A knowledge-based system for MVS dump analysis

by N. G. Lenz
S. F. L. Saelens

A new domain in software problem determination can be automated by means of this knowledge-based system. The system imitates a human problem solver by using the same tools and the same diagnostic approach as the experts use, including the processing of human-readable data. The application is fully integrated within the target Multiple Virtual Storage (MVS) operating system to ensure user acceptance. A large variety of knowledge is contained in the system, ranging from pattern-recognition knowledge to basic MVS knowledge and problem-solving strategies. The diagnostic approach is based on a model of software problem situations and on diagnostic reasoning methods adopted from the medical application domain. The goal of the project was to solve a significant part of the problem resolution process automatically, rather than to build yet another tool for use in software problem determination. This system is a first step to further automation in this area.

Three groups of people are involved in the development of a knowledge-based system, users, experts, and knowledge engineers. The users consult a knowledge-based system to solve their problems. It is well-known that user acceptance of a knowledge-based application is not easy to attain. Lack of acceptance is for various reasons, some of which may be psychological. Therefore, the project must be very carefully positioned, and the users must be involved early in the development process to ensure that their expectations are met. The integration of the knowledge-based system into the working environment of the user is a key point.

The development process for a knowledge-based system differs substantially from the staged development process for traditional software. Experts and knowledge engineers work together to bring the system into being. The experts need not have data processing skills, but they do have to contribute the application-specific expertise. The knowledge engineers elicit the knowledge from the experts and map it to an executable form, for example, a set of rules.

We describe a knowledge-based system called the MVS Dump Analyzer and some experiences with it, along with the intricacies involved in the practice of knowledge-based programming.

The task of solving MVS software problems

Since its introduction in 1973, the IBM Multiple Virtual Storage (MVS) operating system has become large and powerful, but also complex. The complexity becomes obvious in error situations. Typically, MVS is run in a big computer installation. The installation is managed and maintained by a dedicated group of MVS operators and system programmers who have to solve software problems. It can also be difficult for application programmers to resolve error situations.

The availability of competent people for problem resolution is a bottleneck in computing centers. In small computer installations, the skill may not be available at all. In big computer installations, experts typically are very busy. It takes years of experience to acquire the knowledge that system programmers need for MVS problem resolution. Our goal was to remedy this bottleneck situation by providing MVS problem-solving expertise in the form of a knowledge-based system. Thus, MVS problem resolution can be partially automated.

Application and system problems can be diagnosed by analyzing the dump containing the problem situation.

To handle MVS problems, the problem-solving process is organized into several stages. Our investigation showed that in each stage approximately 90 percent of the arriving problems are solved. The other 10 percent must be routed to the subsequent stage. The following are typical stages:

1. *Help desk*—A person having a problem usually contacts the help desk first. The problems are similar to these: "I cannot print my data set on printer xyz," or "My application panel does not come up, although I did not change anything." The latter problem might result from a version update of a specific tool. Often several people have the same problem. In that case the help desk will immediately know what to do.
2. *System support*—For about 10 percent of the cases, system programmers have to take a closer look at the problem. They consult detailed manuals, sometimes analyze a dump, and solve the majority of the problems. The rest must be forwarded to the IBM service organization.
3. *First-level IBM support*—This organization receives customer problems and resolves them to a certain extent. For example, a database is queried to find out if the problem is already known. Difficult problems or those not found in the problem database are passed to the second-level support.
4. *Second-level IBM support*—Here the sophisticated problems are analyzed. The outcome can be manifold, for example, a new problem which must be fixed by IBM, a customer error in the use of a specific software component, or an installation problem.

The knowledge required to solve help desk problems is typically shallow and rapidly changing. We decided to concentrate on the other areas, where more thorough reasoning is necessary. By including all basic MVS knowledge, we ensure that the knowledge stays relevant for a long period of time.

Problem categories. In the MVS environment all software problems can be categorized according to symptoms in the following way:

- Abend (abnormal termination)
- Wait
- Loop
- Program check
- Message
- Incorrect output
- Other (for example, hardware, teleprocessing problem, etc.)

A problem may fall into more than one of these categories. In some cases a dump is automatically generated. For other cases a dump can be produced in order to analyze the problem.

Both application and system problems can be diagnosed by analyzing the dump containing the problem situation. In many cases this analysis is sufficient to resolve the problem. In some cases further investigation is needed, based on the results provided by the dump analysis. The limits of the dump analysis are reached when a human problem solver must look into the program logic to find out what the programmer meant to do.

Because not all system programmers are experienced dump readers and because analyzing a dump is very time-consuming, even for a specialist, we decided to build a knowledge-based system for MVS dump analysis. Dump analysis is complex; no algorithmic solution is available today. Therefore, dump analysis is suitable for a knowledge-based system. The knowledge to analyze a dump is not volatile. This stability is important to ensure the maintainability of the system.

Knowledge acquisition

A major point that distinguishes knowledge-based from "classical" program development is the necessity of knowledge acquisition. It corresponds to system analysis and partly to the design phase in traditional programming. The ultimate goal of the knowledge acquisition phase is to have a knowledge model. The knowledge engineer analyzes the expert's knowledge and at the same time puts together a logical construct of it. This process is a combination of analysis and synthesis and is usually performed in a highly iterative way.

Performance is pure experience and the most tacit knowledge.

Since we had to elicit different kinds of knowledge, we used various techniques:

- Open interviews (experts talk about their work, no directed questioning)
- Closed interviews (with directed questions to clarify details)
- Expert lessons on selected MVS topics
- Books and system literature
- Example protocols (watching how the expert proceeds to solve a problem)
- Expert reviews of protocols and written notes
- Iterative enhancements of the prototype
- Completion of selected parts of the application by the expert

The last way is only possible if a framework for the part of the knowledge to be completed is defined. This way is probably the most preferable, since no information is lost by the knowledge engineering process. The two methods mentioned prior to the last one have an advantage from a cognition point of view: Humans are more inventive in criticizing or commenting on a solution than in creating one.

The knowledge engineering method we used is best reflected in an article by O. E. Laske.[1] The article, which is about building knowledge-based systems, distinguishes three dimensions of expert knowledge:

- *Task environment*—The physical and organizational environment in which the expert works
- *Competence*—The expert's intrinsic, tacit professional knowledge
- *Performance*—The step-by-step problem-solving procedure of an expert, i.e., his or her experience

Each of the three dimensions had to be identified and acquired in a different way. With regard to our project, these three dimensions of expert knowledge have the following meanings.

Environmental knowledge reflects the context of solving MVS software problems. It consists of the service organization structure and the tools used to solve the different tasks. This knowledge is not explicitly coded in rules. It is reflected in the way we implemented and designed the MVS Dump Analyzer and in the way we address its users. This environmental knowledge was retrieved by visiting customer and IBM computer installations and service organizations. The main technique used here was the open interview.

Competence is theoretical in nature. Competence knowledge was collected by closed interviews and from teaching sessions, from reviews, and from literature. As there is a vast amount of MVS documentation, knowing which topics are important is also expert competence. Typical examples of competence knowledge are the control block structure and chaining, the use of supervisor calls (SVCs), the operating system error recovery, or the status of a resource.

Competence mostly includes analytical MVS knowledge, but it also includes basic heuristics. (For example, a task with a nonzero completion code typically has a problem, but there are exceptions.) In our system this knowledge is coded in rules and is also reflected in the structure of the data.

Performance is pure experience and the most tacit knowledge. An expert may solve a problem without being able to tell why he or she proceeds in a particular way or any other way. This compiled knowledge can be retrieved by watching the expert at work. In our case we produced protocols and captured the screen images of the data at which

the expert looked. We printed these screens and reconstructed the diagnostic reasoning with the expert during reviews.

This kind of knowledge is partly coded in rules and partly reflected in the module structure. It covers heuristics (rules of thumb) and strategies.

The MVS Dump Analyzer

The developed MVS Dump Analyzer offers the following functions:

- *Identification of the failing module*—The *failing* module need not be the module issuing the error. Knowing what module is failing is the key to resolution of the problem, because the module owner must repair the error. The name of the module identifies the owner of the failing piece of software.
- *Analysis of the problem to determine the cause*—Often problems are not isolated. One error situation can cause another and so on. The latest error that is seen on a screen is sometimes only a consequence of an earlier malfunction. Our system analyzes the problems as far as it is possible in order to determine the cause.
- *Generation of a technical problem description*—Independent of the ownership question, a problem can only be fixed if relevant information for the maintainer is provided. The MVS Dump Analyzer describes each problem in a detailed manner. (See Figure 7 later in the paper for an example.)

Using the MVS Dump Analyzer results in a number of benefits. First, inexperienced application and system programmers can locate the problem without any help from an experienced system programmer. The MVS Dump Analyzer provides expertise and supports learning by its explanation capabilities.

Second, it relieves experienced system programmers from routine tasks because it contains the human expertise to perform a standard analysis of problems. The MVS Dump Analyzer is strong for the "bread and butter" type of problems. It gives the expert the time to solve the really difficult problems.

Further, the problem-solving process becomes more systematic. The MVS Dump Analyzer output consists of a condensed error description. Similar errors are described in a similar way because the process is consistent. This error description helps in identifying already known problems.

Human expertise on the machine. Tools are available on MVS to help analyze dumps. We did not want to build another tool. Rather we wanted to provide a solution to the problem of analyzing MVS dumps by building the system on top of the tools.

The MVS Dump Analyzer can be used in interactive or in batch mode.

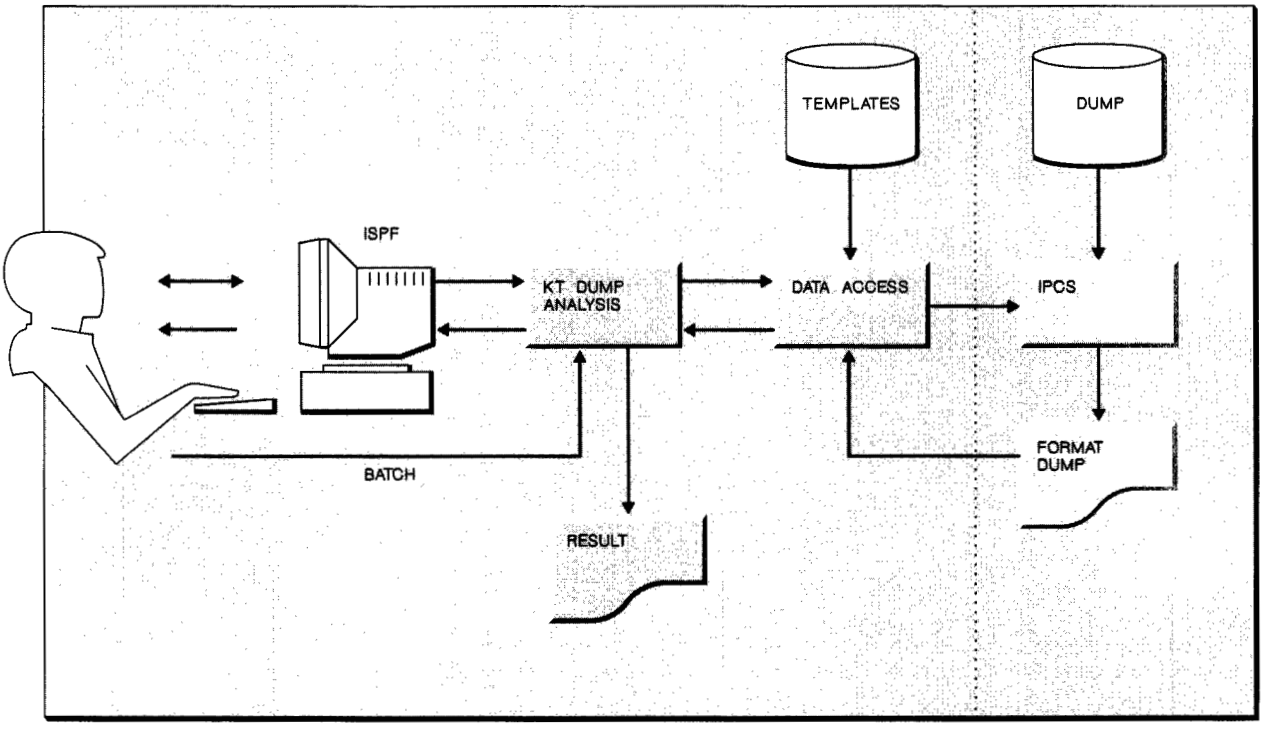
The question of user acceptance has to be carefully addressed. After having studied what others experienced in related projects, the following two points seemed to be most important:

- The effort for a user to consult the system must be minimal.
- The system must run at the location where the problem data are located.

To ensure user acceptance of the MVS Dump Analyzer, the system must run so as to be integrated with the MVS environment. All problem data available on the machine must be accessed automatically. In contrast to many other prototypes, the MVS Dump Analyzer avoids user interactions for obtaining data that are available on the system. The MVS Dump Analyzer can be used in interactive or in batch mode. Batch mode lets the analyzer sequentially process several dumps in the background or overnight.

We tried to automate the work of a human dump analyzer and to imitate it on a computer as closely as possible. Human experts use tools that often provide their output only in human-readable form. We decided to rely on the same tools as the human experts do, even if the tools did not offer a machine-readable interface. A considerable effort had to be made to access the problem data and tool output. This access is performed by keeping "syntax" descriptions of the output data structure in templates

Figure 1 MVS Dump Analyzer structure



and then matching those templates against the current text or printout. The result of this "reverse engineering" is put into structured fields that are then transformed into a format suitable for knowledge-based system processing.

Structure of the MVS Dump Analyzer. The considerations described in the previous subsection led to the global system structure shown in Figure 1. The dashed line represents the former man–machine interface for the human problem solver. During his or her analysis, the human expert uses the Interactive Problem Control System (IPCS)[2] to format and summarize various parts of a machine-readable dump. This process is interactive. The expert looks at some pieces of information, then draws conclusions and decides which piece of information is to be looked at next. The MVS Dump Analyzer proceeds in the same way and uses the same interface as represented by the dashed line. Its structure is as follows:

- The front end of the MVS Dump Analyzer consists of an EXEC (a program) written in REXX[3] and an input/output part. The MVS Dump Analyzer can be used interactively or in batch mode. It can be invoked with the front-end EXEC. The EXEC makes all of the necessary allocations, provides output data sets, and prepares the setup of batch jobs. The input/output part is written in KnowledgeTool™[4] and uses the Interactive System Productivity Facility (ISPF).[5] The flow of panels is controlled by rules.
- The heart of the system is the knowledge-based dump analysis part, written in KnowledgeTool. There are procedural parts and rule procedures. The rules are executed with forward chaining. The data are kept in structures called classes. Instances of the data, called class members or working memory elements, are described later in this paper.
- The data access part is written in REXX and supplies the information required by the KnowledgeTool rules from the MVS data sources.
- IPCS is part of the MVS base operating system. With IPCS, dump data (for example, trace, storage, control blocks, etc.) can be formatted and

summarized. The output is captured in temporary data sets which are processed with the help of "templates." A REXX template interpreter transforms the IPCS output into a structured data format. Then the data are turned into class members. These class members are directly used by the KnowledgeTool rules.

Modeling a software problem. The MVS Dump Analyzer handles many different types of software-error situations such as those listed earlier. In order to describe the variety of software problems, the MVS Dump Analyzer uses a very simple and general model shown in Figure 2. The parts of the model and what they do are now described.

The *requestor* is the culprit causing a problem. Information on the component, loadmodule, module, date, and service level is most important for a problem description.

The *server* usually appears as the module issuing, for example, an abend SVC. Since it is only detecting a bad situation and not causing it, it is of less interest than the requestor.

As a *resource* there usually appears storage, a module, a symbolic name, a data set, a device, and so on.

In the *context*, information concerning the address space and task is collected.

The *request* is some action against a resource, usually an SVC or another System/390™ instruction. Examples are a load SVC for a module, a branch into a module, or a getmain SVC for storage.

The *reply* is the response of the server to the request. If the server cannot satisfy the request, the reply describes the encountered problem, for example, an abend SVC, a program check, or a message.

Note that this model is recursive in the sense that if a server itself issues a request, it changes its role for that moment into the role of a requestor, being served by another server governing some other resource.

Figure 2 Model for software problem situations

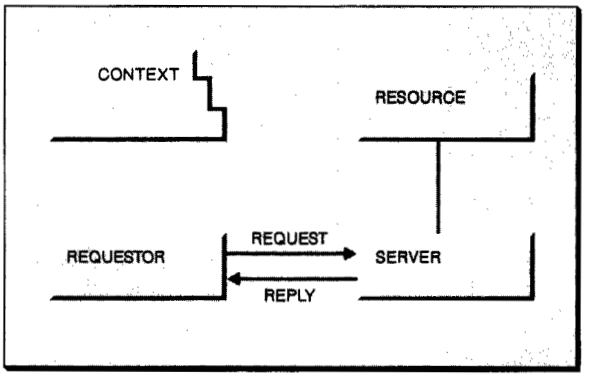
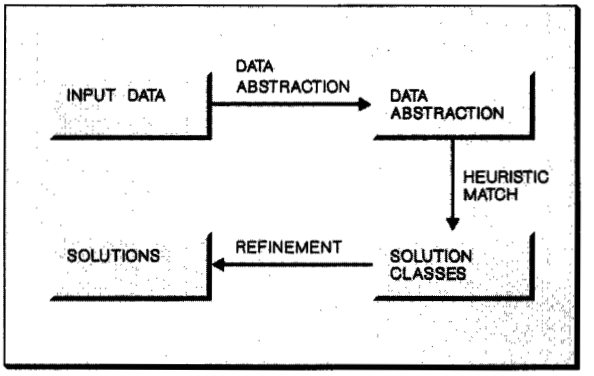


Figure 3 Heuristic classification

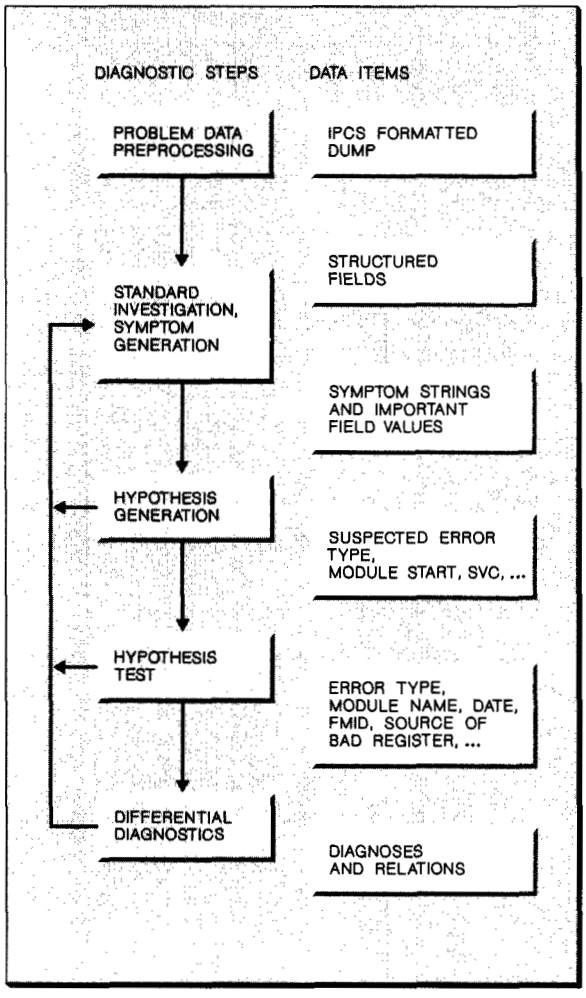


All problems handled by the MVS Dump Analyzer are structured according to the model for software problem situations. This setup is reflected in the data structures discussed later and also in the output of the system shown later in the sample session.

Diagnostic approach

Diagnosis is one of the classical application domains of knowledge-based system technology. Many of the diagnostic knowledge-based systems are based on an associative approach because a functional model suited for deep reasoning is either not available or too complex. A well-known associative approach is "heuristic classification."[6] The basic idea is depicted in Figure 3.[7]

Figure 4 Diagnostic approach



For more detailed investigations such as dump analysis, it is necessary to employ a more detailed approach. In References 8 and 9 the medical diagnostic shell MED2 is described. We have adapted some of the ideas developed there to our problem domain.

Figure 4 illustrates a coarse outline of the diagnostic approach. The right side shows in an exemplary way the data items produced by the diagnostic steps during dump analysis. Each diagnostic step is now described.

Problem data preprocessing—Before any rules can be applied, the problem data must be filtered and transformed. This action is necessary to reduce the tremendous amount of data contained in a dump. Only relevant information is passed to the MVS Dump Analyzer. Earlier in the paper we described how the data access part of the system obtains the problem data and makes the data available to the reasoning part in a structured form, which is then mapped to class members.

Standard investigation and symptom generation—In medical diagnosis a physician starts an examination with a standard set of questions. Analogous to this situation, the MVS Dump Analyzer investigates a standard set of items first. These items are, for example, traces and summaries provided by the tool used to read the dump. The goal of this investigation is to collect a set of indications for problem areas and the corresponding symptoms. The clues gathered in this step are the basis of subsequent detailed analysis.

Hypothesis generation—The heart of the diagnostic approach is to "hypothesize and test," where hypotheses are generated in a first step and tested in a second step. Hypotheses contain, for example, the suspected error type, the suspected request causing that error, and the suspected module containing the request of a potential problem. Especially important is a good guess for the start of the module, since in most cases a human-readable text at the beginning of a module, called the *eyecatcher*, contains valuable information.

The goal is to carry the analysis as far as possible toward determining the cause. If there is a cascade of problem situations, not only is the most recent one investigated, but the problems are tracked back as far as possible. For example, if there is an abend SVC and the situation causing that abend SVC can be identified, it is preferable to describe the latter situation rather than the abend SVC.

Hypothesis test—Each of the generated hypotheses is tested. Alternative values for attributes such as error type or module name are enumerated, then evaluated heuristically, and the most probable value is chosen. For each problem, sufficient relevant evidence must be found; otherwise the hypothesis is rejected. Performing the hypothesis test may lead to gathering more symptoms and also to creating new hypotheses.

Differential diagnostics—After testing the hypotheses, questions such as "Is one problem a follow-on problem of another?," "Are there two different versions of one problem situation described, and which should be merged?," and "Which problem is presumably the most important one?" are resolved in the differential diagnostics step. In general, this can result in a tree of problems with relations as arcs connecting the problems.

Implementation of the knowledge base

We implemented our knowledge base with KnowledgeTool, which is a language extension of PL/I. It offers the possibility of representing knowledge in production rules and supports the forward-chaining paradigm. It is a compiled language and uses the efficient Rete match algorithm as in OPS5.[10] Rule procedures and procedural code can be mixed. KnowledgeTool is very well-suited for implementing knowledge-based systems integrated into other environments, since KnowledgeTool has all of the connectivity offered by PL/I. See Reference 4 for a detailed description of the KnowledgeTool language.

Classes of data. In the MVS Dump Analyzer there are three major classes of data:

1. On the lowest level, we have the *dump symptom class*. Dump symptom class members represent single facts which are obtained from the problem data preprocessing step. Examples are a register value, a piece of memory, and the program status word (PSW) at the time of error. The dump symptom information may also be combined into groups of facts that belong together. Examples are a trace table entry or other summary information.

   Dump symptom class members contain data describing a specific field and its context. These data include information on the address space, task, job, control block, field name, and value.

2. The *problem symptom class* describes a suspicion of a possible problem. Since suspicions may arise from various sources, the symptom information must be consolidated to allow further common processing. A problem symptom is a working hypothesis. It triggers a detailed investigation. Either the possible problem is recognized as a real problem and results in a problem class member as described below, or it is not a problem situation or just another appearance of an existing problem, and the problem symptom class member is deleted.

   Problem symptom class members contain context information (as described above for dump symptom class members) and basic information belonging to the potential problem. This information includes a unique problem identifier, an address of the failing instruction, a reference to register values at the time of failure, load module information, chronological chaining of the problem symptom class members, and other more specific information.

3. The last class is the *problem class*. Each encountered problem is described in a problem class member. It contains all of the information that is displayed on the output screen at the end of the analysis. The information is structured according to the model for software problem situations as shown in Figure 2. Examples of the data kept in problem class members can be seen later in Figures 7, 8, and 9. Furthermore, problem class members contain information used to explain field values and internal information such as the unique problem identifier and chronological chaining of the problem class members.

Rule example. Figure 5 shows an example of a rule in the MVS Dump Analyzer. The `WHEN` part is the condition under which the rule will fire. If the rule fires, the action part of the rule, which consists of the statements between `BEGIN` and `END`, is executed.

This specific rule checks to see if there exists a dump symptom class member with an object name `tcbshsum`, a control block name `tcbshort`, and a field name `cmp`. If such a class member is available, the rule is fired: A `TCB summary` (task control block summary) class member is created, its fields are initialized, and the status is marked `unprocessed`.

The existence of the newly created `TCB summary` class member will then cause another rule to fire and start the investigation of the corresponding task control block.

Figure 5 Rule example

```
WHEN ( s1->dsymptom ( s1->object.name   = tcbshsum          ,
                      s1->cb.name       = tcbshort          ,
                      s1->field.name    = cmp               ))
BEGIN;
/* This rule creates a TCB class member for each TCB entry in the                */
/* 'TCB SUMMARY' from the IPCS command 'SUMMARY FORMAT TCBSUMMARY'                */
/* for the appropriate asid.                                                      */
  probcnt = probcnt + 1;
  Allocate dump_tcb SET(s9);                 /*Create a TCB summary class member */
    s9->tcb_num  = probcnt;                  /*Each TCB has a unique number      */
    s9->tcb_jbn  = s1->object.jobstep;
    s9->tcb_asid = SUBSTR(s1->object.aspaceid,5);
    s9->tcb_tadr = s1->cb.addr;
    s9->tcb_cmp  = s1->field.val;
    If match(s2->dsymptom (s2->object.name    = tcbshsum          ,
                           s2->cb.name        = tcbshort          ,
                           s2->field.name     = rtwa              ,
                           s2->cb.addr        = s1->cb.addr       ,
                           s2->object.aspaceid= s1->object.aspaceid))
     Then do;
       s9->tcb_rtwa = s2->field.val;
       free s2->dsymptom;
     End;
     Else
       s9->tcb_rtwa = h_0_8;
    s9->tcb_stat = st_t_unproc;              /*Mark it unprocessed               */
  free s1->dsymptom;
END;
```

Finding and interpreting module headers

Here we discuss the problem of how to identify patterns in the dumped storage. These patterns are module headers. In describing the hypothesis generation and test, we already mentioned that the module headers contain valuable information. They are readily identified by the human expert. Typically modules have at their beginning a so-called "eyecatcher," a comment containing the name of the module, the date, the service level, and possibly other information in human-readable form. There is no common structure for these module headers and the code surrounding them, but after some practice a human can identify the beginning of the module and the header information. Identifying the module start address and header information of the failing module is especially important, since the offset of the failing instruction in the module, module name, date, and service level is key information for describing or identifying a problem.

Usually a guess is made about the start address of the module obtained by a heuristic analysis of register contents. Some typical elements of a module header such as a date are very easy to identify. Other items are not as distinctive, for example, the name. In a first approach we coded a set of rules that worked on the hexadecimal representation of storage and tried to decide whether a header was there by identifying sufficient plausible parts of the header. It turned out that taking the raw hex dump as input for this kind of investigation worked, but the reliability of the identification was limited. New cases were always coming up that were not identified correctly. These cases then had to be covered by additional rules.

An alternate approach was to modify the coding of the input data. We built a rule-based tokenizer, which splits up the investigated storage area into tokens such as "name text," "long text," "date," "store multiple," and "other hex code." As an

experiment we fed these token strings into a neural network (running on a personal computer under DOS). The network was a three-layer perceptron with sigmoidal nonlinearities.[11] It had five inputs and was trained with back propagation to identify the occurrence of a module name as the middle token of the inputs. It turned out that the network was able to identify all of the names in the training and in the test set. Thus this kind of pattern recognition can be solved with neural network technology.

As an alternative to the neural network approach, we developed a rule procedure working on the token strings that separates relevant information from irrelevant information. It turned out that this rule procedure worked more reliably than the first approach. The major reason was that many disturbing side effects, which would have required special handling in the original approach, were filtered away by the tokenization step. For the module header recognition and analysis, the key question was in which way the input must be coded to get optimal results. Here token strings turned out to be the best way to code the input. Attaining the optimal coding may require intensive preprocessing of the input data.

Sample session

This section illustrates how the MVS Dump Analyzer appears to the user. The dump used in this example shows how an analysis by the system approaches the cause of a problem. The dump must be available in machine-readable form. The MVS Dump Analyzer can be invoked from any ISPF panel. Then an input screen appears (see Figure 6) telling the user what information to provide:

1. In the *interactive mode*, the user wants the result of the analysis to be displayed on the screen. Then the user can get explanations on field values or go into an IPCS session to do further work on the dump. In *batch mode*, a job will be started in the background, and the result will be returned in a specified data set.
2. If the dump was recently obtained on the current system, the MVS Dump Analyzer can take advantage of information available in memory. This applies, for example, to shared memory containing the nucleus or link pack area. These storage areas are often not available in the dump, but the information in active memory may still be valid for the analysis. If the answer to this question is yes, this information will be used by the MVS Dump Analyzer.
3. In interactive mode the name of the dump data set must be entered. In batch mode there may be a whole list of dumps; therefore, an ISPF editor session is entered, where the dump names can be specified in a data set. This is taken as input in batch mode.

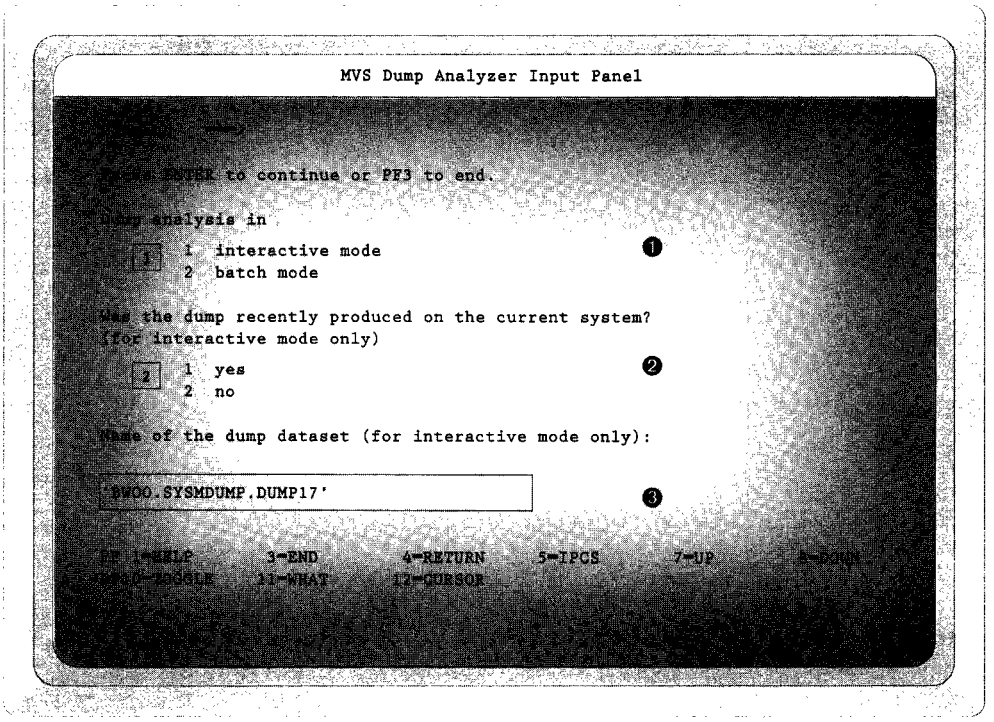
Between the input screen and the final diagnosis, the user need not answer additional questions. All needed information is retrieved automatically. Depending on the complexity of the problem, about 10 to 50 IPCS commands are issued during the analysis, and about 300 to 2000 rules are fired. The system starts with a standard investigation of some summaries. Then it generates some hypotheses and looks into memory areas and control blocks.

After some minutes, depending on the dump size and the computing power of the local CPU, the MVS Dump Analyzer presents the result of the diagnosis. In both batch and interactive mode the result is written to a data set. For the interactive mode the following screens are displayed. First, a "primary diagnosis" screen is shown. It gives global information about the analysis that will follow, as there is dump-related information such as name, title, date, time, type, the number of problems that were found, how many different address spaces and tasks are involved, and which primary symptoms the MVS operating system has assigned to this problem.

In this dump example, the MVS Dump Analyzer finds three problem situations. On the succeeding panels each problem is presented according to the model for software problem situations as shown in Figure 2. Two different output formats are supported. For the novice user there is a *descriptive form*, where a problem is presented on two screens. For the experienced user a problem is presented in *condensed form* on one screen. Problems appear in time-reversed order so that the latest problem is displayed first. Problems displayed next are problems preceding the last.

Figure 7 shows the most recent problem: The requestor is the loadmodule IKJEFT04. It was built on day 012 in 1988 and has a maintenance level (PTF level) of UY17336. At the address 01DA774C, the requestor IKJEFT04 issued a request, which is the System/390 instruction "Test under Mask"

Figure 6 MVS Dump Analyzer input screen



(TM H002(GR5),H01). The request ended with a SYSTEM ABEND 0C4 and reason code 0010, which is a segment-translation exception. It means that the knowledge-based system found the System/390 instruction tried to access a storage segment that does not exist. Indeed, the requested resource `storage` is shown to be `nonexistant`. It was referenced by general-purpose register 5 (GR5), which contained a bad address (040C0002). The context information finally tells us that the problem occurred under the user identification (userid) BWOO in the address space 050 within the task (TCB) 009D1B20.

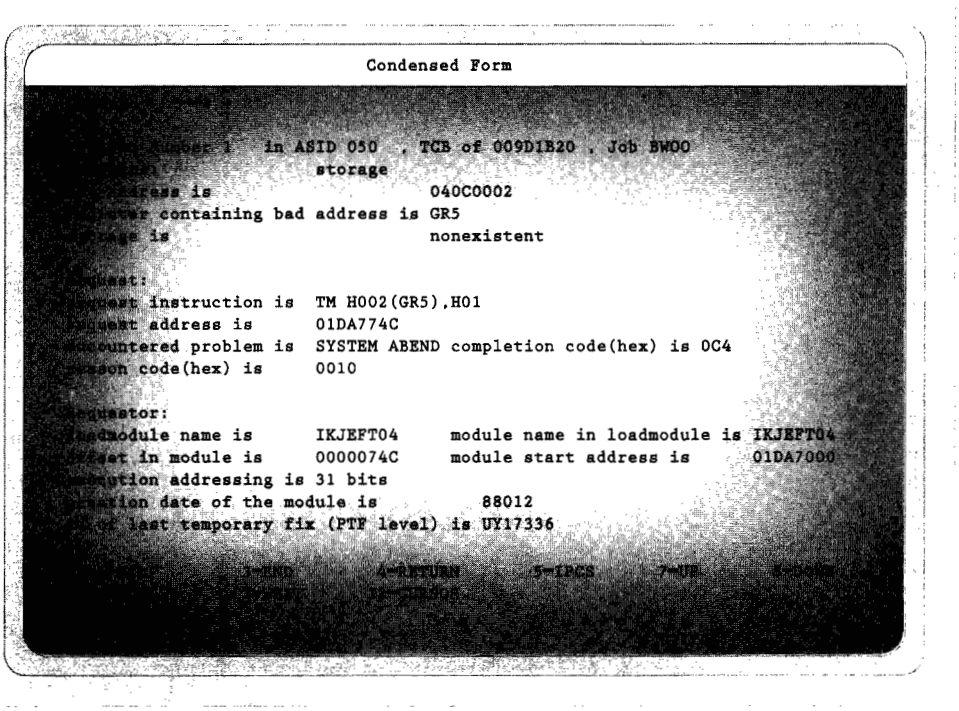
The SYSTEM ABEND 0C4 problem was a follow-on problem of the event described in Figure 8: Here a SYSTEM ABEND 0C1 happened, which is an operation exception. An operation exception occurs when the processor tries to execute an invalid instruction. The requested action was a `branch`. The resource is a `module` to which an attempt was made to give control. However, the branch address 00000050 in general-purpose register 15 (GRF) is invalid. The difficult part here is to find out where the branch came from. But the requestor was identified to be a loadmodule named CONTROL; it branched to the bad address.

There is another problem that caused all the trouble (see Figure 9). It triggered the other follow-on problems. Here the requestor CONTROL issued a request LOAD that ended in a SYSTEM ABEND with completion code 806. The message manual[12] tells us that the LOAD went wrong because of an invalid parameter. And indeed, the name of the desired resource is mutilated; it appears as . . CONTRO. The fact that no module with that bad name could be loaded into memory (problem 3) led to the bad branch and SYSTEM ABEND 0C1 (problem 2). The recovery of problem 2 finally led to problem 1.

The user can "toggle" with function key 10 (PF10) between the condensed form (as shown in the examples) and the descriptive form, which presents the same information fields in a more textual form.

Besides this descriptive form, novice users can use the explanation facility. For every output

Figure 7 MVS Dump Analyzer result: Problem 1



value there is an explanatory text. It can be viewed by putting the cursor on the value field and pressing WHAT (PF11). A panel is then shown containing a description of the theoretical background of the field. Furthermore, it contains the context-specific reason for the field value. This reason is either a causal relationship or information on where the value was retrieved.

Project history

We started working on the MVS Dump Analyzer in April 1988. Considerable time was spent to carefully position the project. Knowledge acquisition started in July and coding in September 1988. A first prototype was finished in April 1989. The prototype used the Expert System Environment product as front end and covered only parts of the functions, but we obtained the proof that the concept worked. During the rest of 1989 we had several test installations available for prototype evaluation. Work was resumed on the MVS Dump Analyzer in 1990. An ISPF front end and batch mode were provided, the performance enhanced, and the functions improved to cover a larger variety of problem types.

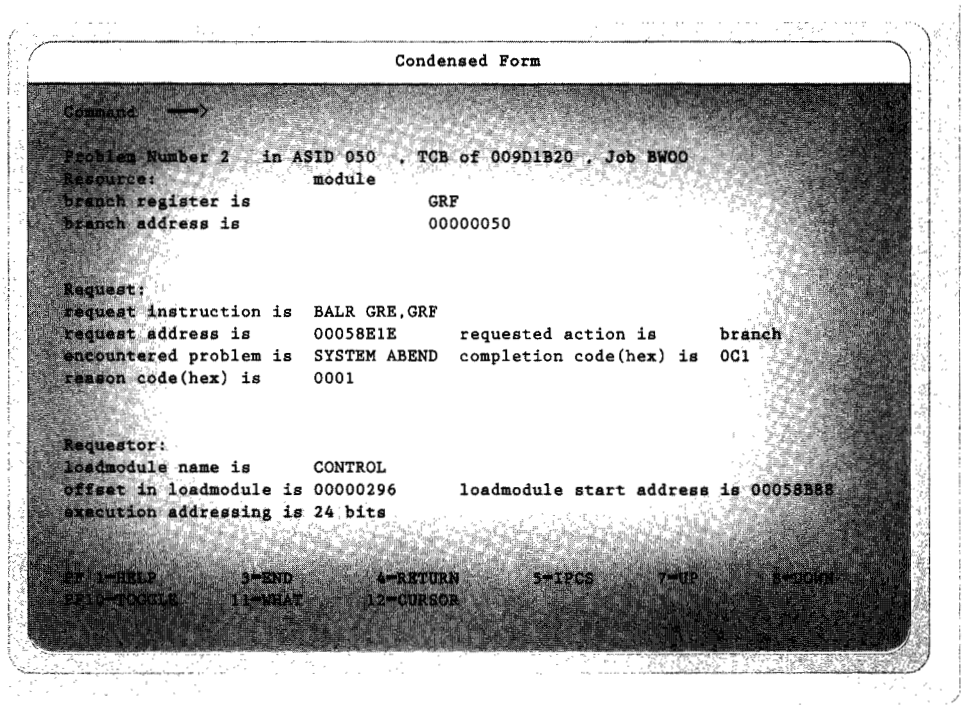
Two to three developers have been working on the system, two of them concerned with knowledge engineering. One expert helped on dump analysis for about four months in total; another expert helped with MVS problem-solving knowledge.

The current version has 5000 instructions of procedural PL/I code, 5000 instructions of KnowledgeTool code distributed among 350 rules, and 7000 lines of REXX code. Note that more than 50 percent of the code produced is conventional. It covers mainly the problem data preprocessing part. For integrated solutions a fair amount of traditional coding is necessary.

Evaluation

The prototype has been evaluated with hundreds of dumps. More than 90 percent of the not-component-specific dumps are analyzed successfully. The success ratio for component-specific dumps is lower because they often contain only very special and limited information, which cannot be analyzed by an approach based on general MVS

Figure 8 MVS Dump Analyzer result: Problem 2



knowledge. The human expert calls for the component specialist in these cases.

According to one of our experts, the experienced problem solver saves about 30 minutes of time per dump. Often the problem description resulting from the analysis is sufficient to identify a known problem immediately. In such cases, the problem solvers need not look into the dump at all.

Conclusion

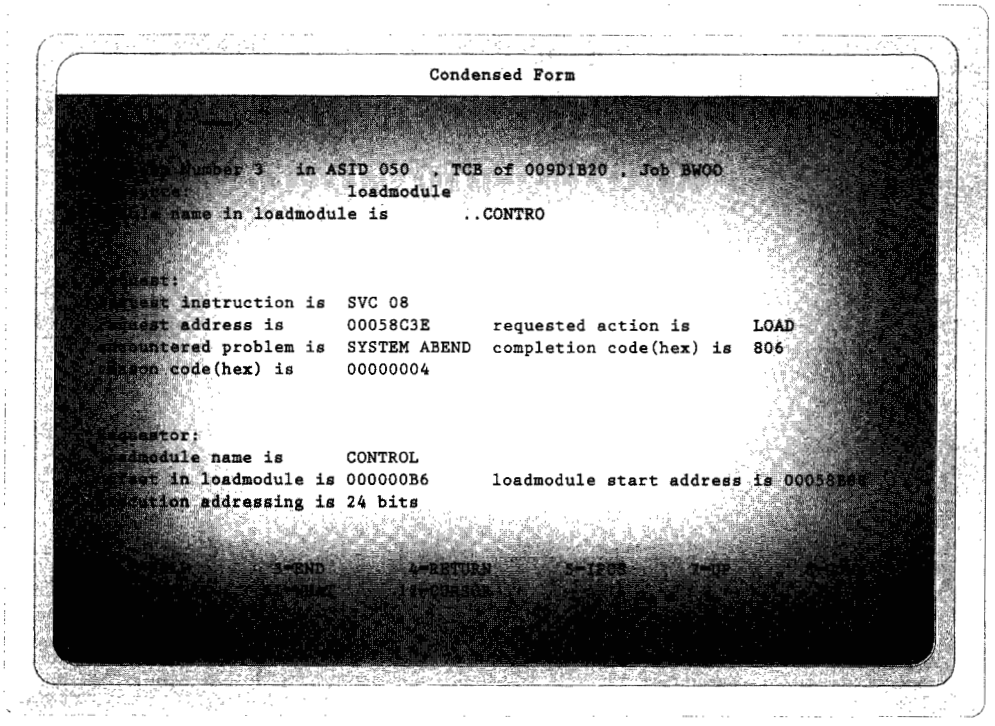
Knowledge acquisition is probably the most difficult part in the development of the system. It is time-consuming and hard to plan. It is not easy to get an expert's time and commitment for cooperation. Experts are not all alike: some are articulate; some are more example-driven. We had the best experiences when iterating many examples. But this method works only if the expert supports decisions on the general approach after having worked on the examples. If the expert claims that there are always exceptions, the example-driven approach does not lead to the goal.

The programming productivity for the 10 000 instructions of KnowledgeTool code was 310 instructions per person-month. This productivity number includes the ongoing knowledge engineering effort and various other development activities, but not the data access part. Since the total KnowledgeTool code consists of procedural parts and rule parts of about equal size, a developer produced about 150 PL/I instructions and 10 KnowledgeTool rules per month.

From our experience, a rule-based approach improves programming productivity overall as well as for the knowledge-based parts. For example, control of the user dialog can be implemented very efficiently with rules. The effort spent for coding the procedural parts of a knowledge-based system is often underestimated. A further difficulty is getting sufficient test data and testing the system. Tests for efficiency or optimality may be necessary.

In order to get user acceptance, a careful positioning of the application and some user education is necessary. A number of common misun-

Figure 9 MVS Dump Analyzer result: Problem 3



derstandings have to be overcome. For example, there is the expectation that an "expert" system especially helps in solving the most difficult problems. Users must have a realistic view of what a knowledge-based system can do for them. Routine problems can be solved automatically with a knowledge-based system, thus giving the expert the time to concentrate on the difficult problems.

Novice users typically are more willing to accept a new technology. Since heuristics may not work correctly in *all* cases, there is no guarantee that every single output field value must be correct. In rare cases it can happen that a field contains a false value. Users must be educated to understand that the result of a problem analysis has to be treated as a whole set of information. In most cases the result is still a valuable description of a problem, even if one of the field values is obviously false.

Experienced problem solvers are proud of their capabilities and tend to claim that they are better than the "expert" system. Therefore, they do not immediately see the value of such a system. The MVS Dump Analyzer can relieve them from routine tasks and allow them to analyze dumps that otherwise would not have been looked at because of a lack of time. The performance of a new and often complex application can be a further problem. Offering an integrated and easy-to-use solution is the key to success.

Acknowledgment

Many people have contributed to the success of the MVS Dump Analyzer project. We would especially like to acknowledge the cooperation of our experts, J. Antholz and B. Pierce. Furthermore, H. Bublitz, V. Schoelles, C. Mueller, and J. Woods have contributed substantially to the system.

Cited references

1. O. E. Laske, "A Three-Phase Approach to Building Knowledge-Based Systems," *CCAI* (Communication and

Cognition) **5.2**, Babbage Institute for Knowledge and Information Technology, Ghent, Belgium (1988), pp. 19–30.
2. *MVS/ESA Interactive Problem Control System (IPCS) User's Guide*, GC28-1833, IBM Corporation (1988); available through IBM branch offices.
3. *TSO Extensions Version 2, REXX User's Guide*, SC28-1882, IBM Corpkeporation (1988); available through IBM branch offices.
4. *KnowledgeTool Application Development Guide*, SH20-9262, IBM Corporation (1989); available through IBM branch offices.
5. *Interactive System Productivity Facility, Version 2 Release 3, Dialog Management Guide, MVS*, SC34-4112, IBM Corporation (1987); available through IBM branch offices.
6. W. J. Clancey, "Heuristic Classification," *Artificial Intelligence* **27**, 289–350 (1985).
7. T. F. Thompson, W. J. Clancey, *Applying a Qualitative Modeling Shell to Process Diagnosis: The Caster System*, Department of Computer Science, Report No. STAN-CS-87-1169, Stanford University, Stanford, CA (1986).
8. F. Puppe, *Diagnostisches Problemlösen mit Expertensystemen*, Springer, Berlin (1987).
9. F. Puppe, "Diagnostik—Expertensysteme," *Informatik Spektrum* **10**, Springer, Berlin (1987), pp. 293–308.
10. L. Brownston, R. Farrell, E. Kant, N. Martin, *Programming Expert Systems in OPS5, An Introduction to Rule-Based Programming*, Addison-Wesley Publishing Co., Reading, MA (1985).
11. R. P. Lippmann, "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine* (April 1987).
12. *MVS/XA Message Library: System Codes*, GC28-1157, IBM Corporation (1987); available through IBM branch offices.

Norbert G. Lenz *IBM Germany, P.O. Box 1380, 7030 Boeblingen 1, Germany*. Dr. Lenz is a staff programmer in the Advanced Software Development department of the IBM laboratories in Boeblingen. Before working on the MVS Dump Analyzer knowledge-based system he was concerned with knowledge-based system technology transfer and with the development of advanced operating system structures. He received his doctor's degree in mathematics at the University of Mainz in 1982 and joined IBM in the same year.

Serge F. L. Saelens *IBM Germany, P.O. Box 1380, 7030 Boeblingen 1, Germany*. Mr. Saelens is a staff programmer in the IBM laboratories in Boeblingen. Since 1988 he has worked in the Advanced Software Development department on the MVS Dump Analyzer knowledge-based system. He joined IBM in 1983 and worked until 1988 in the European Banking Systems organization. Mr. Saelens obtained his "Burgerlijk Elektrotechnisch Ingenieur" diploma at the University of Ghent, Belgium, in 1982.

Reprint Order No. G321-5439.