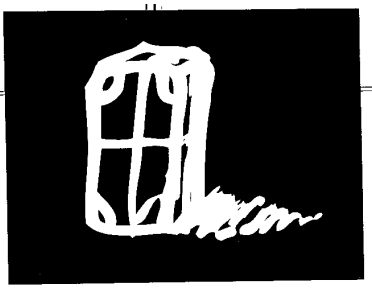


Reengineering User Interfaces

ETTORE MERLO, *Ecole Polytechnique de Montréal*
PIERRE-YVES GAGNÉ, *McGill University*
JEAN-FRANCOIS GIRARD,
Centre de Recherche Informatique de Montréal
KOSTAS KONTOGIANNIS, LAURIE HENDREN,
PRAKASH PANANGADEN, *and* RENATO DE MORI
McGill University

◆ *Most developers would like to avoid redesigning a system around a new interface. But turning a character-based interface into a graphical one requires significant time and resources. The authors describe how this process can be partially automated, giving the results of their own reverse-engineering effort.*

Many older programs, especially those in data processing, have a character-based user interface. Reengineering these interfaces to make them more user friendly would extend the life of the systems in which they are embedded, but it can require significant effort. Developers must understand how the old interface was conceived and implemented, and what constraints the new interface must respect to remain compatible with the rest of the system. To make the level of effort acceptable, developers need automatic or semiautomatic tools for reengineering.

In this article, we report on an effort to develop such tools. Our work was part of the IT Macroscope project, managed by DMR Group, Inc., whose overall objective was to produce a comprehensive set of methodologies, tools, and learning materials that would support organizations in better managing their business processes through information technology. Our focus — the reverse engineering and reengineering part of the project — was to define an interface-reengineering process that would let developers shift from a character-based paradigm to one based on graphical objects.

This work, which was conducted at the Computer Research Institute of Montreal, went beyond merely restructuring or repackaging the interface. We had to provide some way for developers to gain an in-depth understanding of the existing interface. This, in turn, required that we provide a suf-

0740-7459/95/$04.00 © 1995 IEEE

ficient level of abstraction and an inference capability. Our process had to be flexible enough to work with a range of source code and target languages.

SCOPE AND OBJECTIVES

As part of the work to develop a reengineering process, we investigated the feasibility of reverse-engineering a user interface. Our specific tasks included

♦ *Creating and refining a process to obtain the structural and behavioral specifications that correspond to the original interface.* We wanted developers to be able to extract user-interface specifications from source code, convert them into graphical specifications, and generate a new user interface. This task involved developing an interface-specification language[1] that uses an object-oriented approach to represent user-interface structures and Milner's process algebra[2] to model user-interface behavior. It also involved developing a set of specialized analysis tools that would let programmers approximate program properties, including data- and control-flow analysis, multivalued constant propagation,[3] and slicing.[4] These tools can be used to automatically extract some information or, when automation is not feasible, to assist in manual reengineering.

♦ *Developing a research prototype based on the process and testing it on a real system.* The prototype consisted of parsers, analyzers, and code generators and was implemented using Reasoning Systems' Refine environment. The test system we selected was part of a larger management-information system. The company was interested in reengineering a character-based interface into a graphical one, while preserving the same back end and extending the system so that the interface could run on a remote platform. The part of the system we studied consisted of six programs — a total of 24,000 lines of code — and two communicating transactions (sequences of programs executing and transferring control to one another). The programs ran in the Cobol/CICS environment and used the CICS Basic Mapping Support system for terminal display and user interaction.

Our specific constraints were working within a new client-server architecture and aiming for zero modifications to the application.

REENGINEERING MODEL

Figure 1 shows a model of the reengineering cycle, which consists of the following phases:

♦ *Understand the original system (interface code) and extract code fragments.* This involves parsing the system (analyzing its syntax) to obtain an abstract syntax tree representation of the source code, extracting the interface fragments from the rest of the system, and performing flow analyses to gather flow information that will be used in later parts of the cycle.

The fragments are produced by separating the interface code from the rest of the system and making the interface-code fragments subtrees in the source-code AST. The separation task may be difficult if the interface and application code are very interdependent. It may also be relatively easy if the interface functions are isolated in specially identified modules with a clear interface to the rest of the system. Slicing produces integration constraints at an interface cut point — the boundary between two consecutive statements, only one of which belongs to the interface. These constraints must be satisfied when the new interface is integrated with the original code. In general, they involve old dependencies, consistency of accessed routines and data structures, and so on.

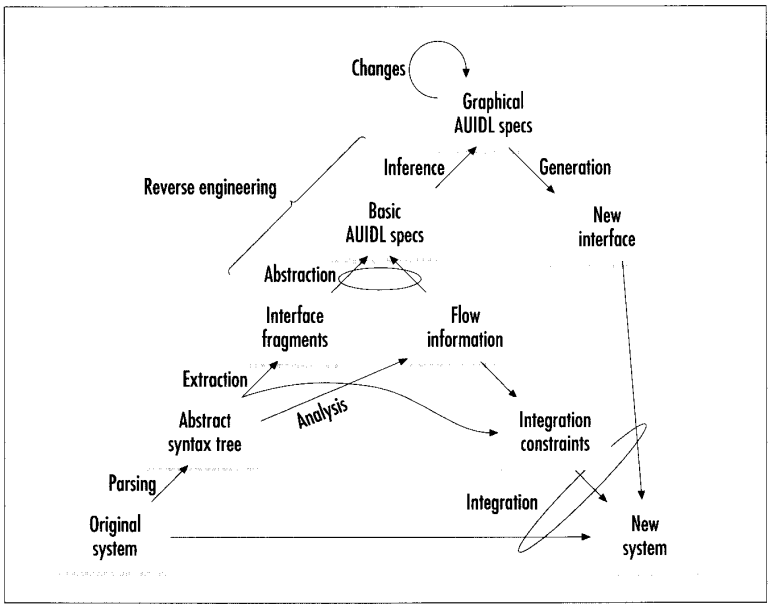
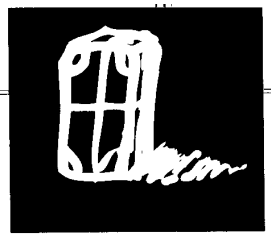


Figure 1. *Reengineering model. Reengineering begins with the original system (lower left corner) and continues through to the implementation of the new system. A key part of the process is the reverse engineering of the old interface, which involves extracting the interface specification, converting it to specifications in the Abstract User Interface Description Language, and then converting the basic AUIDL specification to a graphical AUIDL specification in terms of what the user perceives as screens.*



♦ *Perform abstraction and inference to obtain interface specifications.* This phase takes the extracted interface code fragments from the extraction phase and converts them into graphical specifications in terms of screens.

As Figure 1 shows, abstraction and inference form the core of reverse-engineering operations. They essentially shift the paradigm from the world of control flow to the world of observable behavior from the user's point of view. From interface fragments and flow information, abstraction produces basic specifications in the Abstract User Interface Description Language,[1] a language we designed to provide an abstract representation of the original character-based interface. Inference converts the character-based, or *basic*, AUIDL specifications into *graphical* AUIDL specifications that use graphical constructs to describe objects, screens, and screen sequences that the user can perceive.

AUIDL LETS YOU AUTOMATICALLY GENERATE NEW GRAPHICAL INTERFACES TO AID IN PROTOTYPING.

♦ *Introduce improvements to the interface.* During this phase, developers can change the graphical AUIDL specifications. In our case, we did some restructuring by adding mouse interaction in the form of pushbuttons to the function keys. We could have done much more automatic restructuring but we had limited our prototype to function-key restructuring. Our main goal was to determine the feasibility of the reengineering approach; more sophisticated restructuring would have required a process similar to interface design and involved analyzing programmer tasks and considering ergonomical issues — both of which were beyond the scope of our project.

♦ *Generate a new graphical interface.* Developers can automatically generate a new graphical interface from the graphical AUIDL specifications using standard code-generation techniques. This is particularly important for prototyping to quickly evaluate interface redesigns. In our environment, we implemented a specific code generator that takes as input the AST produced by parsing graphical AUIDL specifications and outputs Easel code.

♦ *Integrate the new interface into the original system.* The difficulty and thus length of this phase can vary. In our case, it was practically negligible because we were able to use Easel Corp.'s Easel, an environment that lets you develop GUIs that run on desktop computers under Windows, OS/2, or DOS and that communicate with IBM mainframe applications through an Easel 3270 terminal emulator. No integration effort and no modification of the mainframe application were required.

In general, though, we expect integration to be more time-consuming than this. Developers are likely to have to change the mainframe code, including initializing parameters, supplying data, and starting new server processes that the client may need. Further, they must take care to make the communication protocol to exchange data and control explicit. This means generating either automatically or semiautomatically the extra code on both the client and server sides.

Moreover, the new code cannot interfere with the system's original behavior. Developers can use integration constraints to check that old dependencies have not been altered. For example, they can check that the names of variables in the new code do not conflict with existing names and that the names and the initial values of parameters do not alter the original control flow except for the desired interface behavioral changes.

This model is a variation of a general reengineering model,[5] which we tailored to user interfaces by adding flow information and integrating constraints across the reverse-engineering and forward-engineering branches. In general, the model was successful. Our main difficulties were in choosing the representation language, defining specific flow analyses, and refining the abstraction and inference processes. The remainder of the article is devoted to these issues.

AUIDL

As Figure 1 shows, AUIDL was a big part of our reengineering model and the key to our success in reverse-engineering an interface. With AUIDL we were able to express interface behavior in a hierarchical, formal way, add ergonomic improvements to the interface design (such as mouse manipulation), and have the flexibility to evolve as new dialogue models and control structures were created. AUIDL also gave us a sufficient level of abstraction to handle a variety of languages.

In Figure 1, we refer to basic and graphical AUIDL specifications. AUIDL can represent a range of user-interface types, including the main two, character-based and graphical. The main reverse-engineering task becomes how to map the existing interface, written in a specific language, to this abstract and general representation language. With this flexibility, AUIDL is suitable for several roles within the reengineering cycle: It can be a target language for reverse engineering, a working specification language for redesigning the interface, and a specification language for generating an interface for a specific platform.

Representation. AUIDL lets you represent structure with object-orientation's inheritance and describe behavior with Milner's process algebra.[2] Interface structure captures the possible relations among interface entities, while behavior describes the interface's dynamic aspects. Thus, AUIDL unites two major design paradigms. It also lets you formally describe complex interactions as direct manipulation (moving an object) and time-

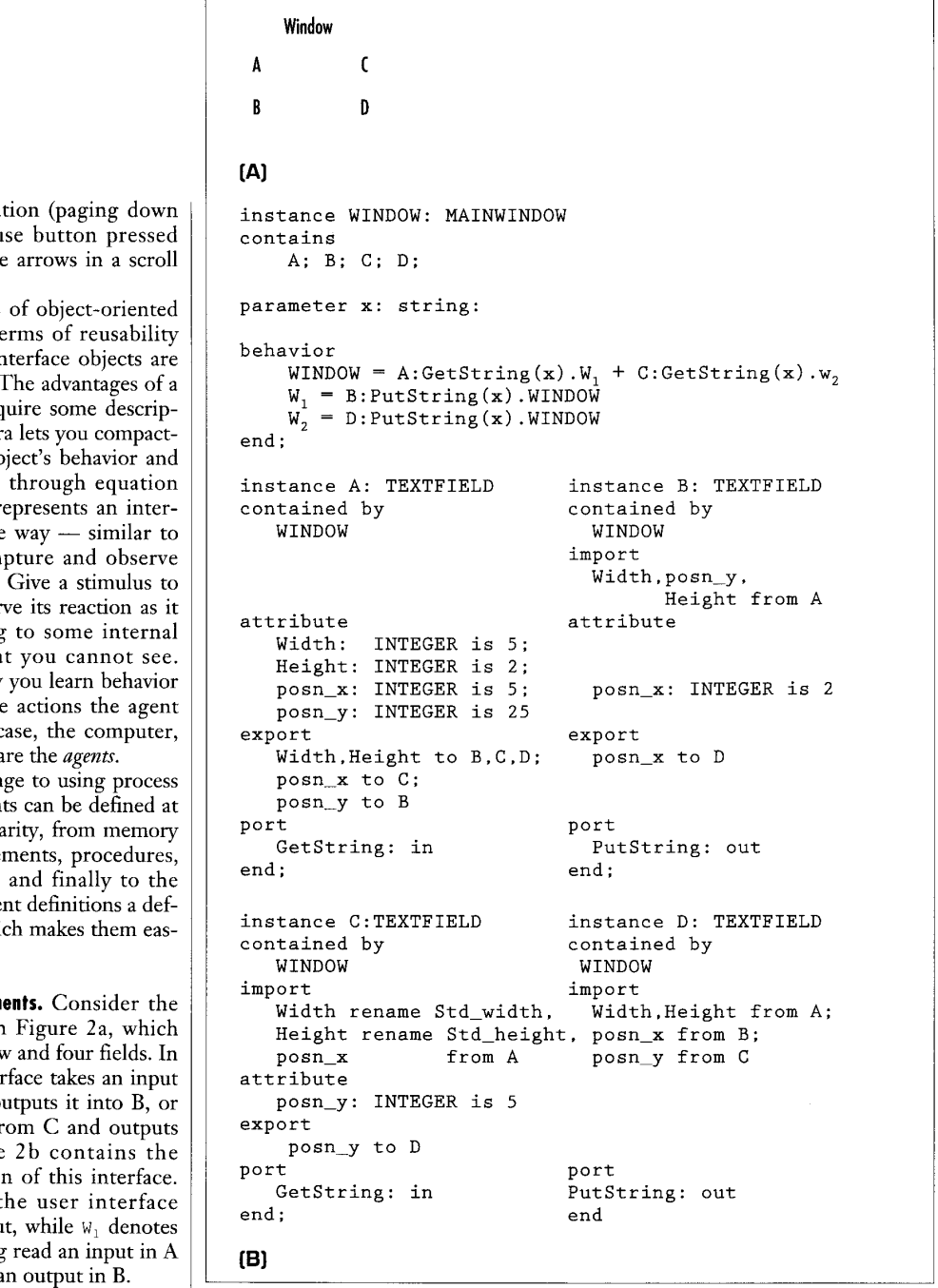


Figure 2. *(A) A simple user interface and* **(B)** *its description in AUIDL.*

sensitive manipulation (paging down by keeping a mouse button pressed and pointing to the arrows in a scroll bar).

The advantages of object-oriented specifications in terms of reusability and hierarchy of interface objects are fairly well-known. The advantages of a process algebra require some description. Process algebra lets you compactly describe each object's behavior and redesign behavior through equation rewriting. It also represents an interface in an intuitive way — similar to the way people capture and observe behavior naturally: Give a stimulus to an agent and observe its reaction as it behaves according to some internal programming that you cannot see. Thus, the only way you learn behavior is by observing the actions the agent performs. In this case, the computer, interface, and user are the *agents*.

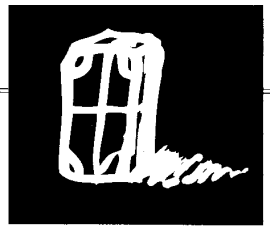
Another advantage to using process algebra is that agents can be defined at any level of granularity, from memory cells through statements, procedures, modules, systems, and finally to the user. This gives agent definitions a definite hierarchy, which makes them easier to analyze.

Specification elements. Consider the simple interface in Figure 2a, which consists of a window and four fields. In the figure, the interface takes an input from field A and outputs it into B, or it takes an input from C and outputs it into D. Figure 2b contains the AUIDL description of this interface. `WINDOW` denotes the user interface waiting for an input, while W_1 denotes the interface having read an input in A and ready to write an output in B.

Within AUIDL, you define a class hierarchy of graphical objects in the interface. Two mechanisms explicitly describe the spatial organization of display objects: *containment* and *importation*. Containment means an object is contained in another (an object is inside a window). To express containment, we introduced two keywords, `contains` and `contained by`.

Importation lets an object import an attribute and its value from another object. This provides an easy way to maintain the consistency of common attributes. With this mechanism, you can keep two fields aligned or ensure that they have the same size or color, for example. Again, two keywords, `import` and `export`, describe the relation in both objects.

A behavior section describes how an object reacts to events through a list of equations. On the left side of these equations, an identifier repre-



sents an agent in a certain state, in this case, WINDOW. The right side shows what interactions are possible from this state and the state after the interaction. An action that takes place in a specific object is denoted by <object_name>:<action_name>. In the behavior section for WINDOW, for example, A:GetString(x) means that object A performs the action of reading a string (GetString) on parameter x. The "+" indicates nondeterministic alternatives; that is, you can choose A or C to start with, but once started you must proceed as W_1 or W_2, respectively (because after an input, nothing else can be done until the corresponding output is observed). The dot denotes sequencing, so

```
WINDOW = A:GetString(x).W1 +
C:GetString(x).W2
```

means that WINDOW behaves as a nondeterministic choice between reading a string x in A and behaving subsequently as W_1 *or* reading a string x in C and behaving subsequently as W_2, where W_1 outputs x in B and proceeds as WINDOW, and W_2 outputs x in D and proceeds as WINDOW.

```
MAPSET_W      DFHMSD
                 MODE=INOUT,
                 LANG=Cobol,
                 TERM=3270-2,
                 CTRL=(FREEKB,FRSET),
                 TIOAPFX=YES
M1            DFHMDI
                 SIZE=(018,080),
                 LINE=004,
                 COLUMN=001
              DFHMDF
                 POS=(001,010),
                 ATTRB=(ASKIP,NORM),
                 LENGTH=025
                 INITIAL= "NO.EMPL."
```

(A)

```
instance M1 :    MAP
contained by
       MAPSET_W
contains
       Field1;  UNNAMED_Field2;     Field3;

attribute
       SIZE_LINE: INTEGER is 18;
       SIZE_COLUMN: INTEGER is 80;
       LINE: INTEGER is 4;
       COLUMN: INTEGER is 1;
end;

       ...

instance UNNAMED_Field2:  UNNAMED_FIELD
contained by
       M1

attribute
       POS_LINE: INTEGER is 1;
       POS_COLUMN: INTEGER is 10;
       STATUS: STAT_ATTR is ASKIP;
       INTENSITY: INT_ATTR is NORM;
       LENGTH: INTEGER is 25;
       INITIAL: STRING is "NO.EMPL.";
end;

....
```

(B)

Figure 3. *(A) A sample file in the Basic Mapping Support language and (B) the BMS file translated into an AUIDL structural specification.*

APPLICATION ISSUES

Using our prototype, programmers extracted structural properties, in the form of flow information, by analyzing the information sent to the screen. With the prototype, we provided tools such as specialized control-flow analysis, multivalued constant propagation, and slicing.

We captured the interface's behavioral properties through transition graphs derived from the system control-flow graph. We then used inference to transform these graphs into a graphs whose transitions are in terms of screen displays.

Structural analysis. In our environment, as in many old systems, interface code was scattered over several Cobol programs that interacted somehow with the user, but the CICS package supplied an easily identifiable interface to the rest of the system. We decided to consider all CICS statements as belonging to the interface and the remaining Cobol statements as belonging to the application system. This criterion had several advantages. It was easy to implement, it was not ambiguous, and it made sense semantically in the Cobol/CICS world. Other criteria might involve evaluating properties of the dependency graph, such as slices. However, rather than having a complex separation criterion with deep implications for the application code, we opted to use this simple criterion along with a complete dependence analysis and supply the analytic tools to the programmers.

The CICS environment posed special difficulties, however, because of the type of display mechanisms it uses.

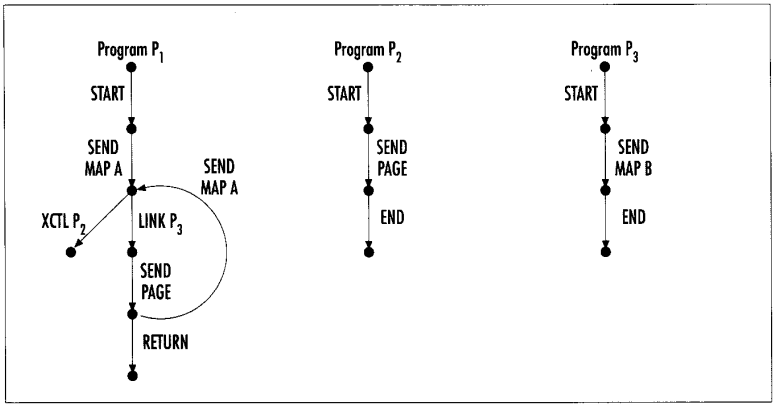


Figure 4. *User-interface transition graph. A transition graph represents the interface's behavioral description in terms of states and transitions. Transitions also describe control flow at this point.*

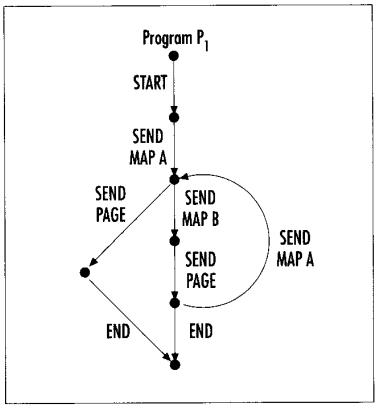


Figure 5. *The transition graph in Figure 4 after behavioral abstraction. After abstraction, the transition graph is in terms of basic AUIDL action only, like SEND MAP and SEND PAGE.*

CICS treats each portion of screen as a single processable unit, a *map*. The static description of a map, including the fields that compose it, is written in Basic Mapping Support language. Figure 3a shows a sample BMS file.

Maps or sets of maps are accumulated in memory through the statement SEND MAP and are sent to the screen with the statement SEND PAGE. The display operation in CICS is essentially the result of sending some maps to the screen using some SEND MAP commands followed by a SEND PAGE command. Input operations consist of reading some maps through the RECEIVE MAP command. At runtime, the accumulated maps and the pages sent change according to program execution.

By tracing the sequence of SENDs using control-flow analysis, we were able to identify the maps or map sets for a particular screen. However, their names are often hidden in variables, which generally have a number of permitted values. Because these values represented interface structures that could be displayed to the exclusion of each other, we had to identify the permitted variable values at any given point in the application program. To do this, we used a specialized analysis based on multivalued constant propagation. The box on p. 70 explains more about MVCP analysis and issues specific to our environment.

Figure 3b shows the final AUIDL specification translated from the BMS file in Figure 3a.

Behavioral abstraction. Abstracting the interface's behavioral properties requires shifting perspective from the world of control transfer to the world of behavior and interaction. The control-flow graph is a good representation of control, while the interface's *transition graph* is a good representation of interface behavior. The transition graph represents interface behavior in terms of states and transitions and is used to produce the basic AUIDL specifications.

Figure 4 is a sample transition graph that corresponds to three programs P_1, P_2, and P_3 belonging to the same CICS transaction (sequence of programs executing and transferring control to one another). The box on p. 70 gives more details on transactions.

In the graph, START identifies the unique entry point to a program; END terminates a program's execution; RETURN, LINK and XCTL represent the interprocedural control-transfer instructions; and the CICS statement SEND MAP prepares maps A and B to be displayed on the screen by a SEND PAGE statement.

Behavioral abstraction consists of removing the interprocedural control information from the transition graph. In this case, we must remove the arcs XCTL and LINK so that the final transition graph contains only behavioral information. Arcs containing XCTL or LINK statements are replaced with the complete transition graph corresponding to the called programs.

As Figure 4 shows, the edge XCTL P_2 in program P_1 has been replaced by the SEND PAGE and END edges in P_2. To replace LINK, the LINK P_3 edge has been replaced by SEND MAP B in program P_3.

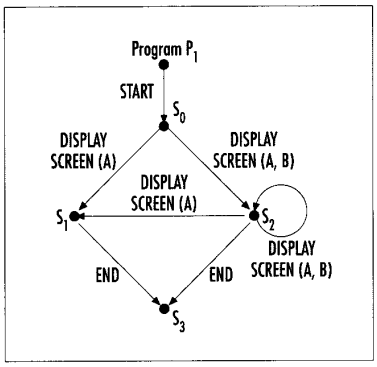


Figure 6. *The transition graph in Figure 5 after inference. Inference is the process of converting the transitions that are in terms of basic AUIDL actions to transitions in terms of graphical AUIDL actions like DISPLAY SCREEN.*

Figure 5 shows the graph in Figure 4 after abstraction. Only basic AUIDL actions, like SEND PAGE and SEND MAP, appear as transitions. The XCTL and LINK statements have been replaced, and the RETURN and END edges in programs P_1 and P_2, respectively, have their endpoints merged and relabeled END.

Inference. Inference consists of iteratively traversing the transition graph to collect the map sets that can be encountered on paths between SEND PAGE or RECEIVE statements[6] and producing a transition graph with only

INTERPROCEDURAL ISSUES AND MVCP

Analysis involving multivalued constant propagation was crucial to obtaining structural descriptions. MVCP's main role was to handle the interprocedural dataflow aspects inherent in a Cobol/CICS environment. Specifically, flow information can come into a code fragment from many calling points and from parameters that are sometimes passed among programs through shared memory and pointers. In Cobol/CICS, these calling points are represented by PERFORM, LINK, and XCTL statements.

Handling transactions. Further complicating the issue is the presence of *transactions*, sequences of programs executing and transferring control to one another. Figure A shows a sample transaction, which is defined in terms of three programs. Each of these programs transfers control to the others using LINK, XCTL, and RETURN constructs. Within one program, the PERFORM construct acts as a call subroutine mechanism. Thus, we considered all the PERFORM, LINK, and XCTL constructs as procedure calls.

Modifying the control-flow graph. MVCP analysis proceeds in the following manner. MVCP flow information is tagged by a string that summarizes the activation stack of procedure calls. We could then use these call strings to distinguish among flow information from different calls, thus taking into account interprocedural aspects. We used the output of MVCP analysis — variables together with their possible values — to modify the system control-flow graph. Figure B shows a sample control-flow graph. For each SEND MAP statement that contains a variable, we added nodes in the control-flow graph with the corresponding constant map values detected by MVCP analysis. Figure C shows the resulting graph. For example, suppose MVCP analysis revealed that the $_X$ in SEND MAP M_X in Figure B could be a 1, 2, or 3. We simply change the node SEND MAP M_X to three corresponding alternative nodes: SEND MAP M_1, SEND MAP M_2, and SEND MAP M_3, as shown in Figure C.

Although the interprocedural transfer of control is particularly tricky to analyze in the CICS environment, MVCP analysis would be useful to any environment that lets you display operations of structures on the screen or lets you store structure names and use them through variables. Of course, each environment comes with its own set of problems. We found MVCP analysis suitable for environments with sequential transfer of control and with procedures that do or do not have parameters. We have not addressed issues concerning pointers to dynamic structures, the dynamic structures themselves, or recursive procedures.

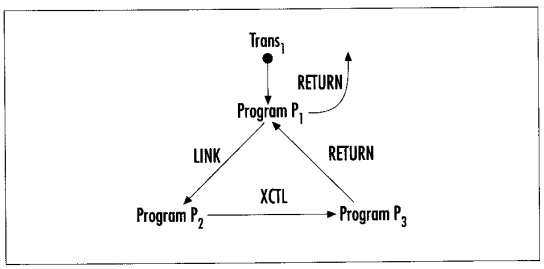


Figure A. *Sample transaction. Three programs make up this transaction, each of which uses certain constructs to transfer control to the others. This greatly complicates the structural and behavioral analyses of AUIDL specifications.*

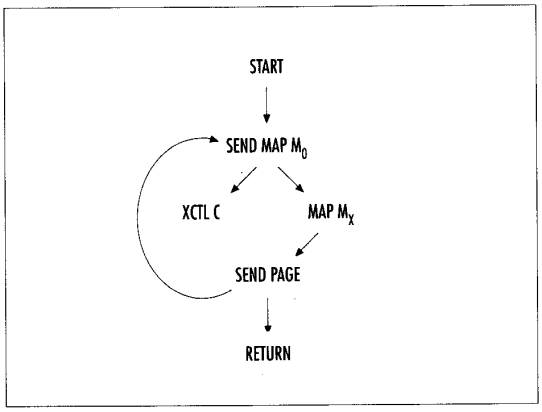


Figure B. *Sample control-flow graph that contains SEND MAP variables.*

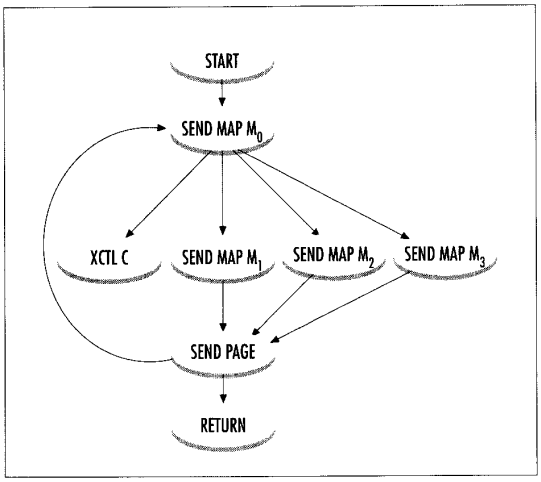


Figure C. *Graph in Figure B after MVCP analysis, in which SEND MAP variables are replaced with constants.*

graphical AUIDL actions as transitions. Figure 6 shows the graph in Figure 5 after inference is applied. Only graphical AUIDL actions like `DISPLAY SCREEN` appear.

Representation choices. Although we accessed AUIDL behavioral specifications through graphs, programmers can opt to display them in terms of process-algebra equations in text format using this simple transformation algorithm:

- Take each node in the transition graph and make it an agent in process-algebra equations. Every node in the transition graph is given a unique name. A "+" in the equation indicates a branch in transition graph. Graph vertices represent AUIDL behavioral states, and edges represent interface communication actions.
- An edge between two nodes, N_1 and N_2, corresponds to the equation N_1 = action.N_2.
- Iteratively traverse all nodes and edges in the transition graph to produce the equations.

The advantage of process-algebra equations is that they can be manipulated as text. The advantage of transition graphs is that they are natural mechanisms for visualization and manipulation in a graphical environment.

Problems. Inference in our case gave rise to specific problems, including how to solve interprocedural issues and how to treat `ERASE`, a statement that stops the flow of maps along a certain path in the transition graph.

To solve the interprocedural problems, we used a technique similar to what we did during MVCP analysis: tagging the flow information with call strings and mapping and unmapping procedural parameters.

We solved the `ERASE` problem by collecting two kinds of map flow information for each statement.[6] One kind represents the current state of the screen in terms of maps; the other represents the cumulative flow of maps since the last `SEND PAGE` was encountered in the flow graph.

Figure 7 gives the AUIDL behavioral specification for the transition graph in Figure 6, where `SCREEN(A,B)` denotes the screen that corresponds to the CICS maps A and B, and `DISPLAY` is the action of screen visualization.

RESULTS

As part of our work, we developed a research prototype of our approach. The system consisted of

- extensions to the Refine/Cobol parser to analyze CICS constructs;
- MVCP and dependencies analyzers;
- programs for extracting interface fragments and performing abstraction and inference;
- an AUIDL parser; and
- the Easel code generator.

After using the prototype on the test system, we identified many different functionalities in the interface as screens, including

- Data exchange only: 14
- Data exchange and function keys: 3
- Data exchange, function keys, and headers: 5
- Combinations of data exchange, function keys, headers, and error messages: 178

This corresponds to a total of 13 CICS maps: eight maps for data exchange, three maps containing headers, one map defining function keys, and one map containing an error message. These results suggest that if data exchanges, function keys, headers, and error messages are considered as different kinds of dialog boxes, the complexity of the interface specifications is still low. If, on the other hand, you consider every screen as a distinct object, you would get 200 different screens and a very complex interface specification.

We included automatic restructuring of function keys into pushbuttons. Other forms of automatic restructuring involving headers and error messages could easily be implemented.

We validated the extracted AUIDL specifications by inspection. Afterward, the user selected six data-exchange screens and one error screen for automatic interface generation on an IBM PC under OS/2. We then tested the generated interface using a coverage criterion, which says that the set of test cases, when executed, should cause the flow of control to go through paths whose union covers the entire program. We were able to demonstrate to the user that the new interface correctly corresponded to the original system in both layout and functionality. Our correspondence criteria were

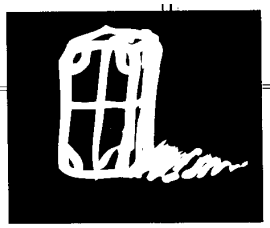
- Every interaction (I/O) possible in the old screen is possible in the new one.
- The intrascreen navigation possible in the new screen was also possible in the old screen and vice versa.
- The sequence of screens (interscreen navigation) possible in the new interface was also possible in the old system and vice versa.

```
S0 = SCREEN(A):Display.S1
     + SCREEN(A,B):
     Display.S2

S1 = S3

S2 = SCREEN(A):Display.S1
     + SCREEN(A,B):
     Display.S2 + S3

S3 = nil
```

Figure 7. *Example of AUIDL behavioral specification in the form of process-algebra equations for the transition graph in Figure 6.*

Achievements. The most visible result of our work was the proof that our interface reengineering approach was valid. We successfully extracted graphical interface specifications that correctly represented the existing system from the 24,000 LOC of Cobol/CICS source code. These specifications, furthermore, were suitable for automatically generating a new interface on a PC.

Our approach is also suitable for



other source code and target languages. Changing the version of Cobol would require modifying only the Cobol parser. To handle the CICS language, for example, we simply modified the Refine/Cobol parser. Changing the programming language could be slightly more complicated because it might affect other operations in the reengineering model, like abstraction and inference. However this is true only if the semantic model changes. If developers stick to imperative languages, which have procedures and parameter-passing mechanisms, modifications should be slight.

Another advantage of our approach is that it separates interface and application code, which in conjunction with the use of the Easel environment, let us generate new interface code for a PC without modifying the old code. If we had had to modify the application code on the mainframe, our task would have been more complex.

OUR APPROACH SEPARATES THE INTERFACE AND APPLICATION CODE, SO YOU CAN GENERATE NEW INTERFACE CODE FOR A PC WITHOUT MODIFYING THE OLD.

Finally, because AUIDL specifications are also available in text format, programmers can easily modify them. This adds flexibility to our automated restructuring rules. Developers can use AUIDL for fast interface prototyping after reverse engineering.

Limitations. Our approach is limited by the approximation inherent in static analysis. CICS is a highly dynamic model of user interaction. Our challenge was to use static analysis to determine dynamic behavior, the possible sets of maps a user could perceive on the screen and their sequence at execution time. The static approximation may identify screens that were artificially introduced by our approach and that correspond to paths that may not be executable at runtime. Although we did not observe such screens in our specifications, they may appear and must be removed manually.

We have not investigated ways to improve this approach, such as analyzing traces and profiles of program execution. Using dynamic analysis would also remove this problem, but it would yield less complete results because you can dynamically analyze only a finite number of paths. Thus, both static and dynamic analysis suffer from drawbacks. A comparison of the two might be a worthwhile research topic.

Another limitation is that our approach is currently only semiautomatic. In our test case, programmers specified the result of flow analysis for certain routines. These turned out to be helpful in weeding out parts of the system that were coded in assembly, didn't contribute to the interface at all, or contributed in a very straightforward manner, such as sending pages on the screen. Assembly routines in our system took about 3,500 LOC. This programmer assistance saved us from writing an assembly analyzer, which generally should be done.

We also required manual assistance in MVCP analysis to deal with statements like `LINK program_variable` MVCP analysis depends on the construction of a system control-flow graph, which in turn depends on the results of the analysis itself (the possible values of `program_variable`). The `LINK program_variable` node is expanded into several nodes `LINK` P_n in the control-flow graph, where P_n represents all the programs in the system. A programmer can reduce this expansion to the number of programs he or she *believes* can be accessed from such a statement. A more elegant solution would be to consider a higher order MVCP analysis in which both the control-flow graph and the results of the analysis are incrementally constructed.

Finally, we had to limit the Cobol/CICS computational model so that we could analyze the source code. However, these limitations were not overly restrictive; they simply called for programming styles that would avoid certain structures. Fortunately, these styles often matched the companies' internal programming guidelines.

Although we have described our reengineering model in the context of a specific environment, it as well as AUIDL and the analysis tools are portable. Further research is needed to study slicing and dependence analysis, since these approaches can be valuable in integrating new interface code into the original system.

Some other directions for further work include

- Introducing new design concepts that are totally absent from the character-based design, such as multithread dialogues and concurrency.
- Measuring and evaluating existing systems for ease of interface reengineering. Using the results of flow analysis, several interface diagnostic tools could be developed. Tools could also detect extra execution paths, which might lead to navigations that were not originally designed but that happen to be available.
- Extending the approach to address more client functions. Our target interface architecture followed the client-server paradigm. Other functions on the client side, such as help and data validation, might be migrated to the interface.

We believe our reengineering approach will let the interface of a legacy system evolve as new interface technologies emerge. This extends the life of the system and improves its overall quality. As a bonus, the knowledge gained during reverse engineering will greatly enhance interface maintenance over the years. ◆

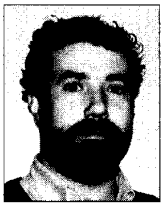
ACKNOWLEDGMENTS

This work has been performed at Centre de Recherche Informatique de Montréal as part of the IT Macroscope project, at McGill University, and at Ecole Polytechnique of Montreal. We thank Alain Thiboutôt for his contribution to slicing and specification inference, Jonathan Levine and Abdu El-Wahidi for their contribution to the AUIDL parser, and Kristina Pitula for her part in the structural analysis of AUIDL specifications.

This work has been funded in part by the IT Macroscope project, managed by DMR Group Inc., (Montreal) and in part by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

1. E. Merlo et al., "Reverse Engineering of User Interfaces," *Proc. Working Conference on Reverse Eng.*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 171-179.
2. R. Milner, *Communication and Concurrency*, Prentice-Hall, New York, 1989.
3. E. Merlo et al., "Multi-Valued Constant Propagation for the Reengineering of User Interfaces," *Proc. Conf. Software Maintenance*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 120-129.
4. K. Gallagher and J. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Trans. Software Eng.*, Aug. 1991, pp. 751-761.
5. E. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, Jan 1990, pp. 13-17.
6. E. Merlo, P. Gagné, and A. Thiboutôt, "Inference of Graphical AUIDL Specifications for the Reverse Engineering of User Interfaces," *Proc. Int'l Conf. Software Maintenance*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 80-88.



Ettore Merlo is an assistant professor of computer engineering at Ecole Polytechnique de Montréal, where his research interests include software reengineering, software analysis, and artificial intelligence. He also participates in the Program Understanding project undertaken by IBM Toronto's Software Solutions Laboratory, Center for Advanced Studies. The project, which addresses the problem of program understanding through reverse-engineering technologies, involves a team from CAS and five research groups working cooperatively on complementary reverse-engineering approaches. All the groups are using the source code from IBM's SQL/DS relational database-management system as the legacy system.

Merlo received a Laurea from the University of Turin (Italy) and a PhD from McGill University, Montreal — both in computer science. He is a member of the IEEE Computer Society.

Pierre-Yves Gagne is programmer analyst with the DMR Group and an MSc candidate at McGill University. His research interests include reverse engineering and object-oriented programming.

Gagne received a BA in mathematics and economics from McGill University.

Jean-Francois Girard is a researcher at the Computer Research Institute of Montreal. His interests include software maintenance, reverse engineering and object-oriented programming.

Girard received a BSc in mathematics and computer science and an MSc in computer science, both from McGill University.

Kostas A. Kontogiannis is a research assistant and PhD candidate in computer science at McGill University. He also participates in the Program Understanding project undertaken by IBM Toronto's Software Solutions Laboratory, Center for Advanced Studies. His research interests are the development and application of pattern-matching techniques for code and plan localization, mathematical logic, specification languages, artificial intelligence, and expert systems.

Kontogiannis received a BSc in mathematics from the University of Patras, Greece, and an MSc in artificial intelligence from the Catholic University of Leuven in Belgium. He holds a fellowship from IBM Canada's Centre for Advanced Studies.

Laurie Hendren is an assistant professor of computer science at McGill University, where she leads the McCAT optimizing/parallelizing compiler project. The compiler is being used to experiment with new compiler analyses and transformations and as a testbed for applying compile-time analysis to the reverse-engineering of C programs. As a student at Queen's University, Kingston, she participated in the development of Q'Nial, a Nested Interactive Array Language. As part of her PhD dissertation, she implemented a prototype compiler that automatically translated sequential imperative programs to parallel programs for the BBN Butterfly.

Hendren received a BSc and MS from Queen's University and a PhD from Cornell University — all in computer science.

Prakash Panangaden is an associate professor of computer science at McGill University. His research interests include concurrency theory, linear logic, parallel programming, and the relationships between physics and computation.

Panangaden received an MSc from the Indian Institute of Technology, Kanpur, and a PhD from the University of Wisconsin, Milwaukee — both in physics. He also received an MS in computer science from Cornell University.

Renato De Mori is a professor of computer science at McGill University and director of the university's School of Computer Science. He is also the principal investigator for the Montreal research node in the program understanding project undertaken by IBM Toronto's Software Solutions Laboratory, Center for Advanced Studies. His research interests are in stochastic parsing techniques, automatic speech understanding, connectionist models, and reverse engineering. He is also on the editorial board of *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *Signal Processing*, *Speech Communication*, *Pattern Recognition Letters*, *Computer Speech and Language*, and *Computational Intelligence*.

De Mori received a doctorate in electronic engineering from Politecnico di Torino. He is a fellow of the IEEE Computer Society.

Address questions about this article to Merlo at Ecole Polytechnique de Montréal, Dept. de Genie Electrique Genie Informatique, 6079, Succ. Centre Ville, Montreal, Quebec, Canada; merlo@rgl. polymtl.ca.