RE-Analyzer: From source code to structured analysis

by A. B. O'Hare
E. W. Troan

The RE-Analyzer is an automated, reverse engineering system providing a high level of integration with a computer-aided software engineering (CASE) tool. Specifically, legacy code is transformed into abstractions within a structured analysis methodology. The abstractions are based on data flow diagrams, state transition diagrams, and entity-relationship data models. Since the resulting abstractions can be browsed and modified within a CASE tool environment, a broad range of software engineering activities are supported, including program understanding, reengineering, and redocumentation. In addition, diagram complexity is reduced through the application of* control partitioning*: an algorithmic technique for managing complexity by partitioning source code modules into smaller yet semantically coherent units. This approach also preserves the information content of the original source code. It is in contrast to other reverse engineering techniques that produce only structure charts and thus suffer from loss of information, unmanaged complexity, and a lack of correspondence to structured analysis abstractions. The RE-Analyzer has been implemented and currently supports the reverse engineering of software written in the C language. It has been integrated with a CASE tool based on the VIEWS method.

Although the use of terms such as *reverse engineering* and *reengineering* is relatively recent,[1] the types of activities they denote can be traced back to the first time someone was required to change a program that someone else had created. That part of the software development life cycle commonly known as maintenance continues to consume the majority of all resources, time, and money spent on software production. Most of us in the software industry are all too familiar with statistics such as: for every dollar spent on software development, nine dollars are spent on maintenance;[2] 55 to 90 percent of the total life-cycle workload is expended on maintenance;[3] and 47 to 62 percent of the total time spent on maintenance activities involves efforts to comprehend the original source code.[4]

As the size of the software increases, so does the time required to comprehend it. As a result, the cycle time and cost of making an enhancement have less to do with the projected size of the modification than they do with the total size of the software that is to be updated.

Although good documentation is undoubtedly beneficial, the realities of the problem are that even when design documentation does exist, it is often hopelessly out of date with respect to modifications of the related source code. Even good documentation will be missing details, the importance of which only becomes evident long after the authors of a project have left it. With the development of integrated computer-aided software engineering (CASE) tools, it is possible to provide

a more dynamic form of design documentation for at least some classes of software applications. However, these tools do not address the problem of maintaining the mountains of legacy code with little or no accurate documentation for it in existence today. Unfortunately, the significant advantages that can be achieved over conventional development techniques by using CASE technology are often precluded by this lack of support for legacy code.

Clearly, there is a need for techniques and tools that facilitate the task of understanding software systems for which the most accurate, if not the only, source of information is the original source code.[5] Such tools and techniques will lead to substantial reductions in the amount of time and money spent on software maintenance as well as significant improvements in software quality.

Within the world of CASE tools, the methodology used to represent or model the software system being developed is crucial to both the creation and comprehension of software systems. If the methodology supported by the CASE tool provides effective concepts and representations for constructing a model or design of a software system, it should be similarly effective in helping one to understand that same model. Given this rather basic premise, it seems reasonable to conclude that it would be extremely useful if one could produce a mechanism whereby source code could be automatically transformed into an abstract model using the same representational scheme as that of a CASE tool methodology. Although many researchers and CASE tool vendors have reached the same conclusion, there is enormous variability in the methodology chosen, the transformations made, and the level of integration with a CASE tool.

Most CASE tools support a structured analysis methodology such as those of Gane and Sarson,[6] DeMarco,[7] Hatley and Pirbhai,[8] Ward,[9] or ESML.[10] Most of these methodologies also provide extensions to support the modeling of real-time systems and thus fall within the class of methodologies commonly called structured analysis for real-time systems (SA/RT).[11,12]

SA/RT methodologies involve the modeling of three different aspects of software, i.e., process, control, and data. Each aspect represents a different view or dimension of the system being

TERMINOLOGY

The following definitions (except for that of program understanding) are from Chikofsky and Cross.[1]

Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.

Reverse engineering is the process of analyzing a subject system to
(1) identify the system's components and their interrelationships,
(2) create representations of the system in another form or at a higher level of abstraction.
Two subareas of reverse engineering are *redocumentation* and *design recovery*.

Redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternative views (for example, data flow, data structure, and control flow) intended for a human audience.

Design recovery is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher-level abstractions beyond those obtained directly by examining the system itself.

Restructuring is the transformation from one representation to another at the same relative abstraction level, while preserving the external behavior (functionality and semantics) of the subject system.

Reengineering also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring.

Program understanding is the activity of examining a software system for the express purpose of comprehending all or part of its construction. It is actually an objective rather than a well-defined process and, as such, it is supported by the various reverse engineering processes described above.

modeled. Specifically, processes are modeled using data flow diagrams (DFDs); state transition diagrams (STDs) are used to model control; and entity-relationship (E-R) diagrams or similar notations are used to model the data aspect of software systems. Although these views are clearly distinct with respect to their domains, they are not strictly orthogonal. For example, part of the process model involves both control and data. Similarly, part of the control model involves processes and data.

This paper describes the RE-Analyzer, which was designed and implemented by the authors as part of a larger software development methodology project. The RE-Analyzer automatically reverse engineers source code into graphic and textual representations within a CASE tool supporting an SA/RT methodology. That is, it transforms source code into a set of data flow diagrams, state transition diagrams, and entity-relationship data models within the design database of a CASE tool. Since the resulting representations can be browsed and modified within the CASE tool environment, a broad range of software engineering activities are effectively supported, including program understanding, reengineering, and redocumentation.

Background

Because of the obvious importance of the maintenance problem and, therefore, the importance of reverse engineering activities, a vast number of techniques and tools have been developed throughout the history of computing to aid these activities. These efforts range from the early cross-reference report generators to recent attempts to provide more robust solutions.

A well-known class of reverse engineering aids, commonly called program understanding tools, typically provide on-line, graphic representations of source code, i.e., a control flow graph, a procedure-calling structure graph, and possibly some other dependency graphs. Examples of commercially available tools of this type include Hindsight**, PROCASE**, and Logiscope**. The representations generated by such tools simply filter information and provide little or no abstraction of the underlying source code. That is, analyzing complex source code produces equally complex graphs or other representations. The complexity of the resulting representations is a result of the monolithic nature of the representations themselves. For example, a single control flow graph will be produced for all of the source code being analyzed. For large systems with dozens or even hundreds of subroutines, the resulting graph will be extremely dense. This problem is mitigated to some extent by the use of graph zooming operations and hypertext-like traversals from a node on a graph to the associated source code.

Another class of program understanding tools (e.g., Refine**) constructs a database of information from the source code and supports *ad hoc* queries made to the database. The principal advantage of this approach is that the queries can be saved and then reused on other software systems to identify similar patterns (e.g., error-prone sequences of code). However, any abstraction of the source code must be performed by manually constructing the proper queries. Also, such tools are usually text-based, i.e., they do not provide any standard graphical representations of the software that has been analyzed.

Perhaps the most serious drawback of most program understanding tools is their failure to preserve any of the knowledge or understanding obtained while using the tool. Other than retaining queries, there is no provision for automatically preserving what the user has learned about the subject system. Ironically, by failing to preserve and maintain what is learned about a system during their use, such tools perpetuate part of the very problem they are designed to address.

Another disadvantage of these tools is that the representations they support, if any, are not consistent with those typically used to produce the source code in the first place. Program understanding and forward engineering activities are often treated as separate and unrelated tasks that do not require the same concepts and abstractions. How much better it would be though, to allow both tasks to be carried out using the same basic vocabulary and representational scheme. As suggested above, one clear way to achieve this end would be to use a CASE tool and the methodologies it supports as the primary environment for both forward and reverse engineering activities.

The most common approach to integrating reverse engineering with a CASE tool supporting an SA/RT methodology is to generate a *structure chart* (e.g., C/Rev** used with the Teamwork**

CASE tool). Structure charts represent the architecture of a software system by depicting the modules in a system and their interactions, i.e., which modules are called by other modules, what parameters are passed in a call, and what values, if any, are returned. The basic concepts represented in a structure chart are: modules, call-return interactions among the modules, and parameters and return values associated with each call-return interaction (referred to as a data or control *couple*).[12] Structure charts are particularly useful in making design decisions about the modularization of a system. However, they do not depict any information about the internal structure of modules, the structure of the data, the shared or global data used by the system, or the environment of the system being modeled. Further, a single structure chart is used to represent an entire system since the elements of a structure chart are not decomposable. For any but the smallest software systems, the resulting structure chart rapidly becomes so complex that it must be supplemented with annotated connector symbols and tables to explain their meaning so that the chart can be spread across multiple pages.

A related approach involves producing a variant of flowcharts in addition to structure charts from Pascal code.[13] However, the resulting structure charts provide only structural information (i.e., no data or control couples are produced). Also, the flowcharts provide little more than a one-to-one mapping of source code statements to flowchart objects that yields no significant abstractions.

There have also been some efforts to produce some part of the SA/RT representations, i.e., DFDs, from source code. In one case, a method has been described for deriving DFDs from structure charts of Pascal code.[14] This approach relies heavily on the lexical scoping of nested procedures and is not easily generalized to other procedural languages. In another case, a prototype of an automated system for representing Ada** tasks using extended DFDs has been described.[15] However, the focus is on modeling concurrency with DFDs, and the bulk of the source code information is thus ignored.

To date, systems that automatically generate all three of the basic SA/RT views of a system, i.e., process, data, and control, are only proposals or in the early stages of development (e.g., MAPR[16] or the considerably more ambitious Desire system[17]).

Overview of the RE-Analyzer

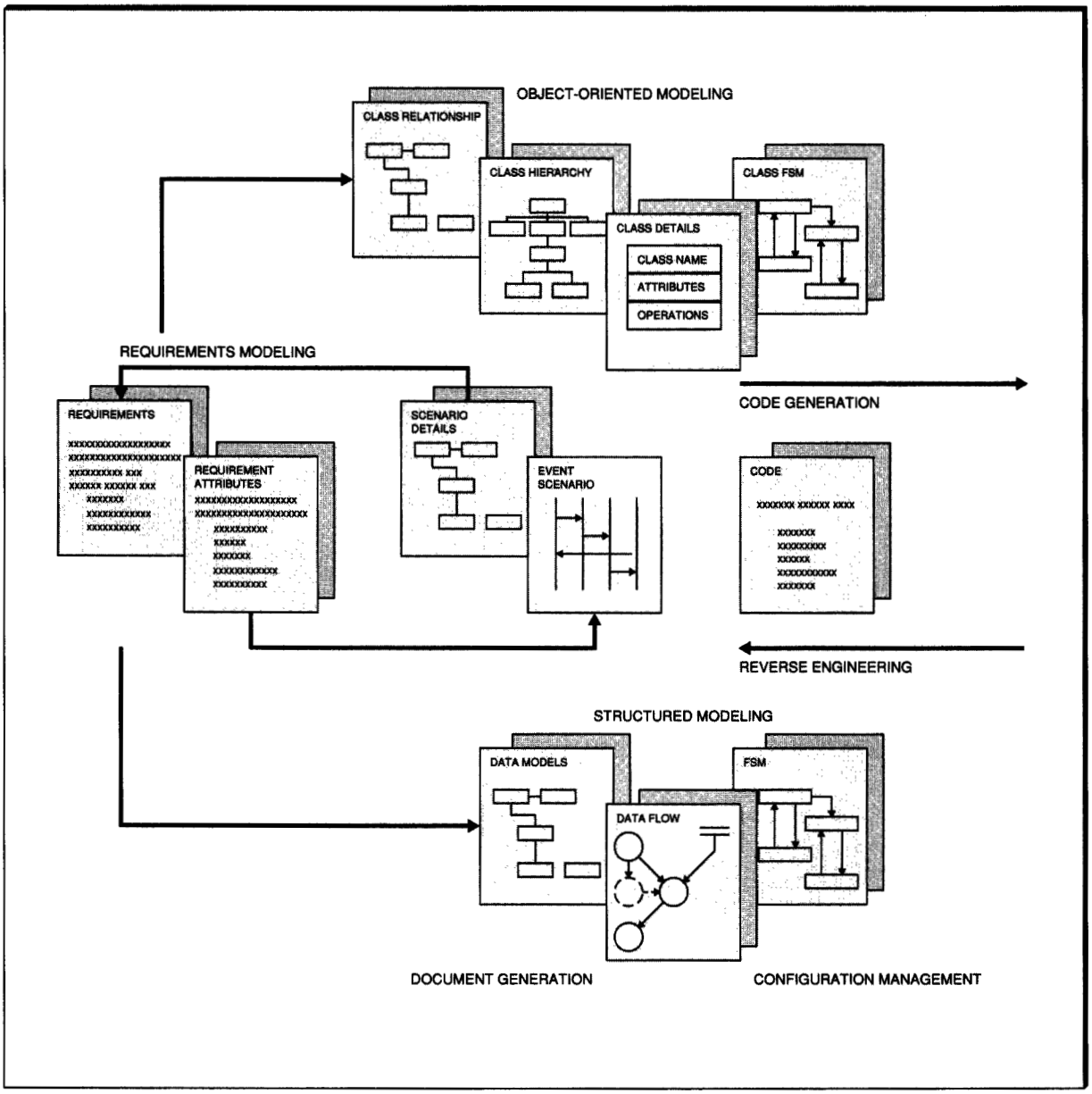
The primary objective of the RE-Analyzer is to support reverse engineering that is fully integrated with the SA/RT constructs of the VIEWS method.[18] As illustrated in Figure 1, VIEWS encompasses both SA/RT and object-oriented development methodologies as well as a common requirements modeling technique. The VIEWS SA/RT methodology is referred to as the Real-Time Structured Development Method (RTSDM).[19] RTSDM was initially based on ESML[10] and then extended to allow for a more precise semantic interpretation of models created with the methodology. Such precision is critical not only for reverse engineering but also in order to eliminate the discontinuities between analysis and design in traditional SA/RT.[20]

During the development of VIEWS, the RE-Analyzer was originally conceived of as a way to extend the life-cycle coverage of VIEWS to include reverse engineering. In fact, the RE-Analyzer was designed and developed after much of the VIEWS method had been completed. Currently, the RE-Analyzer supports the reverse engineering of American National Standards Institute (ANSI) C source code on Operating System/2* (OS/2*) and Advanced Interactive Executive* (AIX*) environments.

Figure 2 shows the basic organization of the RE-Analyzer system. The inputs to the RE-Analyzer include the source code to be analyzed and data on existing objects extracted from the current design database in the CASE tool. The source code to be analyzed must be syntactically correct, and the tool must have access to any libraries of include files or macro definitions that are referenced. As a rule, the source code should be compilable though it need not be linkable. The output of the RE-Analyzer is a data set that, after being entered into the CASE tool as input, constitutes a comprehensive SA/RT model of the source code that was analyzed.

The first step in the reverse engineering process involves accessing the repository of the CASE tool to obtain information about previously defined objects. The primary purpose of this information is to avoid name conflicts with objects that al-

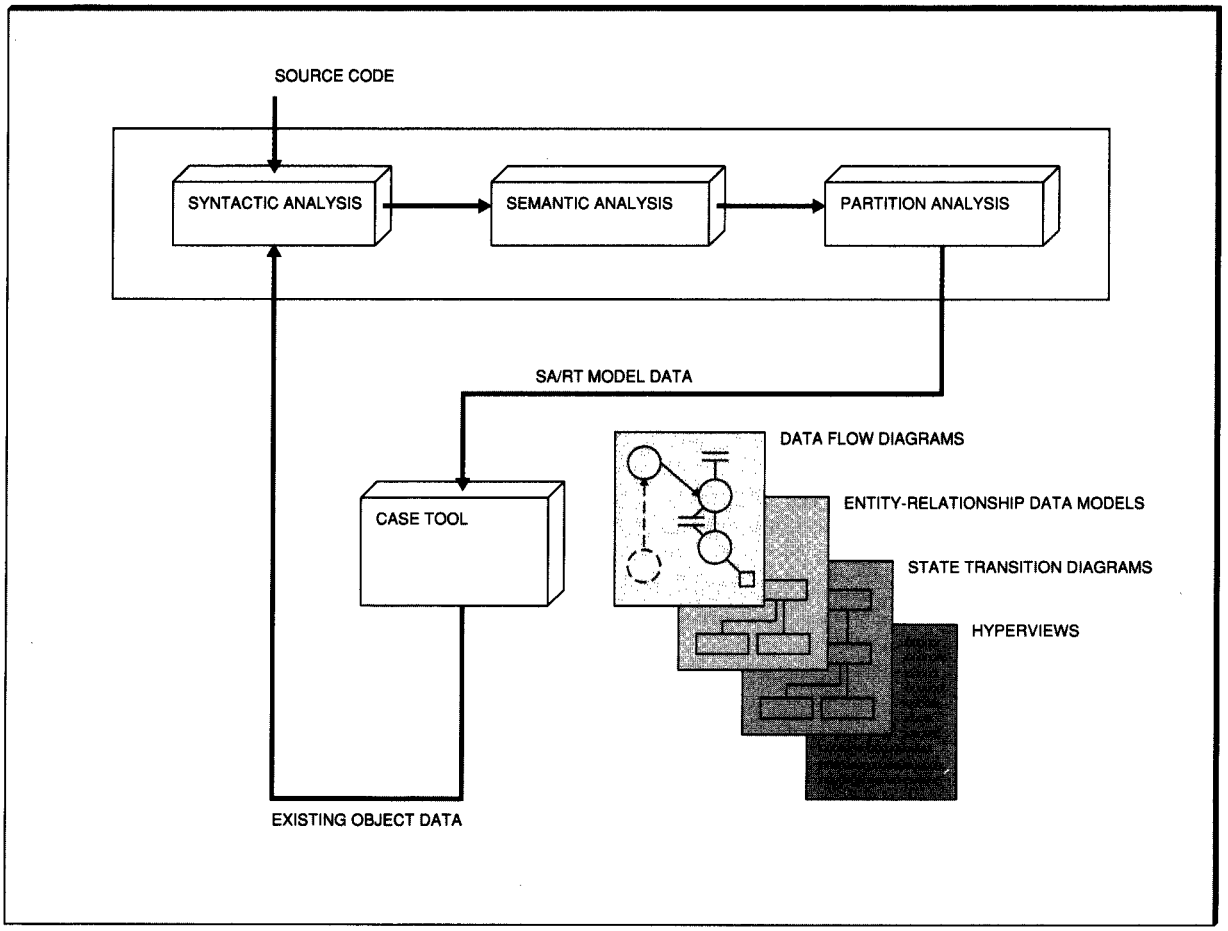
Figure 1 The VIEWS method



ready exist in the CASE tool. It is also necessary in order to support incremental reverse engineering. Incremental reverse engineering is the ability to process different modules of a software system at different times (as opposed to all at the same time). This ability also allows the user to apply the RE-Analyzer to a subset of the source code.

The next few stages of processing are analogous to those used in conventional compilers, i.e., preprocessing, syntactic, and semantic analysis of the source code. At this point the system produces equivalent intermediate code in a canonical form that is then analyzed for *control partitions*. Control partitions are semantically coherent col-

Figure 2 RE-Analyzer system organization



lections of the source code based on an analysis of control flow (including procedure calls). This process is deterministic and successively abstracts sequences of source code into new partitions until a specific level of control flow complexity is achieved for all partitions. This process is carried out in a bottom-up fashion for each function. It begins by grouping together simple statements and expressions. Those base partitions are then grouped into larger partitions according to the control flow of the source code. This grouping continues until a single parent control partition is created for each function. It is important to note that the partitioning process preserves the semantics of the original source code. This process is described in more detail in the next section.

Control partitioning serves two critical purposes. First, it reduces a large part of the task of representing source code within an SA/RT framework to one of representing a control partition. That is, most of the source code becomes encapsulated within one or more hierarchies of control partitions. By establishing a general technique for representing a control partition in an SA/RT framework, it is quite easy to apply that technique to the more complex hierarchies. Second, it helps to manage the complexity of the reverse engineering result by decomposing large sequences of source code into several smaller sequences. Although a control partition is not guaranteed to conform to the magic number seven plus or minus two,[21] (e.g., there may be more than seven processes in a DFD), it does effectively reduce the complexity

of the resulting graphic representations. This is in sharp contrast to other approaches to reverse engineering (e.g., C/Rev or automated flowcharting tools) that provide no automated decomposition mechanism and yield only large, complex representations.

Finally, the results are transformed into a data set that is merged into the data repository of the CASE tool. In effect, this data set contains all the information that would have been produced by the CASE tool if a user had entered the equivalent diagrammatic and textual information. The only exception is that no specifics on diagram layout are produced by the RE-Analyzer. This information is generated by the CASE tool itself using an automated graph layout facility.

Although the RE-Analyzer does not require the user to intervene during the reverse engineering process, it is possible for the user to alter and build upon the resulting models after they have been loaded into the CASE tool. This feature offers several distinct advantages over approaches that require the user to interact with a reverse engineering tool that is separate from the target CASE tool. First, the reverse engineering results are reproducible, i.e., they will be the same for any user starting with the same source code. Second, the user makes modifications within the framework of a CASE tool using the same concepts and abstractions that are employed during forward engineering. Third, this approach allows users to maintain revision histories of their modifications to the SA/RT models created by the RE-Analyzer. This history includes comments, notes, descriptions, etc. that can be added by the users to reflect the knowledge they gain as they strive to comprehend the software system.

The key characteristics of the RE-Analyzer include source code abstraction, a high level of integration with a CASE tool, fully automated generation of SA/RT-compatible representations, the preservation of all source code information, and incremental reverse engineering support.

Control partitions

Subroutine and function definitions containing hundreds of lines of code are not unusual. Even relatively small functions can contain such complex control constructs that they defy efforts to comprehend their code. Representing such functions as a single DFD would be rather pointless simply because of the resulting diagram density. What is required is some way of mapping a flat function onto a hierarchy of DFDs (with a single ancestor) while avoiding tangles of control structures.

As with large function definitions, one of the more difficult problems with representing complex control structures is how to decompose them into manageable units. STDs and similar formalisms have been criticized for just this reason. When a single STD is used to represent the specification for a very large and complex system, it is easy to understand why a two-dimensional diagram could be considered overly complex. Instead of attempting to devise a formalism that supports some concept of subspecifications for control, we have capitalized on the existing decomposition capabilities supported in DFDs to distribute control specifications across different levels of a data process hierarchy.

The key characteristic of a control partition is that all of its control information is local.

The basis for our modeling of and decomposing control constructs is the concept of a control partition that can be recursively decomposed into other control partitions (e.g., two or more nested loops). Each function or subroutine definition within the source code being analyzed will contain a single control partition that may, in turn, contain lower-level control partitions.

The key characteristic of a control partition is that all of its control information is local, i.e., there cannot be a branch operation with either a source or destination that is outside the scope of the partition. This restriction on the formation of control partitions is important, allowing them to be treated as black boxes that behave in a standard way with respect to control. Thus, the execution of the code within a control partition must begin at the top, and, if they end at all, they end at the bottom, though the route taken in between might

vary. The primary purpose for this restriction is to eliminate situations involving complex control interactions among sibling partitions. Given the restriction imposed by control partitions, it is possible to model the control interactions among partitions using a single control process and no direct control flow among the data processes representing partitions.

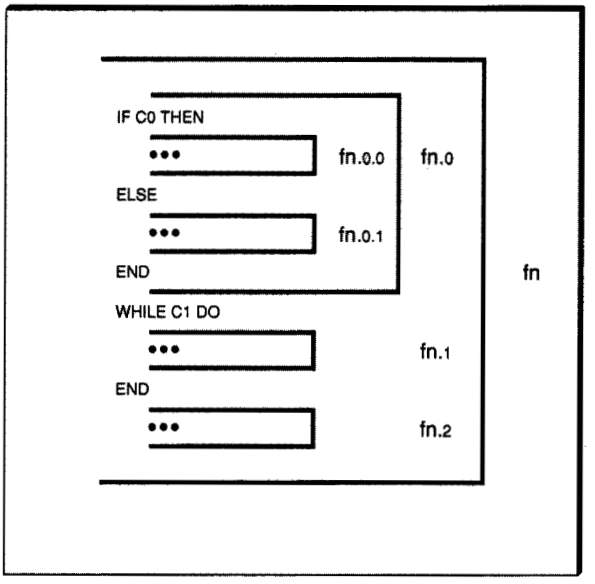
Figure 3 illustrates the basic concept of control partitions (indicated by open boxes). The code in the example constitutes the body of a function definition. The ellipses represent a sequence of statements that contain no control constructs. A single, top-level control partition, fn, contains three subpartitions (fn.0, fn.1, and fn.2). The first of these subpartitions is further decomposed into lower-level control partitions, and, depending on the actual code, these partitions might also be decomposed into still lower-level control partitions. Eventually, each control partition will be represented as a separate data process. The top-level control partition will be mapped to a DFD representing the function definition from which it was derived. If the control partition has subpartitions, its corresponding DFD will contain data processes representing each of its immediate subpartitions. In this way the control partition hierarchy is mapped to a data process hierarchy.

Unfortunately, the control partitioning process is not as simple as the above example suggests. In order to ensure that each control partition conforms to the restriction described above, it is necessary to correctly handle all control constructs, including goto, break, and continue statements. Since such statements can cross the scope of one or more control constructs, the control partitioning process must follow the entire flow of control as it constructs each partition.

Reanalyzing source code

This section illustrates the results produced by the RE-Analyzer. As noted earlier, the system transforms a given source code module into a series of inputs to a CASE tool to yield an alternative representation using concepts and abstractions from structured analysis. Thus, three fundamentally different views of the original source code module are created to represent process structure and data flow, control flow, and information or data structure. For RTSDM, these different views

Figure 3 Control partitions



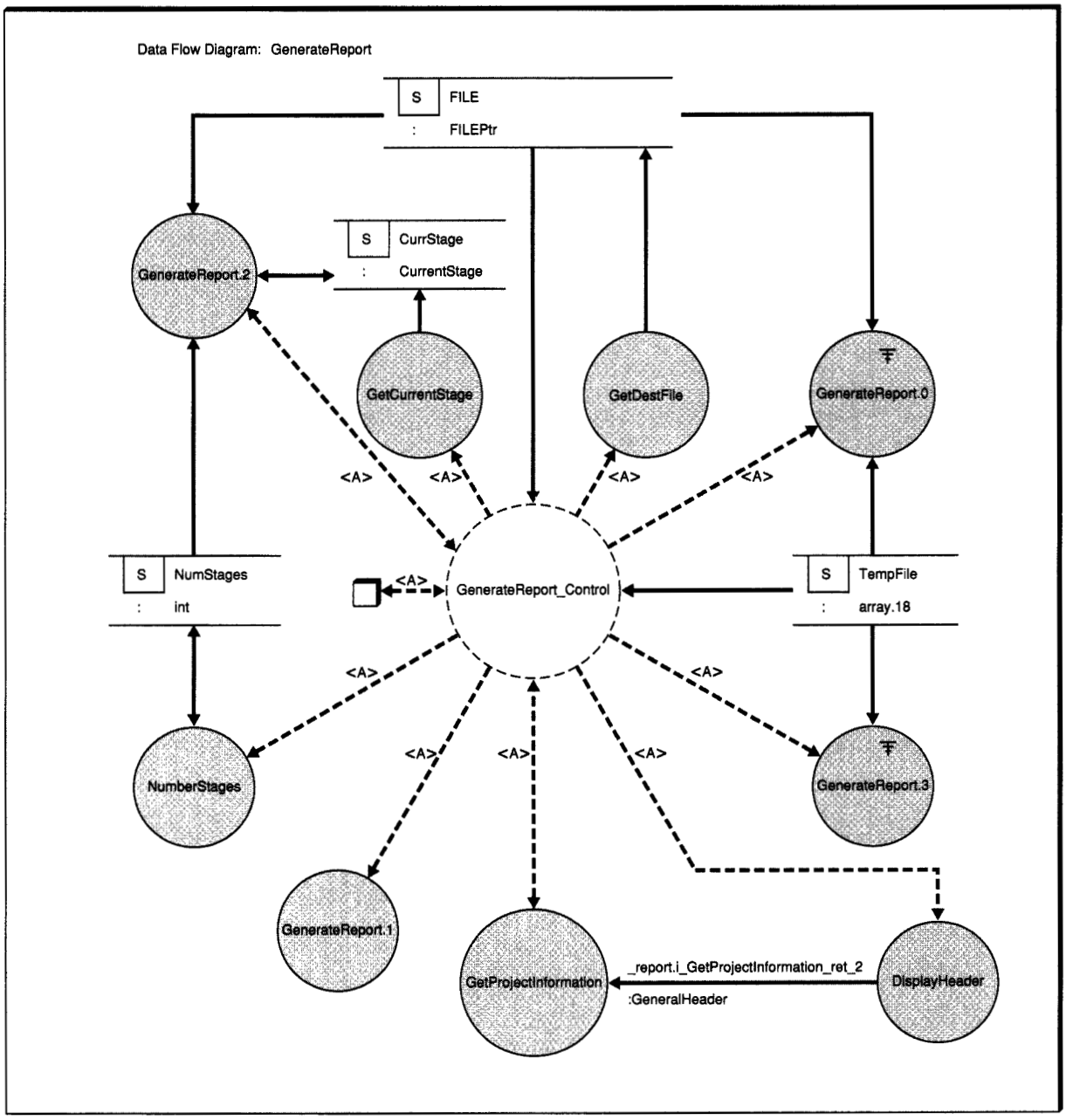
are managed by distinct, yet highly integrated modeling techniques called, respectively, data flow diagramming, state transition diagramming, and entity-relationship data modeling.

For each view, we will briefly describe only the pertinent aspects of the associated modeling technique and discuss examples of some of the results produced by the RE-Analyzer system. In addition, we will provide an example of a HyperViews report that provides cross-reference information spanning all of the modeling techniques and is particularly useful for the task of program understanding. Note that the modeling technique descriptions are necessarily cursory in nature and are not intended to represent complete descriptions of either RTSDM or SA/RT in general.

The examples are based on results produced by reverse engineering source code for a project in a graduate software engineering class at North Carolina State University. Briefly, the project was to build a prototype of a software risk management tool. One of the function definitions from this code is shown in the Appendix.

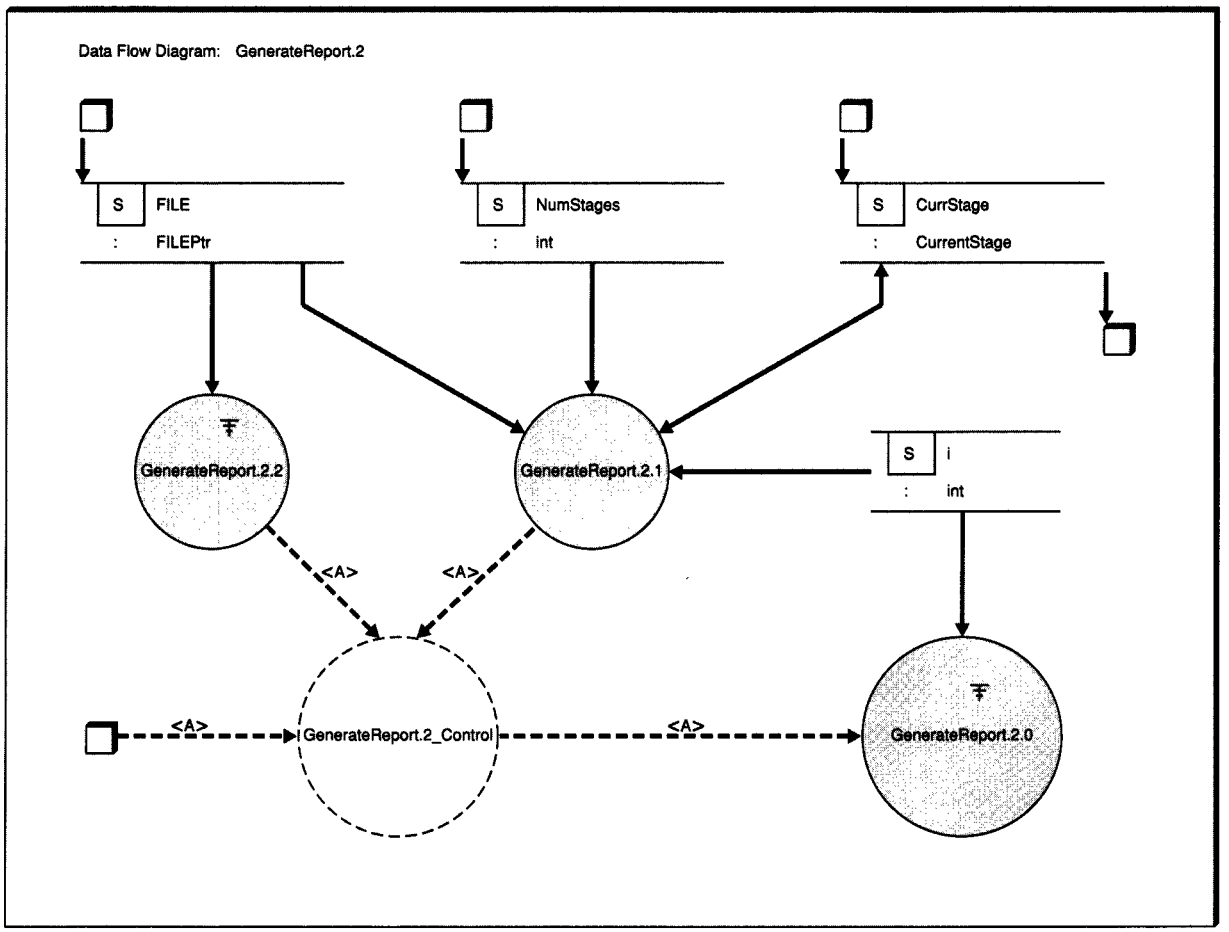
Process model. Figure 4 is an example of a DFD produced by the RE-Analyzer system. Some manual repositioning has been done on the nodes and links to make the diagram more compact and readable, but no information content has been

Figure 4 A sample DFD produced using the RE-Analyzer



modified in any way from that produced by the RE-Analyzer. The DFD represents the top level of the function `GenerateReport` (see the Appendix). There is no input to the function (i.e., no off-page connectors with data flows connecting them to a process or store) other than an activation prompt that indicates how the control process is itself controlled. The single control process manages

Figure 5 The decomposition of process "GenerateReport.2"



the activation of nine different data processes. The precise behavior of the control process is represented by its associated STD.

Four data stores represent variables that are local to the top level of the function, and each variable has its own associated type. For example, the data store named `CurrStage` is of type `CurrentStage`. The data store is both read and written by one data process and just written by another data process. In this particular example, most of the data flows are neither named nor typed since they are connected to data stores that determine their type. The only exception is the data flow between the `GetProjectInformation` and `DisplayHeader` data processes at the bottom of the diagram. The data flow represents the fact that one process is producing input for the other process. The name of the data flow was synthesized by the RE-Analyzer since, in the source code, one function call was an argument of the other.

The type, `array.18`, for the store named `TempFile` is not a user-defined type, rather, it was an unnamed type for which the RE-Analyzer synthesized a name. Data types are discussed below as part of the section on data modeling.

Only some of the data processes were explicitly defined by the user, i.e., `NumberStages`, `GetDestFile`, `DisplayHeader`, `GetProjectInformation`, and `GetCurrentStage`. The other data processes were created by the RE-Analyzer as abstractions of a

Figure 6 Minispec for the lower-level process "GenerateReport.3"

```
/* Process Name: GenerateReport.3 */

GenerateReport.3( )
{
    sprintf ( SystemCommand , "clear ; more %s", TempFile )
    system ( SystemCommand )
    unlink ( TempFile )
}
```

Figure 7 Process instance hierarchy for the process "SMerror"

```
PROCESS SMerror contains:

    SMerror_Control  { Ctrl }
    SMerror.0
      SMerror.0_Control  { Ctrl }
      SMerror.0.0
      CloseProject
        CloseProject_Control  { Ctrl }
        CloseProject.0
      SMerror.0.1
    SMerror.1
```

collection of source code statements. By convention, such synthesized processes have names that begin with that of their parent process and then a decimal suffix that is unique within the process hierarchy, e.g., `GenerateReport.1`.

Figure 5 shows the DFD for one of the synthesized data processes, `GenerateReport.2`, which also contains processes that are abstractions of source code. In this DFD, the data flows from off-page connectors represent process inputs and outputs. By convention, a data store connected to an off-page connector represents either an input parameter or a duplicate of one at a higher level in the process hierarchy. An example of the latter can be seen in Figure 5, where the data store `FILE` is a duplicate of the one shown in Figure 4.

In some cases, a synthesized data process will have no corresponding DFD. This occurs when the source code abstracted by the data process contains only simple sequential control information and can be represented as a sequence of one or more expressions. For example, the data process `GenerateReport.3` decomposes into a sequence of three expressions as shown in Figure 6. This code sequence, or *minispec*, is a partition of source code that was created by the RE-Analyzer and that existed as part of a more complex collection of statements in the original source code. Unlike traditional SA/RT methodologies where only the primitive processes have a minispec, the RE-Analyzer also provides a minispec for each top-level process derived from a user-defined function. This minispec is actually the original source code defining the function, and it may be easily accessed within the CASE tool. Processes annotated with an inverted tree symbol in the upper right corner are primitive or ground processes (e.g., `GenerateReport.3`).

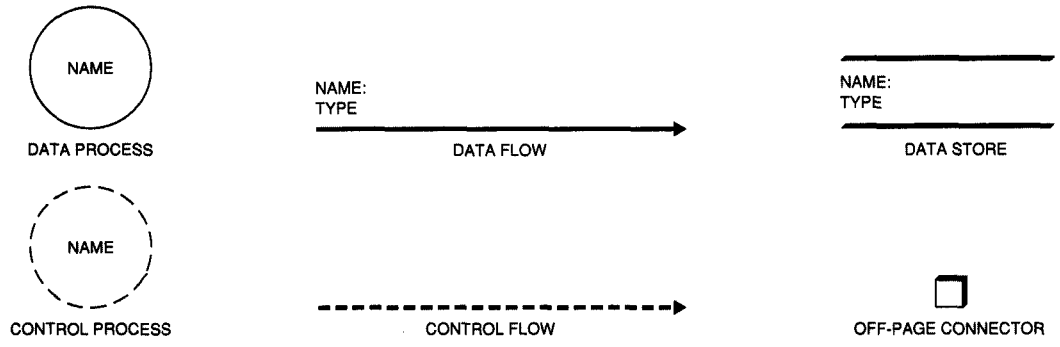
The RE-Analyzer only represents built-in functions, such as `sprintf` or `system`, if so directed by the user prior to analyzing the source code. In the examples presented here, the representation of built-in functions was suppressed so that they appear only within minispecs and not as data processes in a DFD.

Figure 7 shows the process instance hierarchy for `SMerror`. The hierarchy includes control processes as well as data processes and uses indentation to indicate levels within the hierarchy. The process names furthest to the right are leaf nodes and are defined only in terms of a minispec as shown above. Note that there are lower-level synthetic processes that were created within this one process as an abstraction of some source code.

Figure 8 shows the process composition hierarchy for `GenerateReport`. This differs from the process instance hierarchy shown above in that all synthetic processes are filtered out of the list so that only those functions that appear explicitly in the source code are shown.

PROCESS MODELING IN RTSDM

Data flow diagrams (DFDs) provide a functional model of a system. In their simplest form, they represent a system in terms of a hierarchy of processes and their respective inputs and outputs. Although the fundamental concepts of data flow diagrams are hardly new, their utility is reflected in their continued popularity.



A *data process* represents a function that operates on its inputs (represented by data flows) to produce zero or more outputs (also represented by data flows). A data process may be *decomposed* into its own DFD to show how other data processes are used in its construction. Decomposition is a powerful form of abstraction identical in nature to the concept of procedural abstraction present in all structured programming languages. It is important to note that each DFD represents a separate data process. Thus a data process on the DFD is often referred to as a *child process* and conversely, the process that is the subject of the DFD is referred to as the *parent process.*

Unlike many SA/RT methodologies, data processes in RTSDM are *reusable*. That is, instances of a given data process may appear on more than one DFD. The concept of reusable processes is important, not only as a mechanism for supporting reuse, but also as a crucial concept of a robust representation of procedural abstraction. For example, a recursive process can be effectively modeled by placing an instance of a process on its own DFD.

A *data store* represents passive data that may be read and written by processes, for example, a simple variable, an external file, or a data structure. Data stores in RTSDM may be further qualified as being *static, static reference, dynamic,* or *global* which provide information about the scope of the store and its persistence. In addition, a data store may also have a type associated with it.

A *data flow* represents an interaction between one data process and another data process or a data store. As with data stores, a data flow in RTSDM may be given a type designation.

An *off-page connector* represents a connection to a process or store at another level in a process hierarchy. In RTSDM, an off-page connector is analogous to a placeholder for a procedure parameter. Since data processes in RTSDM are reusable, a single off-page connector may represent a connection to more than one process or store in more than one process hierarchy.

A *control process* represents the control portion of data process. For example, control processes are required to express sequential processing behavior or the processing of events or interrupts. A control process decomposes into a state transition diagram where its precise behavior can be modeled.

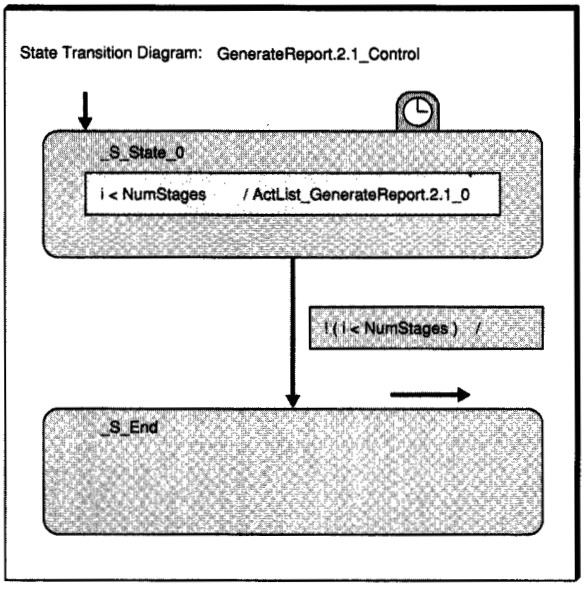
A *control flow* represents the transmission of control signals or events between control or data processes, or both. A *control prompt* is similar to a control flow except that it represents the communication of a specific control imperative, e.g., activate, deactivate, suspend, or resume. Each prompt is indicated by a different letter in angle brackets near the control prompt, e.g., <A> denotes an activate prompt.

Figure 8 Process composition hierarchy for the process "GenerateReport"

```
PROCESS GenerateReport contains:

  GetDestFile
  SMerror
    CloseProject
  GetProjectInformation
  DisplayHeader
  GetCurrentStage
  NumberStages
  GetStageInfo
```

Figure 9 The state transition diagram for "GenerateReport"



In addition to the above, there are several other types of information that can be readily accessed by a user. Most of these types are in the form of reports, but information can be entered or modified as well as viewed. They include the following:

- *Data process details*—contains a process description, creation data, and details of stores used in the process (e.g., type, storage class, and initial value)
- *Data process X-Ref*—provides a general cross-reference listing of data processes
- *Store visibility report*—lists all of the stores that appear on the DFD for a given process and then all of the processes and stores that are above the process in the process hierarchy

Various balance reports are also intended to be used during forward engineering, to identify potential errors in the number and type of process inputs and outputs.

The CASE tool provides context-sensitive, hypertext-like facilities that allow the user to easily move among the different diagrams and text-based forms or reports. For example, from a DFD a user may traverse to the associated minispec, the DFD of a child process, the definition of a type of a store or a data flow, the STD of a control process, balance reports, or a cross-reference index. Clearly, such capabilities are not only of value in forward engineering but for reverse engineering as well.

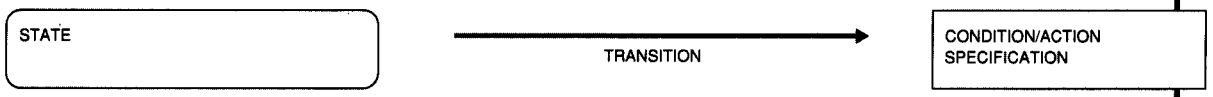
Various reverse engineering activities are also supported by allowing the user to add and modify information. For example, higher-level DFDs may be added to document how the system interacts with other, external, systems. Textual descriptions can be entered for key processes and other objects, and notes and annotations can be added to diagrams. Requirements can be entered and linked to elements in the DFDs. Elements with synthetic names may be given more meaningful names. Unnecessary or inessential elements may be deleted. All of these capabilities help to capture and preserve what users learn as they advance their understanding of the subject system.

Finally, restructuring is facilitated because the synthesized processes may be used as recommendations to the user on how to restructure the source code using separate procedure definitions. In addition, the DFD for each synthesized process shows which local variables are accessed by the corresponding source code.

Control model. Figure 9 shows the state transition diagram (STD) generated by the RE-Analyzer tool for a `for` loop in the bottom half of the source code appearing in the Appendix. This STD contains only two states—an initial state (`_S_State_0`) and a final state (`_S_End`). The transition contained within the initial state indicates that the next state will be the same as the current state, i.e., control remains

CONTROL MODELING IN RTSDM

A State Transition Diagram (STD) is a graphical representation of a finite automata[23] and is used to specify th behavior of its associated control process appearing on a DFD. As shown below, an STD is composed of states and transitions.



A *state* symbol represents one possible state of the system being modeled. In RTSDM, states may be adorned with other symbols to denote an initial state, an exit or final state, a transitory state, and a suspended state. These state qualifiers serve to distinguish different interpretations of an STD and help to establish a precise semantic model. For example, the STDs produced by the RE-Analyzer do not contain suspended states. That is, control processes are assumed to be continuously active until they pass control on to some other process.

A *transition* is used to indicate which states may succeed a particular state. Transitions are typically annotated with a *condition/action specification* which specifies the conditions that must exist before a transition to another state can occur and, optionally, the actions that are to be taken when the transition occurs. Initial actions may be associated with an initial state. The condition in this case is omitted since it is, by definition, the activation of the associated control process.

within the same state as long as the indicated condition is satisfied. The transition to the final state occurs when the condition `i < NumStages` is no longer true. Note that there is no action associated with the transition to the final state.

Although states are used to represent points where control flow branches to one or more different locations, the names of states are arbitrary and have no mapping to any object in the original source code. The user is responsible for supplying meaningful state names.

The clock-shaped symbol that appears on the initial state indicates that it is also a *transitory state*. That is, transitions can occur only after all actions associated with the last transition in or into this state have completed. This type of state is used to model a single thread of control.

State `_S_End` has an arrow symbol (going from left to right) instead of a clock symbol; it denotes an exit or final state. For our purposes, a final state indicates the point at which an STD releases control until it is reactivated.

Figure 10 The action details from the "GenerateReport.2.1" STD

```
Action Name:          ActList_GenerateReport.2.1_0

Description:          (none)

Action Statements:
  1  Prompt: <activate >GetStageInfo
  2  Prompt: <activate >GenerateReport.2.1.0
```

Figure 10 shows the activation list for the action `ActList_GenerateReport.2.1_0` that appears in the initial state of the `GenerateReport.2.1` STD. This list simply enumerates the processes that are to be activated when the transition occurs. Note that the order is important since the second process is not activated until after the first activated process has completed. The transition is said to have completed when all of its actions have completed.

Other, more complex control structures can be easily represented as an STD. One of the main

Figure 11 Entity details for "CurrentStage"

```
Entity Name: CurrentStage

E-R Categories: prinfo.h

Description:   (none)

Structure connection type: and

Attributes:
  1  ReasonLeft : array [255] of char
  2  DateLeft : array [10] of char
  3  DirectionMoved : enum_Delta
  4  PrevAbstractStageName : array [255] of char
  5  NextAbstractStageName : array [255] of char
  6  StageName : array [255] of char
  7  ToolInvocationString : array [255] of char
  8  ToolName : array [255] of char
```

values of the STDs produced by the RE-Analyzer is its integration with the process and data models. That is, the STD for each control process on a DFD can be quickly accessed and viewed; the conditions and actions of an STD similarly relate to entity definitions and data processes. Also, in spite of the great familiarity that most developers have with common control idioms (e.g., if-then-else, while-do, or repeat-until), the graphical representation afforded by STDs can make complex control structures easier to follow.

Data model. The entity-relationship (E-R) model[22] is the basis of the data modeling technique supported in VIEWS. Each type definition found in the source code is represented as an *entity*. This is in keeping with the traditional notion of an entity as representing a class of objects.

For each module processed by the RE-Analyzer, a separate E-R diagram, referred to as an *entity category diagram*, is produced containing one entity for each type defined in the module. Since the VIEWS E-R modeling technique represents entity attributes as subordinate detail rather than as objects on an E-R diagram, most of the data model produced by the RE-Analyzer does not appear in graphic form. Thus, no relationships between type definitions are modeled on the E-R diagrams. However, one important relationship among type definitions, i.e., type composition, is modeled as will be shown below.

Since the VIEWS E-R modeling technique supports a form of lexical scoping for data store names, variables appearing in the source code that have the same name but different scopes can be easily represented.

The subordinate detail generated by the RE-Analyzer for each entity includes a list of categories that reference the entity (i.e., modules that reference the corresponding type) and a specification. An entity specification may take any one of three different forms: (a) *simple* specifications represent a simple type equivalence or a simple structure (e.g., an array or a pointer); (b) *enumerated* specifications represent enumerated types; and (c) *structured* specifications represent either compound types (e.g, records) or a union of types. A structured specification contains one or more attributes that are ANDed together to form compound types or ORed together to form a union of types. Attributes have a name and a type as described above for entities. Figure 11 shows the entity details text form for the compound type `CurrentStage` referenced by a store in Figure 4. This text form displays the immediate structure of the entity and allows the user to enter or edit information. These representations are taken directly from type declarations and structure specifications appearing in the source code.

Figure 12 shows the entity specification hierarchy for `CurrentStage`. This hierarchy shows the complete composition of the entity, i.e., its immediate structure and the structure of each of its components. In this example, the entity has a structured specification with eight attributes forming a compound type as indicated by the structure connection type. The attribute `DirectionMoved` has a type of `enum_Delta` that has an enumerated specification as shown below the attribute entry. Indentation is used to indicate levels in the type hierarchy. Recursive type definitions are handled by displaying a single cycle of the definitions and annotating the first reference with two asterisks. The specification hierarchies are inferred by the CASE tool from the SA/RT model generated by the RE-Analyzer.

The RE-Analyzer must synthesize entity names and specifications for certain classes of type declarations found in source code. For example, the type `array.18` found in Figure 4 was defined as an array of characters. This definition avoids the necessity for RTSDM to support arbitrary type declaration syntax in DFDs and helps to minimize the

diagram space needed to specify type information. Similarly, the RE-Analyzer will synthesize an entity and simple specification for pointers. For example, the store type FILEPtr in Figure 5 was defined as a pointer to a file type.

In addition to allowing the user to easily view the subordinate details of any entity or its complete definition hierarchy, a cross-reference index and a completeness report are provided. The latter indicates incomplete or missing details, e.g., attributes that were not assigned a type, arrays with no bounds specification, or bit fields with size information.

Aside from completing the SA/RT model, there are several other ways in which the data model produced by the RE-Analyzer can facilitate both forward and reverse engineering. For example, the data model is the basis for cross-reference reports on both type definitions and variable declarations as described in the next subsection. Also, the data model simplifies reengineering efforts by eliminating the need to enter required type information and making it readily available to users so that they can easily view the associated type definition for data stores and flows. Finally, the E-R data model can be completed by simply adding relationships among the entities already created and placed on entity category diagrams by the RE-Analyzer.

HyperViews. The HyperViews facility of VIEWS provides an intelligent cross-reference capability that spans all of the modeling techniques. Given a particular object such as a data process, a store, or an entity, it dynamically produces a report indicating where the object is referenced in any of the modeling techniques supported by VIEWS and how it is used.

Figure 13 is an excerpt from the HyperViews report for the entity CurrentStage. The report lists all of the data stores and the processes whose DFD

Figure 12 The entity specification hierarchy for "CurrentStage"

```
Entity Name: CurrentStage

E-R Categories: prinfo.h

Description:   (none)

Specification Hierarchy:

   Structure connection type: and
     ReasonLeft : array [255] of char
     DateLeft : array [10] of char
     DirectionMoved : enum_Delta
          { Forward,
          Backward,
          First,
          Current }
     PrevAbstractStageName : array [255] of char
     NextAbstractStageName : array [255] of char
     StageName : array [255] of char
     ToolInvocationString : array [255] of char
     ToolName : array [255] of char
```

Figure 13 HyperViews for the entity "CurrentStage"

```
CurrentStage has HyperView Links to:

Class or Entity
---------------

  CurrentStage - member of Categories: prinfo.h
    Component of:

    Instances:
      Store:  Stage - Parent Process: RunTool
      Store:  ThisStage - Parent Process: GetStageInfo.0
      Store:  CurrStage - Parent Process: ChangeStage
      Store:  CurrStage - Parent Process: GenerateReport
      Store:  CurrStage - Parent Process: GenerateReport.2
      Store:  Stage - Parent Process: DisplayMenu
:
:
```

Figure 14 HyperViews for the data process "GetCurrentStage"

```
GetCurrentStage has HyperView Links to:

Process
-------

  GetCurrentStage  Parent: RunTool
  GetCurrentStage  Parent: GenerateReport
  GetCurrentStage  Parent: DisplayMenu
```

contains the store, i.e., the name and context for every variable that is declared to be of type CurrentStage.

Figure 14 is an excerpt from the HyperViews report for the data process GetCurrentStage. The report lists all of the processes whose DFD contains the process, i.e., every place where that data process is used. This particular example also illustrates how data processes are reused in different DFDs.

Figure 15 is an excerpt from the HyperViews report for the data store PrStore. The report lists all of the processes whose DFD contains the store and the child processes that read or write to the store. Thus, it not only provides a cross-reference indicating where the store is referenced, but it also details how it is used in each case.

Similar reports are provided for every other basic SA/RT element as well, e.g., data and control processes, entities, conditions, and actions. This type of intelligent, cross-reference capability is valuable for both forward and reverse engineering activities.

User experiences, limitations, and future plans

Relatively large code sets have been successfully processed, including one networking application that had more than 26 000 lines of code and the source code for the code analysis part of the RE-Analyzer, which has over 17 000 lines of code.

Beta test users of the RE-Analyzer have found it to be useful in program understanding work. One user was given a piece of monolithic, poorly structured code to restructure so that it would be more reusable. The RE-Analyzer was used to graphically view which code segments were sufficiently isolated so that they could be moved from in-line code to separate functions. The tool also made it relatively easy to tell which global variables a new function accessed so that they could be passed as parameters instead of declared as global variables. Another beta tester used RE-Analyzer to locate dead code. In the data flow diagrams produced, dead code appeared as processes or clusters of processes with no data flows representing inputs or outputs. Other, more subtle cases (e.g., inputs but no outputs or side effects) were also identified with slightly more effort. Also, an ongoing project is using the RE-Analyzer to produce consistent and comprehensive documentation so that (a) recommendations can be made on how best to restructure the system and (b) new developers can become productive more rapidly.

Some limitations will be resolved in the near future as enhancements are made to the existing tool. One such limitation requires the input of the RE-Analyzer to be preprocessed. Thus, the RE-Analyzer is not able to capture and effectively represent compiler directives that include other files and defined constants and macros, or that control conditional compilation. Because many programmers use such constructs to effect a form of abstraction, we consider it important to augment the system so that it can capture and preserve these abstractions.

Another limitation stems from the fact that no SA/RT methodology addresses the representation of pointers. Although VIEWS provides a simple representation of pointers within its E-R data modeling technique, we hope to develop a more robust formalism that will address related issues such as aliasing.

Finally, the RE-Analyzer does not attempt to represent the semantics of built-in functions found in the source code. This limitation is not significant in the context of sequential programs. However, in cases where a program creates concurrent processes, the RE-Analyzer does not make full use of the SA/RT capabilities to model such behavior.

Other planned enhancements include support for languages other than C, more extensive use of E-R diagrams for representing relationships among defined types, and some simplifications of the re-

Figure 15 HyperViews for the data store "PrStore"

```
PrStore has HyperView Links to:

Data Store
----------

  Parent Data Process: GetStageInfo.0
    Written-by:
    Read-by: GetStageInfo.0.0

  Parent Data Process: NewProject.1
    Written-by: NewProject.1.1
    Read-by: (off-page connector)

  Parent Data Process: NewProject.4
    Written-by: (off-page connector)
    Read-by: NewProject.4.0

Parent Data Process: NewProject
    Written-by: NewProject.1
    Read-by: NewProject_Control, NewProject.4

  Parent Data Process: OpenProject.2.2.2
    Written-by: (off-page connector)
    Read-by: OpenProject.2.2.2.0

  Parent Data Process: OpenProject.2.2
    Written-by: (off-page connector)
    Read-by: OpenProject.2.2_Control, OpenProject.2.2.0, OpenProject.2.2.1,
             OpenProject.2.2.2
:
:
```

sulting SA/RT representations such as removing control processes that have no internal transitions.

A separate type of enhancement involves the integration of the RE-Analyzer or the VIEWS method, or both, with other types of tools. For example, since the RE-Analyzer relies on static analysis, it does not directly address those cases where the reverse engineering process might be enhanced by providing dynamic analysis capabilities. However, a tool that provides such capabilities could be used to complement the RE-Analyzer by annotating appropriate parts of the SA/RT model with the information gleaned from the dynamic analysis.

Conclusion

The reverse engineering system presented here realizes the goal of automatically transforming legacy code into data flow diagrams, state transition diagrams, and entity-relationship data models within a CASE tool supporting structured analysis for real-time systems. This approach to reverse engineering promises to have a significant impact on software development since it greatly facilitates a wide variety of maintenance and reengineering activities, including redocumentation, restructuring, program understanding, and program enhancement. Even the conversion of legacy code written in a structured language to an object-oriented language can be made easier with the use of this system in conjunction with the VIEWS object-oriented analysis and design methodology.

It was noted that the key characteristics of the RE-Analyzer include source code abstraction, a high level of integration with a CASE tool, fully automated generation of SA/RT compatible representations, incremental reverse engineering support, and the preservation of all source code information. These characteristics lead to several other significant properties of the RE-Analyzer system. Since the user enjoys a common set of

modeling techniques and operations for both forward and reverse engineering activities, only a single vocabulary is required for both classes of activities. The CASE tool supports the creation and management of revision histories that facilitates the preservation of knowledge gained during the reverse engineering process. And finally, the system facilitates the injection of CASE technology into a development organization by providing a relatively easy way for users to move from the source code with which they are familiar to the SA/RT abstractions that can be used to model it within a CASE tool.

Although the current implementation of the RE-Analyzer system works for code written in the C language, the underlying approach can be generalized for any procedural programming language.

Finally, the results of this project have given us considerable confidence that the goal of a coherent, integrated reengineering environment can be realized. Specifically, the other half of the reengineering cycle, i.e., code generation, appears to be quite tractable for the VIEWS structured analysis methodology.

Acknowledgments

The realization of the RE-Analyzer relied heavily on the VIEWS method and all of its outstanding characteristics. Thus, we are greatly indebted to all the other VIEWS team members: Sylvanus Bent, Sharon Haerle, and Kim Mui. We would also like to thank David Strube for believing that it could be done and JoAnn Havran for keeping the faith. And last, but certainly not least, we thank all of our adventurous early users.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Advanced Software Automation, Inc., PROCASE Corporation, Verilog, Reasoning Systems, Inc., Cadre Technologies, or U.S. Department of Defense.

Appendix: Source code for the function "GenerateReport"

```
void GenerateReport ( void )
{
    int i;
    int NumStages;
    GeneralHeader Header;
    CurrentStage CurrStage;
    char DestFileName[200];                              /* User entered file name */
    char TempFile[(sizeof("/tmp/") + 15)] = "";          /* Temporary filename */
    char SystemCommand[(sizeof("/tmp/") + 15) + 10];     /* Holds a system() command */
    FILE * File;                                         /* Output file for the report */

    if ( !( File = GetDestFile() ) )
       {
            /* User wants it on the screen. Send it to a temporary file for now */
            tmpnam(TempFile);
            File = fopen(TempFile, "w");
       }

    if (!File)
            SMerror("Couldn't open file", NonFatal);

    /* Now generate the file. The header is a piece of cake. */
    DisplayHeader( GetProjectInformation() );
    CurrStage = GetCurrentStage();
```

```
    /* Now step through the stages and print out info on each */
    NumStages = NumberStages();

    for (i=1; i<NumStages; i++)
       {
            CurrStage = GetStageInfo(i);
            fprintf(File, "%s       ", CurrStage.DateLeft);

            /* This makes it "Moved into" or "Backtracked into" depending on which
               direction the user went at the time. Silly little nicetie. */
            if (CurrStage.DirectionMoved == Forward)
                    fprintf(File, "Project moved into %s stage\n",
                      CurrStage.NextAbstractStageName);
            else
                    fprintf(File, "Backtracked into %s stage\n",
                      CurrStage.PrevAbstractStageName);
            fprintf(File, "          - %s\n", CurrStage.ReasonLeft);
       }

    fclose(File);

    if (TempFile[0])
       {
            /* We used a temp file. Clear the screen and display the file through
               more. Erase the temporary file when we're done */
            sprintf(SystemCommand, "clear ; more %s", TempFile);
            system(SystemCommand);
            unlink(TempFile);
       }
}
```

Cited references and note

1. E. J. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software* **7**, No. 1, 13–17 (January 1990).
2. G. Parikh, "Making the Immortal Language Work," *International Computer Programs Business Software Review* **7**, No. 2, 33 (April 1987).
3. F. J. Buckley, "Some Standards for Software Maintenance," *Computer* **22**, No. 11, 69–70 (November 1989).
4. G. Parikh and N. Zvegintzov, "The World of Software Maintenance," *Tutorial on Software Maintenance*, G. Parikh and N. Zvegintzov, Editors, IEEE Computer Society Press, Los Alamitos, CA (1983), pp. 1–3.
5. T. A. Corbi, "Program Understanding: Challenge for the 1990s," *IBM Systems Journal* **28**, No. 2, 294–306 (1989).
6. C. Gane and T. Sarson, *Structured Systems Analysis: Tools & Techniques*, Improved Systems Technologies Inc., New York (July 1977).
7. T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press, New York (1978).
8. D. Hatley and I. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing, New York (1988).
9. P. T. Ward, "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing," *IEEE Transactions on Software Engineering* **SE-12**, No. 2, 198–210 (1986).
10. W. Bruyn, R. Jensen, D. Keskar, and P. Ward, "ESML: An Extended Systems Modeling Language Based on the Data Flow Diagram," *ACM SIGSOFT Software Engineering Notes* **13**, No. 1, 58–67 (January 1988).
11. E. Yourdon, *Modern Structured Analysis*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1989), pp. 417–423.
12. L. Peters, *Advanced Structured Analysis and Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1987).
13. F. Lanubile, P. Maresca, and G. Visaggio, "An Environment for the Reengineering of Pascal Programs," *Proceedings of the IEEE Conference on Software Maintenance—1991* (August 1991), pp. 23–30.
14. P. Benedusi, A. Cimitile, and U. De Carlini, "A Reverse Engineering Methodology to Reconstruct Hierarchical Data Flow Diagrams for Software Maintenance," *Proceedings of the IEEE Conference on Software Maintenance*, Miami (October 1989), pp. 180–189.
15. G. Canfora, A. Cimitile, and U. De Carlini, "Reverse Engineering and Data Flow Diagrams in ADA Environment," *Microprocessing and Microprogramming* **30**, 357–364 (1990).
16. M. Platoff, M. Wagner, and J. Camaratta, "An Integrated Program Representation and Toolkit for the Maintenance of C Programs," *Proceedings of the IEEE Conference on Software Maintenance—1991* (August 1991), pp. 129–137.
17. T. J. Biggerstaff, "Design Recovery for Maintenance and Reuse," *IEEE Computer* **22**, No. 7, 36–49 (July 1989).
18. A product based on the VIEWS method is expected to be available shortly from VSF Ltd. in Hampshire, England.

19. S. W. Haerle, A. B. O'Hare, and S. G. Bent, "Successive Disambiguation—Adding Precision to Real-Time Structured Analysis Methodologies," *Proceedings of Structured Development Forum XIII,* Philadelphia (August 1993).
20. R. B. France, "Semantically Extended Data Flow Diagrams: A Formal Specification Tool," *IEEE Transactions on Software Engineering* **18**, No. 4, 329–346 (April 1992).
21. G. A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Review* **63**, 81–97 (1956).
22. Peter Chen, "The Entity-Relationship Model Toward a Unified View of Data," *ACM Transactions on Database Management* **1**, No. 1, 9–36 (March 1976).
23. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Co., Reading, MA (1979).

Accepted for publication September 22, 1993.

Anthony B. O'Hare *IBM Networking Hardware Division, P.O. Box 12195, Research Triangle Park, North Carolina 27709 (electronic mail: ohare@vnet.ibm.com).* Dr. O'Hare received his Ph.D. in computer science from the University of Wisconsin–Madison. Since joining IBM in 1989, he has worked in the areas of test automation and software engineering methodologies and tools, and is currently technical leader of the VIEWS project. Prior to that, he was a principal investigator at the Unisys Paoli Research Center working on parallel, symbolic computation and on the BrAID and IDI intelligent database systems. He was also a senior member of the technical staff at the Microelectronics Computer Technology Corporation (MCC) on the Logical Database Language (LDL) project. His interests include software engineering, knowledge-based systems, expert databases, and parallel and distributed systems.

Erik W. Troan *Department of Computer Science, North Carolina State University, Box 8206, Raleigh, North Carolina 27695-8206 (electronic mail: ewt@sunsite.unc.edu).* Mr. Troan is an undergraduate studying computer science at North Carolina State University in Raleigh. After graduating from the North Carolina School of Science and Mathematics in 1991, he worked part time on the VIEWS project through June 1993.

Reprint Order No. G321-5535.