Software Fault Diagnostic System

Charles E. Matthews
Fifth Generation Systems, Ltd.
8 Duborg Drive
Markham, ONT L6C 1R4 Canada
cmatt@attglobal.net

Abstract

This paper examines the role of design knowledge in the software life cycle. How is knowledge about a program design affected by changes to the source code? How can program design knowledge be used by a diagnostic system to identify failing software components? Which attributes of design knowledge are relevant for modeling in a design repository?

This paper addresses these questions with respect to an actual system that was developed to investigate the acquisition, capture, and representation of design knowledge for commercial software designs. Existing components of the system include a central repository for the program design model, a knowledge acquisition module for capturing functional and design specifications and a rule-based diagnostic component for isolating software faults.

Keywords: Design Modeling, Diagnostic Systems, Knowledge Acquisition

1. Introduction

Software developers have responsibilities that extend beyond the act of writing some set of machine interpretable instructions that represent the implementation of a computer program. In addition to program development, the software lifecycle for any non-trivial computer application should also include the specification and documentation of the program requirements, the documentation of the functional specification (which is the satisfaction of the requirement specification), the documentation of the program internal design, and the specification of application or acceptance tests. This paper focuses upon the representation of the program internal design in machine-readable form. Although the common use for design documentation is to preserve the design knowledge for the purpose of transferring that knowledge from one human being to another, there is value in representing the design knowledge in machine-readable form. If machine-readable design knowledge exists, intelligent tools can use that knowledge to assist humans with maintenance and diagnostic tasks.

Current research into diagrammatic representation and reasoning is an active field of investigation. This paper explores the application of some of the techniques from diagrammatic reasoning research to the design of intelligent tools for developing commercial software products. Specifically, this paper describes the objectives and architecture for a rule-based expert system that can assist a user in diagnosing faults within software applications. The value of machine-assisted maintenance can be significant if (or when) the attributes of the human-centric processes are prohibitively expensive or inherently unreliable.

2. Historical Beliefs

A common belief within the commercial software community is that the act of fault isolation and diagnosis is, inherently, a human ability. It is an art that is enhanced with experience and can be taught to new apprentices into the field. During the initial period of research and development of expert systems, early investigators also concluded that software diagnosis was not a practical objective for expert systems for a number of reasons.

Belief 1: Software faults are all unique. Once a software defect is fixed, it is gone. Therefore most diagnostic knowledge has a limited lifespan. Once a defect is fixed, the unique set of deductive actions that identified that defect is no longer needed.

Belief 2: Code changes are design changes. Each code change requires an update to the diagnostic rule base in order to maintain accuracy.

Belief 3: The cost needed to create a reliable diagnostic knowledge base is prohibitively expensive in time and resource.

3. Fault vs. Diagnosis

A software fault (or defect) is an artifact that exists within the design or within the implementation of the design. The programmer views a software fault as a specific set of source code lines. If the source code lines change, the fault changes – it disappears or becomes a different fault. Therefore, one could say that all software faults are defined as a specific representation of source code. (If the fault is missing functionality, then the fault is a missing representation of source code.) Therefore, software faults are unique.

For example, consider the following sequence of Pascal code. The function **FormatXmlStr** is supposed to search a given string for all occurrences of the **'&'** character, replace each occurrence with the character string **'&'**, and return the resultant string. However, an error exists in the first line within the *while* code block.

```
function FormatXmlStr(sIn : string) :
    string;
var
  idx : Integer;
  src : string;

begin
  //
  //  Search for the & character.
  //
  Result:='';
  src:=sIn;
  idx:=Pos('&', src);
  while (0 < idx) do
  begin
    //  The following line is
    //  incorrect.
    Result:=Result +
            copy(src, 1, idx) +
            '&amp;';
    src:=copy(src, idx+1, length(src)-
         idx);
    idx:=Pos('&', src);
  end;
  Result:=Result + src;
end;
```

The design knowledge that function **FormatXmlStr** represents could be stated as an assertion:

```
FormatXmlStr
    has_intention
        replace_&_in_string;
```

This code would translate the input string **'s & t'** into the corresponding string **'s && t'** which is incorrect behavior. The code will work correctly by replacing the first line of the *while* block with:

```
Result:=Result +
        copy(src, 1, idx-1) +
        '&amp;';
```

This is only one of the possible source code changes that could correct the fault. In all cases correcting this line of source code removes the fault without requiring any update or correction to the design knowledge for the function **FormatXmlString**. The design assertion for **FormatXmlString** was true before the source code was changed, and it remains true after the source code was changed. Because this particular software fault relied upon the specific line that was in error, this fault was unique to the source code before the correction occurred.

On the other hand, software diagnosis can be viewed as a procedure. In many cases the process of diagnosis is the process of recognizing when a given pattern or sequence of events is incorrect. Diagnosis does not explicitly depend upon the particular source code under examination. Diagnostic rules or procedures can be generically applicable to different faults and source code representations. Diagnostic procedures can also become more efficient when they use design-specific knowledge, but this type of knowledge is not absolutely necessary. Therefore, software diagnosis is not unique.

Consider the following diagnostic rule:

```
(defrule r-1 "Diagnostic rule 1"
  (funcObj (name ?func)
           (has_intention
                "replace_&_in_string")
           (inputStr "s & t")
           not (outputStr "s &amp; t"))
  =>
  (printout t
      "Function: " ?sub " is failing. "
      crlf)
)
```

This rule could identify that **FormatXmlString** has a fault by using the previous assertion of design knowledge about **FormatXmlString**. This diagnostic rule does not identify which line in the function is faulty, but it doesn't necessarily need to do so. By identifying which code segment potentially contains the fault, the rule focuses the programmer's attention within the appropriate area of the program.

Multiple faults within the same area of the design can use the same design-specific diagnostic reasoning to identify the different faults. The diagnostic reasoning procedures search for program specific patterns that occur repeatedly for different faults.

4. Code Changes vs. Design Changes

One of the important properties of diagrammatic representations or models is that they represent *abstractions*, i.e., they abstract out some of the information and represent other information. [2] Thus, the nature of an abstraction is that it differs in some fashion from the thing that it abstracts. Therefore, a change to the program code does not necessarily imply the need for a corresponding change to the program code's abstraction, i.e., the design model. As illustrated in the previous example, changing the defective line of source code did not affect the assertion about the program design for the function **FormatXmlString**. In reality, most code changes do not affect the design model.

In a previous study, the author analyzed four years of source code changes to an existing commercial software application. [8] The conclusion from this study was that the majority of source code changes have no effect upon the representation of the design model.

5. Knowledge Capture Cost

The belief that the cost to create a reliable diagnostic knowledge base is prohibitively expensive is based upon two presumptions: (a) the diagnostic knowledge can only be created by the expenditure of additional developer time (a scarce resource) and (b) the minimum amount of time that is necessary to represent any useful diagnostic knowledge is significantly large in comparison to the amount of time that is available.

An essential component for any toolset that uses design knowledge is a knowledge acquisition module that can assist with the task of knowledge capture and representation. The objective of the knowledge acquisition module is to reduce the knowledge capture resource cost. One way that the acquisition module can reduce this cost is to build or to infer knowledge automatically from existing artifacts of the design process.

As an example, consider the design object model for a software program. The object model is an abstraction of the program implementation, it is a normal by-product from most programming projects, and it is design knowledge. Thus, the appropriate toolset can infer design knowledge directly from the object model. If the object model exists in a file, an import utility can read and translate the model into the knowledge representation schema.

In the event that an import utility for the model file cannot be created (e.g. the modeling utility uses a proprietary file format), the knowledge acquisition tool could still reduce the cost for capturing the model knowledge by presenting the designer with some simple question/answer sessions about the model.

6. Design Knowledge and Diagnostic Knowledge

The various roles that design knowledge and models perform within engineering and programming activities is studied widely. Chandrasekaran has defined a Functional Representation framework for describing objects, properties and causal relations. [1] [4] He argues that a precise language is necessary for describing objects, properties and causal relationships. *"When systems fail, reasoning about how to fix them involves reasoning about*

malfunctions. When designers are designing, they look for components that can achieve certain functions. Predicting how systems would behave under various conditions of use or abuse also often requires knowing what the functions of the device are."[1] Whether or not a formal language is necessary in order to capture "all" functionality and intent for an object, this paper presents evidence that the representation of at least some of the functionality for an object allows some automatic inference of diagnostic knowledge.

Novak asserts that expertise in solving problems is characterized by the ability to set up or to represent a problem. [7] In addition, he uses models of abstract software components to investigate the task of automatic program generation. [9] *"We argue that real programs are based on the use of multiple views of data, and indeed, that multiple views of actual objects as different kinds of abstractions are common in design problems of all kinds."*[2] This paper contends that it is possible to represent the different abstractions to which Novak refers as part of the design model. Even if the design model fails to encode all of the different abstractions for an object, the features that are encoded are available for diagnostic use.

In all of these cases, design knowledge or models are the essential component to studying how objects and devices interact in complex systems. The representation of knowledge identifies the constraints that define the interactions. This paper asserts that similar constraints are identified and represented that aid in diagnostic tasks for software faults.

7. Program Modeling

It is fairly easy to conceive of a knowledge base in which the rules for diagnosing faults are customized and tightly coupled to the particular device/program that is being analyzed. Essentially, most model-based reasoning systems take this approach. The drawback is the direct dependence of the diagnostic rules upon the device model.

An alternative approach is to identify meta-rules that are generically applicable to the problem domain. When the design-specific knowledge is added to the knowledge base, the interactions between the design-specific knowledge and the meta-rules produce a diagnostic knowledge base for the application. Obviously, the critical aspect of this approach is the identification of appropriate meta-rules. If the rules are too generic, then it is difficult to achieve good diagnostic performance. If the rules are too specific, then the knowledge base is coupled too tightly to a specific design – increasing the difficulty in maintaining the knowledge base as the design changes and evolves.

A deep and thorough understanding of the problem domain must guide and influence the definition of the meta-rules. However, the meta-rules do not necessarily need to encompass all possible aspects of the problem domain or all possible design models that could be defined for that problem domain. The minimum sufficiency for the meta-rules is that they must cover all design modeling constructs that are used by the specific design that is being modeled. Let us examine some of those constructs.

Subsystem Packaging: A common practice for large applications is to modularize and group modules of similar features and functions into the same package. These packages may be called subsystems, components or libraries. The particular term is unimportant. The relevant aspect is that the features and functions within the package have an inherent similarity.

For example, to identify the subsystems within an application, use the following description.[3]

```
(system
    (has-name  Coverscan)
    (has-subsystems
        (GUI, SDM, AS, RGM, IPC))
)
```

Subsystem Message: One of the commercial designs that the author studied was a multi-threaded system. Each of the thread streams was a separate subsystem, and they communicated with each other by sending messages to an intra-process controller that queued the messages for execution for each thread. Because this messaging mechanism was an essential element in the design model, the diagnostic rules needed to account for it.

[1] [1] p. 1.
[2] [9] p. 9.

[3] The sample model representations that appear in this paper are representations only. Do not attach much importance to the particular syntax of the language. The purpose is to represent the intent of the model rather than how to propose any specific modeling language.

```
(subsystem
    (has-name  SDM)
    (handles-message
        (LoadFileMsg,
         LoadHeaderMsg,
         GetRegionInfoMsg,
         SelectForAnalysisMsg))
)
```

External Interface API: When a collection of functions or classes is packaged into a library, the library often publishes its external interface. This external interface identifies those functions and classes that are available for access by other users.

```
(library
    (has-name  libACV)
    (has-proc
        (dwACVSetup,
         dwACVSetCheckValue,
         dwACVClose))
    (has-class (CacvObj, CacvCounter))
)
```

Asynchronous Events: Event handling is a common requirement for many applications. Almost all GUI-based applications implicitly handle the common windowing events, e.g. mouse actions, screen painting, etc. However, some applications need to respond to explicit events like serial communication I/O operations and program interrupts from special purpose hardware equipment. Most likely, the model must account for all of the explicit events. The model could ignore many of the implicit windowing events. However, the model should account for any windowing event that initiates any significant processing. The decision point for deciding which events are important enough to model is left to the designer.

```
(subsystem
    (has-name  GUI)
    (handles-event
        (ReqLoadDesign,
         ReqAnalyze,
         ReqLoadRegion))
)
```

The objective for the model representation is to represent the design model in programmatic form. For an object-oriented design, it is possible to describe the class hierarchy and the class interrelationships. For subsystems or library modules, identify the external interface: subsystem messages, library APIs, common library routines, etc. Identify the mechanisms by which the outside world communicates with the subsystem and, in return, how it communicates with the outside world.

8. Generic diagnostic rules

The expert systems literature contains a number of references to different meta-rules for diagnosing faults in complex systems. [6] Some of the simpler rules are:

1. Divide and conquer – If you know the expected path of reasoning, select a point in the middle. If the parameters appear correct at the focus point, the problem probably occurs later in the reasoning path.
2. Causal analysis – If you see this, you must do that.
3. Case-based reasoning (lazy inference) – The previous time when I saw this, the problem was that.
4. Easter egging – Browse around. Look for problems.

Another class of meta-rule is known as a *Focus of Attention Heuristic*. [10] The focus of attention rule starts with a set of possible faulty components and reduces that initial set to a smaller set by the elimination of those components that have a low probability of occurrence. For example, assume the following design assertions:

```
(subsystem
    (has-name  SDM)
    (handles-message
        (LoadFileMsg,
         GetRegionInfoMsg))
)

(subsystem_message
    (has-name  LoadFileMsg)
    (is-implemented-by
        (ClearGlobalInstanceLists,
         ssOpenDsnFile,
         ReadFileInfo))
)

(subsystem_message
    (has-name  GetRegionInfoMsg)
    (is-implemented-by
        (HandleGetRegionInfo
         ssInitSelections,
         GetDefinedModulesList,
         copydbpkg,
         GetStateMachineList,
         MakeRootObject,
         SearchForModuleNames,
         copydbobj,
         copydbmod))
)
```

According to these assertions, the subsystem **SDM** handles two messages – **LoadFileMsg** and **GetRegionInfoMsg**. These messages are implemented by the functions listed in their respective **is-implemented-by** attributes. If no other information is available other than the statement that the **SDM** subsystem has a fault someplace, then all of the code within that subsystem could be the source for the fault. However, if it is known that the fault symptoms appear when the subsystem is processing the message **LoadFileMsg**, the user should focus attention upon the functions involved in implementing **LoadFileMsg**. There is no claim that the fault source will truly be found within this set of functions – only that it is most appropriate to focus attention upon them until additional data focuses attention somewhere else.

9. Conclusions

Two significant benefits occur if the design knowledge for a software system is available in machine-readable form. First, large software systems are, by their nature, extremely complex. As systems grow in complexity, the number of possible interactions between the different parts of the system tends to grow, also. After some point it becomes extremely difficult for anyone to understand and to manage the complexity of the software without assistance. Paper documents and class diagrams provide some assistance, but the ability to navigate the actual software by encoding the design knowledge should provide significantly better results.

Second, a characteristic of the software industry is: the individuals who are responsible for diagnosing, maintaining, and extending a software system, are rarely the same individuals who originally designed and implemented the system. Even if the programmers remain with the same company, they are transferred to other projects. Their knowledge about the design of the system is lost – unless that knowledge is documented in some fashion. If the design knowledge is documented such that it is machine-readable, then intelligent programs can be built that use that design knowledge to assist the programmers who must diagnose or evolve the software.

10. Future Directions

The author has developed a knowledge-editing tool for specifying the design knowledge for a system. The editor exports the design knowledge either as an XML file or as a CLIPS[4] file of *deffacts* statements. An embedded CLIPS expert system acts as a diagnostic assistant to a user who is debugging a software fault in the design. The expert system reads the design knowledge file, queries the user for initial symptoms and guides the user to the area of the design that is most likely causing the fault.

The knowledge editor needs to be enhanced with a dialog generation module. This module will contain domain-specific rules. These rules provide two functions: a) initiate question/answer sessions with a program designer for the purpose of acquiring the design knowledge, and b) implement model constraints that check for the existence of inconsistent or incomplete design knowledge.

The diagnostic assistant needs to be embedded within a graphical interface. The current system is command-line driven. The future system will display

[4] The C Language Integrated Production System (CLIPS) is an expert system shell. Originally developed by NASA in 1985, the current system is maintained as public domain software.

sections of the design model graphically while guiding the user through the diagnostic task.

Although the initial prototypes require a Microsoft OS platform, an objective for future work is platform independence. Therefore, all future work will be done with ANSI C++ and Tcl.

11. References

[1] B. Chandrasekaran, J.R. Josephson, Representing Function as Effect, **AAAI-96 Workshop on Modeling and Reasoning about Function**, Portland, OR, August 1996.

[2] B. Chandrasekaran, Diagrammatic Representation and Reasoning: Some Distinctions, **AAAI Fall 97 Symposium Series, Diagrammatic Reasoning**, Boston, MA, 1997.

[3] B. Chandrasekaran, J.R. Josephson, V.R. Benjamins, Ontology of Tasks and Methods, **Proceedings of KAW '98, Eleventh Workshop on Knowledge Acquisition, Modeling and Management**, Banff, Alberta, Canada, April 18-23, 1998.

[4] B. Chandrasekaran, J.R. Josephson, Function in Device Representation, to appear in **Journal of Engineering and Computers**, Special Issue on Computer Aided Engineering.

[5] D. Dodson, 3D Aids Cooperation with Diagrams that Think, **'Thinking with Diagrams' Colloquim, IEE**, London, Jan. 18, 1995.

[6] P.E. Johnson, D. Volovik, I.A. Zualkernan, C.E. Matthews, Design Knowledge for Discovering Troubleshooting Heuristics, **Proceedings of the IASTED International Symposium on Expert Systems Theory & Applications**, June 26-28, 1989, Zurich, Switzerland.

[7] H.J. Kook, G.S. Novak, Representation of Models for Expert Problem Solving in Physics, **IEEE Transactions on Knowledge and Data Engineering**, vol.2, no. 1 (March 1991), pp. 48-54.

[8] C.E. Matthews, **Effect of Source Code Changes upon a Program's Design Model**, internal paper, 2001.

[9] G.S. Novak, Interactions of Abstractions in Programming, **Lecture Notes in Artificial Intelligence**, vol. 1864, pp. 185-201, Springer-Verlag, 2000.

[10] D. Volovik, I.A. Zualkernan, P.E. Johnson, C.E. Matthews, A Design Based Approach to Constructing Computational Solutions to Diagnostic Problems, **Proceedings of the National Conference on Artificial Intelligence**, July 29 - August 3, 1990, Boston, Massachusetts.