Copyright © 1982 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 61, No. 6, July–August 1982
Printed in U.S.A.

Automated Repair Service Bureau:

Software Tools and Components

By R. F. BERGERON and M. J. ROCHKIND

(Manuscript received July 15, 1981)

Complex software systems can be built from components, where a component is a software entity that performs one type of function in a general, usually table-driven, way. If the components are defined well and a standard interface is provided between them, the resulting product is flexible. It is easy to understand and easy to change. The performance of the system is a function of that of the components. The new Loop Maintenance Operations System Front End (LMOS-2) is a transaction processing system developed in this way. The components, described in this article, are a mask handler, a data validator, a database management system and a transaction-oriented filing system. Transactions are written in a high-level language designed for the purpose. The LMOS-2 system runs on the UNIX operating system. The desired flexibility properties have been realized and have produced important benefits in the course of LMOS-2 development. Moreover, with considerable attention to the performance characteristics of the components, a system has been produced which is as fast and efficient as its predecessor, coded mostly in assembly language.*

I. INTRODUCTION

The software systems developed in the past for the Automated Repair Service Bureau (ARSB) have been well suited to the application. By almost any measure—performance, availability, protection of the database—they are excellent systems. Their major drawback is inflex-

* *UNIX* is a trademark of Bell Laboratories.

ibility—a fault shared with many systems of their generation, particularly production systems built for major applications.

The Loop Maintenance Operations System (LMOS) front end (FE) was handcrafted for the ARSB application. The Bell Operating System (BOS) was built to the requirements of LMOS, as was the BOS Filing System (FS). The original application code was written in PDP* MACRO-11 assembly language. Application programs access and manage data files themselves, calling on FS only for direct operations on data blocks. As a result, database changes ripple through the application code, making change expensive and error prone. Another consequence of making LMOS very application-specific is that there has been little spin-off of LMOS technology to other projects. This is unfortunate, for LMOS has been an extremely successful product.

During the past decade (but for practical purposes after the LMOS FE system was built and deployed), the *UNIX* operating system was developed, bringing with it new software development tools and new notions of software flexibility and transferability.[1] Several years ago, the *UNIX* operating system supplanted DEC's DOS operating system in our development environment and we began to write new programs in C. The BOS continued to be our production operating system.

It was not enough to change the development environment, though it helped. Costs incurred in modifying the FE, extrapolated into an environment of increasingly rapid change, indicated that an entirely new approach would be necessary. The course we followed in LMOS-2 was inspired in part by the *UNIX* operating system and represents a step forward in software development technology. It is expected to be cost-effective for LMOS and, unlike previous LMOS development work, should have significant side benefits for other Bell System software projects.

The approach, described more fully in the next section, is to build the software system from components, taking care that the components are sufficiently general in function and that they interface neatly together. We avoid the temptation to handcraft another LMOS. Not that the temptation isn't there, for our objective is to buy flexibility without paying in performance or in any other way.

We think we have partitioned the system into components in such a way that the partitioning itself causes no loss of efficiency. We then have only to worry about the performance of the individual components. The challenge, for all components, is to be both general and efficient (either property by itself is easy). This was relatively straightforward for some components (the mask handler and the data valida-

* Registered trademark of Digital Equipment Corporation.

tor), and extremely difficult for others (the database management system and the filing system).

II. DEVELOPING SOFTWARE FROM COMPONENTS

The ability to adapt a system to a rapidly changing environment is related to an ability to build totally new, though similar, systems with relative ease. The difference in capabilities required of an LMOS system at points in time ten years apart might be quite significant. If we can really make LMOS tolerant to change, we should be able to use the same approaches to build new systems.

Our approach is to develop the capability to build "LMOS-like" systems and, in the process, make LMOS itself flexible.

One of the keys to this approach is *modularity*. However, we seek not just to achieve modularity for LMOS, but to identify for a class of systems a set of basic, application-independent, functions. For each function a software component is built, if it does not already exist somewhere.

Another significant factor is *generality*. Our components are script-driven. Each is configured for a specific application by coding the specifications for that application in a special-purpose language. For example, the data validator is configured by coding the validation conditions (which look much like expressions where the variables are data field names) in a validation language.

Finally, we have a standardized format for passing data between components. This guarantees that the output of any component is in a form suitable for input to another component. Hence, many different arrangements of components are possible. Once the software components are on the shelf, it should be possible to build new systems, or change an existing one, rather easily.

Ultimately, we hope that building a software system will be similar to building a house: you start not by designing everything from scratch, but by looking through catalogs of windows, kitchen cabinets, heating systems, etc. You keep costs down, improve quality, and shorten the completion time by limiting on-site construction to building what you can't find ready-made, and to installing what you got from the catalogs.

III. COMPONENTS OF A TRANSACTION PROCESSING MACHINE

In our model of a transaction processor, there is a collection of input cathode ray tube (CRT) terminals at which operators enter transactions. The processor supports a predefined set of transaction types. For each transaction type there is input data, entered on a mask by the operator, and processing by the system, which involves validation, database access, and (sometimes) updates. The output may be a report, directed to a printer or to a CRT, or just an indication to the operator that the desired function has been performed.

We can now identify some of the general-purpose components for transaction processing systems: a mask handler, which is easily programmable to produce input and output masks of arbitrary design, and which manages data flow to and from the CRTs; a data validator, which is capable of taking a set of data (e.g., input data for a transaction) and testing it against a corresponding set of validation conditions; and a database manager, which provides flexible yet efficient access to the system's database. With these components, and another component specific to the transaction to provide control and do any special processing required, we have nearly enough to implement many simple transaction types. The transaction model is this: the named mask, with input data, is on the screen when the SEND key is pushed, initiating processing. The mask handler routes the data to the validator for input validation. If valid, the data are sent to the transaction-specific component, which determines the rest of the execution sequence. That may involve local processing, as well as data transfer to and from the database manager and further validation. Finally, output data will be returned to the mask handler to be put up on the operator's screen or at a printer.

Note that the transaction is served by a number of components, including one that is tailored to it. The standardized component-to-component interface completes the component model of transactions.

IV. COMPONENT INTERFACE

The flow of data between components varies with each transaction. To allow flexibility in the order in which components are invoked and in the passing of data between them, there is a standard mechanism for invoking components, embodied in a subroutine called *invoke*, and a standard representation for data, called a *packet*.

4.1 Packets

A packet is an abstract data type that can represent a data record in which the fields are referenced by their names (as opposed to their positions within the record). Conceptually, a packet is a set of pairs of field names and field values; for example,

TN	201-3862773
NAME	Marc
ADDRESS	1D 301A Whippany.

Operations are provided to initialize a packet, change the value for a named field, and get the value of a named field, among other things. Field values are null-terminated strings. The data structures underlying the packet implementation are not accessible to the user and do not need to be. All manipulation of data in packets is done using the packet operations. Packets provide a natural, easy-to-use mechanism for expressing and interpreting fielded data.

By standardizing on a data representation for all components, we allow the output from one component to be fed into another component without translation or reformatting. Also, common utilities, such as trace routines, can be used on any component.

4.2 Component invocation

All components have the same interface, a subroutine call with three arguments: the name of the component to be invoked, the packet to be operated on, and a flag indicating whether the caller is to wait for completion of the operation or not. The input data and the output data are in the same packet. This permits the use of a particularly efficient data transmission technique using memory shared between the calling and called components. For example, the Database Management System (DBMS) component is called from another component by

invoke ("DBMS", pkt, WAIT);.

This sends the packet pointed to by *pkt* to the component named DBMS. The packet contains fields that indicate to DBMS the operation, file name, key, and also the name-value pairs for the database fields involved in the access. For a "retrieve," for example, the names of fields for which data are desired would be present in the offered packet. The DBMS returns the packet with the values filled in.

V. TRANSACTION SPECIFICATION LANGUAGE*

The Transaction Specification Language (TSL) is a high-level programming language that includes the C language as a proper subset. The "non-C" features of TSL are intended to facilitate writing the transactions of LMOS-2 and similar applications. In particular, transactions written in TSL are easier to write, modify, maintain, and understand than the same programs written in C.

The key features of TSL are as follows:

- A *packet* data type, and operators for manipulating packets.
- String operators (e.g., concatenation).
- An operator for pattern matching.
- Statements to facilitate the use of the DBMS.

In the following, we describe the "non-C" features of this version of TSL. We assume that the reader is familiar with C and its conventions.[2]

* The Transaction Specification Language (TSL) was designed and implemented by J. W. Hunt. This section was adopted from his TSL user's manual.

5.1 Declaration types

In addition to the basic C-declaration types, TSL provides three new declaration types: *packet*, *field*, and *view*. Variables using the *packet* declaration type name instances of the abstract data type discussed in Section IV. The *field* declaration type is used to declare the names of accessed packet fields. Aggregates of field names are declared using the *view* declaration type.

5.2 Operators

Two binary operators are available in TSL that are not available in C: the concatenation operator (//) and the regular expression operator (*matches*). The language also provides an additional interpretation of the "." operator which is used in C to reference members of a structure.

5.2.1 Concatenation

The concatenation operator requires both operands to be pointers to a null-terminated sequence of characters. The value of the expression

$$a \,//\, b$$

is a pointer to a null-terminated sequence of characters consisting of the nonnull characters pointed to by a followed by the characters pointed to by b. This operator is simply an abbreviation for the *strcat* function of C.

5.2.2 Regular expression matching

Transaction programmers are often interested in finding out whether a string is a representative of some set of strings. The *matches* operator provides a general facility for doing just that. As with the concatenation operator, the *matches* operator requires both operands to be pointers to a null-terminated sequence of characters. The regular expression (the second operand of the *matches* operator) is a shorthand for specifying a set of strings. The value of the expression

$$a \textit{ matches } \mathrm{regx}$$

is nonzero if the string pointed to by a is a member of the set of strings specified by the regular expression *regx*.

The regular expressions recognized by TSL are similar to those used in the *UNIX* text editor. That is, "." matches any character, "$" matches the end of a string, "[xyz]" matches "x", "y", or "z", and so on.

5.2.3 Referencing fields of a packet

Suppose *pkt* is a pointer to a packet and LRTN is a declared field name. The term

pkt.LRTN

represents the value of the LRTN field in the packet *pkt*. If the term appears to the left of the assignment operator, the result is to change the value of the LRTN field in *pkt*. For example,

pkt.LRTN = "210" // "3863000";

changes the value of the LRTN field to "2013863000." If the term *pkt.LRTN* appears to the right of the assignment operator, the result is the value of LRTN, which is a pointer to the null-terminated sequence of characters stored as the field value.

5.3 Statements

The data-type *packet* plays a central role in the code written by a transaction programmer. For example, every transaction in LMOS-2 accepts a single input, a pointer to a *packet*. This packet is created by the mask handler (see next section). Transaction programmers also use packets to send information back and forth between other processes in the system. Thus, most of the "non-C" TSL statements provide convenient mechanisms for manipulating the values stored in a packet. In the following, examples from LMOS are used to help describe TSL statements, but keep in mind that these statements are useful in any application that must manipulate packets.

5.3.1 Creating a view for a database call

When retrieving information from a file using the DBMS (see Section VIII), one is required to send DBMS a packet that has all of the fields of the view to be retrieved in the packet. The DBMS forms the intersection of the fields of the view corresponding to a record of the file being queried and the fields in the packet, and returns the values of the fields in this intersection. Therefore, the first time that a retrieve request is made for any particular file, the transaction programmer must make sure that all the relevant fields are in the packet that was sent to DBMS. At worst, this would entail a separate statement to initialize each field of the view to be retrieved.
The TSL statement

make pkt a uif view

changes the value of each field of the view *uif* in packet *pkt* to the null string. If the packet is then used in a retrieve request to the DBMS, the fields that were listed in the definition of view *uif* will be retrieved.

5.3.2 Copying the fields of a view

There are statements that permit transfer of fields of a given view from one packet to another and from a packet to and from temporary storage.

5.3.3 A variant of the "switch" statement

In addition to the *switch* statement of C, TSL provides a similar statement called *rswtch* which allows the programmer to affect flow of control based on the value of a null-terminated character string.

5.4 Experience with TSL

Transaction Specific Language has been used to code all of the transactions for LMOS-2. There are 53 transactions, averaging 425 lines of TSL code. While our evaluation is subjective, we feel that, in comparison to the old assembly language transaction programs, and even in comparison to a few transactions that had been written in straight C, the TSL programs are smaller, more easily written, faster to debug, and more readable. Furthermore, since the facilities of TSL are so well suited to the programming of transactions, there has been no significant loss of efficiency.

VI. MASK HANDLER*

The Loop Maintenance Operations System uses a fill-in-the-blank approach for transaction entry. The terminal operator initiates a transaction by first requesting the appropriate mask (for example, "/FOR TE" requests the trouble-entry mask). The mask that is displayed resembles a paper form: there are blanks to be filled in, and "preprinted" labels. The labels are protected; that is, the operator cannot overtype them. After the operator fills in the blanks, he or she submits the mask by pressing the SEND key. If the transaction has a result to be displayed, another mask (possibly with more blanks to be filled in) is displayed. Otherwise, the operator can clear the original mask and reuse it.

As we said in Section IV, LMOS-2 uses a standard data format, the packet, internally. The mask handler, thus, has two translation duties: When a mask is entered, it gets the data from the terminal in a format native to the terminal (a Teletype† TTY 40/4 or an IBM 3270 terminal) and builds a packet. On output it takes a packet and builds a mask, again in the format native to the terminal. Not only does this translation hide messy details of the terminal from the rest of the system, but it also allows different terminals to be used without affecting any module other than the mask handler itself. In fact, we have a version of the mask handler that works on simple time-sharing terminals.

* The mask handler was designed and implemented by D. A. Rosenthal.
† Registered trademark of Teletype Corporation.

These terminals are inefficient for production use, but they are invaluable for debugging. Once a transaction works with the time-sharing terminal, it also works with the Teletype 40/4 terminal, because its interface with the mask handler is absolutely identical in both cases.

Recall that a packet consists of *named* fields. When the data comes in from the terminal, the fields are not named—they are simply sequenced in the same order that they were in on the mask. To build a packet, then, the mask handler must know what transaction's mask is on the screen, and, for that mask, what the names of the fields are. This information appears in a *mask table*. The mask table also specifies display attributes for each field that determine how the mask is to appear on the screen. This information is used when the mask is output.

The mask table contains one specification for each mask. The following is a sample mask specification:

```
_TE   CLEAR
      1       1      pam    "TE"
                     skip   "TN"
              {16}   unm    TN
      1       30     skip   "RSA
              {3}    uam    RSA
      4       1      skip   "DESCRIPTION"
              {30}   uam    DESCP.
```

The first line names the mask "_TE" and specifies that the screen is to be cleared before the mask is displayed. The underscore before "TE" means that this mask is to be used in response to a form request ("/FOR TE") rather than for entry of the TE transaction (which probably will follow the form request, as soon as the form has been filled out).

Each line of the rest of the specification describes a mask field (not to be confused with a packet field). Columns 2 and 3 are used for x- and y-coordinates or for horizontal distances. Column 4 contains screen attributes. Column 5 contains either a literal to be displayed or the name of a *packet* field to receive some data.

The second line, then, specifies that the letters "TE" are to appear in row 1 of the mask, in positions 2 and 3. Position 1 is reserved for the attribute, "pam," which specifies that the letters "TE" are to be protected and returned when the mask is sent. The third line has no coordinates, so the letters "TN" go in the next available positions, which are 5 and 6. Position 4 holds the attribute, "skip" which means that the letters are to be displayed but not returned to the computer. The fourth line specifies an unprotected field 16 positions long, so it occupies positions 8 through 23. When the mask is sent, data typed into this field by the operator will be inserted into the packet under the name TN. The remaining lines specify labels and unprotected fields for RSA and DESCP. When displayed, the "_TE" mask would appear like this:

```
TE TN                    RSA


DESCRIPTION.
```

After the operator fills out the mask, it might look like this:

```
TE TN 2013861234      RSA 26


DESCRIPTION.
```

When the SEND key is pressed, the screen is read and a buffer consisting of terminal-specific control codes and data is sent to the mask handler. It sees that the first field is "TE" and looks in the mask table for a specification of the TE mask (which is similar to the "_TE" mask described earlier). It builds a packet that looks (conceptually) like this:

```
XACT  TE
DVC   T16
TN    2013861234
RSA   26.
```

The first two fields, XACT and DVC, were not specified in the mask table, but are standard fields used by the mask handler for the transaction name and terminal device name, respectively. The device name is used later to send data back to the same terminal that entered this mask.

Since the value of XACT field is "TE," the mask handler sends the packet to the TE program (written in TSL), where the processing specific to the TE transaction is performed.

It so happens that the TE transaction ends by displaying a TR mask with information about the customer whose telephone number is 201-386-1234. To display this mask, the TSL program changes the value of the XACT field to "TR" in the packet:

```
XACT  TR
DVC   T16
TN    2013861234
RSA   26.
```

It then reinvokes the mask handler by calling "invoke": (see Section IV.)

invoke("MASK", pkt, NOWAIT);.

The mask handler finds the specification for the TR mask in the mask table, uses the data in the packet to build a buffer containing the appropriate terminal-specific control codes, and sends the buffer to device T16. Processing of this transaction then terminates.

The operator, meanwhile, has observed a delay of a few seconds after pressing SEND, and then sees the TR mask with the customer's information that he or she requested. After adding some information by filling in blanks on the TR form (description of the trouble, time when the customer will be at home, etc.), the operator presses SEND again, and the cycle repeats—this time for a TR transaction.

VII. VALIDATOR

The data validator[5] takes a single packet, subjects it to various tests specified in a special-purpose validation language, and returns the same packet augmented with an error code for each test that failed. In LMOS, the validator is called by the mask handler once for each incoming mask. If an error is detected, the code is translated to a short English message by table lookup, the message is displayed at the bottom of the CRT screen, and the transaction is terminated. If the data has no errors, the packet is passed on to the appropriate transaction-processing component. Thus, each transaction programmer may assume that the input is a valid packet.

By convention, the field VALCODE is used by the validator to return error codes. So, for example, if the validator receives the packet

NAME	Mary Smith
ADDR	123 Main St
TN	2913861234

it might return the packet

NAME	Mary Smith
ADDR	123 Main St
TN	2913861234
VALCODE	tnerr.

The code "tnerr" indicates that the value of the TN field failed its validation test (291 is an invalid area code).

The mask handler, upon receiving the packet returned by the validator, determines whether any errors were detected by checking for the presence of the VALCODE field. In this example, it translates the code "tnerr" to, say, "Invalid area code," and displays the message.

Presumably, the operator will correct the TN field on the mask (the old mask values would still be on the screen) and then press the SEND key. This re-entered transaction would be an entirely new transaction; the system would not, and need not, remember the previous, failed attempt to enter the transaction.

The validation criteria for a particular application (e.g., LMOS) are specified in a table which configures the validator for that application. The configuration process involves compiling the validation table with the validation compiler, which produces intermediate code that is interpreted by the validation machine at execution time. Fig. 1 illustrates this architecture.

What makes the validator general-purpose is that all application knowledge is contained in the validation table, so the validator may be used for different applications just by changing the table.

The validation table consists of two columns. The first is an expression, involving field names, number and string constants, and operators, that defines a validation condition. The second column contains an error code that is generated if the validation condition is false. For example:

```
NAME % "[A-Za-z]{4,25}"       badname
ADDR != ""                    badaddr
SAL > 10000 & SAL < 75000     badsal.
```

The first condition requires the value of the NAME field to match the pattern within quotes. The pattern matches if the value consists of between 4 and 25 alphabetic characters. If it does not match, the error code "badname" is generated. The second condition just requires the ADDR field to be present (that is, not equal to the null string). The third condition requires the value of the SAL field to be in the range 10,000 to 75,000—supposedly, anything else is an unreasonable salary.

In the case of the last condition, it would be better to check that

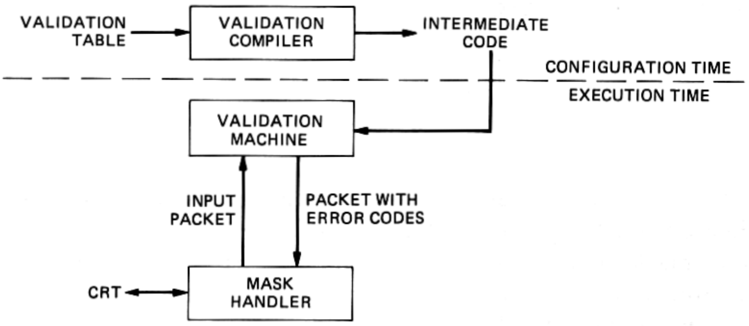

Fig. 1—Validator architecture.

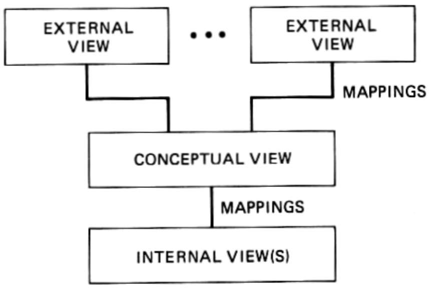

Fig. 2—ANSI/X3/SPARC database architecture.

SAL is numeric before checking its range. To do this, one may indent a condition under another condition:

```
SAL % "[0-9]{1,6}"                 badsal1
  SAL > 10000 & SAL < 75000        badsal2.
```

Now error code "badsal1" is generated if SAL is not from 1 to 6 digits. The indented range check is performed only if the first test passes; if SAL is out of range, error code "badsal2" is generated.

About two dozen operators and built-in functions are provided to code validation conditions. The reader is referred to Ref. 3 for further details.

VIII. DATABASE MANAGEMENT SYSTEM*

The DBMS is packaged as a component and communicates with other components via packets, as illustrated in the example of Section IV. The packet interface is natural, as nearly all the data in the transaction processor's database are fielded. Other properties of the DBMS that are critical to its usefulness are its flexibility and efficiency. These aspects are treated in this section. The DBMS design is described in detail in Ref. 4.

8.1 Structural model

The DBMS is an implementation of the three-level ANSI/X3/SPARC model.[5] (See Fig. 2.) There is an *internal view* of the data, expressed in terms of constructs that naturally describe the layout of data on the storage medium: hash files, linked lists, trees, etc. The *conceptual view* provides a canonical application and an implementation-independent model of the data of the enterprise. The *external views* provide the interfaces to the application programs. They may be structurally different from each other and from the conceptual view, and they may

* The DBMS was designed by T. C. Chiang and implemented by Chiang and A. Weinstein.

change over time with the applications. The only requirement is that it must be possible to map them all from the conceptual view. Of course, one of the external views may coincide with the conceptual view.

The ability to define multiple external views is nice—and in one case we found it essential. The first version of the DBMS was retrofitted into existing front-end software that had had only a low-level filing system. That DBMS provided a relational "new view" of a major file, with additional fields, for several new transactions. It was necessary to do this without changing the preponderance of existing software. Therefore, an "old view" was provided for the old transactions. The old view duplicates the data format presented by the filing system. The retrofit was a success. The system is now running at more than a hundred LMOS sites.

The LMOS-2 DBMS presents several external views as a matter of convenience for different types of transactions. Another advantage of the multilevel model is that the internal representation of data can be changed relatively easily without impacting the external views. It is just a matter of changing mappings. In fact, fairly extensive internal database redesign will be done eventually for LMOS-2.

8.2 The entity-relationship model

At each level in the ANSI/X3/SPARC structure there is a data model. The external views may support several data models, as required by the application: relational, hierarchical, ad hoc, or whatever. We use the Entity-Relationship (E-R) model at the conceptual level.

In the E-R model there are "things," and relationships between things. The entities are represented by entries in a file. Files are connected by named "associations" in our version of E-R. Each instance of an association is a relationship (implemented via pointers or physical contiguity) between one or more items in one file and one or more in the associated file. The associations have properties, including cardinality (1-to-1, one-to-many, many-to-many), and update constraints.

The associations provide an easy way to express the update relationships that are to be maintained automatically by the DBMS. They provide a mechanism for navigation through the database. What is of most benefit is that they allow information to be reflected to the user about relationships between items in separate files. This makes significant performance gains possible. Data Manipulation Language commands take association name as an argument, and allow the DBMS to use information picked up in previous database accesses to eliminate index searches and, in some cases, to satisfy a data request without going to disk.

In a system like LMOS, where there are many interfile relationships, the savings made possible by proper use of the associations may be significant. Analysis of busy-hour data for LMOS shows that nearly 50 percent fewer disk accesses will be required using the association data than would be necessary for a pure relational model in which each file were accessed independently. (In the old LMOS, transactions had direct access to pointers everywhere, the system was tuned for performance over a number of years, and there was no data model. Our problem has been to fit a semantically consistent data model to LMOS *without* sacrificing performance. Our E-R-based system and the old LMOS are equivalent in the number of disk accesses required.)

8.2 Scope of application

The LMOS-2 front end is a medium-sized record-based transaction processing system. The database is in excess of 300 megabytes, organized into 14 external files with 10 associations. External records range in size from a few tens to a few hundred bytes. Internal records may be thousands of bytes long.

The DBMS and the filing system (described below) run on version 4.0 of the *UNIX* operating system, modified somewhat to meet the performance objectives.

IX. THE C FILING SYSTEM*

The *UNIX* file system is a flexible and useful tool. It is not particularly well suited, however, to production transaction processing systems for which efficiency, protection of database consistency, and quick recovery from system crashes are critically important.

Therefore, we have developed a new filing system, the C filing system (CFS). This is a component, interfacing with DBMS and other subsystems requiring access to data in secondary storage. The CFS internal files are stored contiguously, minimizing the number of disk accesses required. Data are transferred across the CFS interface using our version 4.0 *UNIX* operating system's shared-memory facilities, another factor that improves efficiency. Multiple copies of CFS may be run, so that disk I/O may be overlapped.

9.1 The role of CFS in protecting database consistency

Database consistency is defined by a possibly large set of integrity constraints that often, in a complex system, are not formally expressed. The closest we get to a comprehensive statement of them is the set of audit/edit programs that is run against the database to find and correct inconsistencies. Inconsistencies may take the form of incorrect pointer

* The C Filing System was designed and implemented by D. H. Carter.

relationships, incorrect assignment status, failure of sums to balance, etc.

Because database updates are done one at a time, and because typically there are relationships between updates, the database cannot be consistent at every point in time. We can and do, however, identify groups of updates such that if the database is consistent before the group, it will also be consistent after all the updates have been made. These groups correspond to the transactions processed by the system. We can define the transaction, in fact, as the unit of database consistency. Care must be taken by the transaction designer to ensure that this is indeed the case.

Database inconsistency can arise in the following ways:

(*i*) In the event of a software crash, transactions may be left partly done. Inconsistency results if some, but not all, of the updates for a transaction are implemented.

(*ii*) If a database must be reconstructed, updates associated with transactions that had not been completed as of the catastrophe cannot be redone, for the same reason.

(*iii*) If transactions are run in parallel, with their database accesses interleaved, database states that could not have been developed by any serial execution of transactions may arise. These may be inconsistent. If we can guarantee that all database states are equivalent to states that would be produced by some serial execution of transactions, we achieve database consistency.

The CFS provides capability to recover from hardware and software failures and preserves consistency in doing so. It also provides concurrency control: data accesses of transactions executing in parallel are interleaved in such a way that serializability is guaranteed.

9.1.1 Concurrency control mechanism

Locks are applied at the block level by CFS. (The block is the unit of physical data access.) The CFS has "share" locks, which allow other readers, and "exclusive" locks, used when writing. Locks are applied when needed, on behalf of a transaction, and are held until the transaction is complete. This locking policy guarantees serializable transactions.

9.1.2 Logging and recovery

To protect database consistency in the face of system hardware and software failures, CFS delays updating the "real" copy of the file until the transaction signals, via a "commit" message, that it has issued all its update requests and is ready to have them implemented. Until the commit signal is received, updates are held in CFS buffers, tagged by transaction ID. Upon receiving "commit," the logging subsystem gathers all updates for the transaction into a large buffer and writes the buffer to the logging area on disk. If this atomic write is successful, the transaction is considered completed. Writes to the individual files must still be made, and locks are held until these are complete.

The overhead of this logging approach is minimal: one additional write per transaction. Recovery is simple if there has been no database damage. Transactions which have not committed have written nothing to the database. They simply disappear. Transactions which have committed but have not completed disk writes must have them re-applied from the logging area. The recovery process takes less than five minutes.

The strategy for recovering damaged databases involves periodic full copy of the database—daily for LMOS—and logging to tape of all committed updates. The database is reconstructed by applying the log tapes to the most recent backup copy. This may take an hour or two depending upon time elapsed since the backup copy was made.

X. EARLY FIELD EXPERIENCE

The LMOS-2 system, built with the components described in the preceding sections, went into field trial on October 16, 1980, with Michigan Bell Telephone Company. Development of the application and the components had proceeded in parallel over the preceding 15 months. The staffing was split roughly two-to-one between component and application development.

The development experience had met our expectations. With validation, mask handling, and database services provided by special components, and with the high-level TSL language and the simple packet interface, transactions were small, easy to write, and easy to debug. The reduction in lines of source was 7 to 1. Programmers who had experience implementing and maintaining the LMOS-1 transactions found the difference particularly striking. Even inexperienced programmers were able to build working transactions in a very short period of time. One, in his first programming assignment, wrote six transactions in four months.

Success or failure, however, would be determined by behavior of the system in the field. How would LMOS-2 compare in reliability and performance with its predecessor?

The reliability question was answered quickly and positively. On the first day of operation the system suffered five "fatals" (i.e., errors causing interruption of processing.) By the end of the first week, it was getting through an entire day without a fatal; by the end of the first month, its availability surpassed that of some of Michigan Bell Telephone Company's LMOS-1 systems. This compared favorably with the LMOS-1 trial experience, in which months passed before the system could process through the day without interruption.

Problems that did surface in LMOS-2 were almost always fixed within a day, sometimes within hours. The ability to find and fix problems quickly is due at least in part to the component-based system design.

As we indicated earlier in the article, performance of LMOS-2 was a major concern. Our objective was to add flexibility to the system with minimal sacrifice of performance. At the time the trial began, the load presented by the application—a new system serving a single repair bureau—was small: on the order of 300 to 500 trouble processing transactions in the busy hour. Laboratory load testing showed the system to be capable of processing 800 to 1000 transactions per hour at that time. The objective—the standard set by LMOS-1—was 4000 transactions per hour. In the seven months that have elapsed between the field trial cutover and the time of this writing, significant efficiency improvements have been made. The LMOS-2 system now processes more than 4000 transactions per hour in load tests. (The trial application has grown during the same interval and now presents a busy-hour load of 2800 transactions per hour.) Performance of LMOS-2 is approximately as good now as that of its predecessor.

The flexibility of LMOS-2 has served us well in management of the project and of the trial. One example: Much care was taken to ensure that in the early days of the trial the LMOS-2 and LMOS-1 databases were compatible, so that in the event of user-affecting problems at the trial site it would be possible to convert back to an LMOS-1 system within minutes. (This option was used several times: it maintained LMOS availability and gave us additional time to solve problems.)

To add new features to LMOS-2, a database conversion was performed in May and new versions were introduced for about 30 modules. Quick convertibility was given up at this point.

It is obviously desirable to give new standard installations of LMOS-2 a similar initial period during which reversion to LMOS-1 is possible. One way of doing that is to maintain the first versions of all the software used in the trial and use them in initial LMOS-2 installations. Fortunately, the generality of our component interface allows us to install all of the latest versions, except for the DBMS, atop the LMOS-1 database. The resulting system is functionally equivalent to LMOS-1 and can be backed out to it quickly. The new LMOS-2 features become available when the data conversion is performed and the DBMS replaced. We have the best of both worlds: easy convertibility in the early days of an installation, without the need to maintain two versions of the software.

XI. CONCLUSION

We have the beginnings of a catalog of components out of which "LMOS-like" transaction processing systems can be built. The components include a mask handler, a data validator, a database management system, and a filing system. There is a standard interface—packets—between components, and a language for programming transactions that fits the component model. Because the set of systems which we wish to build includes some (like LMOS) that place a premium on performance, availability, and integrity of the database, the database manager and filing system are designed with efficiency and robustness, as well as flexibility, in mind.

The first system to be built from our components is the LMOS-2 FE, which has been field tested since October, 1980, with very promising results.

XII. ACKNOWLEDGMENTS

Development and application of the LMOS software components has been a cooperative effort involving many individuals. Special recognition goes to J. W. Hunt for TSL, D. A. Rosenthal for the mask handler, T. C. Chiang for the DBMS, and D. H. Carter for the C Filing System.

REFERENCES

1. B. W. Kernighan and J. R. Mashey, "The UNIX Programming Environment," COMPUTER *14* (April 1981), pp. 12–24.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J.: Prentice-Hall, 1978.
3. M. J. Rochkind, "A Table-Driven Data Validator," Proc. of COMPCON FALL 1980, IEEE Catalog No. 80CH1598-2C.
4. T. C. Chiang and R. F. Bergeron, "A Data Base Management System With an E-R Conceptual Model," Proc. of Int. Conf. on the Entity-Relationship Approach to Systems Analysis and Design (December 1979), pp. 540–9.
5. D. Tsichritzis and A. Klug, "The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Database Management Systems," Oxford: Pergamon Press, 1978.