CANASTA: The Crash Analysis Troubleshooting Assistant

Michael S. Register and Anil Rewari

CANASTA (crash analysis troubleshooting assistant) is a Digital proprietary knowledge-based system developed by the Artificial Intelligence Applications Group (AIAG) at Digital Equipment Corporation in collaboration with Digital's customer support centers (CSCs). It is targeted to assist computer support engineers at CSCs in analyzing operating system crashes, traditionally one of the most complex types of problems reported by customers. Digital started work on CANASTA in January 1988. A version of CANASTA that assists in the analysis of vms operating system crashes was successful: It is currently deployed at CSCs in over 20 countries and is used to resolve over 850 crash-related customer calls each month. It is estimated that in time savings alone, it saves Digital over 2 million dollars each year.

CANASTA's success largely results from the innovative way in which it integrated different problem-solving modules that model the different types of problem-resolution strategies that experts use in this domain. These strategies include making quick checks (rule based) on whether the crash at hand is because of a known cause, using deeper analysis (decision tree–based) reasoning to resolve new types of crash problems, and checking for similarities among unresolved cases (a form of case-based generalization) that can lead to the identification of new hardware and software bugs. In fact, CANASTA's unresolved crash processor distinguishes it from other expert systems: It directly assists the expert in the generation of new knowledge regarding crash-causing bugs.

CANASTA also integrates different technologies that have not been combined before in this domain. It integrates a remote scripting package and rule-based inference to provide sophisticated automatic data collection that allows it to automatically gather data from the customer's machine thousands of miles away. It uses a rule-based system for quick checks on known problems. It uses a tool that allows experts to quickly encode troubleshooting knowledge graphically in the form of decision trees. It uses database technology to store case-related information that can be accessed later. CANASTA also includes an innovative distributed knowledge maintenance system that automatically collects knowledge from experts worldwide at all CSCs and automatically validates and redistributes this knowledge to all other sites. This approach facilitates the sharing of knowledge across various geographic sites.

In the following sections, we describe the crash analysis problem domain, the functions and architecture of the deployed system, details about the development and deployment stages, and the business payoffs.

The Crash Dump Analysis Problem

When an operating system detects an internal error so severe that normal operations cannot continue, it crashes. For many operating systems, this process involves signaling a fatal condition and shutting itself down in an orderly fashion by saving the contents of the registers, stacks, buffers, and memory at the time of the crash in a crash dump file. The underlying cause for the error can be a failure in user-written code, hardware failure, microcode failure, or an error in system software. *Crash analysis* is resolving the problem, whether it is in the hardware or software, and identifying a fix or a "workaround."

Analyzing crash-related problems is not an easy task because there is no fixed algorithmic method to identify the cause of the crash. Experience plays a large role in identifying the problem. Without CANASTA, the diagnostic process is as follows: First, the support engineer remotely connects (through modem servers) to the customer's machine to read the crash dump file on the crashed system. Once connected, s/he remotely scans the crash dump file and tries to identify the major symptoms of the crash.

The support engineer then checks to see if the symptoms match any known problems (bugs) that were already identified. Such problems are documented in over a dozen different textual databases. The support engineer usually does a key-word search over these textual databases based on the current symptoms. A significant number of problems being investigated by support engineers match previously identified problems. If an appropriate match is found, the support engineer notes the solution specified in the matched database article.

If the support engineer cannot match the current problem to a known problem, then s/he needs to do some deeper analysis. This process involves traversing the stack of procedure calls made prior to the point where the crash occurred, looking at the assembly language instructions and their operands, and locating where the error occurred. Knowledge of assembly language is required, and for many cases, knowledge about how the operating system works is required.

If the cause of the problem is identified, the solution is provided to the customer. For hardware-related problems, this solution usually involves replacing a hardware component. For microcode-related problems, it involves providing access to a new microcode revision level. For software-related problems, it can involve providing a new patch (a relatively small piece of software written to fix a specific problem in the system or application software), making recommendations to upgrade to a new operating system version, or correcting the software configuration of the system.

Unresolved cases are eventually seen by experts. They can try to resolve them by deeper analysis of the problem. However, based on their experiential knowledge about other similar crashes or an analysis of other unresolved crashes in the textual databases, they might be able to see similarities with other crashes and make generalizations that help them in resolving the crash at hand.

The process, as described, has several problems: Previously identified problems that are known to cause crashes are scattered in dozens of different textual databases. On the average, it takes about 30 minutes to scan the various textual databases to check whether the current crash results from a known problem. There are several reasons for this delay. First, many times one cannot identify a known problem because one does not scan the textual database where the right article lies. Second, even when known problem types are described in the textual databases, the descriptions might be incomplete. There is no set format for the articles. Finally, with the constant release of new products and software versions, it has become difficult for experts to document bugs, in a timely fashion, that are introduced by various hardware and software products.

Collecting symptoms itself requires knowledge. The techniques for collecting some of the symptoms are documented (Kenah, Goldenberg, and Bate 1988), but others are not documented and are difficult to obtain. Because of the amount of knowledge required to collect symptoms and identify bugs, often less experienced support engineers arrive at wrong conclusions and give wrong solutions. These errors invariably result in the customer calling back when the problem reappears, which results in increased calls to CSCs. Furthermore, support engineers follow no uniform approach in performing deeper analysis, and many of them have difficulty in performing deeper analysis.

Experts spend more time than necessary in looking at unresolved cases that are escalated to them because they do not have ready access to other similar cases seen in the company. Having access to similar unresolved problems allows them to make generalizations, resulting in faster resolution times.

These problems result in a substantial increase in the average time that a customer has to wait for a solution (also an increase in the time a customer's business can be disrupted), resulting in higher costs for Digital and a decrease in customer satisfaction. There are some tangible benefits to applying AI technology to this domain. Using an automated intelligent system results in quicker analysis times because it is considerably faster than manually searching through textual databases.

With an expert system approach, knowledge can be distributed and accessed easier than it was previously through the use of textual databases. Furthermore, expert system technology allows for much easier maintenance of the knowledge base than conventional software technology. This ability is especially beneficial in this domain because new bugs in hardware and software are identified on a continuous basis. Also, intelligent validation techniques can be used to validate the knowledge as it is entered into the knowledge base.

Using AI techniques also facilitates the generation of new knowledge. By using AI techniques to group similar unresolved cases, experts can now identify new problems faster because they have multiple instances (crashes) of the unresolved problems. This information also results in improved communications between the engineering development groups and the service delivery groups because now the service groups can send multiple instances of a problem to the engineering groups for resolution.

Finally, the automatic collection of data from the crash dump requires a significant amount of knowledge. Maintenance of this knowledge is difficult because it changes frequently. Encoding the knowledge in a rule base enables easier maintenance.

CANASTA provides an architecture that uses AI techniques to obtain these benefits.

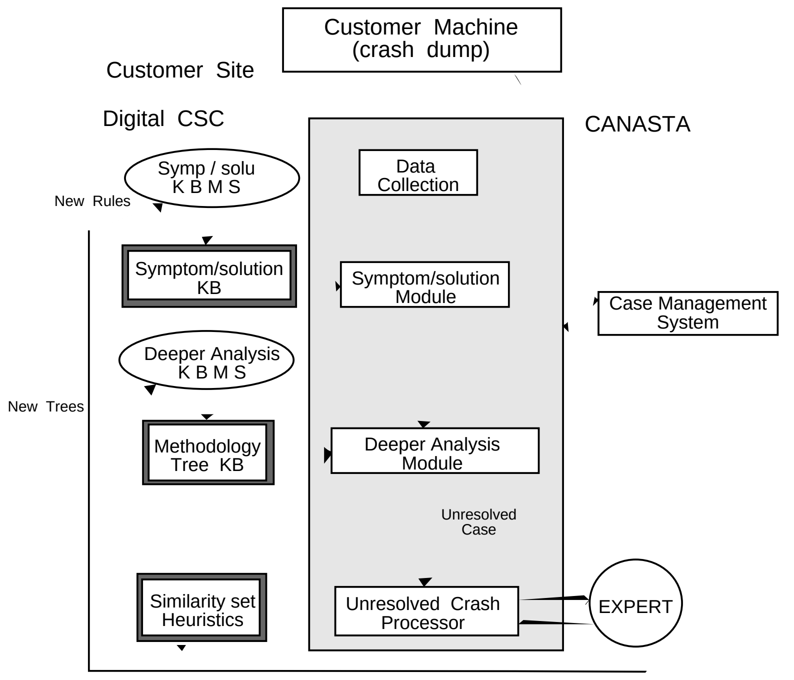

Figure 1. The CANASTA *Architecture.*

Architecture and Functions

In this section, we describe the architecture and functions of CANASTA. In figure 1, the horizontal line at the top separates the customer machine that crashed from the host machine at CSC where the support engineer works. The customer machine can be thousands of miles away. The crash analysis process begins with the support engineer establishing a remote connection to the customer system through a modem line.

Data Collection

The data-collection module extracts data from the crash dump file at the customer machine without transferring the entire dump to CSC. It uses a remote scripting package that enables it to remotely run commands on the customer machine. Figure 2 depicts the interaction between the data-collection module, located at the CSC machine, and the customer's machine.

A rule-based controller controls the sequence of information-gathering activities. It determines what information still needs to be collected

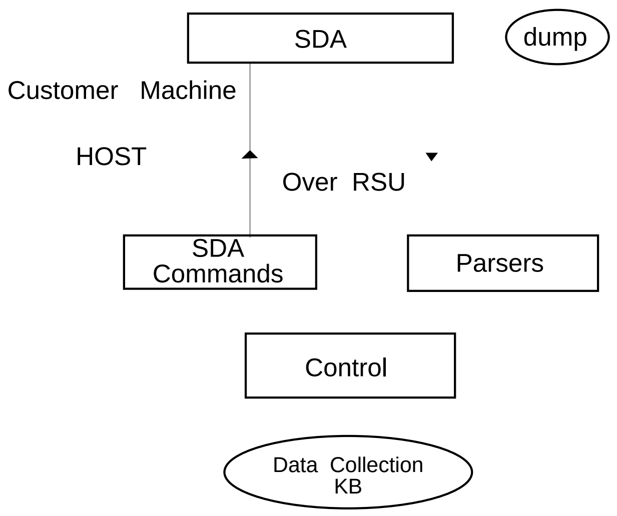

Figure 2. The Data-Collection Module.

and how. There is a large amount of knowledge regarding methods for extracting the relevant symptoms from the dump file for different types of crashes. The knowledge is represented in the form of about 750 OPS5 rules. When the controller determines that a particular piece of information is required from the crash dump at the customer machine, it sends a command to the customer machine over the modem connection using the remote scripting package. The commands usually involve running the *system dump analyzer*, a tool that formats binary crash dump files into ASCII text and displays particular parameters of interest. Output from the system dump analyzer at the customer site is automatically sent back to the data-collection module at CSC by the remote scripting package. This information is parsed, and then based on the parsed value, the controller decides which command to send next to the customer machine. It collects 15 key symptoms that CANASTA uses to make an initial hypothesis about whether the problem is because of a known bug. In case the crash is not resolved by the symptom-solution or deeper analysis modules (described later), it later collects additional parameter values and saves them with the unresolved case so that these additional symptoms are available to experts who look at the unresolved cases.

Using AI techniques (rule-based data collection) has resulted in considerable leverage. Initially, we started out with a procedural representation of the knowledge, but later, it was found easier tomaintain and add data-collection knowledge by representing it declaratively in the form of rules. It was difficult to represent the data-collection techniques using algorithms because too many exceptions had to be dealt with. Using a rule base allows CANASTA to represent this knowledge more conveniently and collect only the parameters that it needs to meet our goals of completeness and speed.

The goals for the data-collection module were to automatically collect data for 95 percent of the crash types with 99 percent accuracy and to collect the initial 15 symptoms within 3 minutes over a 1200-baud modem line. All these goals were met.

Symptom-Solution Module

After the initial set of symptoms are collected, CANASTA invokes the first of its analysis modules. The *symptom-solution module* uses a knowledge base of symptom-solution rules to see if the given crash matches a well-known hardware or software bug. A sample rule is shown in figure 3. If the combination of initial symptoms in the current crash matches a rule, then a hypothesis (the "description" part of the rule) is displayed to the support engineer. For many cases, this hypothesis is, in fact, the conclusion. However, for other rules, a further test is required (the "technique for confirmation" part of the rule) to actually confirm whether the current hypothesis is true. This determination involves running further commands remotely on the customer machine, that is, probing the dump file in more detail. If the test results are positive, then the hypothesis is proved to be correct, and a solution is then displayed (the "solution-recommendation" part of the rule). The rationale for having techniques for confirmation tied to particular rules is to save the overhead costs of performing such tests for every crash when they are only required to confirm specific bugs.

Rule-based pattern matching and heuristic identification of problems give considerable leverage in trying to quickly identify the cause of the problem, especially in a domain where almost half of the problems seen are repeated problems that were previously seen by others. This module is implemented in FOXGLOVE, a rule-based shell developed internally in our group. Currently, there are over 630 symptom-solution rules in CANASTA-VMS. Almost 75 percent of the rules point to software problems that cause crashes. There are rules for problems caused by system software bugs, application software bugs, hardware faults, faulty system configuration parameters, and microcode bugs.

Deeper Analysis Module

If a match is not found in the symptom-solution knowledge base or if

```
IF:
   VMS-VERSION            = (one-of 5.0 5.0-1 5.0-2 5.1 5.1-1)
   BUGCHECK-TYPE          = INVEXCEPTN
   MODULE-OF-CRASH        = ETDRIVER
   MODULE-OFFSET          = x1546

THEN:

   DESCRIPTION:
          "The crash occurs due to a synchronization problem in the
          ETDRIVER while it is stepping down its list of UCB."

   TECHNIQUE FOR CONFIRMATION:
          "Check to see if R5 = FFFFFFFF. This tells us that the UCB
          has already been severed."

   SOLUTION-RECOMMENDATION:
          "You should first recommend the customer upgrade to VMS
          version V5.2 because this patch and 6 others are included in
          this major release!

          If the customer cannot upgrade then send them patch
          number 0055."
```

Figure 3. A Symptom-Solution Rule.

the confirmation test does not succeed, then the *deeper analysis module* is invoked. This module uses a *methodology tree knowledge base*, containing knowledge of how experts troubleshoot crashes, to guide the user in analyzing the crash dump. This module suggests the most appropriate tests to be performed given the particular crash type and eventually indicts a hardware component in the case of hardware problems or narrows the list of possible software causes.

Currently, the deeper analysis module's troubleshooting knowledge is organized into several decision trees that are separated on the basis of high-level symptoms. A graphic view of a portion of the length violation tree is shown in figure 4. At the top of the tree, we only know that this particular crash was caused by a length violation. At run time, CANASTA asks the user questions to determine the reason for the length violation. By the time CANASTA reaches a leaf node in the tree, CANASTA has either isolated the reason for the crash or significantly narrowed the list of possible reasons.

Currently, the tree knowledge base covers a small breadth of crash types but one that captures 80 percent of the problem types seen at

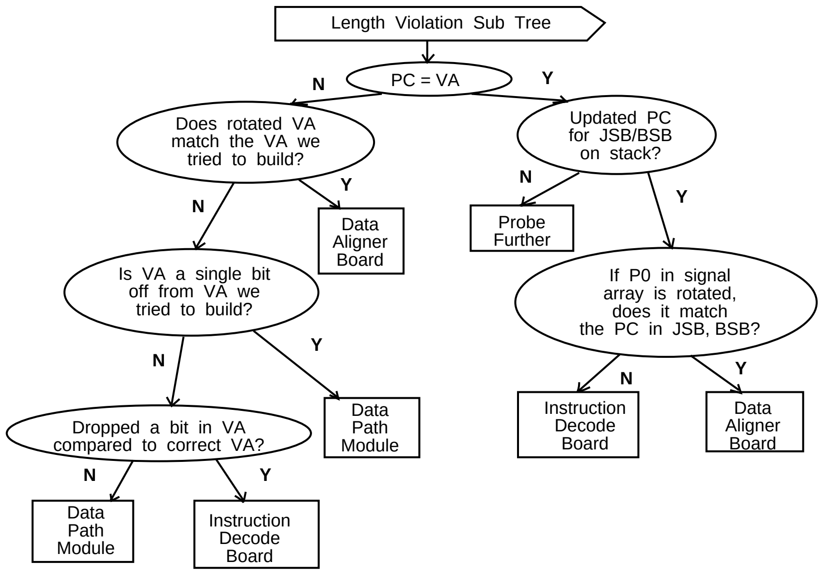
Length Violation Sub Tree
PC = VA
N
Y
Does rotated VA match the VA we tried to build?
Updated PC for JSB/BSB on stack?
Y
N
N
Y
Data Aligner Board
Probe Further
Is VA a single bit off from VA we tried to build?
If P0 in signal array is rotated, does it match the PC in JSB, BSB?
Y
N
N
Y
Data Path Module
Instruction Decode Board
Data Aligner Board
Dropped a bit in VA compared to correct VA?
N
Y
Data Path Module
Instruction Decode Board

Figure 4. The Length Violation Decision Tree.

CSCs. In some places, the tree knowledge base lacks depth. Some of the conclusions it reaches are of a high level, and further probing (by the support engineer) is required to identify a cause at the hardware component level or software module level. Experts at CSCs were provided with a maintenance tool for expanding the depth and breadth of the methodology tree knowledge base.

Unresolved Crash Processor

CANASTA's unresolved crash processor periodically collects unresolved crash cases from all CSCs that run CANASTA and, using heuristic knowledge, groups similar unresolved crashes into similarity sets. It attempts a high-level classification of the cause for each similarity set. These sets are then made available to users worldwide, although the main beneficiaries of this module are the much-overworked expert-level support engineers and the engineers in the software and hardware development groups to whom the unresolved cases are escalated. AI techniques such as the heuristic grouping of cases into similarity sets and the heuristic classification of these sets are the basis of making the unresolved crash processor a powerful tool that is crucial in expediting the process of experts generating new knowledge about crash-causing bugs.

The unresolved crash processor design consists of three subcomponents: the back end, the browser, and the matcher.

The Unresolved Crash Processor Back End. The *back end* collects all the unresolved crash cases and groups them into similarity sets, providing a high-level classification for each set. First, it collects the unresolved crash cases from CSCs all over the world. It then uses a heuristic rule base to group similar cases into sets. The rules in this heuristic rule base are domain specific and include knowledge such as "crashes occurring in the same software module with very similar offsets are most probably due to the same underlying cause." Once the sets are formed, it infers a high-level characterization of the cause for the crashes in each similarity set. It uses a set of heuristics to perform this function; for example:

> If all the crashes in a set occurred in the same minor release of VMS, then the problem seems to be in that minor version of the VMS software (as opposed to hardware or the microcode).

If several of these heuristics are applicable, then the similarity set can have several possible causes attached to it. The back end then removes all sets that have a small number of cases in them, treating them as noise. The remaining sets correspond to the most common types of problems remaining unresolved, as seen at CSCs worldwide.

Finally, the unresolved crash processor back end builds a set of rules, one for each similarity set, that are used by the unresolved crash processor matcher module. As described later, the matcher is used by users during run time to check whether the current unresolved case corresponds to a known set of similar cases. To form a rule that represents a match with a similarity set, the back end takes certain key symptoms of the cases in a set and includes them as conditions in the left-hand side of a rule. Three symptoms—the software module, the offset, and the VMS version—are always included as conditions in the left-hand side of the rule because they are vital to detecting similarity. If there is a single value for this symptom in all the cases in the set, then the left-hand side of the rule will simply have an equality test. If there are multiple values, then a condition of the form "(one-of <symptom> '(<symptom-value1> ...))" is generated. If multiple causes are listed in a similarity set, then a rule is formed for each possible cause.

To illustrate, on the left-hand side of figure 5 is an example of a similarity set. In this example, only the symptoms deemed significant in identifying the problem are included. The right-hand side of the figure contains the two rules that would be generated from this problem description. Two rules are generated because there are two potential causes associated with the problem. The first rule identifies the problem when caused by a microcode problem. The left-hand side includes the basic symptoms (module, vms-version, offset) as well as another

```
((PROBLEM-TEMPLATE:                IF:
cpu-type:        NAUTILUS            module-of-crash = CTDRIVER
vms-version:     4.4, 4.2            vms-version     = (one-of 4.4 4.2)
module:          CTDRIVER            module-offset   = (one-of AA9 A7E)
offset:          AA9, A7E            cpu-type        = NAUTILUS
image:           NETACP)           THEN:
                                     Cause seems to be due to the microcode
                                     version on this CPU family.

(CAUSED-BY MICROCODE IMAGE)        IF:
                                     module-of-crash = CTDRIVER
individual instances follow …)       vms-version     = (one-of 4.4 4.2)
                                     module-offset   = (one-of #xAA9 #xA7E)
                                     current-image   = NETACP
                                   THEN:
                                     Cause seems to be related to the
                                     image running at the time of crash.
```

Figure 5. A Similarity Set (left) and the Two Rules Derived for This Set (right).

symptom that is critical in isolating microcode problems (cpu-type). The second rule is identical to the first except that cpu-type is replaced with current-image. In this case, the current image is critical in isolating software problems that occur when this specific image is running on the system.

Unresolved Crash Processor Browser. The *browser* is an interface meant primarily for expert support engineers viewing the sets of unresolved cases. To the expert support engineers, the automated grouping into sets saves substantial effort and time. When manually looking at unresolved crashes, they do not have other similar unresolved cases available to compare them with. If they did have other such cases, it would be much easier for them to make generalizations and resolve the current unresolved crash. Having similar unresolved cases as a comparison helps in strengthening hypotheses and discarding others. It significantly shortens the resolution time and, thus, accelerates the generation of new knowledge about known crash-causing problems that goes into the symptom-solution or deeper analysis knowledge bases.

Unresolved Crash Processor Matcher. The *matcher* is available to all users during run time. When the symptom-solution and deeper analysis modules do not assist in resolving the current crash, then a user has an unresolved case. S/he can quickly check whether the current crash corresponds to a set of similar unresolved crashes in the unresolved crash processor database. If a match exists, then the user knows that such cases have already been seen at CSCs, and by looking at the comments attached to the particular similarity set of unresolved cases, s/he might find information that would assist him(her) in proceeding further. For example, an expert trying to resolve a set of similar unresolved crashes might place a comment stating that any support engineer seeing similar crashes should get some further parameters to store along with the case that will help the expert in further analysis.

The unresolved crash processor matcher consists of a rule base, with one rule for each similarity set. A rule fires if the symptoms of the current crash match with certain key symptoms of the crash cases in a particular similarity set. The user can then look at the individual crash cases in this set and the comments attached to this set.

Implementing the Unresolved Crash Processor. The unresolved crash processor was written in VAX Lisp, Digital's implementation of Common Lisp (Steele 1989). The unresolved crash processor matcher rule base uses FOXGLOVE rules.

In the latest run of the unresolved crash processor back end, it collected approximately 2500 unresolved cases from 20 CSC sites worldwide. With a threshold of 4 (at least 4 cases in a set), about 90 similarity sets were formed. Our main expert estimates that over 50 rules were generated by him alone within the short time of 8 weeks as a result of accessing these sets through CANASTA. As a comparison, in the previous year when the unresolved crash processor was not available, it is estimated that the knowledge generated by all experts at the U.S. CSC during the course of the entire year was less than 100 bugs (rules).

Case Management Module

All crash cases that are seen by support engineers are saved in a case database. It allows them to browse through both the resolved and unresolved cases seen at their site. There are several important benefits of saving case-related information in a database. CANASTA always saves the status of a case before exiting, including information about whether the crash was resolved using information from outside CANASTA's knowledge bases. Experts can now retrieve all such cases resolved by non-CANASTA means and fill the holes in the CANASTA knowledge bases by adding information about such problems and their solutions. Also, a history of crashes seen on a particular machine is available and can be used when analyzing new crashes on this machine. Finally, it is easy to generate statistics from the database. For example, the most common crash types being encountered or the number of crashes caused by a particular software module can now be generated.

Knowledge Base Maintenance

A knowledge-based application such as CANASTA is only as good as its knowledge base. Critical to CANASTA is the timely and periodic update of its various knowledge bases. New bugs in hardware and software are continuously identified, and previously identified bugs frequently extend to new releases of software or might have better solutions available at a later date. CANASTA has two main types of knowledge bases: a rule base where well-known symptom-solution rules are encoded and a decision tree–based knowledge base where the troubleshooting methodology that experts use is encoded. The development team provided tools and processes to allow the support engineers to maintain these knowledge bases on their own.

For maintaining the symptom-solution knowledge base, a tool was developed that allows experts to enter knowledge of known problems in the form of templates into an exclusive crash-related textual database. The templates are basically textual renditions of the rules in the knowledge base. The CANASTA template consists of essential symptoms that identify a problem and textual attributes such as a description of the problem, the technique for confirming the problem, and the solution. The user first enters the values of only the minimum set of symptoms that are relevant to identify the problem. The tool dynamically checks whether the values being entered are valid (it has validation routines attached to each symptom slot). It also checks for consistency among different symptom values (there are constraints in the relationships between some of the different symptoms). The checks are based on domain-dependent information. For example, version 5.0 of the VMS operating system is valid for the newer central processing units (CPUS). If the user types in VMS 5.0 as the VMS version and then types in the CPU type as VAX 780 (an old CPU that is incompatible with VMS 5.0), then the system indicates a warning. Therefore, even though the value 780 is valid as a DEC CPU, this value is inconsistent with the value of the VMS version already filled in. This type of checking ensures that the conditions in the rule are consistent with each other. The maintenance tool resides at CANASTA sites worldwide, so that crash analysis experts from various locations can enter knowledge into CANASTA. Once the template is filled, the maintenance tool automatically forwards it to a central site where all such templates are collected.

All templates received at the central site are parsed and translated into rules and are then compared for consistency with all the existing rules in the knowledge base. The central-site software has both domain-dependent, as well as domain-independent, heuristics to check for consistency. A new rule is added only if it is found to be consistent with all the other rules in the knowledge base. In this case, the corresponding textual template is also added to the exclusive crash-related textual database, allowing the experts to view the rules in a textual format. They can modify textual entries in the textual database, resulting in corresponding modifications to rules in the rule base. The updated knowledge base and the corresponding textual database are then copied over Digital's internal network by demon processes running at CANASTA sites around the world. The knowledge and textual databases are updated and distributed on a weekly basis.

The symptom-solution knowledge base of previously identified bugs has been exclusively maintained by the experts since September 1989. New rules are added when new bugs are identified in software or hardware. Rules are modified when bugs are found to extend to other CPUs, operating system versions, or software modules; the techniques for confirming the bugs are changed; or better solutions become available. On the average each week, about 5 new rules are entered, and about 10 existing rules are modified.

The deeper analysis knowledge base is maintained using DECtree, a tool developed by Digital. DECtree provides a graphic user interface that allows one to create or modify decision trees. It translates the decision trees into C source code that is compiled and linked to the rest of the CANASTA run-time system.

Development

Early in the project, based on a set of requirements and discussion sessions with several experts, a high-level architecture of the system was designed. We followed many of the suggestions in Kline and Dolins (1987) for identifying the knowledge representation schemes to use. During the first year, we went through several design-implement cycles, similar to the iterative cycles mentioned in Buchanan et al. (1983). The development schedule was tied to base-level releases, with each base level having increased functions or knowledge. By the time we released base level 4 in April 1990, all the modules had been implemented with full function.

The CANASTA team had the full-time commitment of one of Digital's leading experts in crash dump analysis for knowledge-acquisition purposes. The knowledge acquisition was undertaken in two ways: We periodically interviewed the expert, and he scanned the textual databases and placed knowledge about crash-causing bugs into CANASTA using the maintenance tool previously described.

The initial development costs included about six person-years for software development and about two years of expert time for knowledge acquisition.

Testing and Deployment

After each base-level release, CANASTA was tested by both the developers in Marlboro and experts at the U.S. CSC in Colorado. Several types of testing were performed. Each module was tested to check that it was behaving correctly. The developers extensively tested all the modules, except the data-collection module, to verify that they were working as designed. The data-collection module was extensively tested by several of the experts. The experts generated several hundred dumps to test the data-collection module.

After testing module behavior, the different modules in CANASTA were tested on actual customer crashes to verify that they arrived at conclusions that were similar to ones that expert-level support engineers had arrived at. The data-collection module was run on some difficult crash dumps to verify that it could collect all required symptom values. The symptom-solution module was tested on different types of cases to find out how it compared with experts in identifying known problems, both in terms of accuracy and time. To test the unresolved crash processor, our expert collected a number of unresolved cases that he had grouped into similarity sets and had provided high-level classifications for. We ran these unresolved cases through the unresolved crash processor to compare its results with the expert's results. The validation heuristics for the maintenance tool were tested by entering inconsistent symptom-solution rules.

Several criteria were used to measure the success of the deployment effort. First, CANASTA had to be installed at all the major CSCs worldwide. By October 1989, this effort was accomplished: CANASTA was installed at over 20 CSCs in the United States, Europe, Canada, Australia, and Japan. We also wanted CSCs to receive training in using CANASTA. This goal was met by April 1990, with training being offered at most of the major CANASTA sites. We also wanted support engineers to use CANASTA on most crash-related calls seen at CSCs. Current figures indicate that in most CSCs outside the United States, almost all crash-related calls are run through CANASTA. During the summer of 1990, almost 200 crash-related calls were being run through CANASTA every week. We see this goal as being met.

Because CANASTA runs on existing CSC hardware and requires only Digital software, the main deployment cost at CSCs was training. To date, at the U.S. CSC in Colorado, 2 instructors have spent 1 week training about 105 engineers to use CANASTA.

Nature and Estimate of Payoff

There have been a variety of business payoffs since CANASTA's deployment. The bottom line is that the use of CANASTA at CSCs has resulted in substantial savings to Digital in handling crash-related calls. It is estimated that in time savings alone, it is saving Digital over two million dollars each year. Most of the savings directly result from the savings in call-resolution times and increased first-time-correct resolutions. Some of the types of payoffs are hard to quantify, although they are clearly felt by users and managers at CSCs. The most significant of these payoffs are described below.

The average time for handling crash-related calls has decreased. The automatic data-collection module collects the symptoms in less than 3 minutes for most crashes and does so with greater accuracy than most engineers. Cases where a technique for confirmation is not necessary are directly identified within 10 seconds. The rest of the cases require a further test to confirm the initial hypothesis. It usually takes a few minutes to perform the additional test. In both cases, a great deal of time is saved over the previous method of scanning a dozen databases to find a match, which on the average took over 30 minutes.

Experts are able to identify the causes of unresolved cases much faster now by using the similarity sets of unresolved cases generated by the unresolved crash processor. This step leads to a significant decrease in time for identifying bugs in new software and hardware products. Furthermore, the sharing of a common-case database and knowledge base among the engineering staff (the developers of software) and CSCs has resulted in quicker dissemination of knowledge about existing problems and quicker resolution of unresolved cases.

More accurate identification of software problems has resulted in a decrease in the unnecessary replacement of hardware. Besides the cost of the boards, some of which are expensive, this step also saves in the fixed cost associated with sending a field engineer to replace the board. Recently, an expert discovered with the help of the unresolved crash processor that a set of similar unresolved cases was the result of design faults in a hardware product. Catching such hardware design faults early in the release period saves Digital a significant amount of money because a reduced number of hardware components have to be replaced in existing installed systems.

The presence of a case-management facility, where all cases, resolved and unresolved, are saved, has resulted in several advantages: First, support engineers can retrieve cases relating to those that they are investigating by features, such as symptoms, customer ID, dates, and type of resolution. Also, by running the cases in the case database at a site against the weekly knowledge base updates, many unresolved cases are flagged as resolved because rules are modified or added. Support engineers can go back to customers and on a proactive basis suggest changes to their systems that will prevent a repeat of the problem. This sort of tracking was not possible earlier. Furthermore, support engineers at CSCs can now trace the crash history of each customer machine for which calls were reported. Having the crash history of a specific machine can help in isolating the cause of the latest crash in this machine.

The introduction of a distributed knowledge maintenance system allows the incorporation of expertise from multiple sites worldwide. The smaller CSCs in Europe and Asia can now benefit from the large volume of crashes seen at the U.S. CSC. They now share a global case database and the same knowledge base.

CANASTA is perceived as a good training tool at CSCs for new support engineers working on crash analysis. The symptom-solution module, which identifies almost 45 percent of the cases as known bugs, has confirmation techniques for many rules that indicate to the engineers the type of tests that are required to confirm different hypotheses. The decision trees help new engineers in learning about the sequence of tests that are required to confirm different conclusions.

Customer satisfaction has increased because of the quicker resolution of problems and the accurate identification of problems the first time around. The U.S. CSC has been experiencing a reduction in the gross volume of crash-related calls. Several of the expert engineers are attributing this reduction to the increase in first-time-correct resolutions by support engineers who are using CANASTA.

Summary

CANASTA represents a major breakthrough in the way crashes are analyzed at Digital. Using AI technology, CANASTA integrates multiple problem-solving strategies into a single architecture. CANASTA not only assists in initially analyzing a crash, it also provides assistance in the generation and distribution of new knowledge about crash-causing problems.

The success of CANASTA has led us to believe that we can develop diagnostic systems with architectures similar to that of CANASTA to assist support engineers in the resolution of various types of computer hardware and software problems. In fact, an effort is already under way to develop CANASTA-ULTRIX to handle crashes in the ULTRIX operating system (Digital's implementation of UNIX). As we gain experience with several other domains, Digital might well build a shell that is based on this architecture. Such a shell would allow us to release a series of quickly built AI-based diagnostic systems that could be of great use at CSCs.

Acknowledgments

The CANASTA project has been fortunate to have the enthusiastic support of Steve Brissette, one of the leading experts in Digital in the domain of crash dump analysis. We are also grateful to Charlie Gindhart and Mark Fisher, who contributed much of their time toward developing the initial version of the automatic data-collection module. Shanti Subbaraman worked with us during the first year of the project, and we thank her for her efforts in the project during this critical time period. Mark Swartwout, the CANASTA project manager, played a crucial role in helping us access the right people and acquiring the necessary resources.

References

Buchanan, B.; Barstow, D.; Bechtal, R.; Bennet, J.; Clancey, W.; Kulikowski, C.; Mitchell, T.; and Waterman, D. 1983. Constructing an Expert System. In *Building Expert Systems*, 127–167. Reading, Mass.: Addison-Wesley.

Kenah, L. J.; Goldenberg, R. E.; and Bate, S. F 1988. *VAX-VMS Internals and Data Structures*. Burlington, Mass.: Digital.

Kline P. J., and Dolins S. B. 1987. Choosing Architectures for Expert Systems, Technical Report, Corporate Computer Science Center, Texas Instruments Inc., Dallas, Texas.

Steele, G. 1989. *Common Lisp: The Language*. Burlington, Mass.: Digital.