Technical Forum

USING LOGICAL DATA MODELS FOR UNDERSTANDING AND TRANSFORMING LEGACY BUSINESS APPLICATIONS

Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowchart; it'll be obvious.

Frederick Brooks, *The Mythical Man-Month*

Modifying a legacy application is typically an expensive and time-consuming process, even when the required modifications are conceptually very simple. We argue that this problem can be ameliorated by adopting an approach in which *logical data models* of a legacy application are used by software developers to understand, maintain, and transform the software. In addition, we outline the goals and status of the Mastery project at IBM Research, which aims to build a suite of tools for automatically extracting logical models from legacy applications, focusing initially on logical data models.

THE PROBLEM

For the past few years, our group at IBM Research has been investigating tools and techniques for analyzing and transforming legacy business applications, focusing on mainframe-based applications written in COBOL.[1] Such applications are often decades old and implement core business functionality. Yet they are difficult to update in a timely manner in response to new business requirements due to a number of factors that include the following:

- Volume of code in a typical application
- Logical structure of code has deteriorated as updates have accumulated over time
- Functional redundancy
- Structure of code reflects the dated technology on which it was built
- Scarce technical skills

Size

Legacy application portfolios, that is, complete collections of programs and related components,

0018-8670/06/$5.00

can be very large. For example, one IBM customer had a portfolio consisting of 700 interdependent applications, 3000 online data sets, 27,000 batch jobs, and 31,000 compilation units. The sheer volume of information contained in an application of this size makes it impossible for an individual to understand the relationships between all parts of the application.

Deterioration

The logical structure of code and data tends to deteriorate over time as a result of a continuous stream of modifications and enhancements. For large legacy applications, persistent data is the principal coupling mechanism between components of an application portfolio. Yet, as an application evolves to meet new business requirements, the structure and coherence of the data models underlying the code decays faster than the structure and coherence of the basic control and process flow through the application. Perhaps this is because it is relatively easy to add new functionality to an existing application by creating modules that manipulate new data items stored separately from the original application data. The alternative of refactoring the basic process flow through the application to accommodate new requirements typically requires much more intrusive changes.

Redundancy

Over time, applications frequently accumulate a great deal of redundant code (multiple code fragments that perform the same logical function) and redundant data (data structures that represent the same information, perhaps with slight differences, and are scattered throughout the code). Reasons for this redundancy include incomplete integration of information systems following business mergers, performance-driven enhancements to the code, and quick "hacks" when adding new functionality under tight schedules.

Technology

The code structure of legacy applications often reflects the limitations of the programming languages used and the middleware on which it was originally designed to run. In many cases, the code structure dictated by the constraints of legacy languages and middleware renders such systems more difficult to understand and evolve than they would be if they had been implemented on modern platforms.

Skills

As new languages and software systems become popular, it becomes more difficult to find people with skills in legacy languages and systems.

Interest in the use of automated and semiautomated tools to analyze and transform legacy code is increasing. Such tools include program-understanding tools, tools for identifying and extracting semantically related code statements (through techniques such as program slicing[2]), tools for migrating from one library or middleware base to another, tools for integrating legacy code with modern middleware, and so on.

In the remainder of the paper, we first explain the value of logical data models and describe a number of applications of logical models to program-understanding and transformation tasks. Then we describe the Mastery project, which is concerned with developing algorithms and tools for extracting and manipulating logical data and the source code from which they are derived. We conclude with a brief review of related work and some final comments.

VALUE OF LOGICAL DATA MODELS

The Mastery project is concerned with extracting *logical models* from legacy applications. These logical models, which are high-level abstractions of business processes and data relationships, are used together with human- and machine-readable links from these logical models back to their physical realizations in code as the foundation for a variety of program-understanding and transformation tasks (we use "physical" to mean "implementation-related"). The initial focus of the Mastery project is on logical *data* models: abstractions encoding essential data relationships. In this paper we focus on applications of data models because we believe that their utility (relative to process- and control-oriented program abstractions) in program understanding and transformation has been underappreciated. Nonetheless, other concepts of logical models not covered in this paper are also valuable; the information they provide can complement logical data models for many of the applications we consider.

Logical data models are critical for understanding and transforming legacy applications. Consider the UML**-style[3] logical data model depicted in *Figure 1* (UML stands for Unified Modeling Lan-

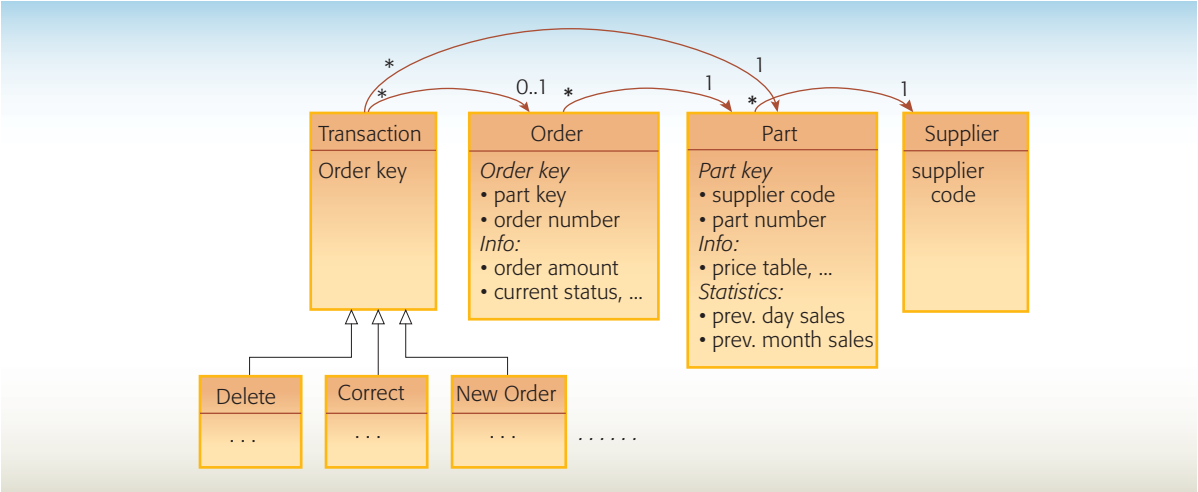

Figure 1
A logical data model for a typical order-processing application

guage**.) This model describes key data structures and their interrelationships for a typical order-processing application. In this case, a batch application processes transaction records pertaining to orders for parts; the processing of a transaction may result in the creation of a new order for a part (New Order), in the correction of an error in an existing unfulfilled order (Correct), or in the cancelation of an unfulfilled order (Delete), and so on.

The application represented by the model in Figure 1 is large (around 60,000 lines of COBOL) and complex. The complexity of the code obscures its essential functionality, which is to process different kinds of transactions pertaining to orders for parts. This functionality is expressed succinctly and at a high level of abstraction by the data model. In other words, the "business logic" of the application is concerned primarily with maintaining and updating certain relationships among persistent and transient data items; therefore, the data model embodies much of the interesting functionality of the application, even though the model contains no representation of code.

It is notable that the logical data model shown in Figure 1 differs greatly from the data declarations in the source code of the application. ***Figure 2*** shows an outline of these data declarations, with the data items linked (links shown using dashed arrows) to the corresponding logical-data-model entities (this figure contains a relevant subset of the logical model in Figure 1). The data declarations are spread over several source files; furthermore, they reveal little about the structure of and relationships between the logical entities manipulated by the application, which is obtainable only by an analysis of the *code* that uses the data. As illustrated in Figure 2, the logical data model adds value by making information that is hidden in the code explicit, such as the following:

- *Logical entities*—The logical entities manipulated by the program include Transactions (i.e., requests to the system of various types) Orders, Parts, and so forth. Physical data items (variables) correspond to these entities; such as `ORDER-BUF` and `ORDER-REC` store Orders (as indicated by the links).
- *Logical subtypes*—Transactions are of several kinds (have several *subtypes*), such as Delete, Correct, and New Order.
- *Associations*—Entities are associated with (or pertain to) other entities, as indicated by the red arrows in Figure 1. Associations have *multiplicities*; for example, the labels on the association from Transaction to Order indicate that each Transaction pertains to zero or one (existing) Orders and that each Order has zero or more Transactions pertaining to it on any given day.
- *Aggregation*—The information corresponding to a single part is stored in two physical records, `PR1-PART-REC` and `PR2-PART-REC`, which are tied together by their `PART-KEY` attribute. This

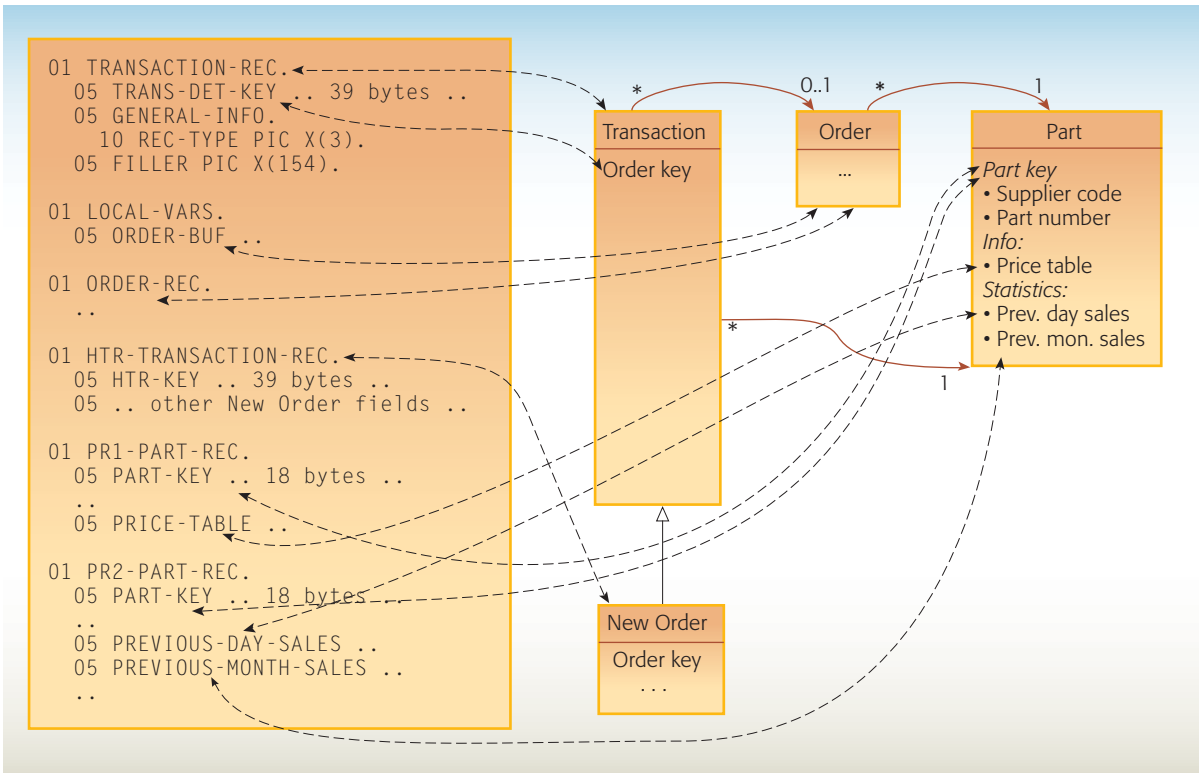

Figure 2
Links mapping entities in a logical data model to declarations in the source code

(perhaps historical) artifact is elided in the logical model, and both records are linked to a single "Part" entity.

- *Integrity constraints*—Although our example does not illustrate it, a logical data model can also include semantic integrity constraints and data invariants (beyond those implied by multiplicities on associations), such as the constraint that the `Order Amount` must be positive.

APPLICATIONS OF LOGICAL DATA MODELS

A logical data model linked back to the physical constructs that realize it can serve as the foundation for a variety of program transformations, either implemented by hand or automated by tools. Examples of such transformations are described in the following subsections.

Data representation changes

In business applications, it is often necessary to update the physical representations of a single logical entity, say, to accommodate an expanded range of values. For example, a 2-digit date field might be expanded to 4 digits to allow for dates after the year 2000, or a serial-number product field, represented using numeric values, might be updated to an alphanumeric type to accommodate additional products. A logical data model and an accompanying logical-to-physical mapping can be used to distinguish unrelated instances of the same physical type (e.g., one 10-digit variable in a program may represent a part number, and another variable of exactly the same physical type might represent a customer number), or logically related instances of *distinct* physical types (e.g., character string and integer representations of a value representing an invoice number). The logical-to-physical mapping can allow for automatic or semiautomatic type transformations, which would otherwise require tedious and error-prone manual code searches.

Addition of attributes to existing logical entities

In this scenario, the physical representations of a logical entity must be updated to accommodate a new logical attribute (e.g., adding an `email_addr` attribute to an existing `customer` entity). This is

usually straightforward in the presence of a logical-to-physical mapping that identifies all physical realizations of the entity in question. The details of the code modification required to accommodate the new attribute may depend on the nature of the physical representation of the type. For example, if the logical entity is manifest as a pair of non-contiguous physical fields, either one of the physical fields might be updated to hold the new attribute. A tool that uses logical-to-physical model mappings to implement attribute update transformations could examine the contexts in which each physical manifestation of the same type is used to suggest an update transformation with the least impact on other parts of the code.

Application integration

When businesses merge, there is usually a need to loosely integrate application software from the premerger businesses, because rewriting applications from scratch to accommodate the joint needs of the merged organization is usually impractical (at least in the short run). In such situations, it is necessary to identify logically related data entities that need to flow between premerger applications (e.g., `customer`, `part`). Clearly, it is easier to determine which entities serve similar roles at the logical level than at the physical level. Consider, for example, a procedure parameter that is declared to be a character string of length one. A logical data model for the parameter that constrains the value of the string to be either `'Y'` or `'N'` provides more useful information than the type of the parameter alone. Similarly, a logical data model that breaks the contents of a 100-character buffer into component logical types, or which distinguishes logical output ("write only") parameters from logical input ("read only") parameters, provides vital information for integrating the two applications. By annotating logical models with ontological information that characterizes the canonical "business semantics" of basic logical entities (as a trivial example, the character string defined above might be annotated with the canonical type `boolean`), automatic or semiautomatic tools can be used to define mappings between related logical models in premerger applications. Annotated logical types can then be used to define runtime "adapter" code that can manage the flow of data between applications. For example, such an adapter might transform the string `'Y'` into the boolean value `true` to allow two premerger applications to be loosely integrated.

Database migration

Many legacy applications store persistent data in data stores such as flat files or nonrelational databases rather than more flexible relational databases. Businesses are often motivated to migrate data from older data stores to relational databases in order to consolidate data from multiple applications more tightly or because they wish to move from the nightly batch processing of transactions to continuous "straight through" transaction processing. In such a scenario, a semantically accurate logical model of persistent data can assist in defining not only a data schema for the new data store but also the code transformation required to access the new data store from existing applications.

Migration to service-oriented architectures

The concept of service-oriented architecture (SOA[4]) entails decomposing complex enterprise-scale software into collections of loosely coupled distributed components. In order to achieve the flexibility promised by the SOA approach, it is important to define natural component interfaces for legacy applications. In transforming a legacy application to implement SOA interfaces, it is typically necessary to address most of the issues that arise in integration, adaptation, and migration scenarios.

Facilitation of program understanding, documentation, and planning

Many large-scale application integration, migration, and transformation projects go awry due to lack of adequate information about the behavior of existing applications and the impact of proposed code changes. Accurate logical models, linked to their physical counterparts, can serve as semantically well-founded program documentation to allow better planning for proposed changes. Like textual documentation, good data (and process models) can help application architects and programmers understand the behavior of an application. Unlike text, however, models can be queried by software to help understand the impact of proposed changes. A simple example might be determining how many distinct applications access data from a particular logical entity for which a change is being considered. A more detailed query might determine the number of different physical types used to implement the logical type as a measure of the difficulty of carrying out a transformation. Yet more sophisticated tools might compute estimated transformation costs or define a candidate transformation plan by

mapping a proposed transformation at the logical model level back to code.

For all the reasons enumerated above, we believe that logical models in general, and logical data models in particular, should serve as the foundation for legacy analysis, understanding, and transformation tools.

THE MASTERY PROJECT

In this section, we give a brief overview of the Mastery project at IBM Research, whose aim is to create data-centered modeling tools and to develop the algorithmic foundations for model extraction. Our long-term goal is also to address process-model and business-rule extraction, in addition to data-model extraction.

Mastery currently consists of two distinct threads: first, research on novel algorithms for extracting logical data models, and second, development of an interactive tool, the Mastery Modeling Tool (MMT), for extracting logical models from COBOL applications and for querying and manipulating these models.

Research on algorithms for extracting logical data models

The goal of the Mastery model extraction work is to algorithmically recover data abstractions from legacy applications that accurately reflect the "natural" business semantics of the application. The level of abstraction that we are aiming for is roughly similar to that found in UML class diagrams, OCL (Object Constraint Language)[5] or Alloy.[6]

In general, it can be quite challenging to construct semantically well-founded logical data models from legacy applications. For example, COBOL programmers frequently overlay differing data declarations on the same storage (using the `REDEFINE` construct). Sometimes overlays represent disjoint unions, that is, situations in which the same storage is used for distinct unrelated types, typically depending on some "tag" variable. However, overlays are also used to reinterpret the same underlying type in different ways; for example, to split an account number into distinct subfields representing various attributes of the account. The disjoint and non-disjoint (reinterpreting) use of overlays can be distinguished in general only by examining the way the data are used in code, and not just by examining the data declarations.

Examples of other challenging problems related to extracting logical data models from applications include:

- Identifying logical entities and the correspondences between program variables and logical entities.
- Identifying distinct variables that aggregate to a single logical entity because they are always used together.
- Identifying associations (with multiplicities) between entities based on implicit and explicit primary/foreign key relationships. For example, a `PartNumber` field in an `Order` record may contain a value that implicitly refers to a corresponding `Part` record containing the same value in its `Number` field. `PartNumber` may be considered a foreign key and `Number` a primary key, which together define an association between `Order`s and `Part`s.
- Identifying integrity (i.e., consistency) constraints on logical entities and their attribute values (e.g., a particular attribute of a particular entity might be an enumeration type and allowed to take on a value only from a given set of values).
- Identifying uses of generic polymorphism, that is, data types designed to be parameterized by other data types. An example would be a `Log` file type that is parameterized by the class of the transaction being logged.

Our basic algorithmic idea is to use data-flow information to infer information about logical types; for example, given a statement assigning A to B, we can infer that at that point in the code, A and B have the same type (i.e., correspond to the same logical entity). Further, if we know that K is a variable containing a key value for a logical type T, and K is assigned to a field F of record R, then we can infer that there is a logical association between the type of R and T. We regard reads from external or persistent data sources as defining "basic entities" from which more elaborate relations can be inferred. The simple ideas outlined above are greatly complicated by various COBOL language features and programming idioms, particularly relating to the use of overlays. These aspects lead to the same memory locations (variables) being used to store values of different logical types (different logical entities) under differ-

ent conditions. We have developed an efficient, flow-insensitive, heuristics-based algorithm for inferring subtype relationships to model such situations. Details of this algorithm are beyond the scope of this paper. We have also developed path-sensitive type inference algorithms, described in References 7 and 8, which deal with these language features and programming idioms precisely and in a semantically well-founded way. We hope to implement efficient versions of these path-sensitive type inference algorithms in the future.

Mastery Modeling Tool

MMT is built on the Eclipse**[9] open-source framework. The tool contains several components. An importer processes COBOL source files and COBOL copybooks to build a physical model that can be browsed inside the tool. A model extractor builds a logical model at the level of a UML class diagram (as described earlier), and links that model to the physical data model. Currently, the model extractor identifies basic logical types, subtypes, and association relations. Extensive model view facilities allow logical models and their links to physical data items to be browsed and filtered from many perspectives. A query facility allows expressive queries on the properties of models and links. Such queries can be used, for example, to understand the impact of various proposed updates to the application from a data perspective. The tool also allows users to edit the logical model, permitting manual refinement of automatically inferred models. A report generation capability generates HTML-based reports depicting the logical and physical models for easy browsing outside the MMT tool.

Because the UML 2.0 specification was still evolving when we began work on Mastery, we opted to use a simplified class model as the metamodel for the Mastery logical model. With the availability of tools such as Rational* Software Architect,[10] based on the UML 2.0 specification, we have supported integration by converting the Mastery logical model to UML 2.0 through a simple converter, using the Eclipse UML 2.0 implementation.[11]

RELATED WORK

There are commercially available tools that assist with data modeling, such as AllFusion** ERWin** Data Modeler[12] from Computer Associates International, Inc., Rational Rose*[13] from IBM, Modernization Workbench**[14] from Relativity Technologies, Inc., and Enterprise Repository[15] from SEEC, Inc. In general, these tools may support forward engineering of applications from data models (e.g., ERWin and Rose), or infer physical data models from data declarations alone (e.g., Modernization Workbench, Enterprise Repository, and Rose). We believe Mastery is unique in its ability to infer a logical data model for an existing application by analyzing how data is used in the code.

The Object Management Group** (OMG**) has an Architecture-Driven Modernization (ADM) Task Force,[16] which is creating standards to support the analysis and transformation of existing applications. The ADM Task Force has defined a road map containing seven standards. The work on Mastery corresponds to the first three standards in that road map as follows:

1. Mastery's physical model of COBOL programs corresponds to the information covered by the Abstract Syntax Tree Metamodel (ASTM)
2. Mastery's model of the batch jobs and transactions that invoke COBOL programs corresponds to the information covered by the Knowledge Discovery Metamodel (KDM)
3. The linkage between Mastery's physical model of COBOL programs and its inferred logical model corresponds to the Analysis Package (AP).

Previously reported academic work in using type inference for program understanding includes References 17–22. Of these, the approaches in References 19–22 are close in spirit to the type inference component of our model extractor. However, our approach has certain distinctions, such as the use of a heuristic to distinguish disjoint uses of redefined variables from non-disjoint (reinterpreting) uses; this distinction is important because the program transformations required to implement a functional change related to a redefined variable are very different in the two cases. Van Deursen et al.[21] describe a system for exploring information produced by type inference that is similar to MMT.

Approaches that do not use type information for inferring logical data models (or certain aspects of logical data models) include References 23–26. Examples of approaches for inferring business rules (as opposed to data models) are found in References 27 and 28.

CONCLUSIONS

Many legacy applications perform essential business functions; yet, due to a number of factors, modifying such applications in order to accommodate new business requirements can be troublesome. Such factors include: the volume of code in a typical application, logical code structure that has deteriorated as updates have accumulated over time, functional redundancy, code structure that reflects the dated technology on which it was built, and scarce technical skills. We have argued that the consequent difficulty of understanding and modifying legacy code can be ameliorated through the use of logical data models. In the Mastery project, we are developing both algorithms for extracting logical data models from legacy COBOL applications and software tools that use the generated models to query and transform the code from which the models are derived.

ACKNOWLEDGMENTS

We thank an anonymous referee for bringing some related work, as well as Fred Brooks' quotation at the beginning of the article, to our attention.

CITED REFERENCES

1. Mastery, IBM Research Division, IBM Corporation, http://domino.research.ibm.com/comm/research_projects.nsf/pages/mastery.index.html.
2. F. Tip, A Survey of Program Slicing Techniques, *Journal of Programming Languages* **3**, 121–189 (1995).
3. M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd Edition*, Addison-Wesley, Reading, MA, (2003).
4. Service-Oriented Architecture, *IBM Systems Journal* **44**, No. 4 (2005).
5. *UML 2.0 OCL Specification*, Object Management Group, Inc. (October 2003), http://www.omg.org/docs/ptc/03-10-14.pdf.
6. D. Jackson, "Alloy: A Lightweight Object Modeling Notation," *ACM Transactions on Software Engineering and Methodology* **11**, No. 2, 256–290 (2002).
7. R. Komondoor, G. Ramalingam, S. Chandra, and J. Field, "Dependent Types for Program Understanding," *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Edinburgh, 2005, in *Lecture Notes in Computer Science* **3440**, Springer, Berlin (2005), pp. 157–173.
8. G. Ramalingam, R. Komondoor, J. Field, and S. Sinha, "Semantics-Based Reverse Engineering of Logical Data Models," To appear in the *Proceedings of the International Conference on Software Engineering (ICSE'06)*, May 2006, ACM Press (2006).
9. Eclipse, Eclipse Foundation, http://www.eclipse.org.
10. Rational Software Architect, IBM Corporation, http://www.ibm.com/software/awdtools/architect/swarchitect/index.html.
11. The Eclipse UML2 Project, Eclipse Foundation, http://www.eclipse.org/uml2/.
12. AllFusion ERWin Data Modeler, Computer Associates International, http://www3.ca.com/solutions/Product.aspx?ID=260.
13. Rational Rose XDE Modeler, IBM Corporation, http://www.ibm.com/software/awdtools/developer/modeler/.
14. Modernization Workbench, Relativity Technologies, http://relativity.com/pages/modernizationworkbench.asp.
15. Enterprise Repository, SEEC, Inc., http://www.seec.com/pdf/SEECEnterprise%20Repository.pdf.
16. Architecture-Driven Modernization, Object Management Group, Inc., http://www.omg.org/adm.
17. R. O'Callahan and D. Jackson, "Lackwit: a Program Understanding Tool Based on Type Inference," *Proceedings of the International Conference on Software Engineering (ICSE 1997)*, ACM Press (1997), pp. 338–348.
18. P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sorensen, and M. Tofte, "Annodomini: From Type Theory to Year 2000 Conversion Tool," *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press (1999), pp. 1–14.
19. G. Ramalingam, J. Field, and F. Tip, "Aggregate Structure Identification and Its Application to Program Analysis," *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press (1999), pp. 119–132.
20. A. van Deursen and L. Moonen, "Understanding COBOL Systems Using Inferred Types," *Proceedings of the International Workshop on Program Comprehension (IWPC '99)* May 5–7, 1999, Pittsburgh, PA, IEEE Computer Society (1999), p. 74.
21. A. van Deursen and L. Moonen, "Exploring Legacy Systems Using Types," *Proceedings of the Working Conference on Reverse Engineering (WCRE'00)* November 2000, Brisbane, Australia, IEEE Computer Society (2000), pp. 32–41.
22. A. van Deursen and L. Moonen, "Documenting Software Systems Using Types," *Science of Computer Programming* **60**, No. 2, 205–220 (2006).
23. G. Canfora, A. Cimitile, and G. A. D. Lucca, "Recovering a Conceptual Data Model from COBOL Code, *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE '96)*, June 10–12, 1996, Lake Tahoe, Nevada, Knowledge Systems Institute, (1996), 277–284.
24. B. Demsky and M. Rinard, "Role-Based Exploration of Object-Oriented Programs," *Proceedings of the International Conference on Software Engineering (ICSE 2002)*, May 2002, Orlando, Florida, IEEE Computer Society (2002), pp. 313–324.
25. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *Proceedings of the Interna-*

tional Conference on Software Engineering (ICSE '99), May 16–22, 1999, Los Angeles, CA, IEEE Computer Society Press, (1999), pp. 213–224.

26. A. van Deursen and T. Kuipers, "Identifying Objects Using Cluster and Concept Analysis," *Proceedings of the International Conference on Software Engineering (ICSE'99)*, May 16–22, 1999, Los Angeles, CA, IEEE Computer Society (1999), pp. 246–255.
27. S. Blazy and P. Facon, "Partial Evaluation for the Understanding of Fortran Programs," *International Journal of Software Engineering and Knowledge Engineering* **4**, No. 4, 535–599 (1994).
28. F. Lanubile and G. Visaggio, "Function Recovery Based on Program Slicing," *Proceedings of the Conference on Software Maintenance (ICSM 1993)*, September 1993, Montréal, Quebec, Canada, IEEE Computer Society (1993), pp. 396–404.

Accepted for publication March 23, 2006.
Published online July 11, 2006.

Satish Chandra
IBM Research Division
Hawthorne, New York

Jackie de Vries
IBM Research Division
Hawthorne, New York

John Field
IBM Research Division
Hawthorne, New York

Howard Hess
IBM Research Division
Hawthorne, New York

Manivannan Kalidasan
IBM India Software Laboratory
Bangalore, India

Komondoor V. Raghavan
IBM Research Division
New Delhi, India

Frans Nieuwerth
IBM Sales and Distribution
Amsterdam, Holland

Ganesan Ramalingam
IBM Research Division
Bangalore, India

Justin Xue
Independent consultant
Edison, New Jersey