Software Reflexion Models: Bridging the Gap between Design and Implementation

Gail C. Murphy, *Member*, *IEEE Computer Society*, David Notkin, *Senior Member*, *IEEE*, and Kevin J. Sullivan, *Senior Member*, *IEEE*

Abstract—The artifacts constituting a software system often drift apart over time. We have developed the software reflexion model technique to help engineers perform various software engineering tasks by exploiting—rather than removing—the drift between design and implementation. More specifically, the technique helps an engineer compare artifacts by summarizing where one artifact (such as a design) is consistent with and inconsistent with another artifact (such as source). The technique can be applied to help a software engineer evolve a structural mental model of a system to the point that it is "good-enough" to be used for reasoning about a task at hand. The software reflexion model technique has been applied to support a variety of tasks, including design conformance, change assessment, and an experimental reengineering of the million-lines-of-code Microsoft Excel product. In this paper, we provide a formal characterization of the reflexion model technique, discuss practical aspects of the approach, relate experiences of applying the approach and tools, and place the technique into the context of related work.

Index Terms—Reverse engineering, program understanding, software structure, program representation, model differencing.

1 Introduction

Complex software systems consist of collections of artifacts. Some of these artifacts are automatically derived from other artifacts; object code compiled from source is a classic example. Most artifacts, however, are manipulated independently of one another, even if they are logically related; design documents and source code are an example. When artifacts are independently manipulated, the artifacts tend to "drift" apart over time. This drift happens neither because software engineers have poor intentions nor because they are lazy, but rather because it is time-consuming, difficult, and rarely the highest priority to maintain logical but implicit relationships among documents.

When a design document drifts, a software engineer attempting to perform a task, such as adding a new feature to the system, has several choices. One choice is to ignore the out-of-date design information and to proceed with the task based solely on information from the source. Using this strategy, the engineer may rely on perusing presumed relevant bits of the code, or may rely on generating a view of the system such as a call graph. Although either of these strategies may work with small systems, on large systems being developed by a team, the engineer may tend to either miss relevant pieces of code or be too overwhelmed by the scale of the extracted view, leading to inappropriate choices and delays during development.

Another approach taken by engineers is to use informal high-level models to reason about the system and the task. Box and arrow sketches of a system, for instance, are often found on engineers' whiteboards. Although these informal models are commonly used, reasoning about the system in terms of such models can be dangerous because the models are almost always inaccurate with respect to the system's source.

A final choice the engineer might make is to use a reverse engineering system to derive a high-level model from the source code. These derived models are useful because they are, by their very nature, accurate representations of the source. Although accurate, the models created by these reverse engineering systems may differ from the models sketched by engineers: An example of this is reported by Wong et al. [37]. The reverse engineered models may not be suitable for reasoning about the task at hand.

We have developed the software reflexion model technique to help engineers perform various software engineering tasks by exploiting—rather than removing—the drift between design and implementation [22].[1] The goal of our iterative approach is to enable a software engineer to produce, within the time constraints of the task being performed, a high-level structural model that is "good-enough" to use for reasoning about the task at hand. The engineer defines a high-level model of interest, extracts (using a third-party tool) a source model (such as a call graph or event interactions) from the source code, and defines a mapping between the two models. A *software reflexion model* is then computed to determine where the engineer's high-level model does and does not agree with

- *G.C. Murphy is with the Department of Computer Science, University of British Columbia, 201-2366 Main Mall, Vancouver BC V6T 1Z4. E-mail: murphy@cs.ubc.ca.*
- *D. Notkin is with the Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98105-2350. E-mail: notkin@cs.washington.edu.*
- *K.J. Sullivan is with the Department of Computer Science, University of Virginia, Charlottesville, VA 22903. E-mail: sullivan@cs.virginia.edu.*

Manuscript received 4 May 1998; revised 18 Dec. 1998; accepted 20 Sept. 1999.
Recommended for acceptance by C. Ghezzi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 106807.

1. The old English spelling differentiates our use of "reflexion" from the field of reflective computing [31].

the source model. An engineer interprets the reflexion model and, as necessary, modifies the high-level model, source model, or mapping to iteratively compute additional reflexion models.

In essence, a reflexion model *summarizes* a source model of a software system from the viewpoint of a particular high-level model. The engineer chooses the high-level view as a suitable view to use to reason about the task at hand. This form of summarization is useful to engineers performing a variety of software engineering tasks. The technique has been applied to support design conformance tasks, to assess software structure prior to an implementation change [22], and to help an engineer at Microsoft Corporation perform an experimental reengineering of the million line-of-code Excel code base [20].

Three key characteristics of the technique are that it is *lightweight*, *approximate*, and *scalable*. By lightweight, we mean that the effort required on the part of the engineer to apply the technique as part of a task is low, typically requiring on the order of hours rather than days. By approximate, we mean that the engineer can use many different kinds of source models and can begin with a coarse mapping between the two models that can then be progressively refined. By scalable, we mean that the technique works well on multilingual systems ranging from a few thousand to over a million lines-of-code. Within the context of an iterative approach, the combination of these features allows an engineer to balance the cost of applying the technique to large systems with the information needed to complete the task at hand.

In this paper, we describe and analyze the technique. To introduce the technique, we begin in Section 2 with an example of its use. Section 3 clarifies the meaning and computation of reflexion models by presenting a formal characterization of a reflexion model system and discussing aspects of its implementation. Section 4 considers the theoretical and practical performance aspects of the approach and our tools. The flexibility of our approach is demonstrated in Section 5 through descriptions of the use of reflexion models in a variety of settings. Section 6 discusses key aspects of the approach. Section 7 considers related work.

2 An Example

To convey our basic approach, we describe how a developer with expertise in Unix virtual memory (VM) systems used the software reflexion model technique to familiarize himself with an unfamiliar implementation, NetBSD. The developer was asked to perform this familiarization step in anticipation of changing the NetBSD's pager implementation to page over the network instead of paging to the filesystem.

The NetBSD system comprises about 250,000 lines of C [15] source code spread over approximately 1,900 source files. The source is thus too large to peruse directly when trying to assess the difficulty of an anticipated change. As is the case with many systems, appropriate documentation and existing expertise with the system are also not readily available.

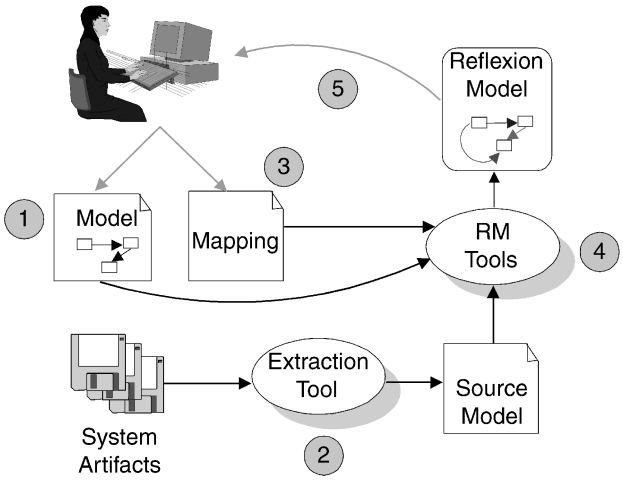


Fig. 1. The reflexion model approach.

To apply the software reflexion model technique to investigate the design of the source, the developer iteratively performed five basic steps (Fig. 1).

First, the developer specified a model he believed, based on his experience, to characterize Unix virtual memory systems. This model consists of a modularization of a virtual memory implementation and the calls between those modules (Fig. 2a).

Next, the developer extracted structural information (called a source model) from the artifacts of the system. Source models are produced either by statically analyzing the system's source or by collecting information during the system's execution. In this case, the developer chose to use the xrefdb cross-reference database tool from the Field [29] environment to extract a calls relation between NetBSD functions. A small awk script [1] was written to translate the output of xrefdb into the input format expected by the software reflexion model tools. The extracted calls relation comprised over 15,000 entries describing calls between over 3,000 source entities (in this case, functions).[2] In the third step, the developer described a mapping between the extracted source model and the stated high-level structural model. The full map is shown below.

```
[ file=.*pager.*      mapTo=Pager ]
[ file=vm_map.*       mapTo=VirtAddressMaint ]
[ file=vm_fault\.c    mapTo=KernelFaultHdler ]
[ dir=[un]fs          mapTo=FileSystem ]
[ dir=sparc/mem.*     mapTo=Memory ]
[ file=pmap.*         mapTo=HardwareTrans ]
[ file=vm_pageout\.c mapTo=VMPolicy ]
```

Each line in this declarative map associates entities in the source model (on the left) with entities in the high-level model (on the right). The seeming difficulty of defining a mapping given the thousands of entities in the NetBSD source model is mitigated in three ways. First, the engineer need only name entities in subsystems of interest. For example, the mapping above does not consider entities from the I/O subsystem. Second, the physical (e.g.,

2. Each entry contained the name of the calling function, the name of the called function, and the file and directory information for both functions.

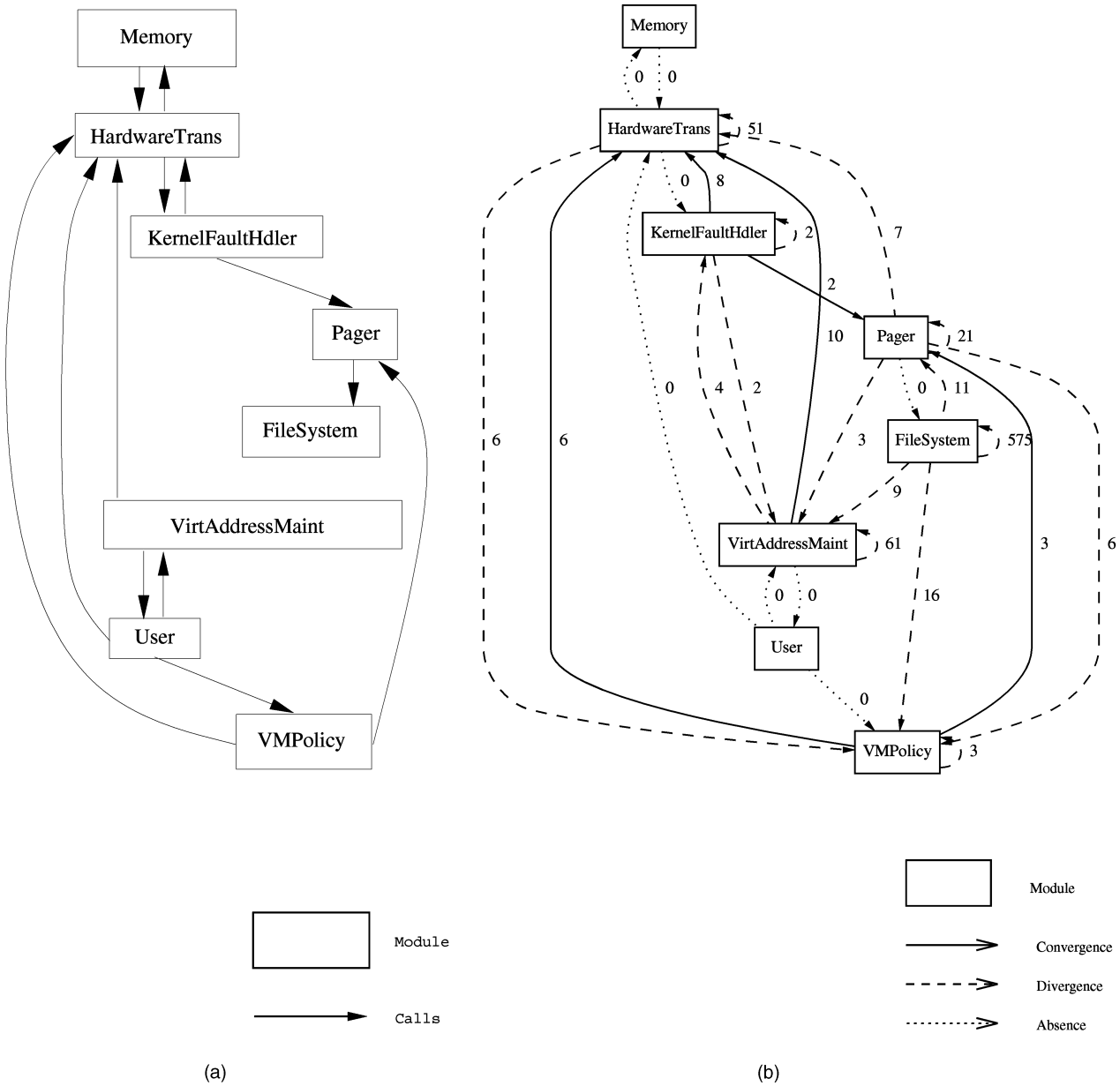


Fig. 2. (a) High-level and (b) reflexion models for the NetBSD virtual memory subsystem.

directory and file) and logical (e.g., functions and classes) structure of the source can be used to name many source model entities in a single line of the mapping. In the mapping above, the developer is using only physical structure, such as file and directory names. Finally, regular expressions can be used to obviate the need to enumerate a large set of structures. For instance, the first line of the mapping states that all functions found within files whose name includes the string `pager` should be associated with the high-level model entity `Pager`.

Given these three inputs, the developer, in the fourth step, uses a tool to compute a software reflexion model that provides a comparison between the source model and the posited high-level structural representation. The computation consists of pushing each interaction described in the source model through the map to form induced arcs between high-level model entities. The computation then compares the induced arcs with the interactions stated in the high-level model to produce a reflexion model.

Fig. 2b shows the reflexion model computed for NetBSD. The solid lines in the reflexion model, called *convergences*, indicate source interactions that were expected by the developer. For instance, the developer correctly predicted interactions between functions in modules implementing `VMPolicy` and functions in modules implementing `Pager`. Dashed lines, called *divergences*, indicate source interactions that were not expected by the developer (e.g., `FileSystem` to `Pager`). The dotted lines, *absences*, indicate interactions that were expected but not found. For instance, no calls were found between modules mapped to `Pager` and modules mapped to `FileSystem`. The number attached to each arc indicates the number of source model interactions mapped to the convergence or divergence; absence arcs are annotated with the value zero. Our tools compute

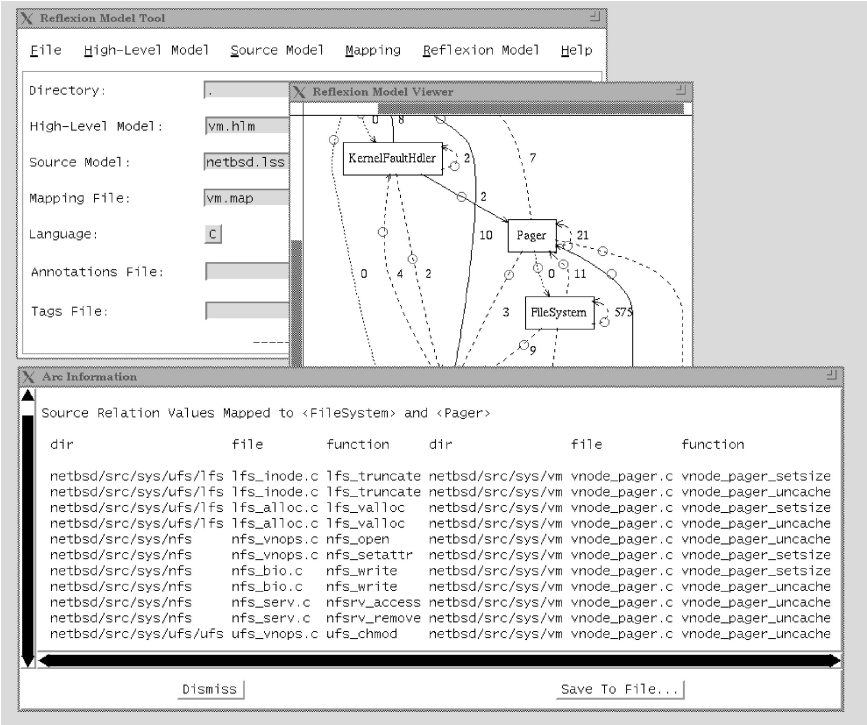


Fig. 3. The reflexion model tool user interface.

and display the reflexion model shown in Fig. 2 from the three inputs in about ten seconds on a DEC Alpha/255.

In the fifth step, the engineer interprets and investigates the reflexion model to derive information that helps the engineer reason about the software engineering task. For example, the screen snapshot in Fig. 3 includes a window "Arc Information" that shows the result of the developer selecting the divergence between `FileSystem` and `Pager`. Values in this window show the calling and called functions, including their directory and file information. Based on an investigation of the software reflexion model, the engineer may decide to refine one or more of the inputs to the computation, and recompute a reflexion model.

In a one hour session, the VM developer specified, computed, and interpreted several reflexion models.[3] Informally, the developer found the representation of the source code as a reflexion model useful in providing a global overview of the structure of the NetBSD implementation. For example, from studying specific divergences, the developer concluded that the implementation of `FileSystem` included optimizations that rely on information from `Pager`. This information is useful for planning modifications to either module. During the one hour session, the developer also indicated a desire for more information about the use of global variables. We later extracted this information and used it to augment the source model. Reflexion models computed on this augmented source model further clarified interactions between the modules of the virtual memory system.

3. In this case, the source model was extracted beforehand, and it was not changed during this session.

3 Formal Characterization and Tool Support

To make precise the meaning and computation of a reflexion model, we present a formal specification of a reflexion model system. A formal characterization of software reflexion models is useful for two reasons. First, it unambiguously and precisely defines the meaning and computation of reflexion models. This allows other researchers or practitioners to both understand the approach and to reproduce the tools if desired. Second, the formalization distinguishes points in the reflexion model computation for which multiple design choices are possible, in essence, identifying a family of systems and providing a base from which different members of the family may be constructed and reasoned about systematically.

In particular, we use the Z specification language [32] to cover three aspects of the formalization: 1) the basic description of reflexion models, 2) the precise definition of the computation that determines convergences, divergences, and absences, and 3) a description of how to compute information needed for displaying reflexion models. A Z specification models the data of a system using mathematical data types and describes the effects of system operations using predicate logic. The specification is decomposed into several schemas linked by a natural language commentary.

In addition, we discuss the particular family members we realized in the tools we have built. The tools consists of several small C++ [34] programs, a user interface implemented in TCL/TK [25], and links to AT&T's graphviz package.

$$[HLMENTITY, SMENTITY]$$

$$HLMRelation == HLMENTITY \leftrightarrow HLMENTITY$$
$$SMRelation == SMENTITY \leftrightarrow SMENTITY$$
$$HLMEntry == HLMENTITY \times HLMENTITY$$
$$SMEntry == SMENTITY \times SMENTITY$$

ReflexionModel

$$rmEntities : \mathbb{P}\, HLMENTITY$$
$$convergences : HLMRelation$$
$$divergences : HLMRelation$$
$$absences : HLMRelation$$
$$mappedSourceModel : HLMEntry \leftrightarrow SMEntry$$
$$smBag : \text{bag}\, SMEntry$$

$$convergences \cap divergences = \varnothing \quad (1)$$
$$divergences \cap absences = \varnothing \quad (2)$$
$$convergences \cap absences = \varnothing \quad (3)$$
$$\text{dom}\, mappedSourceModel = convergences \cup divergences \quad (4)$$
$$\text{ran}\, mappedSourceModel \subseteq \text{dom}\, smBag \quad (5)$$
$$\text{dom}\, convergences \cup \text{ran}\, convergences \cup \text{dom}\, divergences \cup \text{ran}\, divergences \cup \text{dom}\, absences \cup \text{ran}\, absences \subseteq rmEntities \quad (6)$$

Fig. 4. A Z schema describing a reflexion model.

3.1 Reflexion Model

A static schema in Z describes the state space of the system as well as invariants that must be maintained across state transitions. Any system implementing the reflexion model approach must maintain the state components and invariants described in the $ReflexionModel$ schema shown in Fig. 4.

The $ReflexionModel$ schema uses two basic types: $HLMENTITY$, which represents the type of a high-level model entity and $SMENTITY$, which represents the type of a source model entity. Four type synonyms are also defined: $HLMRelation$ and $SMRelation$, which define relations over high-level model entities and source model entities, respectively, and $HLMEntry$ and $SMEntry$, which define the types of entries in the $HLMRelation$ and $SMRelation$. (see Fig. 4)

The variables (above the dividing line) in the schema represent observations that can be made about the state of a reflexion model system. The $rmEntities$ variable describes the set of nodes in a reflexion model. The relation described by the $convergences$ variable defines the set of interactions where the high-level model agrees with the mapped interactions from the source model. The $divergences$ variable describes where the mapped set of interactions from the source model differs from the high-level model, and the relation described by the $absences$ variable defines a set of interactions where the high-level model differs from the mapped source model interactions. The fourth variable, $mappedSourceModel$, is a relation that describes which source model values contribute to a convergent or divergent arc in the reflexion model. The information in $mappedSourceModel$ supports operations that aid an engineer in interpreting a reflexion model. For example, the information in $mappedSourceModel$ is necessary to support a query of the form shown in Fig. 3. The final variable, $smBag$, is a bag that includes all source model values and describes the number of occurrences of each value in the source model.[4]

In addition to declaring the state components of a reflexion model, the static schema includes the definition of invariants (below the dividing line) that must be satisfied by a reflexion model system.[5] The first three invariants state that the values of the $convergences$, $divergences$, and $absences$ relations are disjoint. The fourth invariant states that the investigation of the source model values contributing to a reflexion model arc is supported only for values in the $convergences$ and $divergences$ relations; $absences$ have no contributing source model values. The fifth invariant states that the source model values contributing to reflexion model arcs must either be a subset of, or the same as, the values in the source model stored in $smBag$. The final invariant states that the $convergences$, $divergences$, and $absences$ variables are relations among the entities of the reflexion model. This final invariant is weak enough to

4. This variable helps support the treatment of the range of $mappedSourceModel$ as a multigraph, which is helpful when the input source model is a multigraph. Multigraph source models are conventional for dynamically extracted source models (produced by monitoring a program's execution) and are not unusual for statically extracted source models.

5. The invariants have been numbered in parentheses at the far right of the schema for ease of reference.

$$[SMENTITYDESC]$$

$$MapEntry == SMENTITYDESC \times HLMENTITY$$

$$mapFunc : (\text{seq}\, MapEntry \times SMENTITY) \rightarrow (\mathbb{P}\, HLMENTITY)$$

ComputeReflexionModel

$\Delta ReflexionModel$
$hlmEntities? : \mathbb{P}\, HLMENTITY$
$hlm? : HLMRelation$
$sm? : \text{bag}\, SMEntry$
$map? : \text{seq}\, MapEntry$

$$(\text{dom}\, hlm?) \cup (\text{ran}\, hlm?) \subseteq hlmEntities? \quad (1)$$
$$\bigcup(\text{ran}(\text{ran}\, map?)) \subseteq hlmEntities? \quad (2)$$
$$\forall\, m : MapEntry \mid m \in \text{ran}\, map? \bullet (second\ m) \neq \varnothing \quad (3)$$
$$rmEntities' = hlmEntities? \quad (4)$$
$$smBag' = sm? \quad (5)$$
$$mappedSourceModel' = \{from : HLMENTITY;\ to : HLMENTITY;$$
$$t : SMEntry \mid t \text{ in } smBag' \wedge from \in mapFunc(map?, first\ t)$$
$$\wedge\ to \in mapFunc(map?, second\ t) \bullet ((from, to), t)\} \quad (6)$$
$$convergences' = hlm? \cap (\text{dom}\ mappedSourceModel') \quad (7)$$
$$divergences' = (\text{dom}\ mappedSourceModel') \setminus hlm? \quad (8)$$
$$absences' = hlm? \setminus (\text{dom}\ mappedSourceModel') \quad (9)$$

Fig. 5. A Z schema describing the computation of a reflexion model.

permit the presence of entities of the reflexion model that are not part of the *convergences*, *divergences*, or *absences* relations, enabling an engineer to compute a reflexion model in which one or more entities are disjoint from interactions occurring in the system. For instance, an engineer may wish to compute a reflexion model for a system by describing only the entities and not the interactions between the entities of a system. The resultant reflexion model may include disjoint entities that do not serve as an end-point for any computed divergence.[6]

3.2 Computing a Reflexion Model

The dynamic schema, *ComputeReflexionModel*, in Fig. 5, describes the computation of a reflexion model from three inputs: a high-level model, a source model, and a mapping from the source to the high-level model.

We begin by introducing a data type called $SMENTITYDESC$ that represents the type of a description naming zero or more source model entities. We provide details of the realization of the $SMENTITYDESC$ type in the reflexion model tools in Section 3.4.1. Next, we introduce a type, *MapEntry*, that names zero or more source model entities and associates with these entities zero or more high-level model entities. For example, as described in the previous section, in our current reflexion model tools, an entry in the map names a set of source model entities using regular expressions associated with structural features of the software system (i.e., `file=.*pager.*`); the names of associated high-level model entities are simply listed (i.e., `mapTo=Pager`). Finally, using a Z axiom definition, we introduce a function, called $mapFunc$, that matches entities from the source model to the specified map, producing a set of associated high-level model entities.

Given these definitions, the reflexion model computation may now be defined. The high-level model participating in the computation is described by two variables: the $hlmEntities?$ variable describes the set of entities in the high-level model, and the $hlm?$ variable describes a relation over high-level model entities. Splitting the description of the high-level model across these variables permits an entity to be part of a high-level model without requiring it to interact with any other high-level model entity. The source model ($sm?$) is described as a bag of source model entries describing interactions between source model entities. The mapping ($map?$) is an ordered list of map entries.

The first three predicates after the dividing line in Fig. 5 ensure the four input variables ($hlmEntities?$, $hlm?$, $sm?$, and $map?$) contain appropriate values. The predicate on the first line states that the high-level model ($hlm?$) must not introduce any entities not stated in the high-level model entities set ($hlmEntities?$). The precondition on line two

6. In this case, the computed reflexion model will include only divergences since no interactions were posited.

$$
\begin{array}{l}
\textit{ComputeReflexionModelArcValue} \\
\hline
\Xi ReflexionModel \\
rmArc? : HLMEntry \\
arcVal! : \mathbb{N} \\
\hline
rmArc? \in (convergences \cup divergences \cup absences) \\
\forall\, s : SMEntry \mid s \in (mappedSourceModel(\!| \{rmArc?\} |\!)) \bullet \\
\qquad arcVal! = arcVal! + smBag\ count\ s
\end{array}
$$

Fig. 6. A Z schema describing the computation of the values of a reflexion model arc.

constrains the high-level model entities mentioned in the map to be a subset of the $hlmEntities?$ set.[7] The predicate on line three checks that the map is well-formed, ensuring that each entry in the map names at least one high-level model entity as being associated with a set of zero or more source entities.

The next two predicates describe the values of two of the state variables when the $ComputeReflexionModel$ operation completes. The predicate on line four constrains entities in the reflexion model ($rmEntities'$) to be the same as the stated high-level model entities ($hlmEntities?$). The predicate on line five retains the source model as part of the system state ($smBag'$) so that queries can be performed on the reflexion model, such as determining the number of source model interactions contributing to a reflexion model arc.

The value of $mappedSourceModel'$ (line 6) is computed by pushing each source model entry through the map. Specifically, each entry in the source model is separated into its two participating entities using the $first$ and $second$ operations. The $mapFunc$ function is then applied to determine all of the high-level model entities to which each source model entity is associated. The resulting $mappedSourceModel'$ set consists of tuples of the mapped high-level model interaction and the contributing source model interaction. Once $mappedSourceModel'$ is computed, the values of the $convergences'$, the $divergences'$, and the $absences'$ relations are easily determined through set intersection and set difference operations (predicates seven through nine).

3.3 Displaying a Reflexion Model

The dynamic schema, $ComputeReflexionModelArcValue$, describes an operation to determine the numeric label of an arc in a computed reflexion model. This operation takes a reflexion model arc ($rmArc?$) and produces the numeric output value in the $arcVal!$ variable. A user interface for a reflexion model system might use this operation repeatedly, once for each arc in the union of the $convergences$, $divergences$, and $absences$ relations, to display a reflexion model with associated arc values (as in Fig. 2b), shown in Fig. 6.

One precondition is specified for this operation: The arc for which the value is to be computed ($rmArc?$) must be an arc in the reflexion model.

In the last predicate, the relational image operation (i.e., $(\!| \ |\!)$) is used to determine all source model entities associated with the given reflexion model arc ($rmArc$). The sum of the number of times each of those source model entities appears in the source model is then computed and returned as the value of the $ComputeReflexionArcValue$ operation.

3.4 A Family of Reflexion Models

The formal specification defines a family of reflexion model systems. Different kinds of reflexion model systems result depending on the choices made for representing the source model entity descriptions in a map ($SMENTITYDESC$) and for defining the mapping function ($mapFunc$). We discuss the choices we made in realizing these aspects of the formal model in the tools built to support the technique.

3.4.1 Source Model Entity Description Language

The $SMENTITYDESC$ type is the type of the map used in a reflexion model computation. An important characteristic of the map is that it must provide an expressive way to name groups of source model entities that are then associated with a high-level model entity.

One way to provide the necessary expressiveness in naming source model entities is to leverage the organizational information inherent in the implementation of the software system. Two kinds of organizational information are typically available: physical and logical. By physical, we mean the separation of the software into components for storage on a device. For instance, C [15] source code may be stored, using a Unix or FAT file system, into files that are placed into a hierarchical directory structure. By logical organization, we mean the separation of the software into components using language and environment mechanisms. For instance, C source code is organized into functions, whereas Ada [11] source code is organized into functions and packages. The organizational structure varies for different programming languages, environments, and file systems.

To support these differences between languages, environments, and systems, our tools are parameterized by a specification describing the organization of the software system that is the subject of the reflexion model computation. This specification is written in the *source entity description* language. This language defines a tree-based naming mechanism to denote the physical and logical organizational information recorded about a source model entity.

Each node in a naming tree defines a keyword that describes one aspect of the organizational information. The

7. Since $map?$ is defined as a sequence, the first application of the range operation returns a set of $MapEntry$.

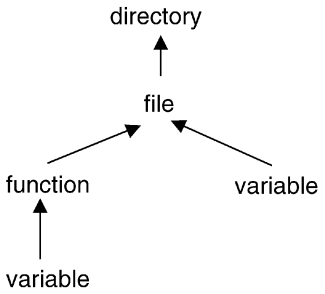


Fig. 7. Source entity naming tree.

defined keywords need only be unique on a path of the tree. For example, Fig. 7 shows a naming tree for function and variable entities within C source code stored in a Unix file system. This tree permits the naming of function entities by specifying values for any combination of physical directory and file information, or logical function name information. Our tools support the use of regular expressions for matching against any possible source entity naming structure that is defined.

When a keyword is used on more than one path in the tree, as is the case with the `variable` keyword in the naming tree of Fig. 7, the naming path used in interpreting a phrase is chosen based on a breadth-first search of the provided keywords.

The source entity description language also affects the source model as the source model must be encoded according to the source entity description language specification. As long as a source model extraction tool is capable of producing an ASCII File, it is generally straightforward to write a script in a text processing language, such as awk or perl [36], to translate the information in the ASCII file into the encoding required by our tools.

3.4.2 Mapping Functions

The formal specification clarifies that the map defined by an engineer between the source model entities and the high-level model entities consists of a sequence of map entries rather than a set of map entries. Characterizing the map as a sequence permits the definition of a mapping function that uses the ordering information in the sequence to produce the set of high-level model entities associated with the first match of a given source model entity to an entry in the map. Treating the map as a sequence is the most common configuration of a reflexion model system used by engineers because it permits the inclusion of "catch-all" entries at the end of the map to associate source model entities not associated by any other entry.

A sequenced map also allows engineers to refine a map by overriding later map entries with early entries. For instance, the Microsoft developer who applied reflexion models to an experimental reengineering of Excel (Section 5) began with a map that stated correspondences between the source and the high-level model at the file-level. As the developer investigated the source through the reflexion model, he added entries that described particular functions within a file that should be associated with a different high-level model entity than most of the functions in a particular file. A sequenced map facilitated this refinement process.

Other mapping functions are also possible. For instance, the reflexion model tools also support a mapping function that returns the set of high-level model entities resulting from the union of all matches found in the map. This definition effectively treats the map as a set of entries rather than a sequence. We have not yet found this treatment useful in practice.

4 Performance

Engineers iteratively specify, compute, and interpret reflexion models. The rate at which an engineer can interpret and iterate reflexion models is dependent, in part, upon the speed of the computation.

Insights into the theoretical complexity of the reflexion model computation can be gained from a consideration of the formal characterization. It is evident from the dynamic Z schema, $ComputeReflexionModel$, that the time complexity of computing a reflexion model is dependent on the cost of forming the $mappedSourceModel$ relation, and on the cost of comparing that computed relation to the high-level model. An upper bound on this time complexity is given by:

$$O(\#(dom\, sm) + ((\#map + t_{comparison}) + (\#hlmEntities)^2)) + O((\#hlm)^2),$$

where $\#(dom\, sm)$ is the number of unique source model values in the source model bag, $\#map$ is number of entries in the map, $t_{comparison}$ is the cost of comparing a source model entity to a source model entity description in a map entry, $\#hlmEntities$ is the cardinality of the set of high-level model entities, and $\#hlm$ is the size of the high-level model relation.[8] Since the number of entities in the high-level model is generally small, particularly in comparison to the size of the source model and the map, the terms involving $hlmEntities$ and hlm can, in practice, be treated as a constant, yielding:

$$O(\#(dom\, sm) \times \#map \times t_{comparison}).$$

Our initial implementation of tools for computing reflexion models performed the computation of a reflexion model in the straightforward manner described by the dynamic schema in Section 3.2. This implementation was sufficiently fast for moderately large systems (with source models consisting of tens of thousands of entries) and small maps (tens of lines), but was not fast enough to support the iterative computation of reflexion models for larger systems or larger maps. For example, an early version of our tools required over 40 minutes on a Pentium II to compute a reflexion model for Excel.

Most of the execution time required to compute a reflexion model is a result of the regular expression matches that must be made between the source model entities and the map given our choice of defining $SMENTITYDESC$.

8. The question marks are dropped from the variable names for simplicity of presentation.

TABLE 1
Speed of Reflexion Model Computations

System	Source Model Entries	Map Entries	Exec. Time (min)	Memory (Mb)
NetBSD	15,656	7	0:02	1.2
restruct. tool	5,855	215	0:03	1.25
Excel	131,042	1425	0:55	2.90

To improve the execution speed of our tools, we addressed this cost in two ways. First, we traded space for time by storing, in a hash table, the match of high-level model entities for a given source model entity the first time a source model entity is seen. Second, we implemented a set of regular expression optimizations within our tools to reduce the number of times a general regular expression library was invoked. These optimizations checked for cases where a simple sequence of string comparisons could be used and for cases where a simple string comparison could eliminate a map entry as a potential match. Further details on the optimizations are available elsewhere [22].

These optimizations enable the computation of reflexion models for large software systems and large maps in less than a minute on a Pentium II (see Table 1). For a large example, such as Excel (see Section 5), the computation of a reflexion model takes just over a minute, requiring just under 3MB of memory. Our tools provide a variety of options to let the engineer determine the appropriate tradeoff between time and space utilization.

5 Experience

The reflexion model approach has been applied to aid engineers in performing a variety of software engineering tasks on a number of different software systems that vary in size and implementation language. Collectively, the variation in these exploratory case studies [39] of tasks, of sizes of systems considered, of implementation languages, and of users builds confidence that the software reflexion model technique may generalize to a useful set of software development scenarios. We discuss four of these case studies in more detail.

5.1 Experimental Reengineering

A software engineer at Microsoft Corporation applied reflexion models to assess the structure of the Excel spreadsheet product prior to an experimental reengineering activity. The traditional methods for a developer at Microsoft to become familiar with the structure of Excel involve reading a high-level document, "Excel Internals," studying the source code, or talking with existing developers of the product. These traditional methods of gaining knowledge of the system were not appropriate for the task facing the engineer because of time constraints and because the engineer could not rely on consistent interaction with a mentor from the development group. Since the Excel product consists of over one million lines of C source code, it was also not tractable to work with direct visualizations of the source or extracted models such as call graphs.

The engineer computed reflexion models several times a day over a four week period to investigate the correspondence between a posited high-level model and a model of almost 120,000 calls and global data references extracted from the source.[9] A detailed mapping file consisting of almost 1,000 entries was produced and was used to guide the extraction of components from the source. The engineer found the approach valuable for understanding the structure of Excel and for planning the experimental reengineering effort. The map produced as part of applying the reflexion model technique was also used to perform parts of the experimental reengineering activity. The engineer continued to apply the technique after the initial use, eventually specifying a high-level model with 16 entities and 114 interactions, and a map with 1,425 entries. Full details on the use of the reflexion model technique for this task are available elsewhere [20].

5.2 Design Conformance

5.2.1 Program Restructuring Tool

We used a sequence of reflexion models to compare the layered architectural design of Griswold's program restructuring tool [10] with a source model consisting of calls between modules. The reflexion model highlighted, as divergences, a few cases where modules in the CLOS [2] source code did not adhere to the desired layering principles. These divergences indicated to the author of the system places in the code where restructurings had not been consistently or completely applied. Identifying these code sections was beneficial as the author of the system was able to revisit the locations and update the code to ensure the implementation encoded the appropriate structure. We are unaware of any other approach that would allow such violations to be found so directly.

5.2.2 SPIN Operating System

We also applied the technique to help developers at the University of Washington determine if the *SPIN* operating system [5] was being developed to plan. The *SPIN* kernel, which is primarily implemented in Modula-3 [7], is layered on top of the DEC OSF/1 operating system, which is primarily implemented in C. Two aspects of the structure of the *SPIN* system were investigated: the calls between Modula-3 modules and the calls from the Modula-3 *SPIN* code to the DEC OSF/1 C code.

The lexical source model extraction technique [19] was used to produce a source model consisting of the two kinds of calls. The high-level model used by the developers was

9. The calls and global data reference information were extracted without consideration of aliasing. We discuss this point in more detail in Section 6.1.

based on the directory structure, which paralleled the desired architecture for the system. This high-level model consisted solely of entities with no interactions posited between the entities. Basing the high-level model entities on the directory structure allowed the map to be automatically generated with a shell script: Each entry in the map associated a directory with a high-level model entity. The reflexion model computed from these inputs had 76 divergences. Looking at these divergences, a *SPIN* team member identified a few suspect interactions, which were discussed with other team members. As a result of the reflexion model computation, the implementation was modified to remove the suspect divergences.

5.2.3 Industrial Use

An industrial partner applied reflexion models to check if a 6,000 line C++ implementation of a subsystem matched design documentation (in the form of a Booch object diagram [4]) prepared prior to implementation. This case was unique in that the reflexion model was fully convergent with the source model. This result is not surprising given that the implementation was small, the designer of the system also implemented the system, and the implementation had not undergone any evolutionary tasks. Fully convergent reflexion models can be helpful in increasing the developer's confidence that an implementation meets desired structural properties.

6 Discussion

Software reflexion models permit an engineer to easily explore structural aspects of a large software system. The goal of the approach is to provide engineers with the flexibility to produce, at low-cost, high-level models that are "good enough" for performing a particular software engineering task, such as restructuring, reengineering, or porting. Three aspects of the approach critical to meeting this goal are the use of syntactic models, the use of expressive declarative maps, and support for querying a reflexion model. In this section, we discuss these aspects, as well as the role of visualization in the use of the technique.

6.1 Syntactic Models

Reflexion models are computed without any knowledge of the intended semantics of the high-level or the source models. The engineer provides the semantics when specifying and interpreting a reflexion model. We chose to use syntactic models rather than semantic models for two reasons. First, syntactic models provide significant flexibility, allowing engineers to investigate many different kinds of structural interactions in a software system, including among others, calls, data dependences, and event interactions. This flexibility has been beneficial in practice. Both the VM developer and the Microsoft engineer first used a calls source model and later augmented the source models with static dependences of function definitions on global data. By adding static dependences, both developers implicitly shifted their high-level model from a calls diagram to a communicates-with diagram. In both cases, the changes were driven by the need to understand, for the task being performed, additional aspects of the system structure.

The second reason for choosing syntactic models was to support the goal of providing a lightweight technique. Syntactic models support this goal because they require minimal specification by an engineer.

Using syntactic models for the comparison has several consequences. One consequence is that the engineer must ensure that it makes sense to compare a selected high-level model with an extracted source model. For example, comparing the calls between modules diagram of the NetBSD virtual memory system (Fig. 2a) with a source model consisting of an extracted calls relation makes sense (Fig. 2b); comparing the same high-level model with a source model representing the `#include` structure of NetBSD would probably be meaningless. In practice, we have not seen engineers attempt inappropriate comparisons.

A second consequence of syntactic models is that the engineer is responsible for selecting, or at least understanding, the kind of the information comprising each structural relation in a source model.

A structural relation, such as the `includes` relation between C files, may be precise, meaning that there are neither any false positives or false negatives.[10] Other relations, such as the C calls relation used by the Microsoft engineer which was extracted without consideration of aliasing, are approximate containing both false positives and false negatives. Extracted relations may also be conservative, containing false positives but not false negatives, or, what we have called optimistic [21], containing false negatives but not false positives. Since the software engineer is involved in the process of extraction, we have not encountered any cases in which the engineer assumes a computed reflexion model is summarizing conservative source information when it is not. Each of these kinds of information in a structural relation can be useful for computing a software reflexion model. A software engineer must simply be cognizant of the kinds of reasoning that any particular kind of information can support. The VM developer, for instance, using an approximate C calls relation in the computation of reflexion models, could not assume that two modules did not interact simply by the lack of a divergence or convergence between the two components in a reflexion model.

While the syntactic nature of the source model induces this consequence, it also helps a software engineer address the issue. As described at the beginning of this section, a software engineer may create a source model that is the union of two or more extracted relations. For example, a software engineer may augment an approximate statically-extracted C calls relation with call information based on profiling. The statically-extracted relation may not contain information about calls through function pointers; the profiling information will contain such information. The union of these two relations may provide the engineer with more confidence in reasoning about the interconnection of two modules.

10. A false positive in an `includes` relation refers to the presence of a pair describing an inclusion of one file in another that does not hold in the source code. A false negative refers to an inclusion that holds in the source but is not represented in the relation.

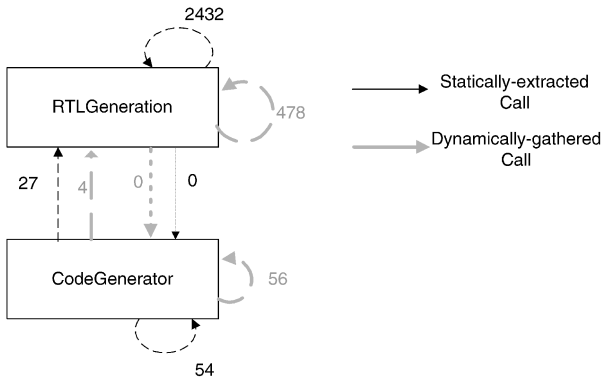


Fig. 8. Reflecting static and dynamic information.

To further aid the software engineer, we have extended the software reflexion model technique to handle typed source and high-level models. In a typed source model, each interaction between source model entities has an associated type, or name. In a typed high-level model, an interaction between high-level model entities may have an associated type (name); some interactions in the high-level model may remain untyped to ease specification of the high-level model. To support a consistent interpretation of a typed reflexion model, if any interaction between two high-level model entities is typed, all interactions between those two entities must be typed. When computing a software reflexion model from typed models, the types from the source model interactions are retained as they are pushed through the map. Each computed arc is then compared with the high-level model. If the arc matches a high-level model interaction that is typed, the types are included in the comparison. If the arc matches an interaction in the high-level model that is untyped, convergences are created corresponding to the induced types from the source model interactions.

To demonstrate how a typed reflexion model may help a software engineer reason about different kinds of information, Fig. 8 shows a snippet of a typed reflexion model computed for the GNU gcc system. This snippet focuses on the interaction between the generation of the intermediate representation by the compiler and the code generator. The source model used in the computation includes two types: one type represents statically-extracted call information and the other type represents call information gathered dynamically from a profiler. The high-level model was untyped. In this case, the statically-analyzed information is approximate (e.g., calls through function pointers are not included), while the dynamic call information contains false negatives. The arcs in the reflexion model resulting from the static information may help an engineer reason about a change task. The arcs resulting from the dynamic information may help an engineer reason about the coverage of various test cases. The arcs resulting from the dynamic information can also help to augment the static calls information; for instance, the dynamic information will include calls made through function pointers. Further details on typed reflexion models are available elsewhere [22].

6.2 Maps

Equally as important as the choice of syntactic models in the definition of the technique was the design of the form and content of the map. The form of the map, chosen to be an entity-to-entity correspondence, affects the range of computations that may be performed, while the content, chosen to be a parameterized declarative language, affects the usability of the technique.

6.2.1 The Form of a Map

A map for a reflexion model computation states a correspondence between some (or all) of the entities of the source model and some (or all) of the entities of the high-level model. This entity-to-entity form for the map was chosen because it supports the projection of interactions from the source model to a higher level of abstraction.

Other choices for the form of this map are also possible. For instance, an engineer may desire to map entities in a source model to interactions within a high-level model. Sullivan describes such a case, where an engineer may model a portion of an integrated programming environment as an `Editor` that may cause an action in a `Compiler` on certain events (top portion of Fig. 9) [35, pp. 11–14]. However, in the implementation, the relationship between the two tools may be implemented as an entity called a `mediator` (bottom portion of Fig. 9). This kind of association is not supported directly by a reflexion model computation. Rather, an engineer must choose one of the following approaches: introduce a mediator entity in the high-level model, associate the source level `mediator` entity with one or both of the `Editor` and `Compiler` high-level model entities, or

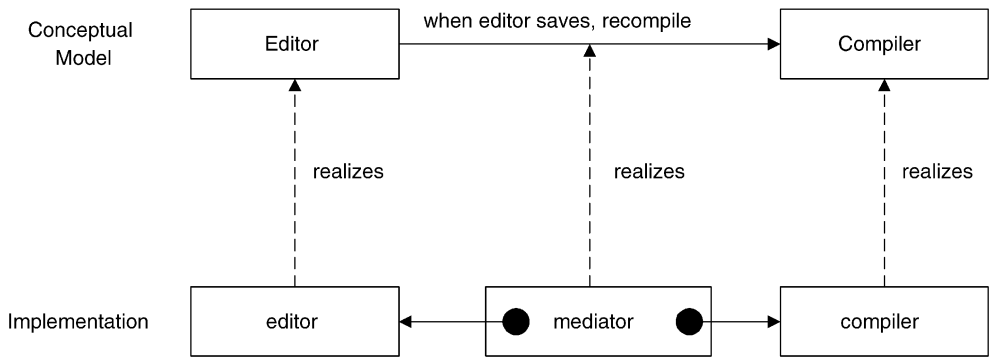


Fig. 9. A mediator-based realization of a programming environment.

preprocess the source model to compute the appropriate interpretation of interactions between the `mediator`, the `editor`, and `compiler` to interactions involving only the `editor` and the `compiler`. Similar difficulties arise if an engineer desires to associate interactions in a source model with interactions in a high-level model. More research is needed to understand when different forms for the map are appropriate and the consequences of the desired forms on a reflexion model computation.

Another design choice made in the current map form was to support the specification of partial maps in which an engineer states only a partial correspondence between the two models. Similar to syntactic models, partial maps are beneficial because they contribute to the lightweight nature of the technique. An engineer, for example, may focus on a particular subsystem important to a task and then successively include other subsystems as necessary by expanding the map. Partial maps are also beneficial in handling scale. Rather than having to provide a full correspondence for a system like NetBSD that contains 3,000 source model entities, an engineer may focus on those aspects of the source model pertinent to the task, enabling an engineer to manage the time spent in defining maps with the benefits accrued from using the technique.

A partial map could also be detrimental if an engineer believes that the map pertains to all entities in the source model. To guard against this situation, we provide queries to help an engineer determine the parts of a source model covered by a map. More information about these queries is available in Section 6.4.

6.3 The Content of a Map

For the technique to be usable, it is imperative that a software engineer be able to specify the content of a map easily, even when there are many source entities. As described before, we chose to implement the *SMENTITYDESC* type by parameterizing the language of the map by a source entity description language. This language is sufficiently flexible to describe the physical and logical organization of a wide-range of source structures. The language has been used, for example, to describe the physical and logical organization of Ada [11], C++, C [15], CLOS, Eiffel [17], TCL/TK [25], and Field structured data files [28], [29] for the purposes of computing a software reflexion model. Furthermore, the parameterization of the map language by the source entity description language permits an engineer to use the vernacular of the system to describe the desired map, making the specification of the map more natural to the engineer. For example, when analyzing a system implemented in C++, an engineer may define a description language to permit the reference of entities by familiar terms, such as file, class, method, or function.

For it to be tractable for the engineer to specify a map for a large system, the source model entity description language must easily permit the description of groups of source model entities. We believe our use of both hierarchical structural information and regular expressions in the source model entity description language provides a good solution to this problem for a large set of interesting systems. Software engineers may peruse the source code and easily state a map. Alternatively, we could have chosen to use one of the existing pattern- or logic-based languages available for describing nodes in an abstract-syntax tree (AST) as a means for identifying source model entities [3], [8]. This approach would allow a more precise description of a source model entity; for example, an engineer might identify a particular call site to be mapped to a particular high-level model entity. The increase in precision of stating a source model entity is offset by two drawbacks of the approach: It may be more difficult to state a map because the engineer would have to understand the AST formulation of the system, and the extracted source model would need to correspond to the AST, limiting the choices of source model extractors that might be used.

Our use of regular expressions allows an engineer to take advantage of naming conventions in the system's source. For systems without strong naming conventions, the use of physical and logical structure can make the map easier to state. The systems for which the source entity description language may not work adequately are those that have neither patterns in naming, nor any structure. We have not found this to be a problem in the systems that we have studied.

Another advantage of the source entity description language is that it allows an engineer to define a semantically approximate map in which an entry coarsely associates source model entities with high-level model entities. For example, an engineer may define an entry that associates, by way of substructure, all C functions in a particular file with a specified set of high-level model entities. This mapping may be approximate because one or more of the functions, to support reasoning about the task, may need to be remapped to a different set of high-level model entities; that is, the mapping by structure alone is not sufficiently precise. The source entity description language thus admits false positives in a map of its ability to name groups of entities. The language also admits false negatives in allowing entities to remain unmapped.[11] Approximate maps that were later refined were used by the Microsoft engineer applying the reflexion model technique to Excel. The engineer was able to determine which areas of the reflexion model required refinement through successive queries for details on the source model interactions mapped to various convergences and divergences (see Section 6.4). The ability to approximately map entities until an engineer finds that a finer granularity is needed is beneficial in permitting an engineer to focus attention, through successive map refinements, on the areas of structure of most interest to a task at hand. Similar to the case of partial maps described above, approximate maps contribute to the lightweight nature of the technique and aid in handling scale.

6.4 Querying Reflexion Models

Reflexion models bridge the gap between an engineer's high-level model and a model of the source. The convergences,

11. The concept of the source entity description language admitting false negatives is related to the partiality of the map. These two concepts differ in the intent of the engineer in defining the map. An engineer defines a partial map when interested in a subset of the system. A map has false negatives when, given a defined subset of the system of interest, the engineer defines a map that leaves desired entities unmapped.

divergences, and absences summarize selected interactions in the source, while the $mappedSourceModel$ captures the connections between the high-level and source model arcs. Based on the summary information, the engineer intersperses two kinds of queries to interpret a reflexion model for a specific software engineering task.

In the first kind of query, an engineer investigates the source model values contributing to a convergence or divergence. Fig. 3 shows an example of this kind of query.

In the second case, an engineer performs queries to determine the source model entities and values that were not included in the reflexion model. Three queries fit into this category. First, the engineer may view a list that describes, for each high-level model entity, the source model entities that have been mapped to that high-level model entity. This list may be used to try to understand absences in a computed reflexion model. Second, an engineer may view a list of the source model values (and, hence, entities) that are not mapped by the computation. This list aids an engineer in understanding what aspects of the source model are not being included in the reflexion model summarization. Third, an engineer may request a quantitative summary of the source model covered by a mapping. This value reports the percentage of the source model that is mapped. This percentage provides a broad basis on which an engineer may assess the amount of information being considered.

Based on the results of the interpretation of the queries, an engineer may either decide to refine one or more of the inputs, computing a subsequent reflexion model, or else may decide that sufficient information has been obtained to proceed with the overall task.

6.5 Visualization

Although the reflexion model technique has a visual component, the graphical visualization of a computed reflexion model has not been essential to the effective use of the technique. The Microsoft engineer, for instance, investigated a series of reflexion models primarily from the textual output of our tools. The engineer drove the investigation of a reflexion model from the counts attached to arcs. This information is easy to understand in textual form by sorting the arcs and associated counts, but may be difficult if the information is only embedded within a graphical layout of a reflexion model. This experience highlights that further research is needed to help determine which style of information display is more effective for different kinds of program comprehension tasks.

7 Related Work

7.1 Consistency Checkers

A number of checking tools compare the structure of a software system with the intended structure. The GRID approach developed by Ossher included such a checker to compare source structure against a GRID description [24]. The Software Landscape environment supports similar functionality through a visual module interconnection language [26]. In both of these approaches, relations between program entities can be extracted and compared directly to relations between low-level design entities. Each of these checkers was particular to the kind of structural diagrams supported by the respective design methods. The software reflexion technique provides a more general structural checker: Fewer constraints are placed on the form of the high-level (design) model, and greater flexibility is supported for associating source model entities directly with high-level model entities of interest.

Sefika and colleagues describe a system, called Pattern-Lint, that supports compliance checking of implementations to a wide-range of high-level models [33]. In Pattern-Lint, an engineer describes the high-level model as two sets of Prolog clauses: one set of clauses defines positive evidence that the source complies with the model and the second set defines violations that indicate noncompliance. An engineer may then compare structural information statically extracted from C++ source code with the predefined models. The system also supports compliance checking through a set of animations of dynamic structural information; for example, calls reported when executing an instrumented version of the program.

The software reflexion model technique differs from Pattern-Lint in several ways. First, in the reflexion model technique, the engineer need only state what is expected for compliance with the high-level model, whereas in Pattern-Lint, the engineer must state what is expected as well as what is not expected for compliance. If an engineer neglects to include a particular kind of violation in a Pattern-Lint model and the violation occurs, it will not be reported by Pattern-Lint, whereas it may be reported as a divergence using reflexion models. Second, in Pattern-Lint both the clauses defining the compliance rules for a model and the association of source model entities with higher-level entities are specified as Prolog clauses. It is unclear whether this description, which may require more time and effort for the engineer to specify compared to our use of syntactic models and a mapping language, affects the scalability of the Pattern-Lint system. Third, with the reflexion model technique, dynamic information collected during the execution of a system may be summarized in terms of the same high-level model(s) in which information extracted statically is viewed. In Pattern-Lint, different views of the system are used to display this information.

7.2 Reverse Engineering

Reverse engineering is the process of creating higher-level abstractions from source code [6]. An engineer may choose to reverse engineer a software system when specifications do not exist as by-products of up-stream software engineering activities or when the system views created during those activities are inconsistent with the source code or are inappropriate for the task being performed.

Lakhotia characterizes 12 reverse engineering techniques based on clustering according to, among other things, the level of automation and the nature of the source information supported by the technique, such as control-flow graphs and data-flow graphs [16]. Of the 12 techniques surveyed by Lakhotia, the Rigi system [18] is the sole technique to operate, semiautomatically, on a generic set of source model relations similar to the software reflexion model technique. We thus focus on a comparison of Rigi and a similar system, called Mercury, built at Microsoft Research.

The Rigi system "supports a method for identifying, building, and documenting layered subsystem hierarchies" [37, p. 47]. Similar to the reflexion model technique, an engineer begins by extracting structural information from system artifacts and by representing that information as a set of relations. These relations are provided as input to the Rigi tool, which displays them as a collection of overlapping graphs. A user then repeatedly determines criteria to apply to cluster elements from a displayed graph of structural information. The criteria may be based, among others, on graph characteristics such as determining the strongly connected components, or may be based on naming conventions of the elements extracted from the source. After applying a sequence of clustering operations, the user can build up a higher-level view of the structural information in terms of the clustered components. This kind of technique is attractive because the user can choose appropriate structural information for the task at hand, and because the technique may be applied in the absence of any knowledge by the engineer of the structure of the source. This bottom-up process of clustering, however, can be overwhelming in the presence of a large source model. Even when significant clustering has been performed to derive a high-level model, it is unclear whether the model is consistent with the user's conceptual view and for which tasks the model is helpful to the engineer:

> For our first experiment, we generated a view of the entire call graph without considering any SQL/DS-specific domain knowledge. The result was not as encouraging as we would have liked. The developers did not recognize the abstractions we generated, making it difficult for them to give us constructive feedback. This reaffirmed our belief that successful reverse engineering must do more than manipulate system representations independent of their domain; the results must add value for its customers. Informal information and application specific knowledge provided by existing documentation and expert system developers are rich veins of data that should be mined whenever possible [37, p. 51].

It appears that most uses of Rigi create high-level abstractions of a software system in a largely or entirely task-independent manner. This is consistent with the conventional view of reverse engineering, where the objective is to get some abstractions to help understand and then modify a system. The clustering tends to be applied somewhat uniformly across the system. This contrasts with the reflexion model approach in which the engineer more aggressively chooses where to refine the information and where to allow inconsistencies in the reflexion model to remain.

A final distinction is that the source model retains its full identity in the reflexion model tools. In Rigi, the clustering information is commingled with the original source model information.

Like Rigi, Weise's Mercury system supports the construction of layered subsystem hierarchies.[12] Mercury differs from Rigi in the operations provided to create the subsystems. Rather than identifying groups of source model entities from which to create a cluster, in the Mercury tool, an engineer posits the clusters and then performs move operations to associate source model entities with the appropriate cluster. The move operations are implemented through a point-and-click interface that allows an engineer to drag a source model entity from one cluster to another as the source is investigated. Mercury, like Rigi, derives arcs between the clusters from the underlying source model information.

The Mercury tool was specifically designed to handle the logical remodularization operations that were performed repeatedly by the engineer using reflexion models to experimentally reengineer Excel. Similar to the reflexion model tool, the Mercury tool requires an engineer to posit the desired clusters—high-level model entities in the terminology of reflexion models. An engineer using the Mercury tool starts by specifying the same input files that are expected by the software reflexion model tools. Once input, however, the map becomes implicit and an engineer performs the desired move operations through the point-and-click interface. More investigation is needed to determine the benefits and limitations of the declarative map approach used in the reflexion model tools versus the direct manipulation of icons used in the Mercury tool, particularly when large changes are made to the map. Unlike the reflexion model tools, Mercury does not permit an engineer to posit interactions between high-level model entities and thus does not support a comparison between interactions in a high-level model and interactions in a source model. It is unclear how the lack of this comparison affects the early investigation of unrefined reflexion models. The Microsoft engineer, for instance, used this information to refine the high-level model in reflexion model computations performed at the beginning of the experimental reengineering task.

7.3 Knowledge-Based Approaches

Another class of techniques to aid an engineer in understanding software structure use knowledge-bases. These techniques may be classified into two categories by the kind of knowledge comprising the knowledge-base. Techniques in the first category encode knowledge as predefined patterns or cliches [30] that are generally domain-independent. Techniques in the second category encode knowledge about the domain and the system design.

Cliche-based techniques may be further subdivided into automated, semiautomated, and manual methods. Automated program understanding techniques, such as Quilici's technique [27], extract design information by matching recorded cliches to the source. These techniques differ in intent from the software reflexion model technique as they primarily focus on building a representation and understanding of the target system within a tool rather than on aiding the engineer in gaining the understanding.

Semiautomated cliche-based techniques use the precoded patterns as a guide to aid an engineer in producing a higher-level view of the system through the successive applications of transformations. An example of a semiautomated cliche-based technique is the ManSART system in which recognizers of architectural styles are used to build architectural views of a system that may then be combined and transformed by an engineer through the application of various operators [38]. Since, like Rigi,

12. Personal communication, D. Weise, 1996.

these semiautomated techniques largely rely on a bottom-up process, they may encounter problems associated with scale and may not produce views appropriate for reasoning about a software engineering task at hand.

An example of a manual cliche-based technique is the Synchronized Refinement approach [23] in which an engineer states an expected design and then successively tries to use semantically-based cliches to rewrite the source code as higher-level pseudo-code to produce that design. When necessary, as driven by information from a code analysis, the engineer may also modify the expected design. This approach is similar in intent to the software reflexion model technique but does not use any automated comparison. Two limitations shared by all cliche-based techniques are the need to determine appropriate representations for cliches and the need to identify and populate the knowledge-bases. Compared to the software reflexion model technique, however, these approaches share the advantage that it may be possible to derive high-level models that contain both behavioral and structural information.

Closer in intent to the reflexion model technique are the category of techniques that use a knowledge-base of facts about a domain and a system to permit an engineer to perform queries at a higher-level of abstraction than the source. The LaSSIE system [9] is representative of this approach. In LaSSIE, the knowledge-base contains information about the domain (i.e., telecommunications), the architecture, the features, and the code of the system. An engineer may use the system to determine the answers to queries, such as which functions implementing a specific domain concept access global variables in a particular file. This type of technique shows promise for supporting an engineer trying to reason about desired changes. Further research may be needed to determine if the effort of building the necessary knowledge base and maintaining its consistency is cost-effective.

7.4 Model Comparison

Software reflexion models result from the comparison of two models at different levels of abstraction. An essential characteristic of the reflexion model technique is its use of a declarative map to associate the two models. The mapping language was designed with two goals in mind. First, the mapping language must be independent of the source programming language; this allows (among other things) the reflexion model approach to be applied to systems written in different languages or even in multiple languages. Second, it had to be concise, since source models may be quite large and the map could not be of nearly the same size as the source model; this was achieved using both logical and physical structures and also regular expressions.

Some other model comparison approaches have quite different objectives.

The Aspect system supports the comparison of partial program specifications (high-level models) to data flow models extracted from the source [13]. The system is able to detect bugs in the source that cannot be detected using static type checking. Aspect uses dependences between data stores as a model of the behavior of a system. For abstract types, the system permits an engineer to express the dependences between data stores in terms of abstract components rather than in terms of the type's representation. An abstraction function is used to relate the concrete components of the type to the abstract components stated in the specification. When the Aspect checker is run, differences between the information extracted from the source and the specification are reported both in terms of the abstract and the concrete components.

Maps in the software reflexion model technique play a similar role to Aspect's abstraction functions. Both maps and abstraction functions permit many-to-many association between entities in models at different levels of abstraction. However, in Aspect's abstraction functions, there is no notion of approximation as is found in the reflexion model maps where coarse-grained maps are supported but may be refined over time. Aspect's abstraction functions also differ from software reflexion maps in that they appear, to the engineer, to be bidirectional; the functions are used to translate source dependences to abstract dependences and are also used to report discrepancies found at the abstract level in terms of the concrete level. Reflexion model maps, on the other hand, are used only unidirectionally to push source model interactions through to the higher-level; the information about which source model values contribute to a reflexion model arc is independent of the map. A bidirectional treatment of the reflexion model map might extend the kinds of queries available to an engineer when successively refining a reflexion model; for instance, it might permit the determination of whether a reflexion model arc is indicative of a relationship holding between all source model entities mapped to the respective high-level model entities or only some of the high-level model entities.

Howden and Wieand's Quick Defect Analysis (QDA) involves the introduction, by an engineer, of comments describing facts and hypothesis about a program's problem domain into the source code [12]. Once commented, an engineer may use an analyzer tool to interpret the abstract program defined by the comments and the control flow defined by the source code to verify hypotheses. In this approach, a special kind of comment, a rule, may be used to associate program-oriented properties with properties of the abstract program. These rules thus play a similar role to the map of the software reflexion model technique. Similar to Aspect, the intent of the map in QDA is to enable automated analysis at the abstract level. To support this objective, the QDA map language, in contrast to a software reflexion model map, is richer and more precise, supporting an association between the conjunction of the states of one or more concrete properties to the state of a single abstract property.

Jackson and Ladd have developed a semantic difference approach for comparing the differences in input and output behavior between two versions of a procedure [14]. Given two versions of a procedure, this approach derives a model of the semantic effect of each procedure consisting of a binary relation that describes the dependence of variables at the exit point of the procedure with variables upon entry to the procedure. The semantic differencing tool compares the relations resulting from each version of the procedure. This tool thus differs from the reflexion model technique in

supporting the comparison of relations at the same level of abstraction rather than the comparison of relations at different levels of abstraction.

8 Summary

The software reflexion model technique permits an engineer to summarize structural information extracted from the source within the framework of a high-level model. This technique is

- lightweight and iterative, in that an engineer can quickly and easily compute a reflexion model and then may selectively refine the inputs and recompute models until the desired information is obtained,
- approximate and partial, in that the map used in the computation of a reflexion model may coarsely associate source model entities with high-level model entities and may not include all entities in a given source or high-level model in any particular computation.

These features combine to provide a scalable approach for comparing implementations to designs that may be applied to systems of either a few thousand or millions-of-lines of code. The utility of the technique has been shown in practice through its use by a Microsoft developer to help with an experimental reengineering task of the Excel spreadsheet product.

The software reflexion model technique provides a means of *bridging the gap* between the high-level models commonly used by engineers to reason about a software system and the system artifacts that are the software system. Closing this gap allows an engineer to produce a view of the system specific to the task being performed. More generally, closing this gap may permit engineers to produce documentation about a system on-demand. The ability to produce documentation on-demand could have a long-term impact on the role that some forms of documentation play in the development process.

The technique also demonstrates that *approximate* structural information can be beneficial to an engineer planning, assessing, and executing tasks on an existing system. The degree and kind of approximate information that is useful is dependent on the task that is being reasoned about. However, relaxing the requirement for conservative information may open up new approaches for accessing structural information from very large software systems.

Acknowledgments

This research was funded from a number of sources, including a Natural Sciences and Engineering Research Council (NSERC) of Canada post-graduate scholarship and research grant, US National Science Foundation grants CCR-8858804, CCR-9506779, CCR-9502029, CCR-9506779, a University of Washington Department of Computer Science & Engineering Educator's Fellowship, DARPA/Rome Labs F-30603-96-1-0314, and a grant from Lockheed-Martin. Equipment was also provided by Microsoft Corporation. This paper is an extended and revised version of a paper that appeared in the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering (1995). The authors wish to thank Robert Allen, Kingsum Chow, David Garlan, Bill Griswold, Michael Jackson, Kurt Partridge, Bob Schwanke, and Michael VanHilst, who each provided helpful comments on earlier drafts of this material. Conversations with Daniel Jackson helped clarify a number of aspects of our work. An anonymous Microsoft engineer applied reflexion models to Excel. Dylan McNamee was our VM developer. Pok Wong applied reflexion models as part of a design conformance task. Stephen North of AT&T provided the graphviz graph display and editing package. They also thank the anonymous referees for their constructive comments.

References

[1] A.V. Aho, B.W. Kernighan, and P.J. Weinberger, "Awk—A Pattern Scanning and Processing Language," *Software—Practice and Experience,* vol. 9, no. 4, pp. 267–280, 1979.

[2] D.G. Bobrow, L.G. Demichiel, R.P. Gabriel, S.E. Keene, and G. Kiczales, "Common Lisp Object System Specification," *Lisp and Symbolic Computation,* vol. 1, no. 3/4, pp. 245–394, 1989, also appears as X3J13 Document 88-002R.

[3] S. Burson, G.B. Kotik, and L.Z. Markosian, "A Program Transformation Approach to Automating Software Re-Engineering," *Proc. 14th Ann. Int'l Computer Software and Applications Conf.,* pp. 314–322, Nov. 1990.

[4] G. Booch, *Object-Oriented Design with Applications.* Benjamin-Cummings, 1991.

[5] B. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System," *Operating Systems Review,* vol. 29, no. 5, pp. 267–284, 1995.

[6] E.J. Chikofsky and J.H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software,* vol. 7, no. 1, pp. 13–17, Jan. 1990.

[7] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson, "The Modula-3 Type System," *Proc. Conf. Record 16th Ann. ACM Symp. Principles of Programming Languages,* pp. 202–212, 1989.

[8] R.F. Crew, "Astlog: A Language for Examining Abstract Syntax Trees," *Proc. USENIX Conf. Domain-Specific Languages,* Oct. 1997.

[9] P. Devanbu, R.J. Brachman, P.G. Selfridge, and B.W. Ballard, "LaSSIE: A Knowledge-Based Software Information System," *Comm. ACM,* vol. 34, no. 5, pp. 34–49, May 1991.

[10] W.G. Griswold and D. Notkin, "Architectural Tradeoffs for a Meaning-Preserving Program Restructuring Tool," *IEEE Trans. Software Eng.,* vol. 21, no. 4, pp. 275–287, Apr. 1995.

[11] *Reference Manual for the Ada Programming Language.* United States Government Printing Office, 1983, MIL STD 1815A.

[12] W.E. Howden and B. Wieand, "QDA—A Method for Systematic Informal Program Analysis," *IEEE Trans. Software Eng.,* vol. 20, no. 6, pp. 445–462, June 1994.

[13] D. Jackson, "Aspect: Detecting Bugs with Abstract Dependences," *ACM Trans. Software Eng. and Methodology,* vol. 4, no. 2, pp. 109–145, 1995.

[14] D. Jackson and D.A. Ladd, "Semantic Diff: A Tool For Summarizing the Effects of Modifications," *Proc. Int'l Conf. Software Maintenance,* Sept. 1994.

[15] B. Kernighan and D. Ritchie, *The C Programming Language.* Prentice Hall, 1978.

[16] A. Lakhotia, "A Unified Framework for Expressing Software Subsystem Classification Techniques," *J. Systems and Software,* 1996.

[17] B. Meyer, *Eiffel: The Language.* Prentice Hall, 1992.

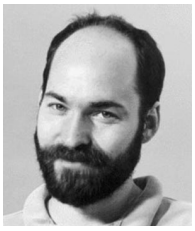
[18] H.A. Müller and K. Klashinsky, "A System for Programming-in-the-Large," *Proc. 10th Int'l Conf. Software Eng.,* pp. 80–86, Apr. 1988.

[19] G.C. Murphy and D. Notkin, "Lightweight Lexical Source Model Extraction," *ACM Trans. Software Eng. and Methodology,* vol. 5, no. 3, pp. 262–292, July 1996.

[20] G.C. Murphy and D. Notkin, "Reengineering with Reflexion Models: A Case Study," *Computer,* vol. 30, no. 8, pp. 29–36, Aug. 1997.
[21] G.C. Murphy, D. Notkin, W.G. Griswold, and E.S-C. Lan, "An Empirical Study of Static Call Graph Extractors," *ACM Trans. Software Eng. and Methodology,* vol. 7, no. 2, pp. 158–191, 1998.
[22] G.C. Murphy, "Lightweight Structural Summarization as an Aid to Software Evolutionm," PhD thesis, Univ. of Washington, July 1996.
[23] S.B. Ornburn and S. Rugaber, "Reverse Engineering: Resolving Conflicts Between Expected and Actual Software Designs," *Proc. IEEE Conf. Software Maintenance,* pp. 32–40, Nov. 1992.
[24] H.L. Ossher, "A New Program Structuring Mechanism Based on Layered Graphs," PhD thesis, Stanford Univ., Dec. 1984.
[25] J. Ousterhout, *TCL and the TK Toolkit.* Addison-Wesley, 1994.
[26] D.A. Penny, "The Software Landscape: A Visual Formalism for Programming-in-the-Large," PhD thesis, Univ. of Toronto, Toronto, Canada, 1993.
[27] A. Quilici, "A Memory-Based Approach to Recognizing Programming Plans," *Comm. ACM,* vol. 37, no. 5, pp. 84–93, 1994.
[28] S.P. Reiss, "Connecting Tools using Message Passing in the Field Program Development Environment," *IEEE Software,* vol. 7, no. 4, pp. 57–66, 1990.
[29] S.P. Reiss, *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development,* Kluwer Academic, 1995.
[30] C. Rich and R. Waters, *The Programmer's Apprentice.* Reading, Mass.: Addison-Wesley, 1990.
[31] B.C. Smith, "Reflection and Semantics in LISP," *Proc. ACM Principles of Programming Languages Conf.,* pp. 23–35, Dec. 1984.
[32] J.M. Spivey, *The Z Notation,* second ed. Prentice Hall, 1992.
[33] M. Sefika, A. Sane, and R.H. Campbell, "Monitoring Compliance of a Software System with its High-Level Design Models," *Proc. 18th Int'l Conf. Software Eng.,* pp. 387–396, 1996.
[34] B. Stroustrup, *C++ Programming Language.* Addison-Wesley, 1986.
[35] K.J. Sullivan, "Mediators: Easing the Design and Evolution of Integrated Software Systems," PhD thesis, Univ. of Washington, Seattle, WA, Aug. 1994.
[36] L. Wall, *Programming Perl.* O'Reilly & Associates, 1990.
[37] K. Wong, S.R. Tilley, H.A. Müller, and M.D. Storey, "Structural Redocumentation: A Case Study," *IEEE Software,* vol. 12, no. 1, pp. 46–54, Jan. 1995.
[38] A.S. Yeh, D.R. Harris, and M.P. Chase, "Manipulating Recovered Software Architecture Views," *Proc. Int'l Conf. Software Eng.,* pp. 184–194, 1997.
[39] R.K. Yin, *Case Study Research.* Beverly Hills, Calif.: Sage Publications, 1984.

Gail Murphy received the BSc degree in computing sciences from the University of Alberta in 1987 and the MS and PhD degrees in computer science and engineering from the University of Washington in 1994 and 1996, respectively. From 1987 to 1992, she worked as a software designer in industry. Dr. Murphy is currently an assistant professor in the Department of Computer Science at the University of British Columbia. Her research interests are in software evolution, software design, and source code analysis. She is a member of the IEEE Computer Society.

David Notkin received the ScB degree at Brown University in 1977 and the PhD degree at Carnegie Mellon University in 1984. He is the Boeing Professor of Computer Science and Engineering at the University of Washington. Dr. Notkin received the US National Science Foundation Presidential Young Investigator Award in 1988, served as the program chair of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering, served as program cochair of the 17th International Conference on Software Engineering, chaired the Steering committee of the International Conference on Software Engineering (1994-1996), served as charter associate editor of both *ACM Transactions on Software Engineering and Methodology* and the Journal of Programming Languages, serves as an associate editor of the *IEEE Transactions on Software Engineering*, was named as an ACM Fellow in 1998, and serves as the chair of ACM SIGSOFT. His research interests are in software engineering in general and in software evolution in particular. Dr. Notkin is a senior member of the IEEE.



Kevin Sullivan received the dual BA degree in mathematics and in computer science from Tufts University in 1987, the MS in computer science, and the PhD in computer science and engineering from the University of Washington in 1990 and 1994, respectively. Since then, he has been assistant professor in the Department of Computer Science at the University of Virginia. His primary research area is software engineering. His overall focus is on modular software design. He currently has projects in component-based software, software engineering economics, infrastructure survivability, and software evolution. He is a senior member of the IEEE.