Expert System - Intelligent Debugger

I. Introduction:

In artificial intelligence, an expert system is a computer system that emulates the decision-making ability of a human expert. Expert systems are designed to solve complex problems by reasoning about knowledge, represented primarily as if–then rules rather than through conventional procedural code. The first expert systems were created in the 1970s and then proliferated in the 1980s. Expert systems were among the first truly successful forms of AI software.

An expert system is divided into two sub-systems: the inference engine and the knowledge base. The knowledge base represents facts and rules. The inference engine applies the rules to the known facts to deduce new facts. Inference engines can also include explanation and debugging capabilities.

Expert System = Knowledge Base + Inference Engine

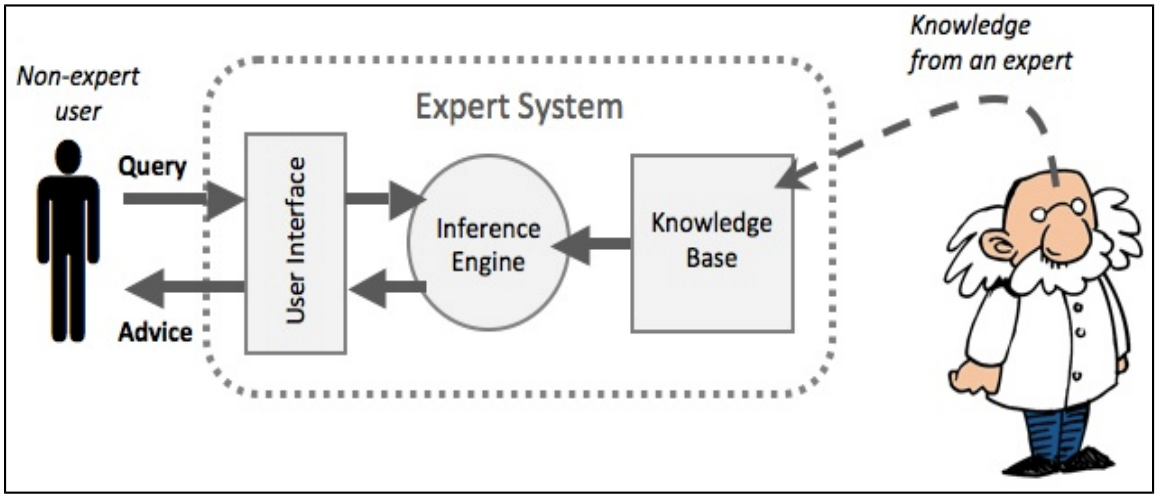

Fig. 1: Expert system schematic model

According to Edward Feigenbaum, Intelligent systems derive their power from the knowledge they possess rather than from the specific formalisms and inference schemes they use. An intelligent agent is composed of architecture and a program which is a concrete implementation, running on the agent architecture.

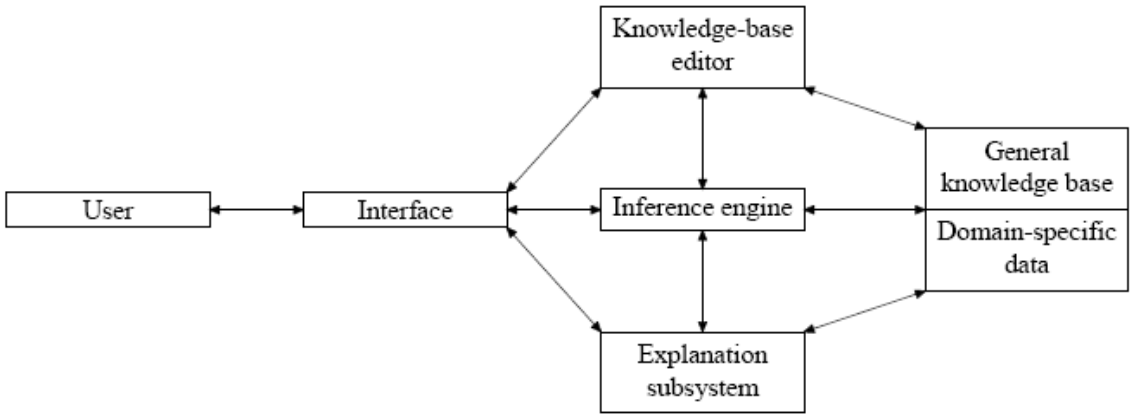

Fig. 2: Architecture of an intelligent system

II. Problem Definition:

Programmers build a basic logic, algorithm and then convert it into a working code. This is then moved onto the compilation and execution phase and is tested for errors in each phase. The programmer has to himself debug to resolve all the errors that arise which can be an overhead in the building time of exhaustive software. This forms the basic motivation for the concept of an automatic intelligent debugger.

The solution comprises of building debugging software that not only points out on the errors but also suggests quick fixes on the problem much similar to the debugging performed in certain IDEs. The technology requires constructing a compiler using different libraries.

III. Innovation:

Automated debugging systems have a long history with interesting results produced by research prototypes and deployed applications. These systems range from tutoring systems that possess detailed knowledge about the individual programs as well as about the typical programmer errors that occur in exactly these programs, over Bayesian Net formalisms that employ statistical results about error reports, to traditional debugging approaches such as Algorithmic (or Declarative) Debugging. The use of model-based diagnosis principles as a basis for software debugging research is the more recent approach. We illustrate the potential of model-based reasoning by discussing several models (differing in expressivity and assumptions on language semantics) that are currently in various stages of realization, from prototype implementations to test use in an industrial environment.

The debugging can be of various types each of which can be briefly described as under -

Functional debugging: The function of a segment of the code is expressed in terms of input state, output state, and proviso. Given a particular input state (i.e., a partial specification of a state of the program), the function will provide a particular output state (i.e., another partially specified state) under the assumption that the proviso is true.

Probabilistic debugging: Runtime errors are a frequent occurrence and are protocol led by a large dump-file which has to be analyzed to fix the bug. A typical dump-file contains a snapshot of relevant parts of memory at the time the illegal operation, e.g., an illegal memory reference, occurred. The goal driving the development of this system is to assist the program analysts to determine the root error in a program segment. A root error is defined as an initial instruction (or sequence of instructions) that led to an erroneous situation.

To reach this goal the DAACS system performs two tasks:
1. A pre-diagnosis task fetches all necessary information from the dump-file, e.g., the type of the operating system violation, or other relevant symptoms.
2. The diagnosis task itself is divided into two phases. In the first phase a logical inference procedure is used to determine feasible execution paths using information gained by the pre-diagnosis. The second phase uses probabilistic inference to rank the paths delivered from the logical phase.

The goal is to compute the probability that a path is faulty, i.e., contains the bug, under the condition that a fault has been detected. We denote this probability by p (P = i | F = true) where P is the random variable associated with the selected path and F a random variable indicating the occurrence of an error. This probability on can be calculated according to the Bayesian rule –

$$p(P = I \mid F = true) = \frac{p(F = true \mid P = i) * p(P = i)}{p(F = true)}$$

Model-based diagnosis: It attempts to provide fundamental knowledge about the relationships between densities in a particular domain. From the point of view of diagnosis, given a particular application domain, these first principles are embodied in the description of
– The structure of the system to be diagnosed, as described by a set of components and their connections
– The behavior of the system, described in terms of the values produced by the system on its outputs for certain input values

IV. Additional Information:

The various features of an intelligent debugger can be listed as follows:

- It greatly reduces the build time of software.
- It doesn't need to rebuild the source code while monitoring inputting parameters and outputting results of the traced APIs in the target program automatically, only monitoring the input and output of APIs
- Exports monitor result as txt format
- Uses VBScript or JavaScript to control Auto Debug
- Source Code level monitor

- Automatic analysis parameter type with PDB files
- Very easy to generate PDB files without source code if you know the API prototype
- Tracing your application with release version
- Supports Debug version and Release version, without the need of source code
- Supports multithreading
- No need to know the prototype of the functions of the API for the user
- Not only trace for exported APIs, but also be effect for undocumented APIs

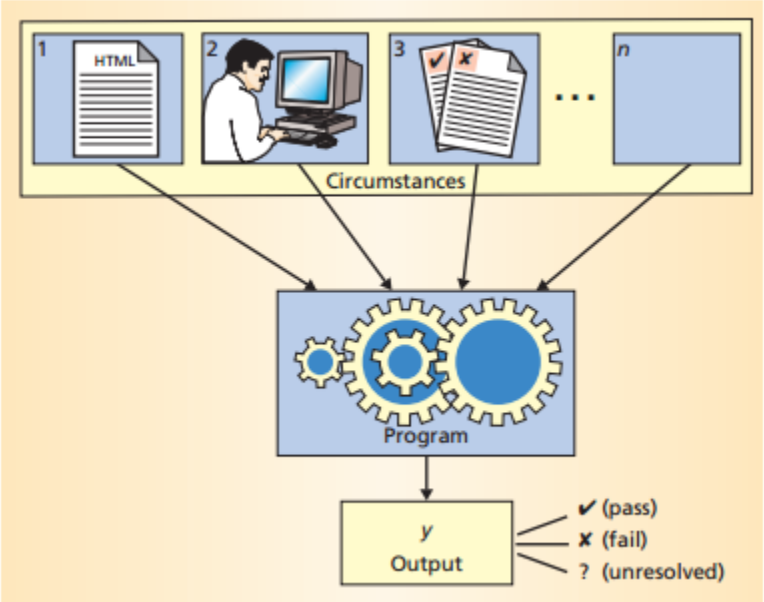

Fig. 3: Circumstances affecting a program's behavior

External failures typically come from program input, user interaction, and program changes. Within each of these categories are myriad circumstances, any one of which could be the root cause of the failure. Intelligent debugging requires a test to prove that each circumstance is really failure inducing. An automated testing environment typically provides such a test, but the number of test runs can be quite large. The trade-off is that, unlike conventional debugging, intelligent debugging always produces a set of relevant failure-inducing circumstances, which offer significant insights into the nature and cause of the failure. Also, if we know something about the structure of the failure-inducing circumstance, fewer tests are required.

Program input is typically easy to access, modify, and evaluate. Adapting smart debugging to run a program on a different input should take only a matter of hours. Examining user interaction history requires external tools that record and play back user interaction. Program changes are moderately easy to access and simple to modify, but altering configurations can be difficult, depending on the tools used.

V. Patents and References:

Patents:

1. US 8752025 B2 Protecting breakpoints in a software debugger

It includes a breakpoint protection mechanism that detects when the program being debugged has been modified to overwrite one or more instructions corresponding to existing breakpoints. When the debugger halts execution of a program being debugged, all of the set breakpoints are checked by determining whether the instruction corresponding to each breakpoint has changed. If any of the instructions corresponding to the breakpoints has changed, the corresponding breakpoint is removed. An optional warning may be provided to the user to inform the user of any removed breakpoints.

2. US 8914777 B2 Forward post-execution software debugger

A method and system debug a computer program by using trace data, which is a recording of the sequence of machine instructions executed by a program during a time period along with the addresses and values of memory locations accessed and modified by each machine instruction. After the time period, the method and system use the trace data to simulate the execution of the program during the time period under the control of a debugger. In addition, the method and system use the trace data to simulate the execution of the program during the time period backwards in time under the control of the debugger.

3. US 7353498 B2 Multi-process debugger

A method of debugging a set of processes is disclosed. There is included providing a first debugger and forking the debugger, thereby creating a set of inner debuggers and an outer debugger. Each of the set of inner debuggers is configured to debug one of the set of processes. There is also included employing the outer debugger to interact with the set of inner debuggers to debug the set of processes.

4. US 6986124 B1 Debugger protocol generator

It is a method for automatically generating front-end code and back-end code that are both compatible with a specification, such as the JDWP communication protocol. First, a detailed protocol specification is written that contains a description of a communication protocol between the front-end code and the back-end code. The detailed specification is then input into a code generator that parses the specification. The front-end code is then automatically generated from the formal specification, and may be written in a first computer language such as the Java™ programming language. The code generator then generates the back-end code, which may be written in a second computer language such as C.

5. US 8291388 B2 System, method and program for executing a debugger

A method for controlling a debugger, the method includes: determining whether to execute a certain breakpoint of the debugger in view of certain breakpoint conditional information and in view of at least one previous visit, during the execution of the debugger, to at least one other breakpoint of the debugger; and selectively executing the certain breakpoint in response to the determination.

References:

- Paper titled 'A Survey of Intelligent Debugging' by Markus Stumptner and Franz Wotawa http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.53.5818&rep=rep1&type=pdf
- Research paper titled 'Automated Debugging: Are We Close?' by Andreas Zeller http://www.csm.ornl.gov/~sheldon/bucket/Automated-Debugging.pdf
- Expert Systems - https://en.wikipedia.org/wiki/Expert_system
- Patents –
  https://www.google.co.in/patents/US8752025
  http://www.google.com/patents/US8914777
  http://www.google.com.ar/patents/US7353498
  http://www.google.com.ar/patents/US6986124
  http://www.google.com.gh/patents/US8291388
  patents.justia.com/patent/8122381
- http://www.generation5.org/content/2005/Expert_System.asp