TPF Dump Analyzer:

A System to Provide Expert Assistance to Analysts in Solving Run-Time Program Exceptions by Deriving Program Intention from a TPF Assembly Language Program

R. Greg Arbon, Laurie Atkinson, James Chen, and Chris A. Guida, Covia Technologies

The TPF dump analyzer (TDA) was conceived in an effort to create an intelligent programming assistant for the transaction-processing facility (TPF) programming environment where IBM System/370 assembly language is used. This particular program represents the first component of the system, which provides expert advice in the domain of solving run-time control dumps (software exceptions) in the TPF environment. This program is used by the application development, run-time coverage, and stability staff members at Covia Technologies to provide more rapid problem diagnosis and resolution to a set of common TPF programming errors. The system has an installed base of nearly 750 users and has proven to be a useful tool in diagnosing commonly occurring errors in the TPF environment.

Covia operates and maintains the APOLLO computer reservation system (CRS) for United Airlines. The APOLLO reservation system, the

world's largest airline computer facility, supports over 60,000 terminals in 45 countries, which generate message rates of as much as 1700 messages per second. APOLLO uses IBM's TPF as the operating system (IBM, 1987) that executes thousands of programs written in IBM 370 assembly language. A staff of approximately 750 programmers write and maintain the reservation system software. The software supports functions such as searches for the lowest fare, airline services, hotel services, car services, airport check in, and the tracking of lost baggage. TDA was developed by Covia to aid in solving run-time control dumps in this environment. To better describe the function of TDA, a brief description of the TPF operating system is necessary.

TPF is used by data processing environments requiring remote access to a large common database, such as airline reservation systems, banking systems, and insurance companies. The units of work in a TPF system are called *entries* and are initiated by commands made by a user such as a travel agent. A typical entry flows through the system in the following way: After the user inputs a command and hits the enter key, the TPF scheduler or control program is ready to process the message. The control program reads the command and determines which set of programs is required to process it. The correct programs are moved from disk to main memory, and a block of storage called the entry control block (ECB) is initialized. (ECB is the primary control medium for an entry in the TPF system and is used by the application programs until processing of the entry is completed.) The execution of the application program then begins. Based on the contents of the input message, control is transferred from one program to another until an output message is formatted and returned to the initiating user's computer terminal.

Problem Domain

When a software exception occurs on the APOLLO system, a program interrupt is generated, and sections of memory are written to tape. This information is postprocessed into a readable format called a *dump* and is generally sent to a programmer for analysis. Hundreds of different types of dumps can occur in TPF, with countless variations of each different type. To identify the problem that is the root cause of the dump, the programmer uses dump-solving strategies and debugging techniques. Experienced programmers can solve a typical dump in minutes, but a novice programmer can require days to solve the same problem. Also, a novice programmer might require assistance from a senior programmer to determine the proper strategy for analyzing the

dump.

The objective of TDA was to develop an intelligent application that could examine a dump, diagnose the error, and recommend a correction. TDA is used by programmers and coverage and stability staff members to reduce the time required to solve a common set of problems. TDA reduces the average time required by analysts to solve these problems and increases the reliability of TPF software testing. These types of systems are thoroughly discussed in Rich and Shrobe (1978), Waters (1982), Green et al. (1983), and many others. TDA is an implementation of a system that is based on these early concepts. TDA is part of a larger ongoing effort to continuously improve the quality of Covia's product and productivity.

The rationale for applying AI to this problem domain was based on previous experience with other applications and on knowledge of the current state of the technology. Previous attempts to create analysis tools were stymied by the difficulty of maintaining complicated procedural code; the lack of necessary skills required to build sophisticated AI programs; and the cost and complexity of the available hardware, languages, and development shells used to produce AI solutions.

Many of the previous analysis tools stopped short of performing any analysis and were actually data-manipulation tools that massaged and translated information into a more useful format for the human analyst. TDA uses some of these existing tools and then proceeds to apply AI to perform intelligent problem analysis.

Application Description

The architecture of TDA is innovative in that it uses a hybrid approach, mixing evidential forward chaining, model-based reasoning, and focused opportunistic search.

TDA Architecture

The architecture of TDA, shown in figure 1, consists of three distinct components: information-gathering utilities, assembly program reconstruction, and problem diagnosis. The *utilities component* reads the dump file and instantiates objects defined within the class hierarchy for use by the diagnostic component. The *assembly program reconstruction component* takes the program from the utilities component in the form of a flat set of hex data and constructs a model of a System/370 assembly code listing. This model contains all the assembly instructions as well as flow relations between different sections of code (we describe it in more detail later). The *diagnostic component*, which includes forward-

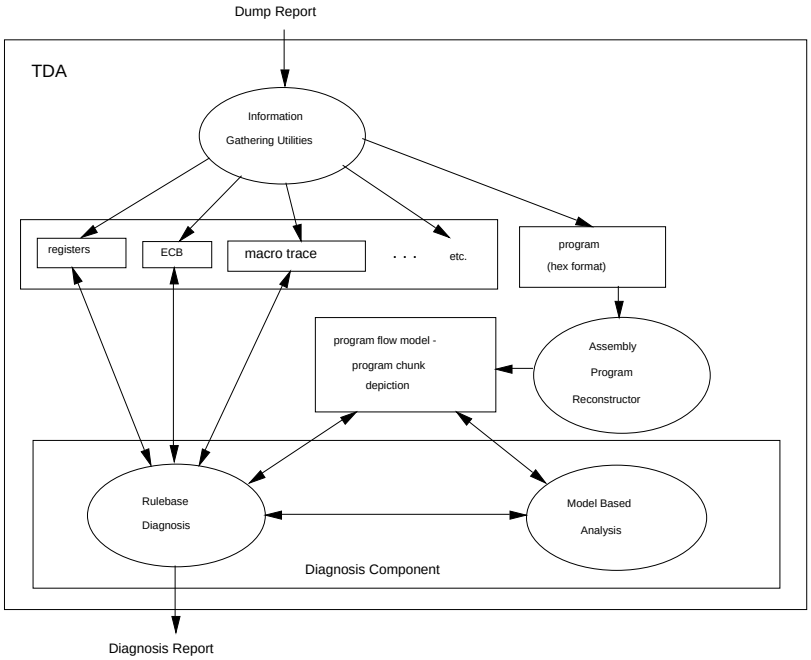

Figure 1. High-Level Architecture of TDA.

chaining rules and model-based reasoning, then analyzes the various data, as well as the program model, to identify the root cause of the problem and provide a recommendation for a solution. A report, consisting of relevant parts of the data, organized by the utilities component and the diagnosis component, is then presented to the user. The intelligent components of TDA include the reconstruction of the program and the execution of the diagnosis component.

Information-Gathering Utilities

TDA was designed and constructed using an object-oriented approach to organize the data given in the dump. The file containing the dump is read once, and each part of the dump that is of use to TDA is instantiated as an object. Specifically, information from the dump that TDA uses includes the following:

General registers: The *general registers* contain the values of the 16- to 32-bit general registers at the time the dump occurred. These registers can be used for base addresses, indexes, or accumulators.

Control registers: Sixteen 32-bit *control registers* are available to the

operating system but not the application programs. TDA uses these registers to determine which system functions were active at the time of the dump.

Program status word: The *program status word* **(**PSW) contains information about the status of the program currently being executed. It includes the instruction address, condition code, and other information used to control instruction sequencing and determine the state of the central processing unit. The PSW is used by TDA to determine the failed instruction.

Storage protection keys: A *storage protection key* is associated with each 4K block of memory. A store instruction is permitted only when the program-access key matches the memory storage key. A protection exception occurs when this action is attempted, and the keys do not match. The dump contains a partial listing of the storage keys associated with each 4K block.

Macro trace: A *macro trace* is a list of the last 250 macro calls executed by the system. This list includes any macro call executed by any program (TPF is a multiprocessor system). TDA extracts the macro calls relevant to the current problem being analyzed. This listing of macro calls is also provided to the user in the output report.

Core blocks (levels): *Core blocks* are data within the blocks of memory currently being accessed by this entry.

Entry control block: The *entry control block* (ECB) is the primary control medium for an entry into APOLLO. One ECB is assigned to each entry into the APOLLO system and represents the entry while in the system. The ECB contains such items as register save areas, error indicators, information regarding related core and file locations, program enter and return addresses, and work areas.

Program: *Program* refers to the program in which the entry failed in hexadecimal format. This information provides the basis for the program flow model, which is described later.

Assembly Program Reconstructor

The reconstruction of the assembly language program consists of stepping through the file of hex data and extracting each instruction. As this extraction is performed, all branch instructions are identified to determine the addresses of labels that exist within the program. These addresses then enable TDA to build the labels that delineate the program into program chunks. TDA must also determine the length of macro calls within the program that are of unknown length. This determination is accomplished by constructing different potential macro call lengths and checking for valid instructions based on these lengths.

```
YCD2000     EQU     *
            L       R5,CE1CR8
            AH      R5,=H'20'
            LH      R6,0(R5)
YCD2100     EQU     *
            CLC     46(R2),=C'DM'
            BE      YCD2200
            TM      6(R5),X'04'
            BO      YCD2000
            CLC     4(R3),=X'FFFF'
            BNE     YCD2300
YCD2200     EQU     *
            MVC     0(9,R5),3(R3)
            LA      R5,9(R5)
            B       YCD2400
YCD2300     EQU     *
            SH      R4,=H'1'
YCD2400     EQU     *
            LA      R3,9(R3)
            BCT     R6,YCD2100
            BACKC
```

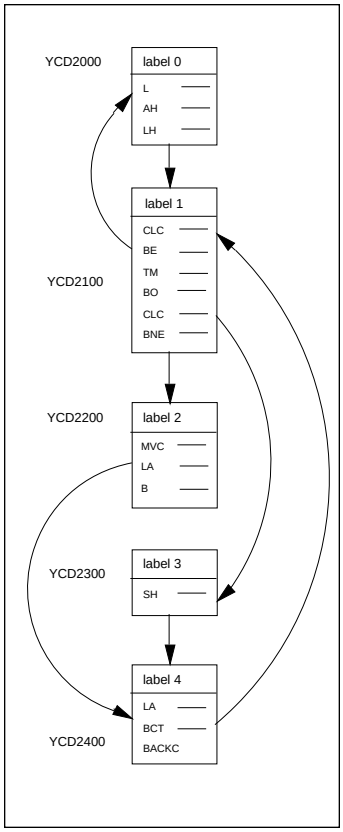

Figure 2. Section of an Assembly Program and the Associated Program Chunks.

Known locations of labels further ahead in the listing are used to ensure that no instructions overlap the address of any label.

An example of the translation of an assembly language program into a set of program chunks is shown in figure 2. The labels seen in the assembly version do not exist in the hex representation of the data because references to them are replaced during compilation with specific addresses. A typical program contains approximately 1000 instructions, which are translated by TDA into approximately 250 program chunks.

By constructing these program chunks, the program can be represented by TDA as a directed cyclic graph (Aho, Sethi, and Ullman 1986; Pearl, Verma 1988), with the program chunks as the nodes and the flow relations (to-from relations) as the arcs. Each program chunk con-

sists of the instructions contained in the particular section of code and flow pointers indicating possible paths of execution both to and from the current chunk. The combination of these chunks forms the set of possible logical flows of the program. It is the model that the reasoning system uses when identifying possible paths of program execution to isolate the root cause of a dump. TDA begins by examining the program chunk that contains the failed instruction. The search space can then be expanded by following possible paths of program execution.

Problem Diagnosis

The diagnosis component of TDA uses a combination of reasoning techniques to determine the solution to the current problem, including evidential forward chaining, model-based reasoning, and focused opportunistic search. It begins with the use of a set of forward-chaining rules that identify evidence of interesting situations that might explain the problem. Sometimes, the solution is determined simply by firing this rule set. Other times, however, these interesting situations require more sophisticated reasoning techniques. For such cases, a search is begun to ascertain further evidence of a problem type by perusing the flow model of the program to follow possible paths of execution. Sometimes TDA identifies a situation where previous assumptions need to be changed and the analysis restarted.

Different sequences of instruction types indicate different types of problems. The objective of the model-based reasoning component of TDA is to step through the program model (represented by the interrelated program chunks) to identify the interesting instructions that identify a problem type. This reasoning is guided by the values of the registers at the time the dump occurred and utilizes the concept of focusing on probable diagnoses to limit the potential scope of the search.

For example, a possible dump type is a *protection exception*. This dump occurs when a program attempts to access an area of memory that the program does not have the authority to access. That is, the storage key for the block of memory does not match the key associated with the application program. One of the problem types that would cause a protection exception is the presence of a loop that contains an increment of a base register combined with a loop that is executed too many times. Once this loop is exited, the base register addresses a new block of memory with a different storage protect key. Thus, when the base register is used in a subsequent instruction and is addressing an area of memory with a different protect key, a protection exception dump occurs. In this case, TDA detects the presence of the loop, finds the in-

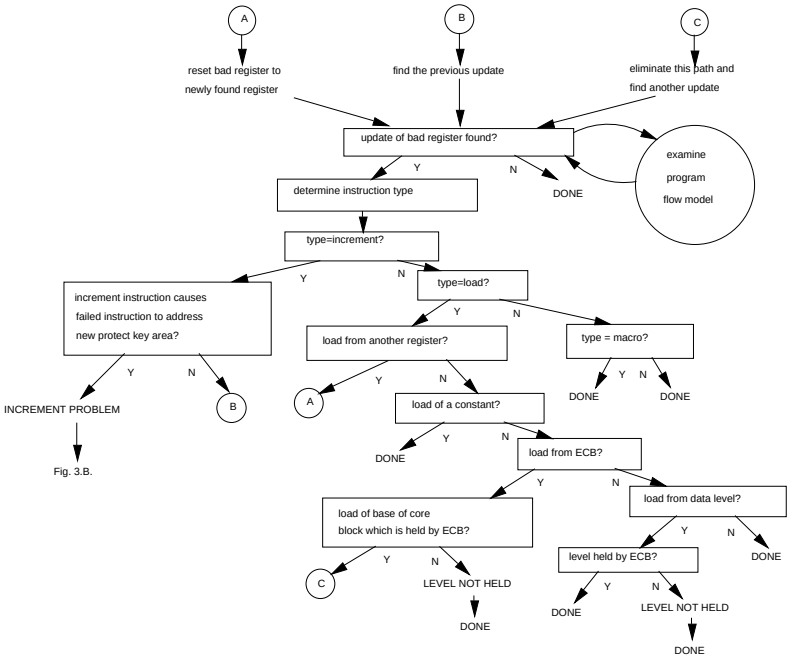

Figure 3a. Decision Tree Used When a Register Contains a Bad Value.

struction that increments the base register, and determines the reason that the loop was executed too many times.

Depending on the type of programming construct for which TDA is searching, different opportunistic search strategies are used that focus on the most probable diagnosis first, as described in de Kleer (1991). For example, when searching for the update of a register that seems to contain a bad value, TDA steps back through the previous program chunks, scanning for the use of this bad register. The search space is then expanded to include each chunk that references the bad register in order of proximity to the current chunk and continues in a breadth-first search manner.

However, when searching for a looping construct, TDA scans forward through the program chunks looking for the presence of a connection that creates a loop. Although it is impossible to determine if a program terminates (the halting problem [Harrison 1978]), it is possible to look for localized sequences of instructions that provide evidence of an incorrect looping construct (for example, decrementing a counter from zero

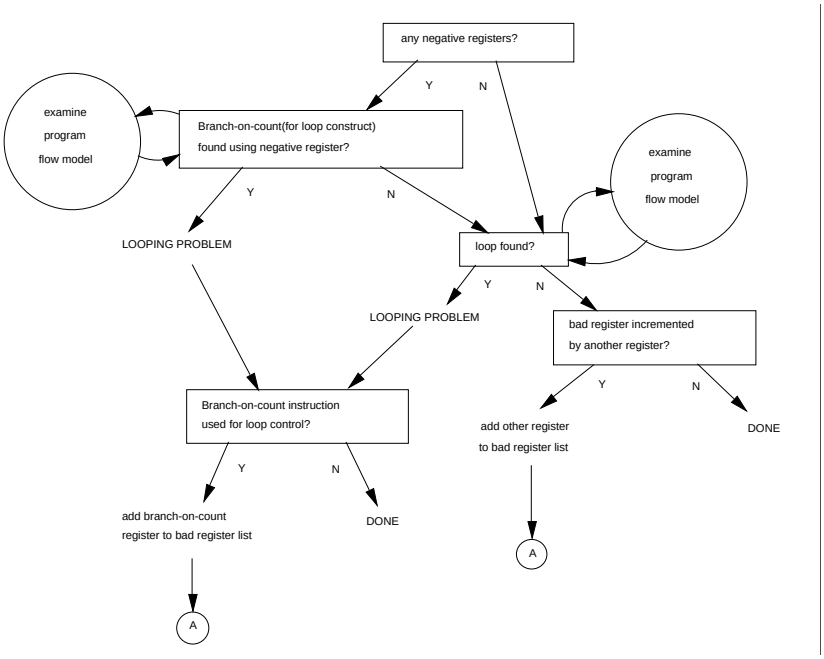

Figure 3b. Decision Tree Used When a Register Contains a Bad Value.

and checking for zero to terminate or not initializing a register correctly.)

Different types of program dumps are distinguished by different flavors of problems. This distinction between dump types allows for different search methodologies when deriving a diagnosis of the particular problem.

These methods include both the selection of where to search and the choice of what parts of the search space to prune. For example, when searching for the solution to a protection exception dump and determining the path that the program was executing before the dump occurred, a path is eliminated for two reasons: (1) a previous instruction is found that would have caused a protection exception before executing the failed instruction and (2) a load of a valid value to a register known to contain a bad value is found.

An example of the reasoning logic for a particular problem is given in figure 3a. If TDA has determined that a dump occurred because a register was being assigned a bad value, then this section of the rule base is used. First, the model-based reasoning component is invoked to find the instruction that last updated this bad register. If such an instruction is found, then the forward-chaining can continue, and the type of the update instruction (that is, increment, load) determines

which set of rules fires next. If, for example, this instruction increments the bad register, then TDA attempts to confirm that the increment instruction caused the register to contain a bad value. If an increment problem is detected, then TDA continues to look for a looping problem, as illustrated in figure 3b. During this diagnosis, TDA might alter the target of its search. For example, if an instruction is found that loads the bad register with a second register (as shown in figure 3a), TDA begins searching for the update of this second register. Another example of TDA altering the target of its search is given in figure 3b, where a looping problem has been detected, and a branch-on-count instruction is being used to control the loop. That is, a register is decremented and tested for zero on each iteration through the loop. In this case, TDA updates the target register to be this loop control register and begins the analysis again. This capability allows TDA to find a root cause of a problem rather than simply the most recent symptom of the problem.

The forward-chaining logic, as well as the methods used to search the program model, were derived from the TDA Expert Group. This group consisted of a select group of TPF analysts considered to be experts in solving dumps or persons with a broad range of knowledge about the TPF system.

During the problem diagnosis, as facts are determined relating to the problem being solved, the report is updated. These facts include such results as there is a looping problem, or register 2 contains a bad value, and it was updated by register 4 at displacement 100 in the program. Other useful information is added to the report. This information includes such items as the instruction that was being executed when the dump occurred, registers that contain bad (or potentially bad) values, and an ordered list of macros executed by this entry. TDA constructs these items through straightforward operations on data in the dump so that the programmer does not have to spend time with these mundane tasks. The programmer is thus freed to work on a more sophisticated analysis of the problem, and the potential for computational errors is eliminated.

Implementation and Development Issues

The actual implementation of the dump analyzer was preceded by a number of analyses to determine where the use of AI could provide a high return on investment. These analyses included a study to determine which of the problems that were being solved by the analysis groups could be automated and solved by an intelligent, online soft-

ware system.

Many problems can occur within the System/370 architecture. Criteria for the inclusion of a problem in the analysis capabilities of TDA encompassed problem complexity, problem frequency, availability of data to determine a solution, and computational complexity of the determination of a solution. With these criteria in mind, a group of experts was selected that represented the diverse application, operations, and support groups within the company. These persons determined which dump types should be addressed and in what order. The overriding criteria for selecting the first dump type were high problem frequency and ease of determining a solution.

Another factor in the design of the system was the impact on the company computer network that would be caused by transferring the data required to solve the problem to TDA and then back to the analyst. This factor led to the decisions to process only minidumps, which are limited to approximately 200K (A full TPF dump on the System/370 can be as large as 32 megabytes.), and to deploy TDA on a mainframe computer using the MVS operating system as opposed to each programmer's workstation (thus maintaining the volume of traffic on the local area network [LAN] at current levels and avoiding potential LAN performance problems).

The development team that constructed TDA consisted of three people: a project lead; and two developers, one full time and the other part time. The development team worked with a group of TPF experts to construct and validate the knowledge base of TDA.

The development effort used a prototyping approach that allowed for early validation of the system requirements and functions and provided the opportunity to include or remove features from the system. The prototypes were continually extended, leading to a final installed system that met the system requirements. The expert group assisted in the validation of each prototype. This process ensured that the analyses that TDA performed were similar to what an expert would do and, most importantly, that the analyses were correct.

The total development time from inception to system installation was 8 months and took approximately 2800 hours.

Application Deployment

Deploying TDA for use by the user community was discussed with the expert group to determine the various impacts associated with bringing a new utility into the analysis environment. It was agreed that among the most important considerations was to minimize the impact to both the

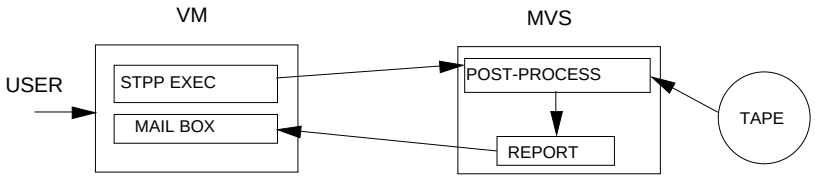

Figure 4. Previous Dump Postprocessing Architecture.

existing dump postprocessing utilities and the user community itself. With this goal in mind, an architecture for installation was derived that would simply intercept the dumps as they were processed.

Previous Dump Architecture

As illustrated in figure 4, the previous dump process involved a user request through the STPP program (run under the VM/CMS operating system), which invokes the MVS dump postprocessor. On completion of the dump processing, the output report (minidump) is sent to the user's mailbox or an output device defined by the user.

New Dump Process, Including TDA

The current process for TDA (figure 5) uses the previous postprocessing architecture. The output report (minidump) from the MVS dump postprocessor is provided to TDA as input. The report is then analyzed, and a TDA analysis report is attached to the beginning of the minidump. This final report is then sent to the user's mailbox or an output device defined by the user.

Delivery Benefits

Many benefits have been derived from this development effort, in problem-solving methods as well as increased programmer productivity.

During the course of the knowledge-acquisition process, experts from diverse application, operation, and support groups were gathered to discuss problem-solving methodologies. This process was educational to all the participants in that it allowed the normalization of each group's analysis techniques as well as the derivation of new techniques owed to the synergistic nature of the meetings. The TDA project provided a formal mechanism for this gathering of experts that did not previously exist.

Prior to approving the TDA project, a detailed cost-benefit analysis was performed that identified a potential for nearly a half a million

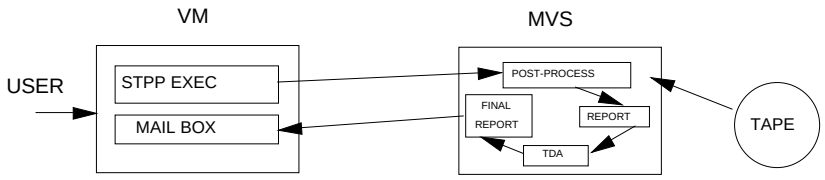

Figure 5. The Current Process for Dump Analysis.

dollars in annual personnel savings because of the automation of the dump analyses during both the development process and normal operations. These savings have been documented and justify the development of the system.

It was determined that TDA saves an experienced TPF programmer an average of one hour during the analysis process and a novice TPF programmer an average of a full day (eight hours). TDA also reduces the amount of supervision required by less experienced programmers when analyzing a TPF dump, freeing senior personnel to focus on more complex problems. (The average time savings for each dump was estimated at approximately six hours.)

TDA has provided productivity improvements in four separate areas of the development and operations process: (1) the analysis of problems occurring on the online APOLLO system, (2) system testing during new project development, (3) general testing of system operations and existing application enhancements, and (4) the installation and loading of new software segments onto the APOLLO system.

Roughly 1000 TPF dumps occur on the online APOLLO system that are analyzed each year. TDA reduces the amount of time required to solve these dumps by 6000 hours (6 hours for each dump, on average). Dumps that occur during new project development are usually encountered during the system-testing phase of the project. On average, 20 development projects occur during a year. Each of these projects encounters approximately 25 dumps during system testing. TDA reduces the total time required to solve these problems by 3000 hours (20 projects x 25 dumps each project x 6 hours each dump). There are 25 application areas, each of which generates 4 dumps each month, on average, during ongoing application enhancement and testing. This process adds another 7200 hours in time savings (25 projects x 4 dumps each month x 12 months x 6 hours each dump). Finally, the time saved during the installation and loading process comes to about 1500 hours each year. There are typically 250 installation and load cycles each year, with an average of one dump each cycle (250 installa-

tion and load cycles x 1 dump each cycle x 6 hours each dump).

Additional nonquantifiable benefits, such as improved customer goodwill and greater system reliability, were also identified, but it is impossible to quantify actual cost savings in these categories.

Highlights of TDA Benefits.

Automated dump analysis: Automated control dump analysis results in increased up time, cost savings, and efficient use of programming resources. These factors are especially critical when viewed from the maintenance perspective. As newer, high-level languages are introduced into our environment (C, PL/1), the level of expertise related to the TPF assembly language programming environment is inexorably deteriorated. An automated analysis system significantly reduces the personnel required to support the tremendous amount of TPF assembly code that has been created over the last 20 years (~ 10 million lines of code).

Rapid problem resolution: The TPF dump analyzer reduces the time required for programmers to diagnose and resolve common programming errors.

Quantified productivity improvements: The greatest savings is nearly half a million dollars annually in personnel costs. These costs savings are realized by saving time and increasing productivity; there has been no actual staff reduction at Covia. The productivity improvements were quantified by determining the amount of time that the programming staff would have spent performing the tasks now done by TDA.

Maintenance Issues

The initial maintenance of TDA will be performed by a member of the Artificial Intelligence Group. It is planned to train a member (or members) of the development staff to maintain the system in the future.

TDA has generated a great deal of interest and excitement in the user community, especially because it provides an analysis aid in an environment that is intimidating and complex. Numerous suggestions have been made, however, for further enhancement of the system to automate the processing of other problem types. These suggestions are being collected and will be addressed during a subsequent development phase.

TDA will also be a continuing effort, which will allow the system to handle a wider variety of dump types. This enhancement and others are discussed in the following section.

Future Enhancements

This section outlines future system enhancements. Such enhancements include extending the diagnostic capabilities, using the system as a code analyzer, and adding a tutorial.

Extend Diagnostic Capabilities

The dump analyzer will be extended to diagnose a wider range of problems than are currently addressed. This task will be accomplished by performing ongoing knowledge acquisition with the community of analysts who use TDA. As further problems are given sufficient description to allow a solution to be encoded in TDA, these newly identified diagnoses will be added. This knowledge-acquisition–TDA enhancement process is envisioned to be a continuous, ongoing effort.

By adding additional problem types that TDA will need to solve, the computational complexity of the reasoning system within TDA will increase. These additions will most likely require the simultaneous assessment of many possibly contradictory solutions, suggesting the addition of a truth maintenance system (Forbus and deKleer 1991) to enable TDA to perform these more sophisticated analyses in an efficient manner.

Use with Other Languages

TDA can be used with the assembled output of any language, such as C or COBOL, that generates native System/370 code because TDA constructs its internal representation of the offending program from the actual System/370 hex representation.

Use as a Code Analyzer

Based on the unanimous comments received from our expert group, we concluded that it would be appropriate to use TDA as a preventative measure, not just as a means of diagnosing existing errors. Thus, TDA would serve a purpose similar to the UNIX LINT utility, although not nearly as extensive as LINT because it has been incrementally improved for most of two decades.

The use of TDA as a code analyzer would not remove the need for the analysis of real-time program exceptions because the body of code being exercised spans 20 years of development and will be replaced slowly over time, if at all. Also, computational restrictions, such as the halting problem and the postcorrespondence problem (Harrison 1978), limit the capability of determining program correctness. Thus, there seems to be a long-term use for a system that diagnoses problems ex post facto.

Provide Tutorial Services

Another enhancement planned for TDA is to include a sophisticated hypertext-based tutorial system that can be used to teach the basics of the dump-solving process to novice programmers.

Conclusion

TDA is the first component in what will be a larger set of programming assistants that will improve the efficiency of the development and operations functions within Covia. To maintain the high levels of service and system up time and stay competitive, it is essential to develop processes and tools that assist existing programming and support staff members in performing their jobs more efficiently.

The objective of TDA was to develop an intelligent application that could be used by both programmers and coverage staff members to reduce the time required to solve a common set of problems. TDA is just one component of a larger ongoing effort to improve quality and reduce costs and time to market.

About Covia and TDA

Covia is an information systems company serving the travel industry in 45 countries.

TDA was developed on IBM PS/2s using OS/2 and DOS. The expert system was developed using Aion Corporation's Aion Development System (ADS). The system is deployed on a mainframe computer running MVS and interacts with other mainframes running VM/CMS.

Acknowledgments

We would like to thank our experts who provided the knowledge and intelligence that was encoded in TDA: Atul Amin, Tammy Homan, Bryan Karr, Steve Murphy, Ky Slickers, Steve Schoenstein, and Jerry Tyra. Additionally, we would like to thank Pierre Campbell, Ralph Henning, and Greg Mally for their help during the installation process; John Bray, Kip Henderson, Kyung Lee, Jerry Looney, Tom Osborne, and Jeannie Smith for their insightful comments during the design and development process; and Phil Marie, Rich Lee, and Brad Boston for supporting our efforts.

Suggestions for Further Reading

Balzer, R. 1990. AI and Software Engineering, Will the Twain Ever

Meet? In Proceedings of the Eighth National Conference on Artificial Intelligence, 1123–1125. Menlo Park, Calif.: American Association for Artificial Intelligence.

Bobrow, D. 1985. *Qualitative Reasoning about Physical Systems*. Amsterdam: Elsevier Science Publishers, North Holland Publications.

Fikes, R. 1990. AI and Software Engineering—Managing Exploratory Programming. In Proceedings of the Eighth National Conference on Artificial Intelligence, 1126–1127. Menlo Park, Calif.: American Association for Artificial Intelligence.

Fox, M. S. 1990. Looking for the AI in Software Engineering: An Applications Perspective. In Proceedings of the Eighth National Conference on Artificial Intelligence, 1128–1129. Menlo Park, Calif.: American Association for Artificial Intelligence.

Hayes-Roth, F.; Waterman, D. A.; and Lenat, D. B. 1983. *Building Expert Systems.* Reading, Mass.: Addison-Wesley.

McDermott, J. 1990. Developing Software Is Like Talking to Eskimos about Snow. In Proceedings of the Eighth National Conference on Artificial Intelligence, 1130–1133. Menlo Park, Calif.: American Association for Artificial Intelligence.

Soloway, E. 1990. The Techies versus the Non-Techies: Today's Two Cultures. In Proceedings of the Eighth National Conference on Artificial Intelligence, 1123–1125. Menlo Park, Calif.: American Association for Artificial Intelligence.

Stallings, W. 1987. *Computer Organization and Architecture: Principles of Structure and Function*. New York: Macmillan.

References

Aho, A. V.; Sethi, R.; and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Reading, Mass.: Addison-Wesley.

de Kleer, J. 1991. Focusing on Probable Diagnoses. In Proceedings of the Ninth National Conference on Artificial Intelligence, 842–848. Menlo Park, Calif.: American Association for Artificial Intelligence.

Forbus, K., and de Kleer, J. 1991. Building Problem Solvers: Program Notes on Truth Maintenance Systems. Presented at AAAI-91 Tutorial on Truth Maintenance Systems, 15 July, Anaheim, Calif.

Green, C.; Luckham, D.; Balzar, T.; Cheatham, T.; and Rich, C. 1983. Report on a Knowledge-Based Software Assistant, Technical Report RADC-TR-83-195, Rome Air Development Center, Rome, New York.

Harrison, M. A. 1978. *Introduction to Formal Language Theory*. Reading, Mass.: Addison-Wesley.

IBM. 1987. IBM System/370—Principles of Operation, 11th ed. Yorktown Heights, N.Y.: IBM T. J. Watson Research Center.

Pearl, J., and Verma, T. 1988. The Logic of Representing Dependencies by Directed Graphs. In Proceedings of the Sixth National Conference on Artificial Intelligence, 374–379. Menlo Park, Calif.: American Association for Artificial Intelligence.

Rich, C., and Shrobe, H. 1978. Initial Report on a Lisp PROGRAMMER'S APPRENTICE. *IEEE Transactions on Software Engineering* SE-4(6): 456–467.

Waters, R. 1982. The PROGRAMMER'S APPRENTICE: Knowledge-Based Editing. *IEEE Transactions on Software Engineering* SE-8(1).