This work appeared as “Reengineering with Reflexion Models: A Case Study”, *Computer 30*, 8, 1997, pp.29-36.

Reengineering with Reflexion Models: A Case Study

Reengineering large and complex software systems is often very costly. Reflexion models let software engineers begin with a structural high-level model that they can selectively refine to rapidly gain task-specific knowledge about the source code. The authors describe how a Microsoft engineer used this technique in an experimental reengineering of Excel.

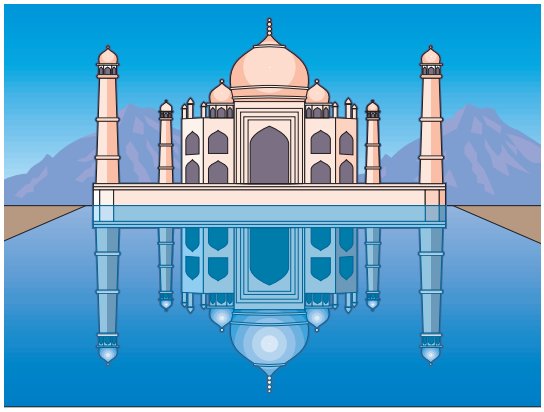
Gail C. Murphy
University of British Columbia

David Notkin
University of Washington

To effectively perform most software engineering tasks on an existing system, a software engineer must have some understanding of the system's source code. However, gaining insight into the source code of large and complex systems typically takes too long and costs too much. In an attempt to address this problem, we developed the software reflexion model technique,[1] which lets software engineers rapidly and cost-effectively gain task-specific knowledge about a system's source code. In this article we give an overview of our technique and then relate how a Microsoft engineer used it to aid an experimental reengineering of Excel—a product that comprises about 1.2 million lines of C code.

Our technique begins with a high-level model, which users define on the basis of a desired software engineering task. We had often seen software engineers use an informal structural model, say sketched on a white board, to begin reasoning about systems. However, reasoning about a task in this way carries significant risk because the model is disconnected from the source. Thus, the next steps in our technique are to extract a model of the source, define a map, and, through a set of computation tools, compare the two models. This lets software engineers effectively validate their high-level reasoning with information from the source code.

To satisfy the need for a quick and inexpensive method, we made the technique "lightweight" and iterative. The user can easily and rapidly access the structural information of interest and can balance the cost of refinement with the potential benefits of a more complete and accurate model. The engineer in our case study—a developer with 10-plus years at Microsoft—specified and computed an initial reflexion model of Excel in a day and then spent four weeks iteratively refining it. He estimated that gaining the same degree of familiarity with the Excel source code might have taken up to two years with other available approaches.



The case study demonstrates not only that our technique and tools can scale up to large real-world problems, but also that our technique is flexible and robust enough to withstand adaptations to suit a particular task or environment.

MOTIVATION

Tool-supported reengineering is usually based on bottom-up reverse engineering approaches in which the source code is analyzed and abstractions of the system are produced using automated and semiautomated techniques. Automated techniques identify structural high-level components by applying either numerical methods[2] or knowledge of structural patterns.[3] With a semiautomated technique,[4] the user is more involved in selecting information of interest from the source code and in clustering it to gradually develop a high-level model.

0018-9162/97/$10.00 © 1997 IEEE

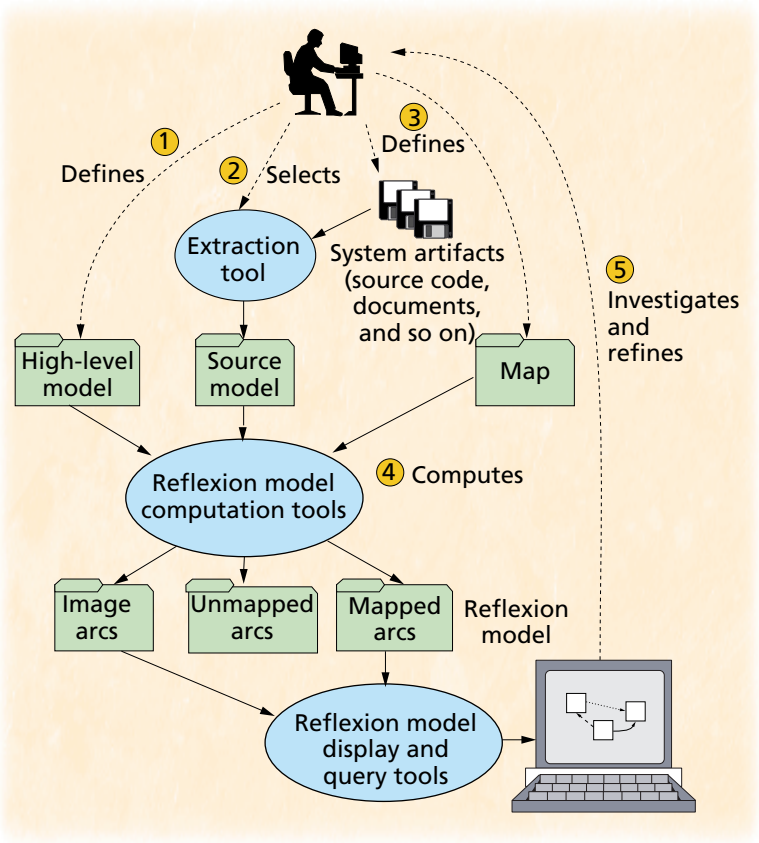


Figure 1. Software reflexion model technique. The user defines a high-level model, extracts a source model, states a map, and then computes a reflexion model. Upon investigation of the reflexion model, the user may iteratively refine the inputs and recompute a new reflexion model.

Both styles apparently have limitations with respect to handling real-world reengineering tasks. Automated techniques are usually based on precoded structural source information (such as data bindings), which is limiting when the reengineering task requires other information (such as event interactions). Semiautomated techniques tend to become increasingly hard to apply as the information extracted from the source grows.

Our technique differs from these by attempting to simultaneously take a top-down and bottom-up approach. Software engineers can quickly form hypotheses from a variety of sources and can then concentrate their effort on getting information about the parts of the system that are relevant to the task at hand.

TECHNIQUE OVERVIEW

The software reflexion model technique has been used to support design conformance tasks and to assess system structure before implementing changes.[5] The software systems involved were fairly large—from several thousand to a quarter million lines of code. As we describe in the next section, we have also successfully applied our technique to a larger system under real-world schedule pressures.

To derive a software reflexion model and iteratively refine it, the user performs five steps, as Figure 1 shows.

Define high-level model

The high-level model describes aspects of the system's structure that aid in reasoning about the software engineering task at hand. This step may involve reviewing artifacts (source code, documents), interviewing experts, or looking at similar architectures—anything that may give information about the system. For example, if the desired activity is to assess how difficult it would be to modify a compiler to generate code for a new platform, the user may choose to reason about the compiler in terms of a dataflow architecture. The High-Level Model window in Figure 2 shows a model of a compiler in which a parser produces an abstract syntax tree (AST) that interacts with a symbol table and code generator to produce object code. This step typically takes 15 minutes to an hour.

Extract source model

The user applies a tool like a call graph extractor or a file dependency extractor to extract interaction information from the source code. This information forms the *source model* that will be compared with the high-level model. In the compiler example, the user is employing a source model that captures calls between C++ methods as an approximation of the dataflow relationships in the high-level model.

Define map

The user then defines a map that describes how entities in the source and high-level models relate. To make this map easier to state, a user may refer to information about both the physical and logical software structures. The user may also employ regular expressions to make it easier to state the map. Figure 2 shows 9 of the 33 lines comprising a map for the compiler. As an example, the first line states that all classes and methods in files with `scanner` in their name will be associated with the Parse high-level model entity. The map is produced manually, but initially it often takes only 10 to 30 minutes to define.

Compute reflexion model

Given a high-level model, a source model, and a map, the user invokes a set of tools to compute a software reflexion model. The reflexion model lets the user see interactions in the source code from the viewpoint of the high-level model. The tools push the interactions in the source model through the map, comparing the mapped interactions in the high-level model with the hypothesized high-level model interactions. The tools consist of three filter programs and eight support programs. They are packaged as separate executables and are accessible through a textual command-line interface as well as a graphical user interface, which is loosely integrated with AT&T's graphviz graph display program.[6]

The Reflexion Model Viewer window in Figure 2 shows the software reflexion model for the compiler example. The model consists of the entities defined in the high-level model and three kinds of arcs. *Convergences*—solid arcs, such as from `Parse` to `AST`—are mapped interactions that agree with the stated high-level model. *Divergences*—dashed arcs, such as from `CodeGen` to `SymTab`—are mapped interactions not stated in the high-level model. *Absences*—dotted arcs, such as from `CodeGen` to

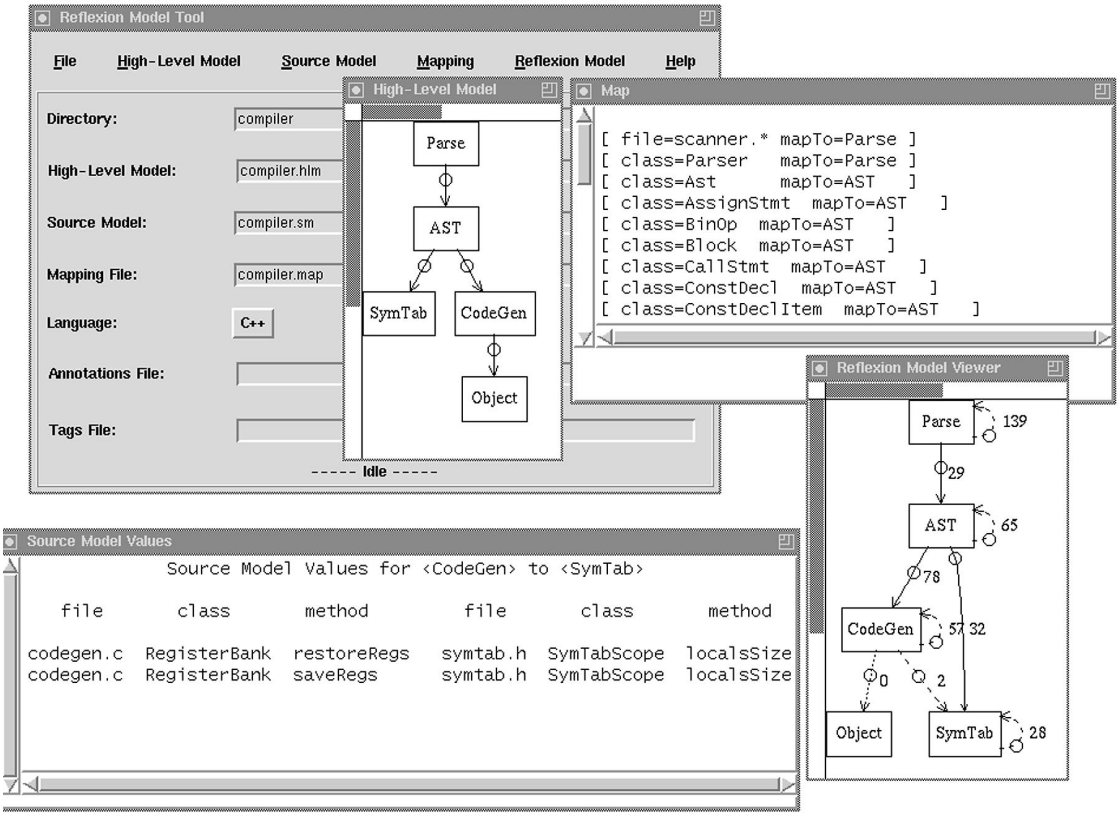


Figure 2. Interface of the software reflexion model tools. The user has specified a high-level model for a compiler and a map; the figure shows the reflexion model computed from these inputs and a source model. The user has selected the arc from `CodeGen` to `SymTab` for investigation. The tools display the calls mapped to this arc. AT&T's graphviz is used for visualizing models. The open circles on arcs are graphviz's arc handles.

`Object`—are interactions stated in the high-level model that do not correspond to any mapped interaction.

The numbers associated with the reflexion model arcs show the number of source model interactions mapped to an arc. This information helps the user assess the system's structure. The "2" associated with the divergence from `CodeGen` to `SymTab`, for example, lets the user assess the degree of unexpected coupling between the modules.

Investigate and refine

Simply viewing a displayed reflexion model does not generally provide sufficiently detailed information for a user to assess, plan, and perform a software engineering task. Typically, she must also investigate the source model interactions mapped to particular arcs in the reflexion model. Sometimes, the information in the reflexion model may reveal missing interactions in the high-level model or deficiencies in the map. For example, a map entry may associate source model entities with the wrong high-level model entity or the map may not be specific enough in capturing the user's intent. This often leads to refinement of the source model, the high-level model, or the map.

The kind of coupling between the `CodeGen` and `SymTab` modules in Figure 2, for example, might affect how the code generator is modified to produce code for a different platform. The Source Model Values window shows the result of a user query about this interaction. Using this information, the user can visit these code locations to help determine the complexity of the task to be performed.

As Figure 1 shows, three files support the display and investigation of a reflexion model.

- The *image arc file* describes the arcs between entities of the high-level model that result from applying the map to the source model.
- The *mapped arc file* describes how interactions in the source model map to arcs described in the image arc file. The information appears when the user clicks on a convergence or divergence.
- The *unmapped arc file* describes the interactions in the source model that are not mapped as part of the computation. A user may access this information to help assess the completeness of the defined map.

Generally, the user iteratively computes and investigates successive reflexion models until she acquires enough structural information for the task being performed.

EXCEL EXPERIMENTAL REENGINEERING

The Excel case study was timely. Although our technique showed promise, we still had many questions about the feasibility of applying it to large systems under the constraints of an industrial environment. About that time, a group at Microsoft was facing the challenge of performing an experimental reengineering of Microsoft Excel. Specifically, they needed to identify and extract components from the source code. To perform this activity within the few months available, the team needed to understand some of the struc-

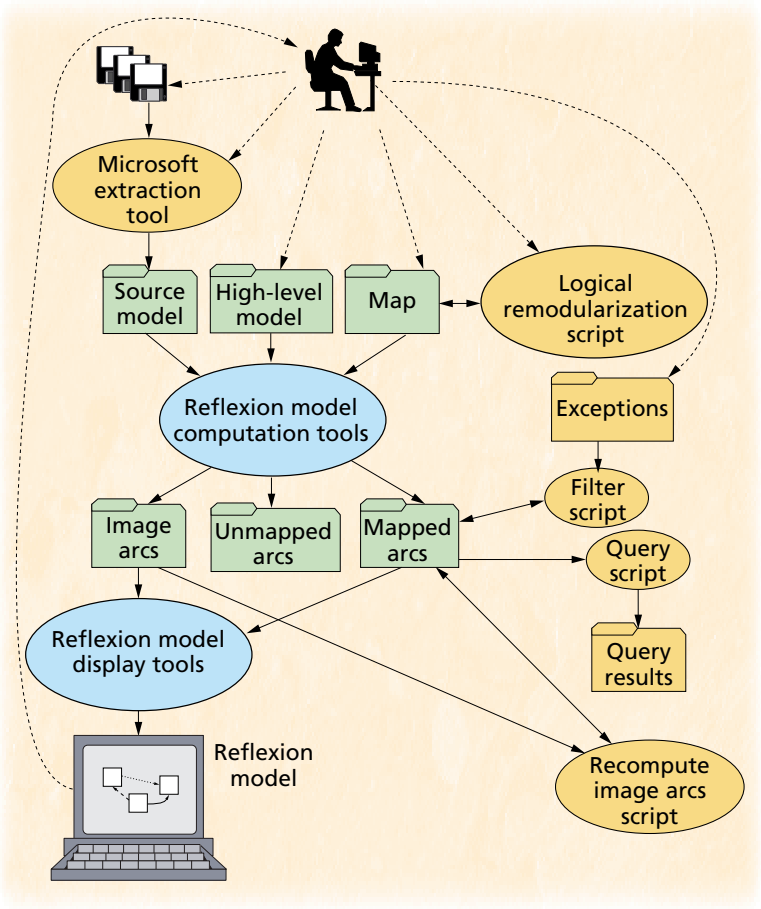


Figure 3. Software reflexion model technique as applied in the Excel experimental reengineering. This process diagram is somewhat different from the generic process diagram in Figure 1. The gold tools and files represent elements the Microsoft engineer added. For example, he created `Query` and `Filter` scripts to support refinements to the reflexion model process and an exceptions file to keep track of the interactions he had reviewed.

ture of the 1.2 million lines of C code. An engineer in this group had heard us present our technique and decided to use it to aid the experimental reengineering.

Figure 3 shows how the engineer adapted the process in Figure 1 to his effort.

Define high-level model

The engineer needed to understand how Excel's C source code divides into static modules and how those modules interact at execution. Traditionally, a Microsoft developer would become familiar with these aspects of Excel's structure by reading a document called "Excel Internals," or he would rely on oral tradition or study the source code. Jon De Vaan describes this process:[7]

> Excel Internals . . . explains the philosophy of a few of the basic things in Excel, like the cell table formulas, memory allocation, a little bit about the layer [a special interface with the operating system that allows Microsoft to use the same Excel core on both Windows and Macintosh platforms]. . . . It's very sparse. We don't necessarily rely on that for people to learn things. I'd say we have a strong oral tradition, and the idea is that the mentor teaches people or people learn it themselves by reading code. . . . Over the course of a project, it goes from mostly truthful to less truthful, and then we have to fix it up. We don't fix it up as we go along on a project. We will give it some attention between projects.

Although this approach may be effective when developing and evolving Excel, it was not appropriate for the experimental reengineering because of the time constraints and because the engineer could not rely on consistent interaction with a mentor from the Excel development group.

Instead, he relied on brief discussions with Excel developers and his development experience. He found it "natural," on the basis of this information, to record an initial high-level model that described some of the modules comprising Excel and the call dependencies between them. Figure 4 shows the high-level model, which consisted of 13 modules and 19 interactions. For proprietary reasons, we have removed the names of all but three modules. Our qualitative examples focus on the interactions among these three, but the statistics we report consider the full set.

Extract source model

Because the Excel high-level model captured calls between modules, the engineer selected calls between functions as the basis for a compatible source model. As Figure 3 shows, he used an internal Microsoft tool to extract the source model. The extracted model consisted of 77,746 calls between the approximately 15,000 functions comprising Excel.

Define map

The engineer defined a map that associated functions in the source model with modules in the high-level model. The map file included entries like

```
[ file=^shtreal\.c mapTo=Sheet ]
[ file=^textfil[ez]\.c$ mapTo=File ]
```

that states that all functions in the `shtreal.c` file are related to the `Sheet` module, and that all functions in the `textfile.c` and `textfilz.h` files are related to the `File` module. Mappings were based on the names of directories, files, and functions.

The engineer defined an initial map of 170 lines after perusing the approximately 400 files comprising the Excel source code. He defined the map in a few hours and did so before installing the reflexion model tools. His inability to compute reflexion models immediately may have affected how he defined the initial map; in other applications of our technique, users have started with maps as small as 10 to 20 lines, which let them begin the iterative investigation and refinement steps earlier.

The engineer spent about a day defining the initial high-level model and map.

Compute reflexion model

Figure 5 shows a small piece of the initial software reflexion model computed for Excel. As expected, the engineer found calls in the source between functions

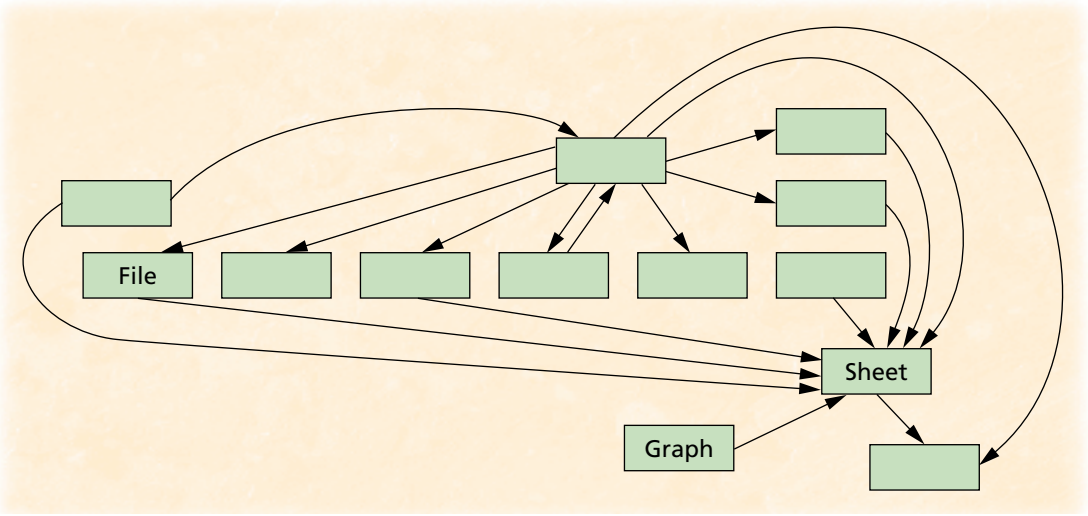


Figure 4. High-level model of Excel. The high-level model provides a hypothesized view of the system structure. The Microsoft engineer wanted to know how Excel's C source code divides into static modules and how those modules interact at execution. For proprietary reasons, the model shows only three named modules.

associated with the `File` module and functions associated with the `Sheet` module (solid arc). The dashed arc from `Sheet` to `File` indicates that functions mapped to the `Sheet` module unexpectedly make calls to functions mapped to the `File` module. Finally, the engineer found no calls between functions mapped to the `Graph` module and those mapped to the `Sheet` module (dotted arc).

The initial Excel reflexion model had 15 convergences, 83 divergences, and four absences. The model summarized about 61 percent of the function calls in the source model. The remaining 39 percent did not match any arc in the map; these unmapped entries are reported in the unmapped arc file, and can be assessed by the user in the context of the task at hand.

The engineer computed the initial reflexion model in 20 minutes using the DOS version of our tools running under Windows NT on a 90-MHz Pentium with 40 Mbytes of memory.

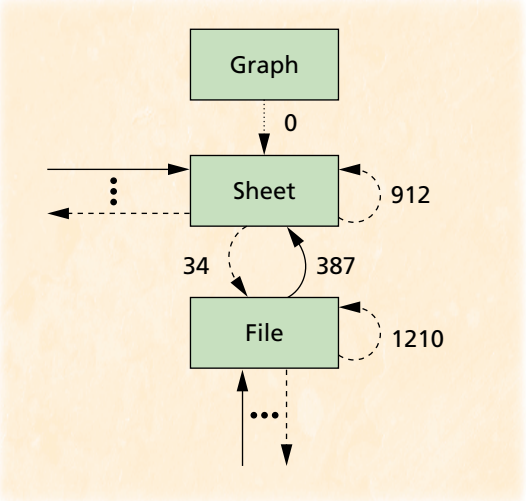


Figure 5. Snippet of the initial reflexion model for Excel. The solid arc from `File` to `Sheet` is a convergence; the dashed arc from `Sheet` to `File` is a divergence; the dotted arc from `Graph` to `Sheet` is an absence.

Investigate and refine reflexion model

In investigating the initial reflexion model, the engineer updated both the high-level and source models and refined the map. He then recomputed the reflexion model and continued the iterative cycles. A cycle typically began with selecting an arc for investigation and ended with a refined reflexion model, although an iteration often skipped steps.

Selecting an arc. The engineer used two methods to select an arc for investigation. Occasionally, mostly in the first week or so, he considered the divergences in the reflexion model. If an interaction was missing from the high-level model, he updated it to reflect the interaction. For example, had the divergence from `Sheet` to `File` represented a reasonable interaction, the engineer might have updated the high-level model to reflect it. This type of investigation and update became less frequent in later refinements because the high-level model stabilized quickly.

More frequently, the engineer selected arcs by sorting through the image arc file (which contains descriptions of convergences and divergences). He selected the arc with the highest number of mapped calls and investigated it using the query script in Figure 3, which he wrote to query the mapped arc file for all calls mapped to the arc. He then qualitatively assessed a subset of these calls by looking at the source code.

Refining the map. The engineer encountered many cases in which functions had been placed in files that no longer represented the module the file was supposed to represent. As a result, he spent significant time refining the map to logically remodularize the Excel system. He wrote a script that inserted an entry in the map that specifically associated a particular function with the appropriate module (Figure 3). Because the entries in the map are ordered, he could insert these logical remodularizations into the top of the map file, leaving the existing map entry for the file unchanged. Then, when one of these functions was seen in the source model, matching stopped at the new map entry.

For example, the engineer found that most of the functions in the `fdefs.c` file should be associated

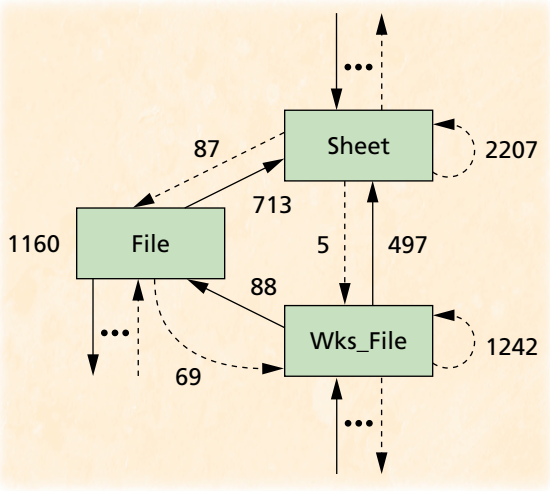


Figure 6. Snippet of a refined reflexion model for Excel. After successive refinements, the reflexion model now includes the `Wks_File` *module. The number of interactions summarized has also increased (larger numbers are associated with the arcs).*

with a user interface module. But, some functions in `fdefs.c`, such as `ExplodeMergeCells`, should be associated with the `Sheet` module. Logically remodularizing this function to the `Sheet` module involved adding the entry

```
[ function=^ExplodeMergeCells$
  mapTo=Sheet ]
```

to the top of the map file.

The engineer refined the Excel map to consist of more than 1,000 entries (from the original 170 entries), refining specific areas of interest and ignoring areas outside the task at hand. Consequently, some parts of the reflexion model represented detailed summaries of the interactions between modules, but other interactions remained fuzzy. In this way, the engineer economically managed the investigation process.

Documenting exceptions. As part of the iterative process, the engineer maintained a file of information that he had investigated and categorized. He called these *exceptions*, using regular expressions to describe particular source model entries mapped to particular arcs in the high-level model. For example, in investigating the divergences from `Sheet` to `File`, the engineer found and categorized calls related to an event-style interaction. He wrote a simple script (Figure 3) to remove the entries in the mapped arc file that correspond to exceptions. Removing these helped him focus on the uninvestigated and unexplained interactions in a reflexion model. We have since added annotations, a generalized version of exceptions, to our tools.[5]

Augmenting the source model. Just over two weeks into the use of the technique, the engineer decided he wanted to consider not only information about the calls between functions in the Excel source, but also references from functions to global variables. He extracted this information using the Microsoft source model extraction tool (Figure 3) and added it into the source model. The resulting source model contained 119,637 entries. Adding the references to global variables to the source model implicitly changed the meaning of the high-level model from a "calls between modules" model to a "communicates with" model. However, even with this semantics change, the engineer had no problem interpreting the later reflexion models.

Results. Figure 6 shows part of the reflexion model after several weeks of refinement. The refined model shows additional source model information, changes to the map, and changes to the high-level model. The new high-level model, for example, includes a `Wks_File` module that comprises some specialized file-handling routines. The interactions summarized between modules have also increased, as indicated by the larger numbers associated with the reflexion model arcs. This increase is the result of augmenting the source model with global data reference information and making the map more specific.

The engineer reported that the technique helped him refine an architectural view of the system and investigate the connection between the architectural view and the source code. He used this view, with the associated map file, as a basis for reasoning about the experimental reengineering activity and for assessing the feasibility of various changes. He also used the map to automate parts of the experimental reengineering activity itself. For example, he used the entries in the map that corresponded to logical remodularizations to place conditional compilation statements into the source code. This aided in isolating potential components for extraction.

The reflexion model also heightened the engineer's understanding of the code base and alleviated the danger of his reasoning in terms of the high-level model alone. For example, he learned that the `Sheet` module requires functionality located in the `File` module in addition to the expected functionality dependence from the `File` to `Sheet` module. He could use this information during planning to upwardly revise any estimates of the complexity involved in separating the `Sheet` component from the code base. This information about the unexpected structural interaction was not available in the high-level model, and it might have been hard to find from the source code alone.

The engineer continued to use the technique even after the original one month. The high-level model grew to 16 entities and 114 interactions; the source model ended up with 131,042 call and data interactions; the map grew to 1,425 entries. The final reflexion model summarized 99.7 percent of the source model entries.

LESSONS LEARNED

The case study gave us several insights into the needs of a user faced with a reengineering activity.

Task-specific views are important

In e-mail communication to us about the technique, the engineer wrote

> Definitely confirmed suspicions about the structure of Excel . . . allowed me to pinpoint deviations . . . very easy to ignore stuff that is not interesting and thereby focus on the part of Excel that I want to know more about.

The engineer found it useful to be able to view the system in terms of a refined reflexion model, yet he also found it valuable to build up an understanding of how the high-level view connected to the source code. Consistent with previous studies on program comprehension, he moved between these levels repeatedly.[8] He also liked having the opportunity to summarize a large database of interaction information from the source code, as long as he was able to refine select parts.

An important side effect of this flexibility is economy. Because the engineer could keep parts of the model fuzzy and refine only select parts, he was able to avoid wasting time collecting information not pertinent to the task at hand.

Graphical and text interfaces are needed

We provided both textual and graphical interfaces to the reflexion model tools. Surprisingly, the engineer drove almost all the investigation of the reflexion model and the source code from textual information. Thus, it might be important to rethink the general belief that graphical interfaces to reverse- and reengineering tools are the best approach.

Explicit, declarative maps help task performance

The engineer used the map to help isolate potential components in the source. We had viewed the map primarily as an input to our technique. Its use to place conditional compilation statements in the source helped the isolation process. This highlights the value of the information that connects the overall view to the source.

Tools and process must adapt to the task

The Microsoft engineer introduced exceptions to better support his exploration of the Excel source code. He also introduced two scripts to support faster recomputation of reflexion models from some of the intermediate files. The architecture of our reflexion model tools as a set of filter programs and the flexibility of our technique made these adaptations possible.

The engineer's use of the reflexion model tools in practice helped focus improvements to both the technique and tools. For instance, we have improved the performance of our tools so that the computation of the reflexion model for Excel now takes fewer than five minutes, compared to the 20 to 40 minutes it first took. This improved speed removes the need for the engineer to define and use specially crafted scripts.

The experimental reengineering case study shows that our technique has practical application. First, the engineer chose to use the technique even when facing extreme pressure. Second, he continued to use the technique beyond the original time (one month) to refine additional parts of the reflexion model for Excel and to compute reflexion models for successive Excel versions. Finally, the engineer stated that the slowdowns he did encounter while performing the experimental reengineering were often due to a lack of up-front understanding. Had the reflexion model technique been used more during planning, he felt that he might have been able to perform the task in less time.

We believe our technique was successful in large part because it uses approximation. This ensures a smooth feedback from the time invested in applying the technique to the results. The more time the Microsoft engineer spent refining the map, the more information he derived. Although this curve is not completely smooth, the engineer was able to gauge the accuracy of the results and use that information to manage the time and effort invested in using the technique.

One question surrounding any case study is whether or not you can generalize its results. We believe our technique can be applied to future efforts similar to the experimental reengineering for several reasons. The source comprising Excel was implemented in C, which is a commonly used language. Like many systems, the Excel source code had been evolving over years and had been implemented by many developers. Finally, the task performed—identifying and extracting components from an existing system—is one that many organizations face. ❖

Acknowledgments

We thank several people at Microsoft for participating in the case study and reviewing drafts of this article: the members of the experimental reengineering team who asked to remain anonymous, and Daniel Weise of Microsoft Research. We also thank Kevin Sullivan for his work on reflexion models, and Alan Borning, William Griswold, Nancy Leveson, Nancy Staudenmayer, and the *Computer* reviewers who commented on earlier case study descriptions and earlier drafts of this article.

This research was funded in part by NSF grants CCR-8858804 and CCR-9506779, in part by a Canadian NSERC postgraduate scholarship, and in part by a University of Washington Department of Computer Science & Engineering Educator's fellowship. Microsoft Corp. also provided equipment.

References

1. G.C. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap between Source and

High-Level Models," *Proc. SIGSOFT Symp. Foundations of Software Eng.*, ACM Press, New York, 1995, pp. 18–28.
2. D.H. Hutchens and V.R. Basili, "System Structure Analysis: Clustering with Data Bindings," *IEEE Trans. Software Eng.*, Aug. 1985, pp. 749–757.
3. D.R. Harris, H.B. Reubenstein, and A.S. Yeh, "Reverse Engineering to the Architectural Level," *Proc. Int'l Conf. Software Eng.*, ACM Press, New York, 1995, pp. 186–195.
4. H.A. Müller and K. Klashinsky, "A System for Programming-in-the-Large," *Proc. Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1988, pp. 80–86.
5. G.C. Murphy, "Lightweight Structural Summarization as an Aid to Software Evolution," PhD dissertation, CS&E Dept., Univ. of Washington, Seattle, 1996.
6. Y-F. Chen et al., "Intertool Connections," in *Practical Reusable Unix Software*, B. Krishnamurthy, ed., John Wiley & Sons, New York, 1995, pp. 299–336.
7. M.A. Cusumano and R.W. Selby, *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*, The Free Press, New York, 1995, p. 109.
8. A. von Mayrhauser and A.M. Vans, "Identification of Dynamic Comprehension Processes During Large Scale Maintenance," *IEEE Trans. Software Eng.*, June 1996, pp. 424-437.

Gail C. Murphy *is an assistant professor of computer science at the University of British Columbia. Her research interests are in software engineering. Her current projects include work on source code analysis, reverse engineering, and software design techniques. She was at the University of Washington while doing the work reported in this article.*

Murphy received an MS and a PhD in computer science and engineering from the University of Washington, and a BSc from the University of Alberta. She is a member of the IEEE Computer Society and ACM.

David Notkin *is a professor of computer science and engineering at the University of Washington. His research interests are in software engineering with a focus on software evolution. His current projects include work on software model checking, software design, and software tools and techniques.*

Notkin received an ScB from Brown University and a PhD from Carnegie Mellon University, both in computer science. He is a member of the IEEE, ACM, and Sigma Xi.

Contact Murphy at murphy@cs.ubc.ca or Notkin at notkin@cs.washington.edu.