful manner, a simple approach is to scan through a program text in order to find program segments according to control structures such as program loops, and to determine which variables are referenced in a segment so that appropriate declarations can be included in a new module. Much of these can be automated and software metrics can help to evaluate the quality of new decompositions so that poor candidates for modules will be rejected. However, this method of breaking a program into modules is unlikely to produce a modular structure which can help much in the future maintenance of a program. If the original program was poorly designed, then modules isolated in this way may be shorter, but they will lack logical cohesion. To be profitable in the long run, system architecture restructuring should produce modules implementing well-defined, preferably reusable, functions and data abstractions. Such restructuring is challenging for programmers and requires a good understanding of both the global and detailed characteristics of a program. In the software maintenance life-cycle, system architecture level restructuring should be done during design recovery, with the help of powerful program analysis tools.

2. Reverse Engineering

Reverse engineering is the process of extracting higher level abstractions from software representations, e.g. design specifications from code or requirement specifications from design and code [8]. The ultimate goal of reverse engineering is to automatically re-create design and requirement specifications from a given program which could further lead to deriving an organization's business rules and procedures, and design decisions. Current

technology is far from achieving this goal. Attempts, however, have been made toward this end by merging traditional program analysis techniques with artificial intelligence techniques, encoding both problem domain and software engineering knowledge [3,9,15,23].

Some low-level design abstractions of reverse engineering such as data flow and control flow relations, function call graphs and cross-reference lists, can be automatically computed from code. These design abstractions help much in understanding a program. Static program analyzers developed in research labs [7] and those commercially available [24] compute low-level program abstractions and present them to a user in the form of lists or via a query language. These program abstractions are also useful in documentation generators which organize code and documentation for ease of comprehension [21].

3. Static Program Analysis

a) Control and Data Flow Techniques

Program analysis methods based on control and data flow relations have a high potential for improving program maintenance [13,19]. The strength of these methods lies in their ability to extract, from a possibly huge program, a portion of code related to a specific task. For example, data flow relations allow a programmer to trace the impact of a data definition on the rest of a program. Tools can display control flow information in graphical form [24] and allow a user to manipulate a program at the level of a program graph.

b) Program Slicing

Program slicing [13,25] decomposes a program much in the same way a programmer would when attempting to understand the role and behavior of certain variables in a program. Slicing uses data flow algorithms to extract lines of program code which play a part in contributing to the value of one or more variables at a certain point in the program. Hence, a slice of the program is a function of (i) a set of variables, and (ii) a statement number. The slice is an executable subset of the program that affects the value of the set of variables at that statement. Weiser [26] has conducted an experiment which showed programmers actually use slicing when debugging programs.

In software maintenance, when trying to understand a program in the absence of documentation, a programmer typically examines the program using a mixture of top down and bottom up strategies. The top down approach provides the context for seeing the "big picture" and the bottom up approach provides the actually understanding of the intention behind the program code. A general scan of the program gives the software maintainer an idea of the overall structure of the program. But to understand exactly how the program variables interact in the program to perform some function, the programmer

has to mentally trace and see how variables are being transformed through the flow of the program. This latter process is precisely what slicing does.

4. Design Recovery

Design recovery is a type of reverse engineering which requires problem domain knowledge and/or software engineering knowledge [3,8]. Design recovery aims at explaining a program at the conceptual level. The process of recovering high-level design abstractions from code requires either user involvement or support of expert systems. As design recovery is expensive in terms of both tools and programmer's time, it should be undertaken only if it proves to be more cost-effective than re-implementation. Design recovery is particularly recommended when:

- system architecture level restructuring is required,
- a system is to be fully understood,
- some of the system functionalities are important enough to be extracted, generalized (e.g. by parameterization) and converted into reusable software blocks,
- a batch system is to be re-designed into an on-line system.

In the context of a maintenance life-cycle, design recovery is done after code level restructuring and low-level reverse engineering together with program analysis.

The key problem areas in research on design recovery appears to be the following:

1. how we can map program structures into problem domain concepts (and low-level concepts into higher-level concepts, etc.), and
2. in what form we should record high-level design documentation recovered from code, so that it helps programmers in long-term program maintenance and it is also easy to keep the documentation up-to-date with an evolving program.

Most researchers accept an assistant approach to design recovery. Static program analysis methods (based on low-level design abstractions) help in program understanding, and are therefore also useful in design recovery. Methods for function abstraction [17] help a programmer in bottom-up recovering of function specifications. Design abstractions recovered from code are recorded within a hypertext system for ease of future maintenance [3,4]. To provide more automation for design recovery, we need some formal representation for mappings mentioned in point 1. above. To facilitate such mappings attempts have been made to build program domain models [3] and to identify meaningful program patterns which can be mapped into higher-level program descriptions [3,15,23].

5. Testing

Testing is the phase where most of the savings in software development/maintenance can be realized. Tools can help in many tasks related to testing such as automatic generation of test data, organizing libraries of test cases, automating regression testing, and analyzing test coverage. XPEDITOR [18] generates test data to traverse control paths, as recommended by the McCabe metrics. ASSET [12] uses data flow information to choose test data and to evaluate test data adequacy.

After program modifications, a library of test cases must be updated as program requirements may have changed and a program must be re-validated. However, we need to run only those test cases which are related to a particular change done to a program. Regression testing tools use control and data flow information to keep a library of test cases up-to-date and to select test cases related to a given modification of a program [11,2]. A Test Coverage Analyzer (TCA) helps identify exactly which parts of a program have been exercised during testing. Test coverage may be checked at different levels, e.g. statement, function or module. The TCA inserts stubs to a program to trace execution of program segments during testing.

McCabe's complexity metrics can be used to indicate the minimum number of test cases needed to cover all independent paths in a program. An optimized test coverage based on the path prefix testing strategy in described in [22].

III. A FRAMEWORK FOR PROGRAM MAINTENANCE

A. Purpose of a Maintenance Framework

Our proposal for overcoming the ad hoc maintenance procedures occurring in companies today is to impose a maintenance framework on the maintenance process. The purpose of such a framework is twofold. Firstly, it should provide an environment that fully supports a systematic approach to maintenance. Secondly, the maintenance framework should enforce a clear segregation of the maintenance phases and explicate the reasons for why different processes are involved in each phase. In other words, the framework should help evaluate the maintenance project and assist in the derivation of an appropriate maintenance methodology. Such an approach to maintenance promotes a clearer perspective as well as better control on the entire maintenance process.

A maintenance environment encompasses both the managerial and the technical aspects of software maintenance. On the management side, we have tasks such as portfolio analysis,

cost estimation and feasibility study, recommendation of a maintenance methodology, resource management, process scheduling, and monitoring of the progress of maintenance work. On the technical side, we are concerned with the derivation of a program maintenance methodology, i.e. a particular combination of methods such as program restructuring, reverse engineering and system re-documentation, and tools to be used by a maintenance team. A framework to support the maintenance environment should support both of these aspects of maintenance.

B. Structure of a Maintenance Framework

Figure 1 provides an overview of the maintenance framework we propose. It embodies both the managerial and technical aspects of maintenance. For a project manager, it provides the means for analyzing a particular maintenance problem and helps in the derivation of a maintenance methodology to be used. The methodology describes the phases of the maintenance processes to be carried out, with each phase having a specific purpose of its own in line with the maintenance requirements. Technically, the framework supports the planning and coordination of the tools to be used for maintenance.

The main components of the framework are a collection of methods and tools for program maintenance, a software metrics evaluator, and an expert system.

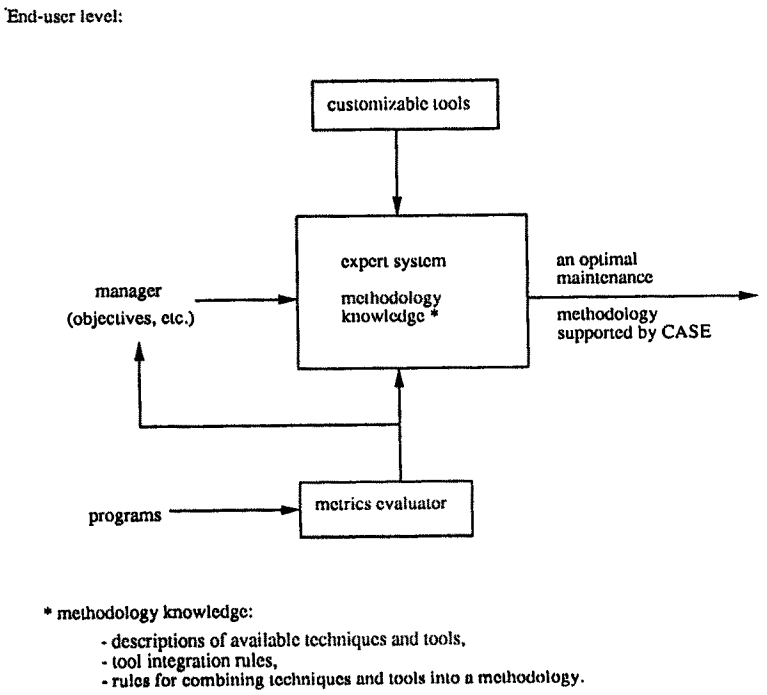

Figure 1. A framework for maintenance methodology

1. Building Blocks for Maintenance Methodologies

A maintenance methodology is defined in terms of methods, techniques and tools to be applied to various components of a system. A methodology is also concerned with scheduling maintenance activities and monitoring the progress of the maintenance process. Table 1 lists typical methods, techniques and tools which may be available within a framework for program maintenance.

Method	Technique	Tool
program restructuring at code level	control flow restructuring	STRUCT
	reformatting	REFORMAT
	renaming variables	RENAME
reverse engineering of low level abstractions	computing the PKB*	RE-PKB
	converting code to PDL**	RE-PDL
	documentation generation	DG
static program analysis	control flow	
	data flow	SPA
	function call graph	
	cross-reference lists	
	program slicing	SLICER
	syntax-directed editing	PDL-EDITOR
design recovery	function abstraction	
	annotations/hypertext system for recording program documentation	DRA
system architecture level restructuring	breaking program into modules	BREAK
code generation	translating PDL into code	CG
testing/debugging	regression testing	RT
	test coverage analysis	TCA
	slicing	SLICER
	dynamic program analysis	DEBUG

* PKB: Program Knowledge Base

** PDL: Program Design Language

Table 1. Possible methods, techniques and tools for the framework

Here are brief descriptions of the tools listed in Table 1:

STRUCT - converts spaghetti code into structured code,

REFORMAT - reformats code using consistent and user defined indentation rules, inserts blank lines, etc.

RENAME - uniformly renames variables across programs; e.g. record variables which are

written to (or read from) the same file are given a single name in all programs within a system,

RE-PKB - computes and stores low-level design abstractions in a PKB (which may be a relational database),

RE-PDL - translates code into a PDL,

DG - generates a customized, low-level documentation from code; uses the PKB to cross-reference code and to incorporate control flow information, function calling graphs, etc. into the program documentation

SPA (Static Program Analyzer) - helps a programmer in analyzing code; SPA is built on top of the PKB; displays program information stored in the PKB in the form of lists; provides a query language to explore design abstractions from the PKB,

SLICER - a tool which computes and displays program slices,

PDL-EDITOR - a syntax-directed editor to view and manipulate the PDL,

DRA (Design Recovery Assistant) - supports function abstraction; provides a system of typed annotations to record high-level design abstractions recovered from code; provides a pattern matching mechanism to help a user interpret code in terms of problem domain concepts,

BREAK - helps a user to break a monolithic program into manageable modules,

CG (Code Generator) - translates PDL into code and integrates codes into an executable program,

RT (Regression Testing) - selects test cases to be executed after program modifications; RT runs test cases, validates real results against expected results and generates test result reports,

TCA - test coverage analyzer: indicates parts of a program which are not sufficiently covered by test cases,

DEBUG - source level program debugger.

2. The Software Metrics Evaluator

The software metrics evaluator assesses the current state of the programs to be maintained. This is one of the preliminary steps in the process since it identifies several important quality attributes of the software such as program correctness, reliability, portability,

efficiency, control and data complexity, modularization, complexity of module interfaces, flexibility, testability, and stability. Certain metrics based on these attributes (e.g. span of control, mean time to repair) measure how maintainable the software is, and form an information source on deciding what tools and techniques should be incorporated for the maintenance task. Other metrics such as the McCabe's cyclomatic complexity metric [5] help in the cost-benefit analysis that the manager would perform.

3. The Expert System

The expert system generates the maintenance methodology based on inputs from the metrics evaluator and the project manager. The metrics evaluator feeds an expert system with data about the quality of programs to be maintained. The project manager also provides information about the computing environment, available resources, objectives of a maintenance task, and other data which has an impact on the choice of the maintenance methodology. With the input from the manager and the metrics evaluator, the expert system infers the most appropriate maintenance methodology for a given situation using what we call **methodology knowledge**.

There are basically two types of knowledge or rules needed to compose a maintenance methodology: process definition rules and process composition rules. **Process definition rules** map characteristics of a maintenance problem such as software quality metrics, objectives of a maintenance tasks, and characteristics of a maintenance team into combinations of available maintenance methods, techniques, and tools to be applied to various components of the system under consideration. **Process composition rules** know about capabilities of tools and about the various ways tools may be integrated (i.e. integration rules, restrictions imposed by implementation, etc.). They know about viable ways of integrating activities into an overall maintenance process and integrating of tools into a CASE supporting this process.

The process composition rules reflect implementation and functional dependencies between tools, and logical dependencies between actions involved in maintenance process. For example, a PKB must be computed before one can analyze a program, generate documentation or do design recovery. Therefore, we need to use the RE-PKB before we can use the SPA, SLICER, DG or DRA. In addition, design recovery requires a PDL, so RE-PDL is a prerequisite for the use of the DRA.

Logical chains of actions involved in the maintenance process can be pictured at various levels of detail. For example, the global network shown in Figure 2 depicts the logical dependencies between maintenance methods:

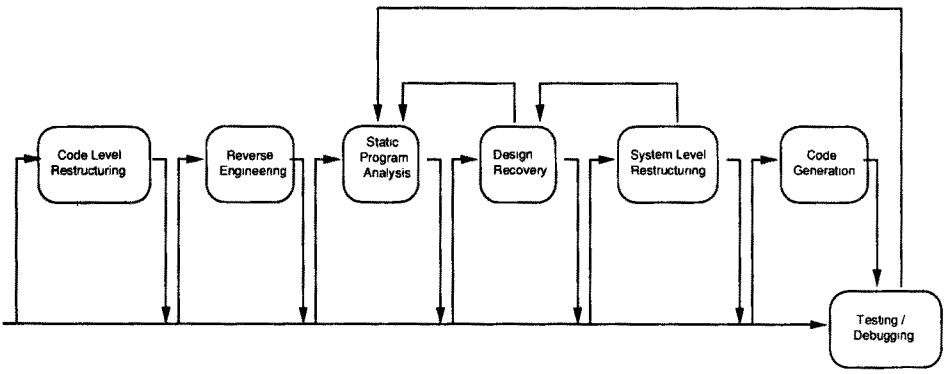

Figure 2. Logical dependencies in maintenance methods

More detailed networks can be constructed to describe logical dependencies between techniques and tools.

C. An Example

As an example of how we can use the framework to generate a maintenance methodology, let us consider a hypothetical situation. System S is written in Cobol and consists of three programs P1, P2 and P3; program P1 consists of four module files: M1, M2, M3 and M4. The manager responsible for the maintenance of system S intends to improve system S for future maintenance since system S contains definitions of some of the important functionalities of the organization, and is in fact the only system that captures this information. Cost is not a limiting constraint for this maintenance task. The manager has also provided the following information about system S:

1. S is 3 years old and has been heavily maintained.
2. It is expected that S will be in use for a few more years.
3. No up-to-date documentation exists.
4. Most of the functionalities solely captured by system S are encoded in M1, M2 and M3.
5. Many functions encoded in M1, M2 and M3 are quite stable within the organization; with small modifications and extensions they might be reused in many similar systems to be implemented. The functionality of M4 is not stable and much of it must be updated.
6. Most of the programmers responsible for maintaining system S have left the company and a new team will be assigned to the job. A programmer responsible for maintenance M4 is still around.

A run of system S through the metrics evaluator provides the following information:

1. S has about 100,000 lines of code.
2. Program control structure of S is poor.
3. Modular structure of P1 is poor:
   a) modules M1, M2, M3, M4 are about 10,000 lines each,
   b) module interfaces involve many external variables,
   c) there is only a small number of procedures (i.e. PERFORMs) and they do not seem to correspond to well-defined functions (i.e. module cohesion is low),
   d) procedures are multi-entry.
4. Module M4 is very complex.

Assuming that we have all the tools available, a maintenance methodology recommended for the above situation might be the following:

General Recommendation:

Most of the system will be analyzed, restructured and re-documented because it contains some vital information and procedures and also because a new team has just been assigned to maintaining system S. Modules such as M4 will be re-implemented. Modules M1, M2 and M3 will be restructured at the system architecture level to extract reusable clusters of functions and data. System maintenance will be done incrementally, using a CASE for program maintenance, customized to the maintenance problem in hand.

A Detailed Maintenance Methodology:

1. For all modules M such that
   - M is not re-implemented from scratch,
   - programmers who maintained M are not around, i.e. for all modules which will be analyzed, re-documented, etc., in particular, for M1, M2 and M3,
   a) do control flow restructuring (using STRUCT) and reformatting (using REFORMAT),
   b) unify variable names across programs (using RENAME),
   c) use RE-PKB to compute low-level design abstractions and RE-PDL to obtain the PDL level representation of code (i.e. Action Diagrams, Nassi-Schneiderman Diagrams, or any other PDL notation used as a company standard),
2. For modules M1, M2 and M3:
   a) use SPA, SLICER, DRA to analyze modules and to recover some of the high-level design abstractions; build a new modular structure (i.e. design PERFORMS and groupings of logically related PERFORMs together with data in program files) by isolating program functions and data; use BREAK and technique of parameterization,

b) fully re-document modules using DRA.

3. Re-implement module M4, consistently with a new structure of modules M1, M2 and M3.
4. The process of redesigning and re-implementing system S is done incrementally; whenever a process milestone is reached, regenerate code (using CG), update a library of test cases (or develop new tests) and do regression testing (using RT and TCA).

D. Tool Integration within the Framework

The concept of a framework for maintenance methodologies requires flexible and customizable tools. Despite their flexibility, tools within a future CASE for maintenance will have to be well-integrated. Tool integration is the only way to assure user-friendlessness of a CASE and smooth transitions between tasks and phases in the maintenance life-cycle.

There are two basic ways for integrating tools: one is through a uniform user interface and the other is through a common data model shared by the tools. Good results can be achieved only if both methods are used. A common user interface is easy to provide. To make tools communicate with one another through a common data representation is more difficult, and this requires a powerful model for the Program Knowledge Base (PKB). One needs to identify program abstractions which will help in the maintenance process and find a data model which will facilitate all types of program manipulations required by the maintenance tools. Not only should the PKB model be general, but it should be language-independent too. Maintenance tools are expensive to implement and it is important to be able to reuse tools across environments for different languages. Models which combine database models (for ease of querying information) with representations suitable for processing tree-like structures [20] are possible candidates. Specifications by grammars, compiler generation techniques and grammar-driven solutions (for syntax-directed editors) can be used to ensure flexibility of a maintenance environment.

V. CONCLUSION

In this paper, we explored a scenario for automating program maintenance. After reviewing the main phases of a maintenance life-cycle, we described the concept of a maintenance framework. The purpose of the framework is to help a manager select an optimal program maintenance methodology for a maintenance problem in hand. A methodology comprises the definition of a detailed maintenance process to be followed by a maintenance team. The maintenance process is further supported by a CASE, customized to a recommended methodology.

This scenario can be realized if tools covering the software maintenance life-cycle are available within the framework. Moreover, tools should be customizable and integrated in terms of a common data model of program information used by the tools. Maintenance tools are expensive to develop and no one company can deliver all the tools required in the framework. We need a way to allow tools from many vendors, specializing in automation of various maintenance tasks, to work together within the framework.

Lack of standardization in tools available today on the market makes tool integration difficult - tools from different vendors cannot be integrated because they use incompatible data models. In the short term, bridge technology can be used to translate a data model used by one tool into the data model understood by another tool. Data model bridges are useful for tools which are not very tightly coupled. For example, tools which are applied in strictly different phases of a maintenance process can be integrated through a bridge. Tightly coupled tools require a common data model. The recent proposal for AD/Cycle is the first step towards industry-wide standards for repositories of data used in application development. If the AD/Cycle is successful, in the future, tools from different vendors may be more amenable for integration than they are today.

ACKNOWLEDGEMENTS

The authors are grateful to the management of CSA Research for providing support and creating an encouraging atmosphere for this research programme. We are also grateful to the following members of DISCS and ISS (both at the National University of Singapore) community for many interesting discussions: Dr Ling T.W., Dr Loh W.L., Dr Lu H.J., Dr Tan C.L., Dr Tan K.P., and Mr Tan W.G.

REFERENCES

[1] Basili, V. and Rombach, H. "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE TSE, vol. 14, no. 6, June 1988, 759-773.

[2] Benedusi, P., Cimitile, A., De Carlini, U. "Post-Maintenance Testing Based on Path Change Analysis," Proc. of the Conference on Software Maintenance, 1988, 352-361.

[3] Biggerstaff, T. "Design Recovery for Maintenance and Reuse," IEEE Computer, vol. 22, no. 7, July 1989, 36-51.

[4] Blum, B. "Documentation for Maintenance: A Hypertext Design," Proc. of Conference on Software Maintenance, 1988, 23-31.

[5] Card, D. and Glass, R. *Measuring Software Design Quality*, Prentice Hall, 1990.

[6] Cashman, P. and Holt, A. "A Communication-Oriented Approach to Structuring the Software Maintenance Environment," ACM SIGSOFT, Software Engineering Notes, Vol. 5, no. 1, January 1980, 319-332.

[7] Chen, Y., Nishimoto, M. and Ramamoorthy, C. "The C Information Abstraction System," IEEE TSE, vol. 16, no. 3, March 1990, 325-334.

[8] Chikofsky, E. and Cross II, J. "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, January 1990, 13-18.

[9] Data Analyst by Bachman Information Systems, Inc.

[10] Elshoff, J. "Improving Computer Program Readability to Aid Modification," CACM, vol. 25, no. 8, August 1982, 512-521.

[11] Fisher, K.F. "A Test Case Selection method for the Validation of Software Maintenance Modifications," IEEE COMPSAC 77 Int. Conf., Nov 1977, 421-426.

[12] Frankl, P.G. et al. "ASSET: A System to Select and Evaluate Tests," Proc. of the Conference on Software Maintenance, 1985.

[13] Gallagher, K. "Using Program Slicing in Software Maintenance," TR CS-90-05, Ph.D. Thesis, University of Maryland, 1990.

[14] Grady, R. "Measuring and Managing Software Maintenance," IEEE Software, September 1987, 35-45.

[15] Harandi M. and Ning, J. "Knowledge-based Program Analysis," IEEE Software, January 1990, 74-81.

[16] Hartmann, J. and Robson, D.J.. "Approaches to Regression Testing," Proc. of the Conference on Software Maintenance, 1988, 368-372.

[17] Hausler, P. et al. "Using Function Abstraction to Understand Program Behavior," IEEE Software, January 1990, 55-64.

[18] Hlotke, J. R. "Complexity Analysis and Automated Verification", Proc. of the Conference on Software Maintenance, 1985 .

[19] Keables J. et al. "Data Flow Analysis and its Application to Software Miantenance," Proc. of the Conference on Software Maintenenace, 1988, 335-347

[20] Jarzabek, S. "Specifying and Generating Multilanguage Software Development Environments," IEEE Software Engineering Journal, vol. 5, no. 2, March 1990, 125-137.

[21] Oman, P. and Cook, C. "The Book Paradigm for Improved Maintenance," IEEE Software, January 1990, 39-45.

[22] Prather, R.E. and Meyers "The Path Prefix Software Testing Strategy," IEEE TSE SE-13, no. 7, July 1987, 761-765.

[23] Rich, C. and Wills, L. "Recognizing Program's Design: A Graph-Parsing Approach," IEEE Software, January 1990, 82-89.

[24] VIA/INSIGHT by Viasoft.

[25] Weiser M. "Program Slicing," IEEE TSE, vol. 10, no. 4, July 1984, 352-357.

[26] Weiser, M. "Programmers Use Slicing when Debugging," CACM, Vol. 25, no. 7, July 1982, 446-452.