A program understanding support environment

by L. Cleveland

Software maintenance represents the largest cost element in the life of a software system, and the process of understanding the software system utilizes 50 percent of the time spent on software maintenance. Thus there is a need for tools to aid the program understanding task. The tool described in this paper—Program UNderstanding Support environment (PUNS)—provides the needed environment. Here the program understanding task is supported with multiple views of the program and a simple strategy for moving between views and exploring a particular view in depth. PUNS consists of a repository component that loads and manages a repository of information about the program to be understood and a user interface component that presents the information in the repository, utilizing graphics to emphasize the relationships and allowing the user to move among the pieces of information quickly and easily.

Software maintenance is broadly defined as any work done on an operational programming system at any time, for any reason.[1] Maintenance begins when the initial development effort ends and the system is put into production. Recent surveys show that expenditures for such software maintenance (including improvements) account for between 50 and 90 percent of the total life-cycle expenditures on the programming system.[2,3]

Many of the activities of a person who maintains software—in adapting the system to new environments or enhancing the system by adding or improving function—are very similar to activities of a software developer during the initial creation of the system. However, there is a difference in that software maintainers are constrained by the framework of the system being maintained. They must work within this framework in developing adaptations or enhancements. A software maintainer must be very familiar with the current system and must fully understand the framework of the particular system. Studies of the activities of software maintainers have shown that approximately 50 percent of their time is spent in the process of understanding the code they are to maintain.[4,5] This is particularly true for code that was not developed using modern software engineering principles (e.g., data abstraction and structured programming).

As a programming system ages, the code for the system becomes increasingly the only true definition of the system, and ancillary specifications for the system become out of sync with the system. A software maintainer must read the code in order to determine the framework for the system. Brooks[6] defines the essence of a software system as a complex construct of interlocking concepts: data sets, relationships among data items, algorithms, and the invocation of functions. The software maintainer must understand these concepts and how they interlock in order to fit an adaptation or enhancement into the framework.

In determining the framework, a programmer is attempting to determine the relationships that exist within the program: where the variables are defined and where these definitions are used; what the interface of a piece of code is to other pieces; which data items are imported and which are exported; what data items are defined globally; where a function is used; what are the interfaces of the function; what is the scope of the definition of a particular piece of data; which internal procedures assign the data a value; which procedures reference the value; where is a file referenced; where is an entity in a database updated. By acquiring this information, the maintainer can begin to define the concepts that are used within the programming system and determine how the concepts have been interwoven to structure the programming system.

Program elements (e.g., functions, subprograms, and procedures) are still being written using a linear-language paradigm and then stored in files. Relationships among these files are understood only by analyzing the statements within the program elements. These relationships that are defined within the program are multidimensional. For example, a procedure E might be called by procedure C, E might also call procedure G and define data used by procedure H, and then E might update data defined in procedure B. Thus procedure E is related in various ways to four additional procedures. The programmer, in the task of understanding, must attempt to superimpose these multidimensional relationships on the linear text for the program. A tool that allows the programmer to easily view the multidimensional relationships that exist within the program would make easier the task of understanding.

The tool discussed in this paper, a Program UNderstanding Support environment, supports the program understanding task by organizing and presenting the program from many different viewpoints (e.g., a call graph for a collection of procedures, a control flow graph for a single procedure, a graph that relates a file to the procedures that use it, a data-flow graph, a use-def chain for a variable). The tool detects the low-level relationships that exist within the program using static analysis techniques, and it consolidates and organizes these relationships and presents them in a user-friendly environment. The user interrogates various pieces of information presented by the tool using a point-and-click mouse interface to point at a piece of information (an object) and click the mouse. The tool responds by providing more information about the object at which the user was pointing, including the object's relationships to other objects. The user can follow relationships between objects, easily moving between high-level and low-level objects.

There are two components to the Program UNderstanding Support environment tool (hereafter known simply as PUNS): (1) a repository and the associated routines to load the repository and respond to high-level queries to the repository and (2) a user interface that presents program information obtained from the repository in a user-friendly manner and supports the programmer during the exploration of the program. The repository component of PUNS must be established prior to utilizing the user interface component.

The two components are designed to be distributed. The repository component resides on a host machine (i.e., a powerful shared machine), able to use the power of the host for efficient repository loading and query response. The user interface component of PUNS resides on a workstation, utilizing the graphical strengths of the workstation in the presentation of information to the user. The components are connected via a communications link. The user interface component sends queries to the repository component over the communication link, and the repository component responds using the communication link.

A research prototype for the PUNS tool supports IBM System/370 Assembler Language. The repository component exists on an IBM System/370 30XX host and runs under the EAS-E Application Development System, which supports an Entity-Relationship data model and provides a language in which to write programs to create and access a database developed using this model.[7] The user interface exists on a workstation (either an IBM PC/AT or PS/2) and uses Microsoft Windows® to support the multiwindow point-click interface.

Several research vehicles have been developed that investigate the presentation of multiple views of a program.[8,9] However, these systems have focused on the initial development cycle and develop the views as the program is developed. A few maintenance environments have been explored,[10–13] most of which have used a database to maintain information about the program. However, to obtain information about the program, the maintainer has been forced to use a query language rather than a point-click interface. Also, it has not been easy to move between views in these environments.

In the first section of this paper, the user interface component for PUNS is described from a user point of view. The second section provides a general description of the PUNS repository and the routines to support the repository. The third section discusses the current research. The concluding section suggests adjunct areas of research.

The PUNS user interface

The PUNS user interface is organized around the notion of a set of objects that are of interest to the programmer who is using PUNS. Examples of objects include a module, a node in a control flow graph, a statement in an assembly unit, a database, a global data structure, and a symbol. Each of the objects is supported in the interface by a separate window. All information that PUNS knows about the object appears in the window for the object. Certain information about the object may appear when the window for the object is first created in response to the user's having pointed at a representation for the object and clicked the mouse. Other information is indicated as being available by its appearing on a menu bar, and it may be obtained by the selection of the item on the menu bar. Objects exist in the context of other objects. For example, a node of the control flow graph for a module has meaning only in the context of the module. The presentation of the windows maintains this context. An object that is known within the context of another object has its window appear within the window of the contextual object.

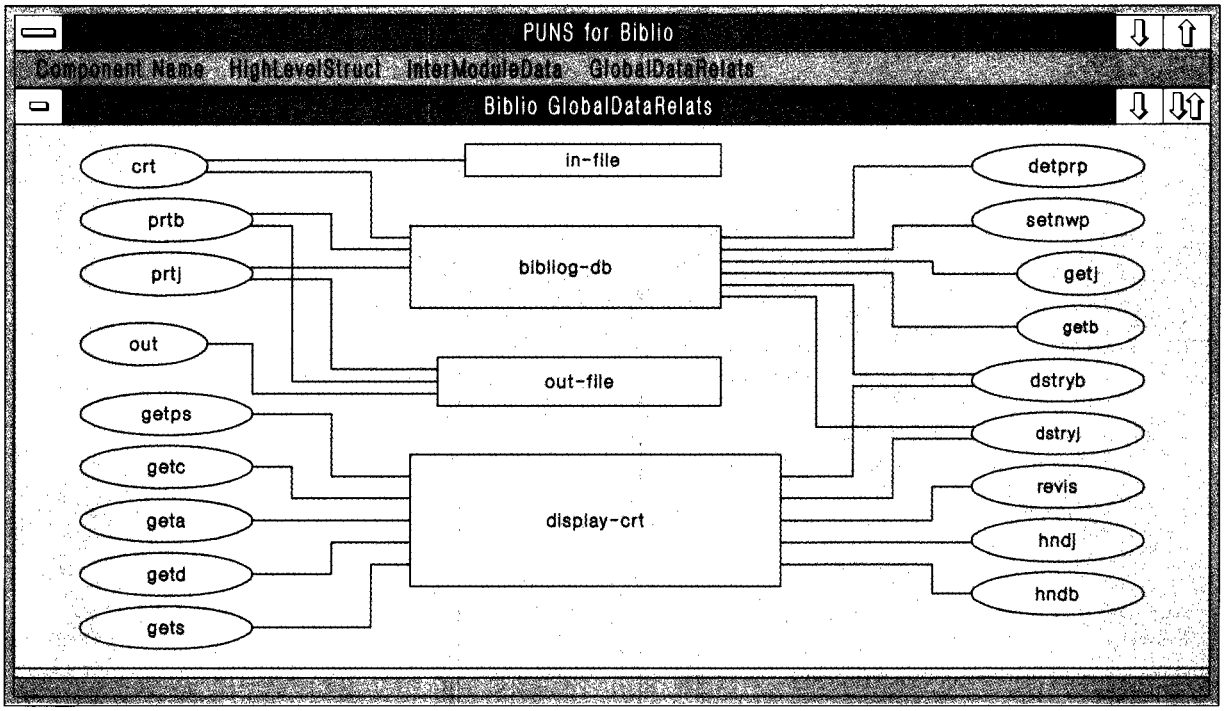
It is assumed that the maintainer using the PUNS user interface has previously used the PUNS repository component to structure a database for support of the user interface. The scenario described next also assumes that the communication link between the two components has been established and that the PUNS user interface component can talk to the PUNS repository component.

Sample scenario. The program being explored in the sample scenario is one that allows the user to maintain and explore a database of bibliographic information. In this discussion, the program is known as the *biblio* program. The series of events that led to the use of PUNS were: (1) a user of the *biblio* program has requested information about a particular reference represented in the database that the *biblio* program manages; (2) the *biblio* program was run and produced a document that contained the citation information for the reference as requested by the user; (3) the user has found that the page numbers given for the reference in the document are incorrect; the page number specification should be 1024–1035 and is given as 1024–10; (4) the programmer who is responsible for the *biblio* program, who has recently assumed responsibility for the program and is not familiar with all its parts, is attempting to determine how incorrect page numbers were established.

A repository for the *biblio* program has previously been created using the PUNS repository component. The programmer invokes the PUNS user interface component and establishes the communication link to the PUNS repository component. The sample scenario details the programmer's exploration of the *biblio* program.

The programmer is first presented with a window that represents any component object. The only option available for selection on the menu bar is the component name. The programmer selects this option and enters the name of the component, *biblio*. This name is sent to the PUNS repository component, which selects the repository for the *biblio* program. The component window is updated to reflect the name of the component to be explored, and the following options become available on the menu bar: high-level structure, global data relations, and intermodule data sharing. The programmer is interested in determining how a piece of information that exists in the database is created, updated, and displayed, and selects the global-data-relations option on the menu bar. This is accomplished by pointing the mouse at the menu item and single-clicking the left mouse button. Within the component window appears a window that displays a graph showing the relationships between the global data elements used in the program (databases, files, screen definitions, and global data structures) and the individual modules of the program. Figure 1 shows the window that is displayed. Each module in the current PUNS context that references a global data element is represented in the graph by an ellipse that contains the name of the module. Each global data element is represented by a rectangle that contains the name and type of the global data element. The arcs (or lines) that connect modules and global data elements represent relationships among the modules and the global data elements. In the current PUNS prototype, the arcs as seen in Figure 1 are undirected. However, each arc should be directed to show the flow of data between a module and a global data element. If a module references data that exist within a global data element, the arc is directed from the global data

Figure 1 Global data relations window



element to the module. If the module supplies (updates or creates) data for the global data element, the arc is directed from the module to the global data element. In cases where the module both references and supplies data, a doubly-directed arc exists between the module and the data element.

To provide another context in which to view the modules, the programmer also decides to look at the high-level structure chart by selecting this piece of information through the menu bar of the component window. Figure 2 shows the high-level structure chart for the *biblio* program. As in the global-data-element graph, an ellipse represents a module, and an arc in the structure chart represents a call-return relationship between two modules. Again, in the current research prototype, the arcs are undirected. The caller in the diagram is placed at a higher level than the callee.

Figure 3 shows the two graphs within the context of the component window. Microsoft Windows supports a feature that allows any window to be maximized to the size of the window in which it is contained. The PUNS user interface uses this facility to allow the programmer to concentrate on the details of a specific window. In Figures 1 and 2, maximized versions of the two graphs are shown. Figure 3 shows the graphs as they exist within the context of the component window.

The relationships provided by the two charts show overall organizational information about the *biblio* program. The programmer uses this information to delve further into the way that the *biblio* program utilizes the database it maintains and points to the database object on the global data relations chart and double-clicks the left mouse button. A new window, one representing the database, now appears. The menu options on this window are: *dbDef* (database definition), *relGraph* (graph of database usage by program modules), *dbObject* (objects defined in the database), *objAttribute* (attributes of the objects defined in the database). Because the programmer is specifically interested in the page numbers for a journal article but is not sure whether the page numbers are represented in the database as a separate object or simply as an attribute of an object, the next step is to select the *dbDef* option on the menu bar. This selection brings up a window showing the da-

Figure 2 High-level structure window

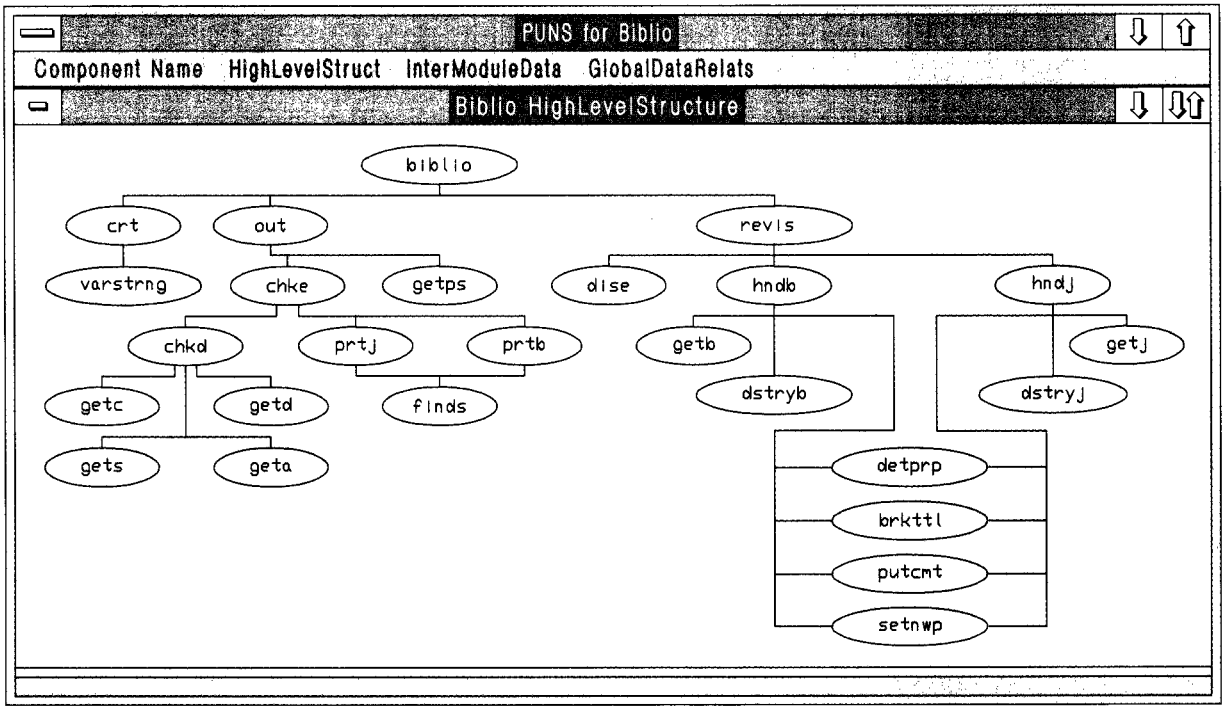


Figure 3 PUNS component window

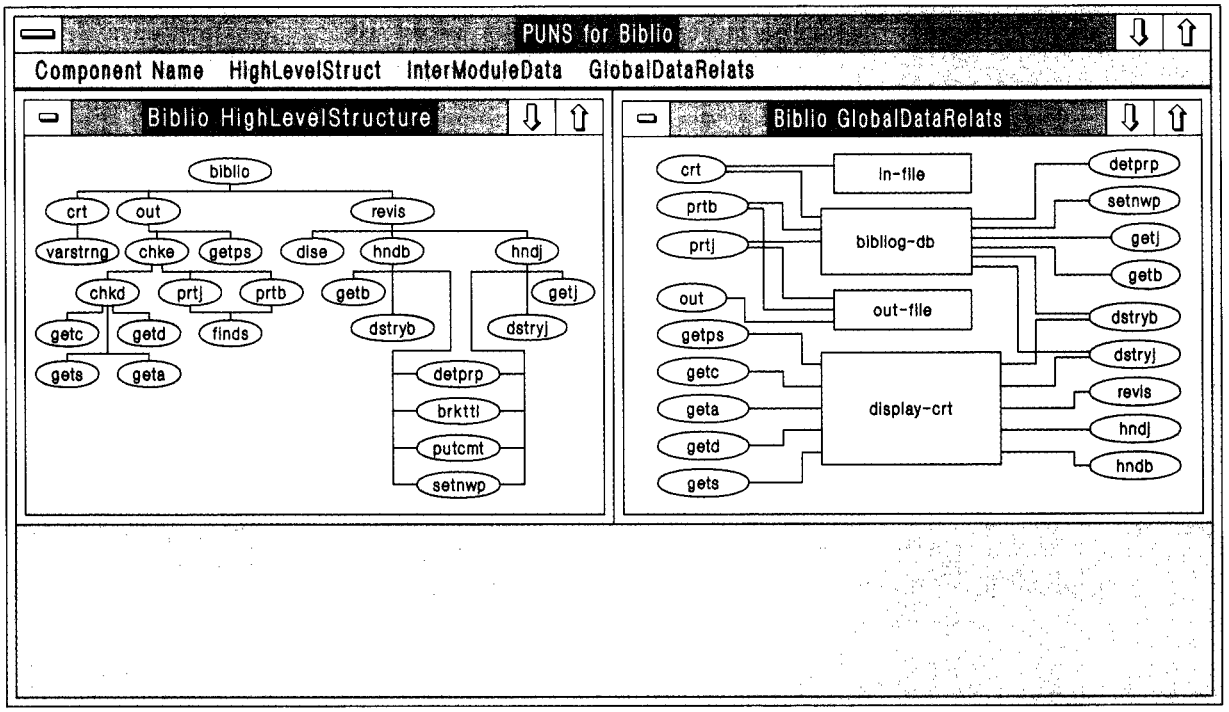
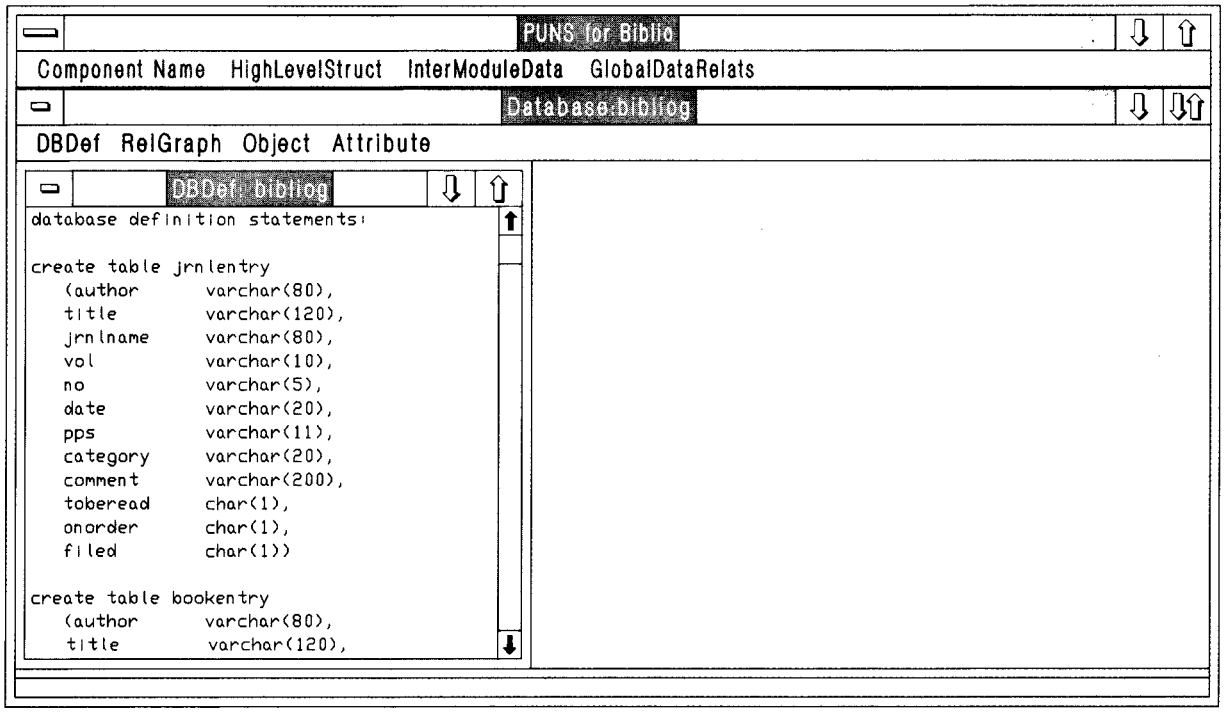


Figure 4 Bibliog database with dbDef window

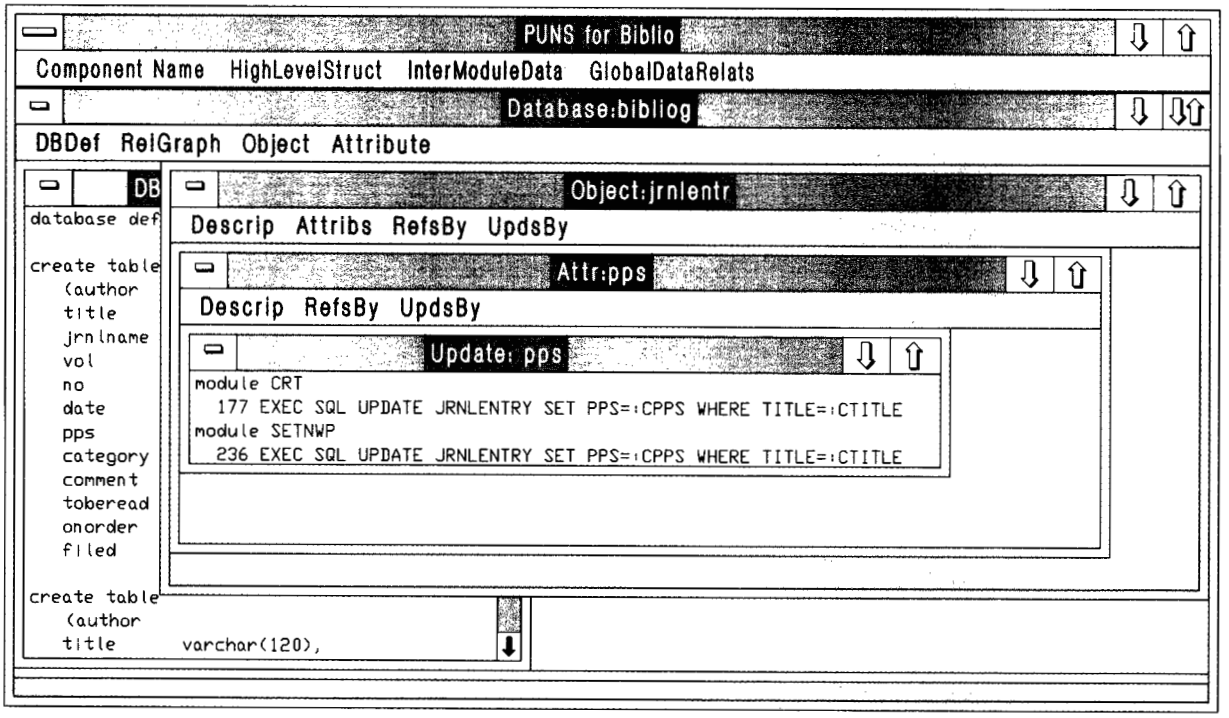


tabase definition statements. The programmer scrolls through the definition and finds the page numbers as an attribute of the journal entry (*jrnlentry*) object. Figure 4 shows the *dbDef* window within the database object window. The programmer finds that the page number attribute seems to be defined to handle a sufficiently large page number specification and thus concludes that the problem of an incorrect page number specification is not the result of any truncation of data within the database. By pointing the mouse at the page number specification attribute (*pps*) and double-clicking, the programmer can bring up two windows contained one within another. The windows represent the object (*jrnlentry*) and the specific attribute (*pps*) associated with the object. The menu bar for the attribute window shows the attribute-specific information that can be obtained: description, references by, updates by. Because the programmer is interested in who updates the page number attribute, the *updates by* option is selected. Figure 5 shows the window that results from this selection.

Had the programmer known the specific attribute specification for the page number specification and the fact that it is an attribute of the journal object, the object and attribute windows presented in Figure 5 could have been obtained more directly. Rather than having to select the *dbDef* option and find the page number specification attribute therein, the programmer could have selected the attribute option on the database object window menu bar and provided the attribute and object specification. The windows shown in Figure 5 would have appeared directly. If the programmer had known the specific attribute specification, but not the object specification, the attribute option could also have been selected. The programmer could then have provided the attribute name, and the user interface would have presented a list of objects in which attributes with the specified name appeared. The programmer would then have selected the appropriate object, and the object and attribute windows shown in Figure 5 would have appeared.

In Figure 5, the *updates by* window indicates that the page number specification field in the journal object is updated in two modules, the *crt* module and the *setnwp* module. For each module, the specific statement(s) that accomplishes the update is

Figure 5 Database object and attribute windows



also shown, but the context in which the update is done is not shown. There are two aspects to context, the context in which the module exists and the context within the module. To see the context in which the module exists, the programmer need only look again at the structure chart shown in Figure 2. To see the context within the module, the programmer points to a statement that updates the page number specification and double-clicks the mouse. Two new windows appear, a window that represents the module object and a window within this window that represents the statement object within the module.

The programmer first chooses to explore the *crt* module and the statement that updates the field within this module. Figure 6 shows the windows that appear when the programmer selects the statement within the *crt* module. The updating statement shows that the page specification attribute (*pps*) is updated using the value in the variable CPPS, which is defined within the program. By pointing to the CPPS symbol and double-clicking the right mouse button, the programmer selects the symbol CPPS as an object of interest, and a window with information about the symbol CPPS appears within the module window. Figure 7 shows this window in its maximized format. The definition for CPPS provides for eleven characters in the variable-length-character-string portion of CPPS. When using SQL with variable-length character strings, the storage for a symbol to supply the value of the variable length character string is defined as a half-word that contains the length of the actual character string followed by a set of bytes that contain the actual character string data. In the current version of PUNS, data-flow analysis is done only for registers; thus the information about the symbol CPPS is cross-reference information only. Data-flow analysis for symbols is a planned extension to PUNS and, when done, the window for the symbol CPPS will provide data-flow relationships for the references to and sets of the symbol.

In this example, the other statements that reference the symbol CPPS appear immediately to precede the statement that uses the value in CPPS to update the database. The programmer brings up the source for the *crt* module, scrolls to the first statement specified in the cross reference and studies the code. Figure 8 shows the windows that the programmer studies,

Figure 6 Module and statement windows

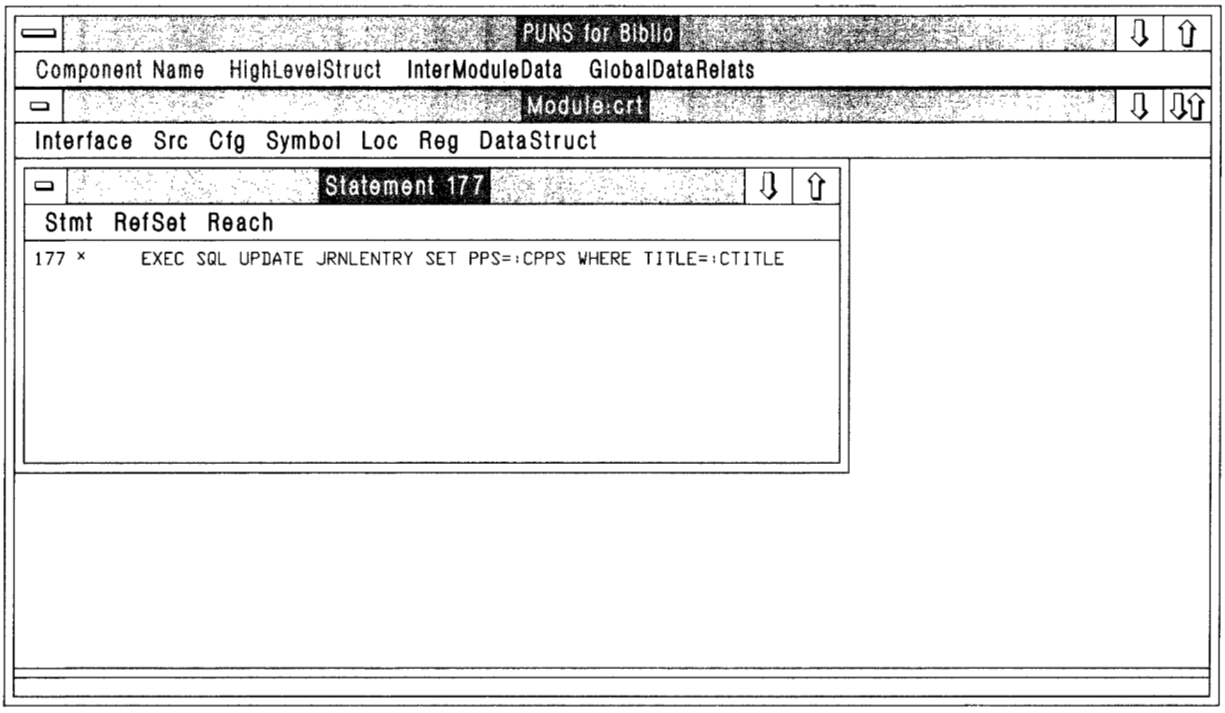


Figure 7 Symbol CPPS window

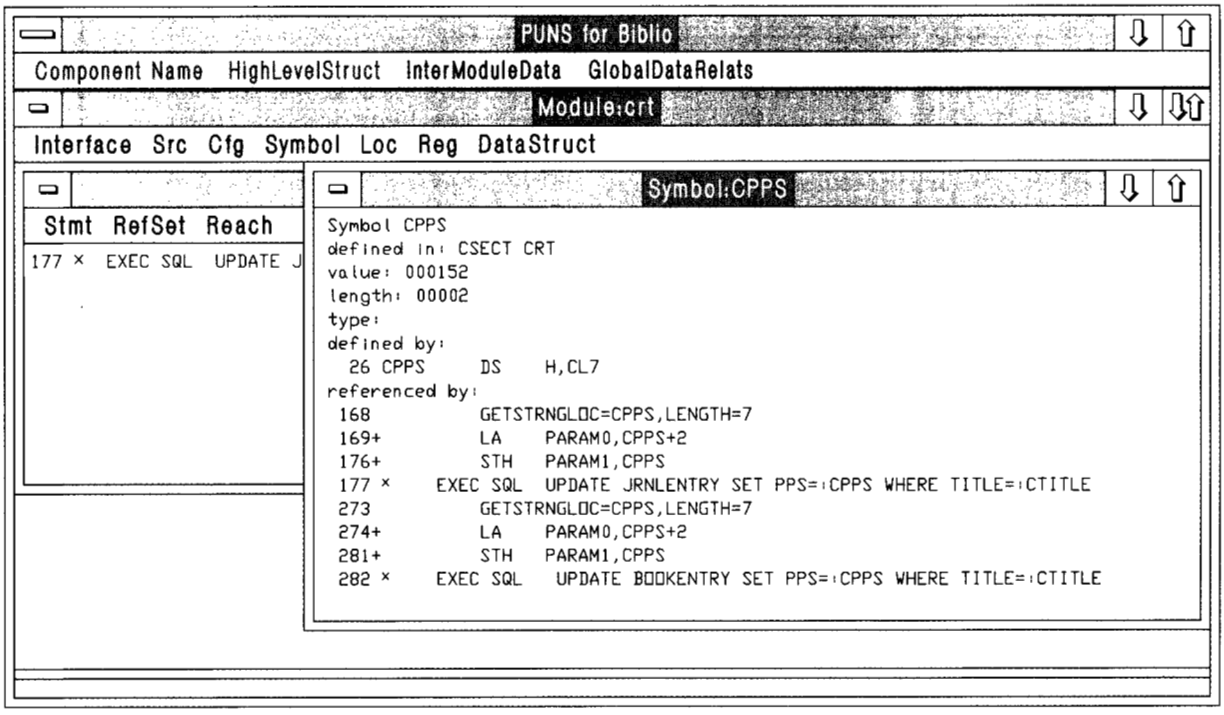
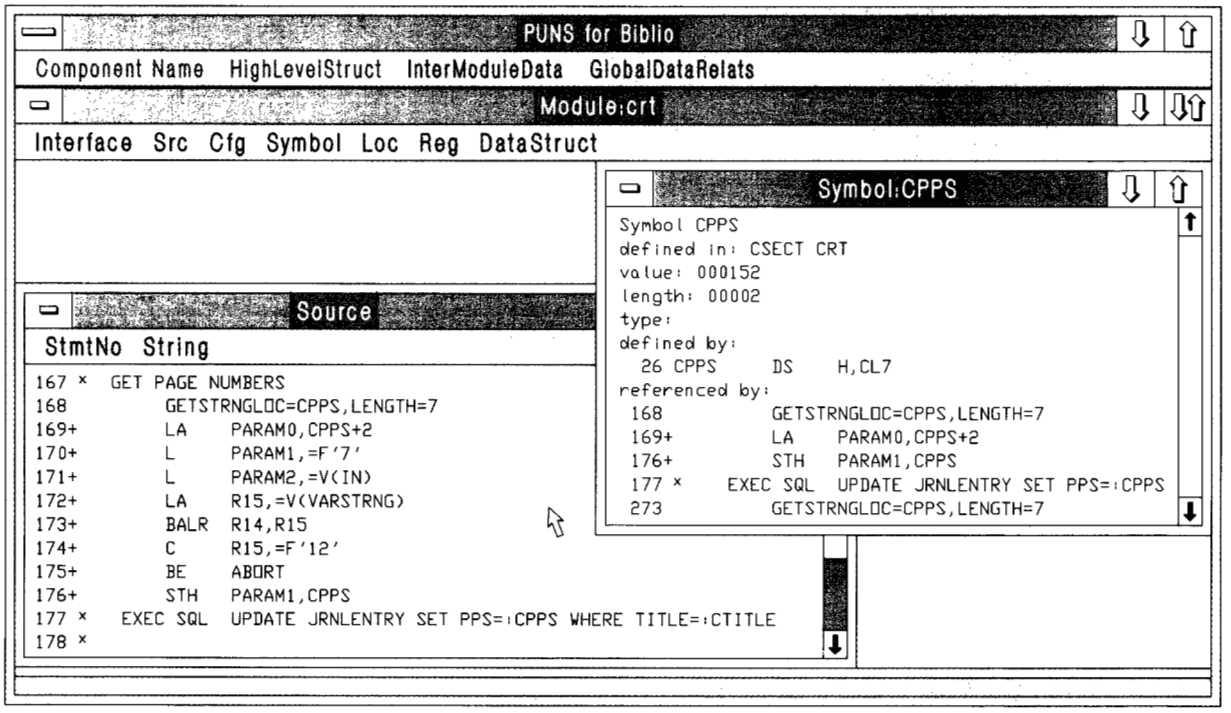


Figure 8 CRT module window



where the symbol CPPS is referenced in a macroinstruction as the operand of the keyword parameter LOC. The expansion of this macroinstruction causes the following actions: (1) it loads the address of the symbol CPPS into a register; (2) a constant value, specified as the operand of the other keyword LENGTH is loaded into another register; and (3) control passes to a subprogram VARSTRNG. When the subprogram returns, the value in the other register is stored into the first half word associated with the symbol CPPS. The programmer thinks that this macro invokes a subroutine that reads a variable-length character string (which is terminated by a special character) from a file and places the character string into the specified location, providing only as many characters as requested. The length specification given is 7. If the length specification indicates a maximum number of characters to be read, the page specification for the reference in which the user is interested would have been truncated. However, if the length specification indicates a minimum number of characters to be provided (including padding with blanks, if fewer are available from the input stream), the page specification has probably been processed properly here, and the problem lies elsewhere. To determine which is the case, the programmer must look at the subprogram VARSTRNG.

On the high-level structure chart for the *biblio* program, the programmer points the mouse at the module VARSTRNG and double-clicks the left button. A window for the module VARSTRNG appears. The programmer first explores the interface of this module to other modules by selecting the interface option on the menu bar of the module window. An interface window appears with menu options: Prologue, ESD, EntryData, ExitData. The programmer selects the *Prologue* option, and a window containing the comments that appear in the program prior to the first storage allocating statement appears. The interface and prologue windows are shown in Figure 9. The prologue seems to indicate that the length parameter specifies a maximum length. However, the programmer decides to investigate the logic of the VARSTRNG subprogram to be sure that the comment accurately reflects what the module does. The programmer displays the control flow graph for the subprogram by selecting the *Cfg* option on the menu bar for the module and displays the source for the module by selecting the *Src* option on the menu bar for the

Figure 9 VARSTRNG interface and prologue windows

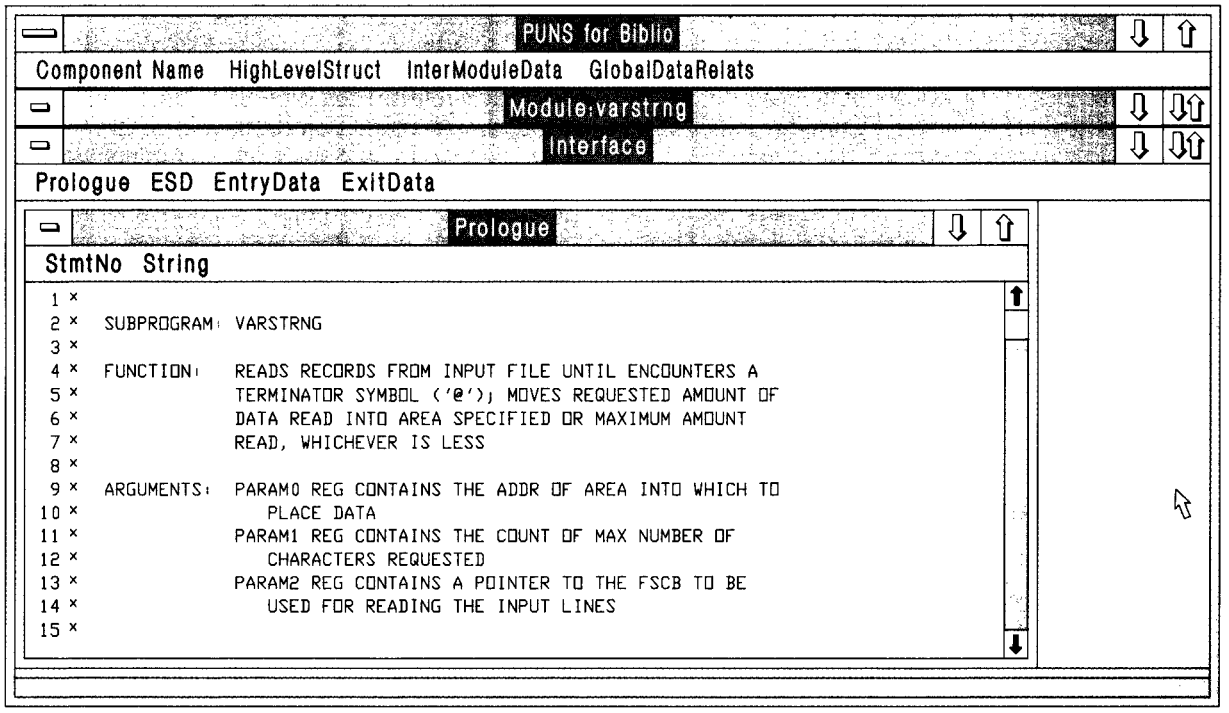
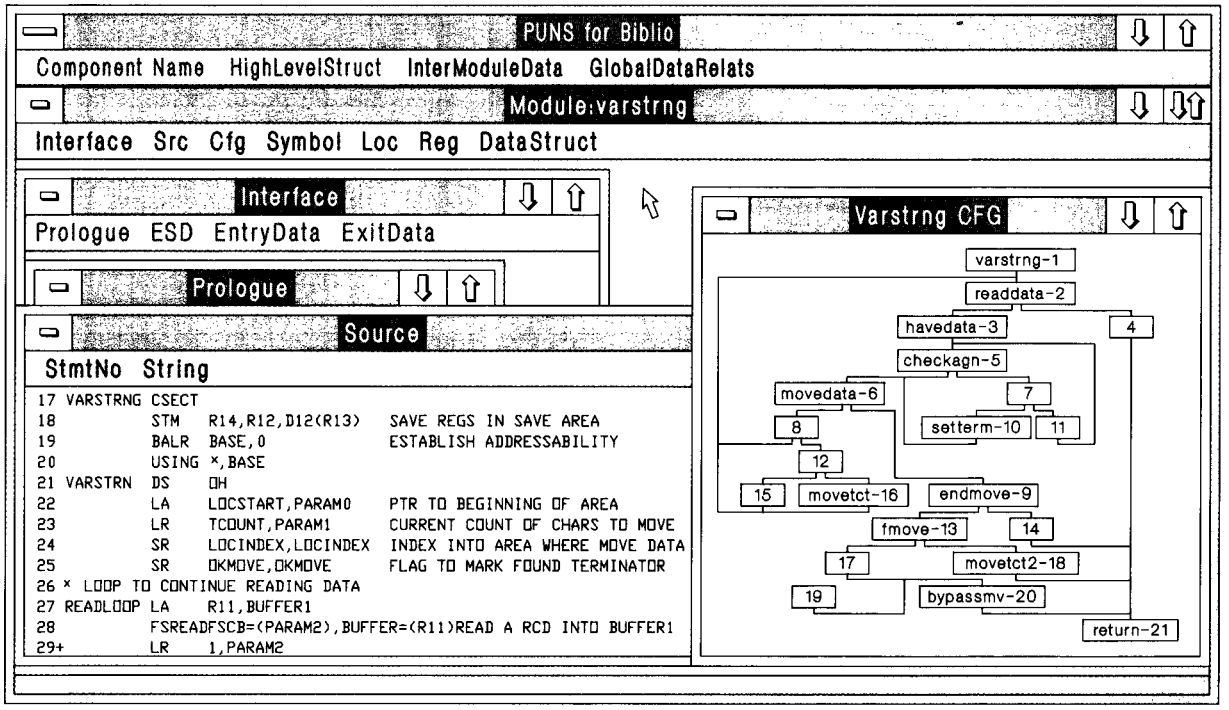


Figure 10 VARSTRNG module window



module. Figure 10 shows the module window and contained windows at this point. The programmer is particularly interested in the use of the length parameter and selects the *EntryData* option on the menu bar of the interface window to see how the parameter register (register 1, in this case) is used initially within the module. Figure 11 shows the contents of the interface window at this point.

The *EntryData* window shows that register 1 is used to set register 5 (TCOUNT) in the subprogram. Thus register 5 is now an alias for register 1, and the programmer explores how register 5 is used in the module. By pointing the mouse at statement 23 in the *EntryData* window (which is the statement that assigns the value in register 1 to register 5) and by double-clicking the left button, the programmer indicates an interest in statement 23 as an object. A window appears for statement 23. The particular concern of the programmer is in how register 5 is used in the module. Thus the *RefSet* option is selected on the menu bar for the statement window for statement 23. A window containing information about where the value placed in register 5 is referenced in the program appears. Figure 12 shows the contents of this window. Using the statement number and node information given in this window, along with the control flow graph and source for the module, the programmer explores the subprogram. From the prologue, one finds that the initial impression that the length parameter specifies a maximum is correct.

At this point, the programmer may wish to explore the other updating of the database to ensure that no truncation occurs there. Once the error(s) in the *biblio* program have been found, the programmer can correct them and test the program.

The schema captures many of the low-level relationships that exist within a program.

User interface summary. The sample scenario just described is intended to provide a feeling for the functions available in the user interface. This sample scenario illustrates the two governing principles followed in the development of the user interface: (1) organizing information about the program around the notion of sets of objects and (2) supporting the point-click technique for exploring an object and moving between objects. In the sample scenario, several of the types of objects supported by the user interface have been illustrated (the component object, the database object, the database object object, the database attribute object, the module object, the module symbol object, the module statement object, and the module interface object). Windows to support these objects and to support information about these objects are also illustrated. The point-click technique for navigation is used extensively.

PUNS repository component

There are three elements of the PUNS repository component: (1) a schema for the repository, (2) a set of routines to load the repository with data, and (3) a set of routines to pull from the repository the appropriate information to satisfy high-level queries from the user interface component. The schema uses an entity-attribute-relationship (EAR) model. The schema captures many of the low-level relationships that exist within a program. The set of routines that load the repository are of three types: (1) a set of routines that scan assembler listings (outputs from the assembler) and load a portion of the repository with information directly discernable from the listing information, (2) a set of routines that are environment-specific (the environment in which the code runs, with a knowledge of ways in which services provided by the environment are invoked) which extract/update information about global objects associated with each assembler listing, and (3) a set of routines that do control flow and data-flow analysis on the information available in the repository. The routines that respond to queries from the workstation interpret each query to determine what data are needed from the repository to satisfy the query. These routines may need to analyze some of the extracted information to determine additional data to extract from the repository. Each of these elements of the repository component is now described in greater detail.

Schema for the repository. The model used for PUNS is an entity-attribute-relationship model. The schema can be depicted using a graph in which each

Figure 11 VARSTRNG interface window

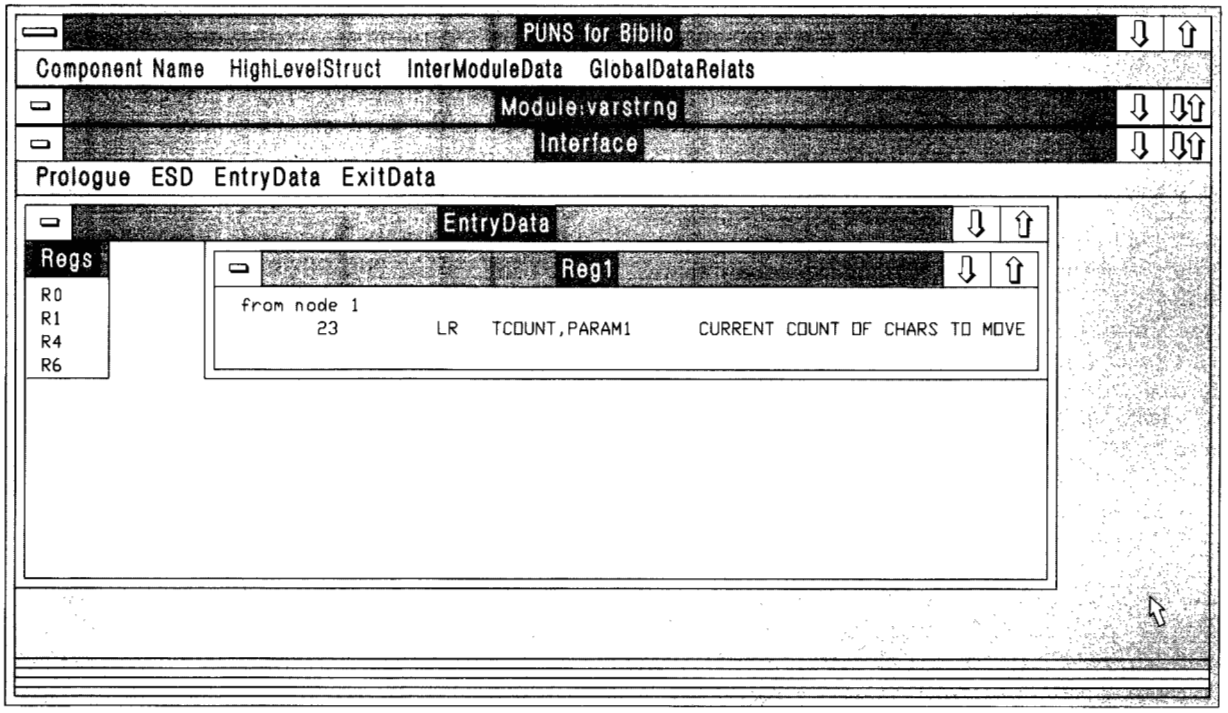
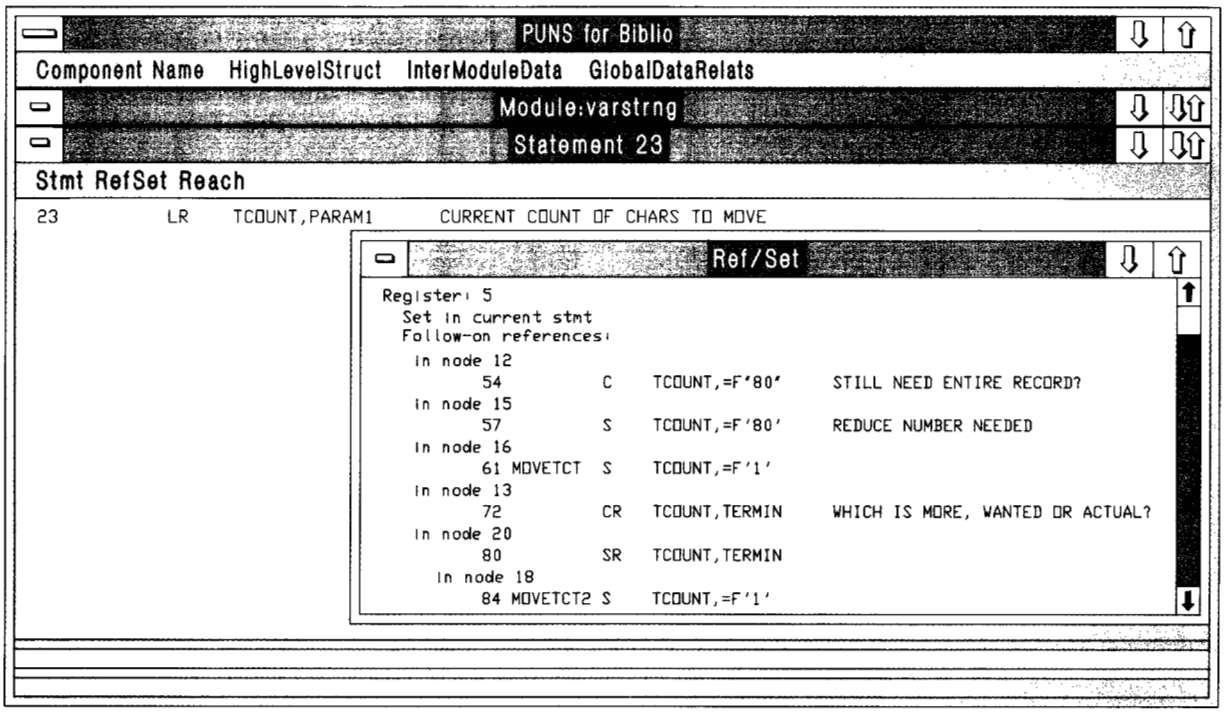


Figure 12 VARSTRNG statement window



node represents an entity that may have attributes associated with it. The directed arcs in the graph represent the relationships between entities. A *relationship* exists between an entity that is a member of a set owned and the entity that owns the set. The owner of the set is shown at the tail of the directed arc, and the member of the set is shown at the head of the directed arc. Each one of the nodes represents a class, and there may be many entity instances each of which will belong to the class. As an example, consider an entity that represents a statement in a higher-level language. In a particular program, there would exist many statements. Each of these statements would be represented in the database by an entity instance as defined by the entity *statement* in the schema. The entity instances would differ in their associated attributes. Each unique statement would have at least one unique attribute (where *attribute* may include the sets owned by the entity instance). In the specific EAR model used by PUNS, the relationships are restricted to one-to-many relationships.

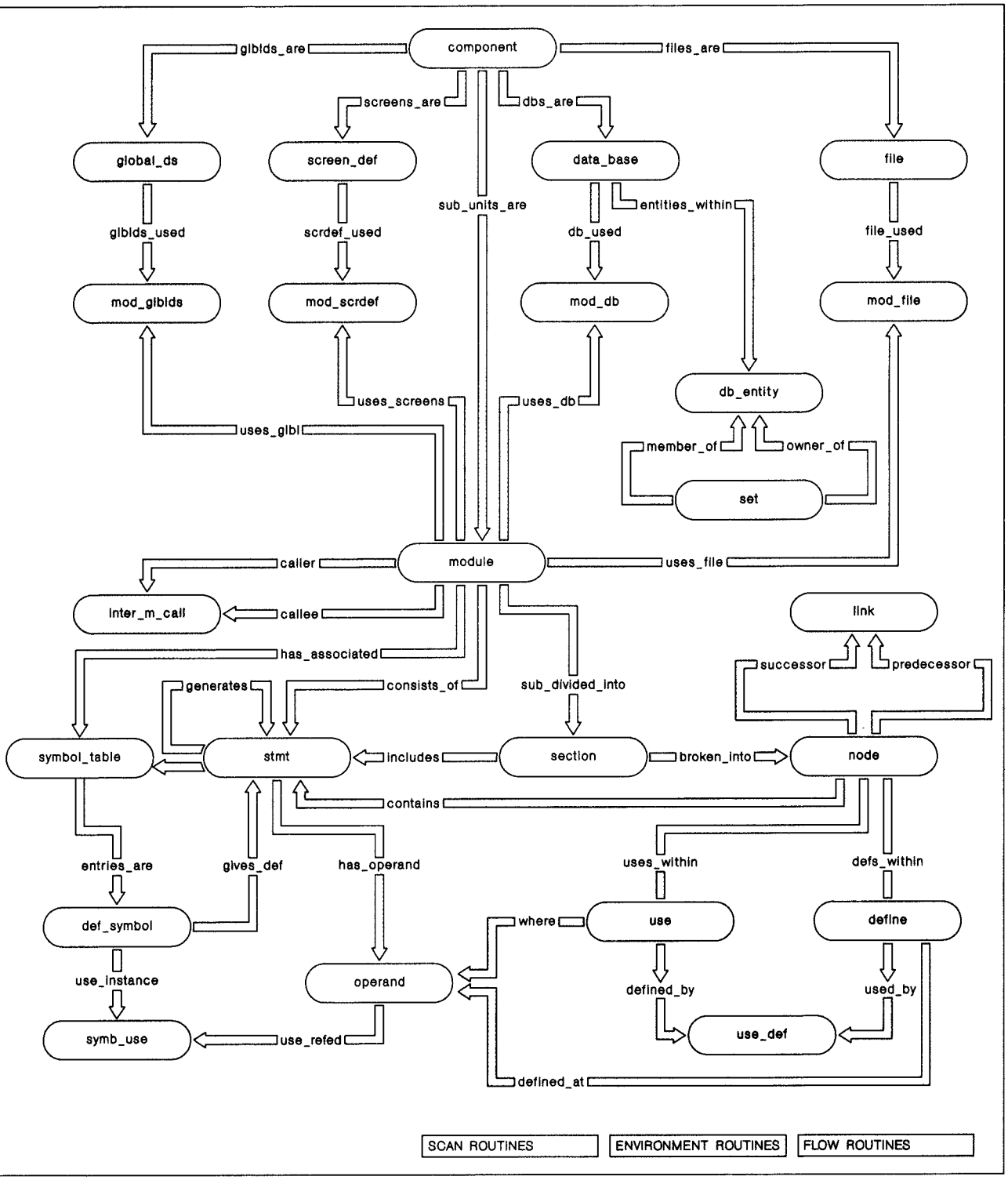
The simplified schema for the repository component of PUNS is shown in Figure 13. Different colors are used to represent different entities. The coloring indicates the routine that created each entity (and thus when and how the information represented by the entity has been derived). The arcs are labeled to specify the relationships which are specified for the schema. The arcs are also colored to indicate which routine established the relationship indicated by the arc. The schema is simplified to remove details that are unimportant to the understanding of the functioning of the tool but necessary to handle idiosyncracies in the language supported. Discussed next are the entities shown in the schema.

The *component* entity serves as the consolidating entity for all component-wide information. If the repository is established to represent exactly one component, there will be only a single instance of the entity in the repository. The *component* entity owns collections of entity instances that represent the basic objects which make up the component: modules, databases, files, screen definitions, and global data structures. For ease of reference, each type of object, as represented by an instance of an entity for that object, is owned by the *component* entity via a unique set. The *component* entity itself would contain the name of the component and might contain functional or ownership information that is solicited from humans at the time the database is established. The sets that the *component* entity owns are: *files_are* with an entity representing a file as the member, *dbs_are* with an entity representing a database as the member, *sub_units_are* with an entity representing a module (generally an assembly unit) as the member, *screens_are* with an entity representing a screen definition as the member, and *glblds_are* with an entity representing a global data structure as the member.

The *module* entity represents a unifying entity for a particular assembly that results in a single module to consider. The only attribute is file name for the source for the assembly. This entity owns a number of sets, many of which relate the module to objects within the module as described later in this paper. The *module* entity is connected to the global domain of the *component* entity. It belongs to the *sub_units_are* set, which is owned directly by the *component* entity. Also defined at the component level are entities that represent objects that may be used globally, i.e., in more than one module. Because a global object may be used by many modules and a module may use many global objects of a particular type, a set of *connector* entities are shown in the graph to resolve the many-to-many relationships. The *module* entity owns a *connector* entity via a set relationship, and the entity representing the global object owns the *connector* entity via a different set relationship. This allows the many-to-many relationships to be established. The sets owned by the *module* entity in this context are: *uses_glbl* to relate a module to a global data structure (connector entity is *mod_glblds*), *uses_screens* to relate a module to a screen definition (connector entity is *mod_scrdef*), *uses_db* to relate a module to a database (connector entity is *mod_db*), and *uses_file* to relate a module to a file used by the module (connector entity is *mod_file*).

In this version of the schema, the only connection between the global objects and the modules that act upon the global objects is via the sets that relate the *module* entity to each of the entities for these global objects. Certainly there are finer relationships that could be expressed in the schema for any of the objects: the information specifying where within the module the object is referenced, the type of reference, and whether the module in fact defines the object. In the schema, we opted to walk through the repository (i.e., knowing the module, find the references to the global object through the local symbol table and determine the reference types and definitions) to determine this information as it was needed, rather than capturing it in the database at the time the database is established. The schema also shows

Figure 13 PUNS repository schema



the *module* entity owning two sets (*caller* and *callee*) that both have as a member the *inter_m_call* entity. This structure captures the use of one *module* entity instance by another *module* entity instance. The *inter_m_call* entity contains a pointer to a *stmt* entity instance representing the point at which the call is made. The *global_ds*, *screen_def*, and *file* entities are handled in a similar fashion in the schema.

The *data_base* entity, which represents a global database is connected in a similar fashion to the modules that reference the database. However, as elements within the database are distinguishable and may serve as focus points for an exploration session, they have been abstracted from the schema definition of the database and exist as separate entities. The *data_base* entity has a textual specification of its schema as an attribute. It owns the *db_used* set for connection to modules (via the *mod_db* entity). The *data_base* entity also owns an *entities_within* set, whose members are the *db_entity* entity. In order to capture the relationships between *db_entity* entities, the *db_entity* entity owns two sets, the *owner_of* set and the *member_of* set. The member of each of these sets is the *set* entity. Such an organization resolves the many-to-many relationships that may exist between entities in the database. In representing a referenced database in the PUNS repository, the EAR model has been selected to describe the referenced database. For a referenced relational database, each *db_entity* entity represents a table, each *set* entity represents a key used to move from table to table. For a referenced hierarchical database, each *db_entity* entity represents a record, each *set* entity represents a parent-child relationship.

The *module* entity is described above in terms of its interconnection with other global objects. As the unifying element for an assembly unit, the *module* entity owns several sets: (1) the *sub_divided_into* set, which associates entities representing individual section definitions within the assembly, i.e., the *section* entity, with the *module* entity; (2) the *consists_of* set, which associates entities representing individual statements in the assembly, i.e., the *stmt* entity, to the *module* entity; and (3) the *has_associated* set, which links the entity representing the local symbol table, i.e., the *symbol_table* entity, with the *module* entity.

The *symbol_table* entity represents the local symbol table for the module. It owns an *entries_are* set which has as a member the *def_symbol* entity which represents the individual symbols defined in the local symbol table. Each *def_symbol* entity is a member of a *gives_def* set owned by the *stmt* entity, which defines the symbol represented. The *def_symbol* entity also owns the *use_instance* set whose member is the *symb_use* entity, which represents a particular use of the symbol (and resolves an otherwise many-to-many relationship).

The *section* entity represents a control section or a dummy section within the assembly. The *section* entity owns two sets and serves as an organizing vehicle for each defined control or dummy section. The sets are the following: (1) *includes* set, which has as member the *stmt* entity and connects those statements defined within the section with the *section* entity and (2) *broken_into* set, which connects the *section* entity to the *node* entity, which represents the set of statements to be treated as a single node in the control flow graph for a control section.

The *operand* entity represents a single operand. It is singled out, as it is the operand that provides the reference to a symbol and also the use or definition of a variable in the data flow. The *operand* entity owns one set, the *use_refed* set, which associates the *operand* entity with its use of a symbol (and resolves a many-to-many relationship). The *operand* entity also participates as member in three set relationships: (1) *has_operand* set owned by the *stmt* entity, (2) *where* set owned by the *use* entity and relating a use of the variable represented by the operand to the operand, and (3) *defined_at* set, which relates the definition of a variable represented by the operand to the *operand* entity.

The *node* entity represents a division of the module according to control flow. Each *node* entity is connected to other *node* entities (to show the flow of control) via the *predecessor* and *successor* sets and the *link* entity which is a member of both sets. This structure reduces the many-to-many relationship of the flow of control among nodes into one-to-many relationships. The *node* entity is also a member of the *broken_into* set that is owned by the *section* entity. It is an owner of the *contains* set that shows the statements within the node and the owner of two sets, the *uses_within* and *defs_within* sets, which relate specific definitions and uses of variables (for data flow) that occur within the node. The definition of a variable is represented by the *define* entity, which is connected to uses of the variable via the *used_by* set, and to the *operand* entity which provides the definition via the *defined_at* set. The use of a variable

within a node is represented by the *use* entity. The *use* entity is connected to definitions for the variable it represents via the *defined_by* set and to the *operand* entity that represents the use via the *where* set. The *use_def* entity connects uses of variables to definitions and is needed to resolve a many-to-many relationship.

Load routines. There are three types of routines that are used to load the PUNS repository: (1) routines to scan the input statements for the modules that make up the component to be described in the PUNS repository, (2) routines to deal with the environmental services used by the modules, and (3) routines to deal with representing the control and data-flow aspects of the modules. These three types of routines are now presented under the headings scan, environment, and flow.

Scan. The routines that scan the individual assembler listings for the component for which the repository is being established do the following: set up the *module* entity instance; set up the *symbol_table* entity instance; set up *section* entity instances for each control or dummy section encountered; set up a *stmt* entity instance for each statement encountered (from macro expansion information contained within the assembly listing also establishing the *generates* set); establish a *def_symbol* entity instance for each symbol defined in a statement; identify operands for each statement that has such and creates appropriate *operand* entity instances; and set up the appropriate *symb_use* entity instance for each operand reference.

As each of the entity instances is created it is placed in the proper set(s) to express the relationship of the entity instance to other entity instances that exist within the repository. In certain cases (for example with the *generates*, *consists_of*, *includes*, and *has_operands* relationships), it is necessary for the routines to establish a context for the processing of information so that as new entity instances are created, the necessary information to connect them to existing entity instances is available.

Attributes are established for each of the entity instances as the instance is created. The collection of attributes for a particular entity instance may not be complete at this time. The scan routines are written to utilize only information about the syntax of the language in which the module is expressed. These routines do not utilize semantic information about the statements of the language used. The semantic information provided by the statements is captured in the repository by the flow routines. For example, the *operand* entity instance contains an attribute that identifies where the operand represents a use or redefinition of the variable represented by the operand. In order to determine how the operand is used, the meaning of the operation on the operand must be established. The scan routines do not deal with this type of information and thus the use/redef attribute information is not established at this time.

When dealing with an assembler language, the use of services provided in the environment is evidenced in the assembly listing by the use of macros.

Environment. When dealing with an assembler language, the use of services provided in the environment (e.g., linking, use of files, and use of databases) is evidenced in the assembly listing by the use of macros. The environment routines are aware of the macros supported within the environments in which the code is to run. By locating the macro calls that have been used (by inspecting the created *stmt* entity instances) and evaluating the paramaters specified on each macro call, these environment routines can create the global objects associated with the component and establish the relationships between these global objects and the particular *module* entity instances that reference the global objects. There may exist several sets of environmental routines that must be run, one set for each of the support subsystems called upon by the modules within the component.

Flow. The routines that establish the control and data-flow entity instances within the PUNS repository must be written to deal with the semantic information associated with each executable statement. This information can either be encoded into the routines or the routines can be written to operate on a separate body of knowledge that provides the semantic information for each executable statement. The second option was utilized in setting up the PUNS re-

pository. A database—the semantic database—was built that contained an entity instance for each executable statement type. The entities were subdivided into those representing data manipulation operations, those representing conditional or unconditional branch operations, and those representing branch-link operations.

In an assembler setting, it may be impossible to resolve completely the control flow and branch-link relationships.

The first task for the flow routines is to identify the executable statements and classify each according to the types specified by the entity instances in the semantic database. Once the branch operations are identified, the basic blocks (nodes) for each module can be established, and by examining branch targets the control flow can be represented by using link-entity instances to interconnect the basic blocks (nodes). The possible entry points for each control section (each represented by a *node* entity instance) can be identified and represented in the repository as can the exit points for each control section.

The next task for the flow routines is to determine the data flow within each node. The semantic database provides the necessary information to classify operands as to use or redefinition. The appropriate *use* and *define* entity instances can be created and associated with the *node* entity instances. These entity instances can be associated with the appropriate operands. Where the use of a data item is satisfied by a prior redefine within the block, a *use_def_connect* entity instance can also be created. Once intrablock data flow is established, interblock data flow can also be established, and the appropriate *use_def_connect* entity instances can be created. In the PUNS context, all data-flow analysis should be optimistic. That is, if it appears that a particular data-flow assertion may be true, it should be captured. In doing this, a weighting can be used to indicate that the assertion is probably true, although not necessarily exact.

The final task for the flow routines is to deal with the branch-link operations and, where possible, to resolve these branch-links within or across the modules and build the appropriate entity instances to express these interconnections.

In an assembler setting, it may be impossible to resolve completely the control flow and branch-link relationships. Inasmuch as the interblock data flow depends on the established control flow, the data flow may also be incompletely determined. Although the current PUNS implementation accepts this incomplete resolution, a better solution is to involve a human expert during the running of the flow routines to achieve a more complete resolution. For branch and branch-link types, where the flow routines are unable to detect a target, the human expert can be queried to provide a target. The information from the expert can be treated using a weighting factor so that the future user of PUNS knows that the information is a best guess.

Query-response routines. The PUNS user interface queries the PUNS repository component to obtain information necessary for developing the windows that are presented to the user via the user interface. These queries may require interrogation of many of the repository-entity instances and the following of many relationships or may be able to be satisfied by attention to only one or a few entity instances. The repository schema should be developed to allow those queries that will be issued frequently by the user interface to be satisfied by accessing only one or a few entity instances. Queries that will be issued more infrequently or at points at which user activities can be overlapped with a wait for response from the query can require interrogation of many repository-entity instances. The current implementation of the PUNS repository has not been optimized to minimize the time for queries that are frequently issued by the PUNS user interface component.

We now discuss a few of the queries that are issued by the PUNS user interface component and then sketch the method used to develop the appropriate information to respond to each query. Our intention is to give a flavor for the types of queries and show how the repository component can navigate the repository schema to provide the requested information.

High-level structure chart query. To satisfy the query to provide a high-level structure chart for the component, each one of the *module* entity instances

existing in the subunits relationship to the *component* entity instance must be enumerated. The caller-callee relationship between the modules is captured by the *inter_m_call* entity and the caller and callee relationships. For each caller module, the appropriate callees must be specified.

Global data relations query. To satisfy the query to provide a graph description showing the use of the global data elements by modules, each one of the global data element entity instances must be enumerated. For each global data element entity instance (*global_ds, screen_def, file, data_base*) the modules to which it is connected via the *xxx_used* and *uses_xxx* sets and the connector entity instances must also be enumerated. If a relationship is defined for the use of a global data element and a separate relationship for the definition of a global data element, the enumeration of modules can specify a direction for the arcs.

Prologue for interface query. To satisfy this query, it is necessary to specify which statements within a module are part of the prologue for the module. Currently, the prologue is interpreted to mean those statements in a module that precede the first storage-allocating statement within the module. Because the statements forming the prologue are contiguous, a specification of the first and last statement numbers and a count of the number of statements in the prologue suffices. The *module* entity for the module for which the information is desired must be located. Then each of the *stmt* entities in the set *consists_of* owned by the *module* entity must be checked until an entity representing a storage-allocating statement is encountered. All checked entities prior to that representing a storage-allocating statement are part of the prologue. The data representation to satisfy the query is then the statement number attribute for the first and last of these entities and a count of the number of entities.

RefSet query for registers within a statement within a module. This query provides reference and set information for registers used within a statement in a particular module. For each register used in the statement, the query identifies the use of the register (reference or set). For a reference register, the query supplies all statements (and nodes within which the statements exist) that previously set the register in the module. For a set register, the query supplies all statements (and nodes within which the statements exist) that reference the register following the set.

The entity representing the module (*module* entity) and the entity representing the statement (*stmt* entity) within the module must be located. Then each of the operands of the statement (*operand* entity) must be inspected to see whether it represents a register. If the *operand* entity represents a register, then the membership of the *operand* entity in either the *where* or *defined_at* sets indicates whether the register is set or referenced in the statement. For purposes of illustration, assume that the operand is set (defined) in the statement under consideration. The *operand* entity then belongs to a *defined_at* set owned by a *define* entity. By looking at each *use_def* entity that is owned by the *define* entity in the *used_by* set, one can find the *use* entity that owns the *use_def* entity in its *defined_by* set. Then tracing back to the *operand* entity associated with the *use* entity, the *stmt* entity that owns the *operand* entity via the *has_operand* set can be determined. To find the node in which the statement exists, one must find the owner of the *contains* set to which the *stmt* belongs. This owner will be the *node* entity representing the node in which the statement exists. Repeating this procedure for each of the *use* entities linked to the *define* entity via the *use_def* entity, all the necessary information for an operand that is set in a statement can be found. For an operand that is referenced in a statement, one begins with the *use* entity and finds the *define* entity linked to the *use* entity via the *use_def* entity. From the *define* entity, one can determine the statement and node information needed to satisfy this query.

Current research

There are many directions that extensions to the current work on PUNS might take. These include performance issues, dynamic information updating, a logging of explorations with a replay option, a checkpoint/restart facility, a notebook facility for recording discoveries, and experimentation with the current prototype.

Performance. To achieve reasonable performance for a system such as PUNS requires an analysis of frequently issued queries and of the expectations of the user as to response for different query types. The repository can be organized so that those queries that are frequently issued or have a user expectation for immediate response can be quickly satisfied. That is, all the analysis to answer the query is done during the loading of the repository. Queries that are less frequently issued or are perceived by the user either to take time to answer or to be capable of being

overlapped with other user activities can be satisfied by analysis on the fly by using the current data in the repository as the source for analysis routines to isolate what is actually needed to satisfy the query.

At one extreme is a repository whose schema allows immediate satisfaction of all queries. A repository built using such a schema would—for even a very small component—be enormous in size. However, the response time would be very short, even on a moderately powerful host. At the other extreme is a repository schema that captures only the most significant relationships; all other relationships must be dynamically derived. A repository built using such a schema would be compact and not require a significant amount of space. However, the number of instructions that would have to be executed to satisfy any but the most basic query would be very large. Only if a very high performance computer were used as host, would the resulting response time be acceptable. Thus a compromise between these extremes in terms of space is indicated.

If a high-performance host is available, a minimal schema can be used, saving disk space for repository storage and allowing the repository to be more quickly loaded.

One interesting point about the design of PUNS is that different repository components can be used, depending on the availability of resources. If a high-performance host is available, a minimal schema can be used, saving disk space for repository storage and allowing the repository to be more quickly loaded. On the other hand, if a low-performance host, such as a workstation with sufficient storage is used, a more comprehensive schema can be used. It takes more disk space and a longer set-up time, but performance during a use of the PUNS user interface component would be acceptable. The intended users for PUNS and the types of available hardware need to be taken into account in the research to produce a tool that will satisfy the performance needs of the users.

Dynamic information updating. The PUNS tool as described does not allow for the dynamic updating of information. The PUNS repository is set up prior to a user session using the current versions of modules of the component and not changed during the user session.

However, there is much to be gained by allowing the user to update incomplete information on the component during an exploratory session. There will always be relationships that cannot be determined via the static analysis done in setting up the repository. Allowing the user to add information to the repository raises the question of the accuracy of the information provided, particularly if more than one user has access to the PUNS repository for a particular component. Should the added information be held as user-specific information and not shared with others using the repository? Should the information have a probability associated with it, so that if it is presented to a user it is flagged as only being a possible truth? What if the information presented by a user affects relationships previously determined by static analysis on the basis of incomplete information?

The question of the need to update the PUNS repository also arises in the case of allowing the user to investigate the impact of changes in the module versions during an exploratory session. Each change would have an impact on the relationships represented in the repository, and there would be a need for incremental updating of the repository. The extent of such incremental updating would be a function of the number of relationships explicitly expressed in the repository. If most of the relationships are derived dynamically from a minimal set of relationships expressed in the repository, the updating would be minimal. However, if almost all relationships were expressed in the repository, a significant amount of reanalysis and updating would be necessary prior to continuing to use the repository.

Logging of an exploration. The current prototype for PUNS has no facility to record the use of the PUNS user interface for a particular exploration. Such a recording could prove to be very beneficial. A programmer using the system who is interrupted during a task could quickly replay the recording of that

prior exploration upon returning to the task to set the context for continuing the task. Even more beneficial would be a facility to allow a replay of an exploration session with interruption to alter the exploration at a point where it went astray or where more information is needed that was not gathered in the initial exploration.

Checkpoint and restart. Not as extensive as logging, the ability to checkpoint the PUNS user interface during an exploration session and then restart at a later point is also a needed extension and subject for further study. Given this facility, the system could be checkpointed at predetermined or user-selected points so that, in case of interruption—a loss of machine facility or a hopeless deadend in an exploration—the user could restart without the need to repeat the entire exploration.

Notebook facility. A user working through any exploration of a system usually takes notes of the significant issues uncovered or needing to be resolved. These notes generally include information about the current point in the exploration and how the information available at that point has led to a particular conclusion. The PUNS user interface provides support for the exploration phase of a system, and the user also needs support for the notetaking activity. It should be possible for the user to select certain pieces of information currently available on the screen, place them into an on-line notebook (probably represented as a viewable, scrollable window), indicate relationships between the items of information, and add comments. What is placed in the notebook may be fairly static information. However, it may also be quite dynamic, thereby allowing a sequence of windows to be captured as the notebook is written and replayed when the notebook is read. The notebook need not be simply a linear text. Rather, it could encompass many of the aspects of hypertext.[14]

Experimentation. The PUNS prototype described in this paper has been demonstrated extensively. However, it has not been used in a production programming environment. There is a need to define, conduct, and analyze experiments for using the tool in such a realistic environment. Such experiments can provide much feedback to the research team from which would come an understanding of such things as relationships that have been ignored, performance, extensions to the tool, and areas of the tool that are not user friendly. Such experiments are contemplated and we hope they will be carried out.

Concluding remarks

PUNS is a tool to support programmers in their effort to understand a program. The PUNS user interface presents many views of a program under consideration and allows the programmer to gain an overview of the program and to explore any aspect of the program in depth. The interface is so structured that the user can move between the overview type information and the detailed levels using a simple point-click interaction with the PUNS tool. The PUNS repository component is designed to provide the necessary information to the user interface component, so that the many views can be made available when requested.

With continuing development and extension the PUNS tool described in this paper should allow programmers to reduce the proportion of time they spend on the program understanding task while increasing their level of comprehension of the program. As such, it should be a valuable tool in any software maintainer's toolbox.

It is interesting to speculate on potential adjacent areas of research. The availability of a tool such as PUNS should prove useful to the community of scientists attempting to understand how people comprehend programs. By recording exploration sessions using PUNS, a machine readable, nonintrusively obtained set of data on sequences of actions taken by an individual trying to comprehend a portion of a program to accomplish a particular task is available for analysis. If the scope of the PUNS tool is sufficient, any desired set of relationships can be explored in any logical sequence, and the tool will not constrain the explorer. The data obtained by recording tool exploration will not be contaminated by the facilities provided by the tool.

Not only can the recording of exploratory sessions provide detailed input for the study of how people go about certain tasks involving comprehension, but also the tool can provide a vehicle for exploring whether a certain set of steps are necessary and sufficient for a particular task. By adding on top of PUNS a component that constrains the user of the tool to following certain relationships at any particular point in the exploration only, the tool could be programmed to force the user into following a certain task structure. Experiments using different task structures could yield valuable information about task content.

A third area of associated research in which a tool such as PUNS could prove useful is that of reverse engineering, that is, the attempt to capture design information from existing code. The PUNS repository component provides much of the relationship information necessary to begin a reverse engineering task. The user interface could be modified (or extended) to maximize user input for cases in which there is insufficient information to make the reverse-engineering decisions. Thus, human guided reverse engineering could be done easily and interactively.

Acknowledgments

I want to thank Don Pazel and Ashok Malhotra for their technical assistance in this work. I thank Tom Corbi for his vision of what PUNS could be and for his continuing enthusiasm and support for the work as it has progressed.

Cited references

1. G. Parikh, *Handbook of Software Maintenance*, John Wiley & Sons, Inc., New York (1986), p. 14.
2. B. P. Lientz, E. B. Swanson, and G. E. Tomkins, "Characteristics of application software maintenance," *Communications of the ACM* **21,** No. 6, 466–471 (June 1978).
3. E. B. Swanson, "The dimension of maintenance," *Proceedings of the Second International Conference on Software Engineering*, San Francisco (October 1976), pp. 492–497.
4. R. K. Fjeldstad and W. T. Hamlen, "Application program maintenance-report to our respondents," in G. Parikh and N. Zvegintzov, Editors, *Tutorial on Software Maintenance*, IEEE Computer Society Press, Silver Springs, MD (1983), pp. 13–27.
5. T. A. Standish, "An essay on software reuse," *IEEE Transactions on Software Engineering* **SE-10,** No. 5, 494–497 (September 1984).
6. F. P. Brooks, Jr., "No silver bullet: Essence and accidents of software engineering," *IEEE Computer* **20,** No. 4, 10–19 (April 1987).
7. A. Malhotra, H. M. Markowitz, and D. P. Pazel, "EAS-E: An integrated approach to application development," *ACM Transactions on Database Systems* **8,** No. 4, 515–542 (December 1983).
8. M. Moriconi and D. F. Hare, "The PegaSys System: Pictures as formal documentation of large," *ACM Transactions on Programming Languages and Systems* **8,** No. 4, 524–546 (October 1986).
9. S. Reiss, "PECAN: Program development systems that support multiple views," *IEEE Transactions on Software Engineering* **SE-11,** No. 3, 30–41 (March 1985).
10. J. Ambras and V. O'Day, "Microscope: A program analysis system," *Proceedings of Hawaii International Conference on System Sciences-20* (January 1987), pp. 71–81.
11. Y. Chen and C. V. Ramamoorthy, "The C information abstractor," *COMPSAC 86*, Chicago (October 1986), pp. 291–298.
12. J. S. Collofello and J. W. Blaylock, "Syntactic information useful for software maintenance," *National Computer Conference 85*, Chicago (July 1985), pp. 547–553.
13. W. Teitelman, *Interlisp Reference Manual*, Xerox PARC, Palo Alto, CA (1978).
14. J. Conklin, "Hypertext," *IEEE Computer* **20,** No. 9, 17–41 (September 1987).

Linore Cleveland *IBM Research Division, T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Ms. Cleveland worked for IBM from 1963–1969, teaching in and managing an internal programmer education group in Poughkeepsie, New York. After leaving IBM, she taught at Vassar College in Poughkeepsie and Polytechnic University in Brooklyn, New York, and served as chairman of Vassar's Computer Science Studies Program. She also spent two years as a guest researcher at the Tokyo Research Laboratory of IBM Japan. She rejoined IBM in late 1986. She has been working in a program understanding group developing demonstration vehicles and prototypes of a tool to assist programmers in understanding programs written in old code. Ms. Cleveland has a B.S. in mathematics from Michigan State University, East Lansing, Michigan, an M.S. in computer science from Polytechnic University, and is currently a candidate for the Ph.D. in computer science at Polytechnic University.

Reprint Order No. G321-5362.