Software Reflexion Models: Bridging the Gap between Source and High-Level Models*

Gail C. Murphy and David Notkin

Dept. of Computer Science & Engineering
University of Washington
Box 352350
Seattle WA, USA 98195-2350
{gmurphy, notkin}@cs.washington.edu

Kevin Sullivan

Dept. of Computer Science
University of Virginia
Charlottesville VA, USA 22903
sullivan@cs.virginia.edu

Abstract

Software engineers often use high-level models (for instance, box and arrow sketches) to reason and communicate about an existing software system. One problem with high-level models is that they are almost always inaccurate with respect to the system's source code. We have developed an approach that helps an engineer use a high-level model of the structure of an existing software system as a lens through which to see a model of that system's source code. In particular, an engineer defines a high-level model and specifies how the model maps to the source. A tool then computes a software reflexion model that shows where the engineer's high-level model agrees with and where it differs from a model of the source.

The paper provides a formal characterization of reflexion models, discusses practical aspects of the approach, and relates experiences of applying the approach and tools to a number of different systems. The illustrative example used in the paper describes the application of reflexion models to NetBSD, an implementation of Unix comprised of 250,000 lines of C code. In only a few hours, an engineer computed several reflexion models that provided him with a useful, global overview of the structure of the NetBSD virtual memory subsystem. The approach has also been applied to aid in the understanding and experimental reengineering of the Microsoft Excel spreadsheet product.

*This research was funded in part by the NSF grant CCR-8858804 and a Canadian NSERC post-graduate scholarship.

SIGSOFT '95 Washington, D.C., USA

1 Introduction

Software engineers often think about an existing software system in terms of high-level models. Box and arrow sketches of a system, for instance, are often found on engineers' whiteboards. Although these 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.

Current reverse engineering systems derive high-level models 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. [WTMS95].

We have developed an approach, illustrated in Figure 1, that enables an engineer to produce sufficiently accurate high-level models in a different way. The engineer defines a high-level model of interest, extracts a source model (such as a call graph or an inheritance hierarchy) from the source code, and defines a declarative 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 the source model.[1] An engineer interprets the reflexion model and, as necessary, modifies the input to iteratively compute additional reflexion models.

[1]The old English spelling differentiates our use of "reflexion" from the field of reflective computing [Smi84].

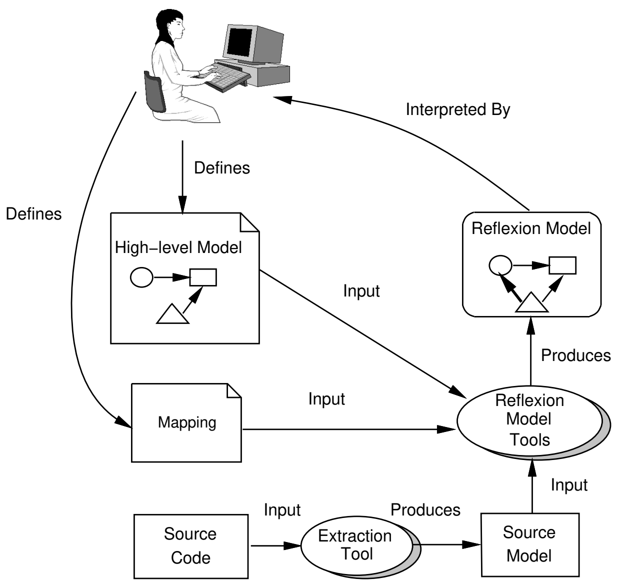


Figure 1: The Reflexion Model Approach

In essence, a reflexion model summarizes a source model of a software system from the viewpoint of a particular high-level model. This form of summarization is useful to engineers performing a variety of software engineering tasks. In one case, an engineer at Microsoft Corporation, prior to performing a reengineering task, applied reflexion models to help understand the structure of the Excel spreadsheet product. In another case, computing a sequence of reflexion models highlighted several places where calls between modules violated the layers of the software architecture that an engineer had perceived to exist.

We have developed several techniques for simplifying the engineer's task of defining high-level models and mappings. We have also developed tools that compute reflexion models for large systems in a minute or two. As described in Section 2, these techniques and tools have made it possible for an engineer, in a few hours, to better understand a system of several hundreds of thousands of lines of code.

To clarify the meaning and computation of reflexion models, we present a formal characterization of a reflexion model system in Section 3. The flexibility of our approach is demonstrated in Section 4 through descriptions of the use of reflexion models in a variety of settings. Section 5 considers the theoretical and practical performance aspect of the approach and our tools. In Section 6, we discuss key aspects of the approach that provide software engineers with the needed flexibility. 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 reflexion models to familiarize himself with an unfamiliar implementation, NetBSD. The system is composed of about 250,000 lines of C [KR78] source code spread over approximately 1900 source files.

The developer first specified a model he believed, based on his experience, to be characteristic of Unix virtual memory systems. This model consists of a modularization of a virtual memory implementation and the calls between those modules (Figure 2a).

A calls relation between NetBSD functions was then computed using the cross-reference database tool (xrefdb) of Field [Rei95], and a small awk [AKW79] script was written to translate the output of Field into our input format. The extracted calls relation consisted of over 15,000 tuples over 3,000 source entities.[2]

Next, the developer defined the following mapping:

```
[ 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

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

model is mitigated in three ways. First, the engineer only named entities in subsystems of interest. For example, the mapping above does not consider entities from the I/O subsystem. Second, the physical (e.g., directory and file) and logical (e.g., functions and classes) structure of the source are used to name many source model entities in a single line of the mapping. In this case, only physical structure is used since C provides little logical structure. Finally, regular expressions are 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, a reflexion model was computed to compare the source and high-level models (Figure 2b). The solid lines show the *convergences*, where the source model agrees with the high-level model. For instance, as the developer expected, there were calls found in the source between functions in modules implementing `VMPolicy` and functions in modules implementing `Pager`. The dashed arrows show the *divergences*, where the source model includes arcs not predicted by the high-level model. For instance, the dashed line from `FileSystem` to `Pager` indicates that functions within modules mapped to `FileSystem` make calls to functions within modules mapped to `Pager`. The dotted lines show the *absences*, where the source model does not include arcs predicted by the high-level model. For instance, no calls were found between modules mapped to `Pager` and modules mapped to `FileSystem`. The number associated with each arc in the figure is the number of source model relation values mapped to the convergence or divergence; absence arcs are annotated with the value zero.

Computation and display of the reflexion model shown in Figure 2 takes twenty seconds on a DEC 3000/300 with our tools, which consist of several small C++ [Str86] programs and a user interface implemented in TCL/TK [Ous94]. Computed reflexion models are displayed using AT&T's graphviz package. The tools allow the engineer to select arcs in a reflexion model, producing displays of the associated source model tuples. The screen snapshot in Figure 3 includes a window "Arc Values" that shows the result of selecting the divergence between `FileSystem` and `Pager`. Values in this window show the calling and called functions, including their directory and file information.

In a one hour session, the VM developer was able to iteratively specify, compute and interpret several reflexion models. (The source model was extracted beforehand, and it was not changed during this session.) 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.

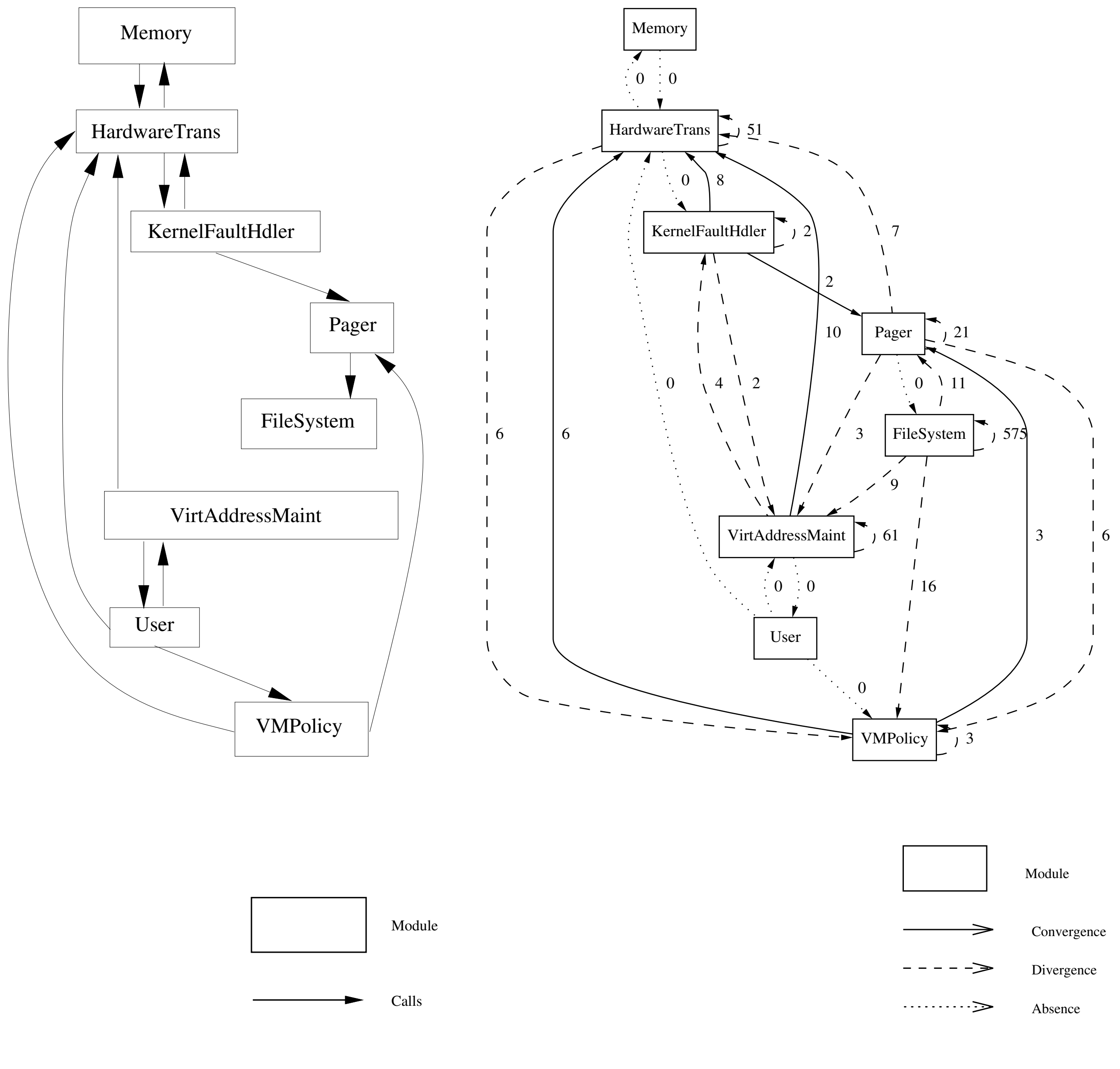
3 Formal Characterization

To make precise the meaning and computation of a reflexion model, we present a formal specification of a reflexion model system using the Z specification language [Spi92].

3.1 Reflexion Model

A static schema in Z describes a state the system can occupy 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 presented below.

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 *HLMTuple* and *SMTuple*, which define the types of tuples in the *HLMRelation* and *SMRelation*.



(a) High–level Model

(b) Reflexion Model

Figure 2: High-level and Reflexion Models for the NetBSD Virtual Memory Subsystem

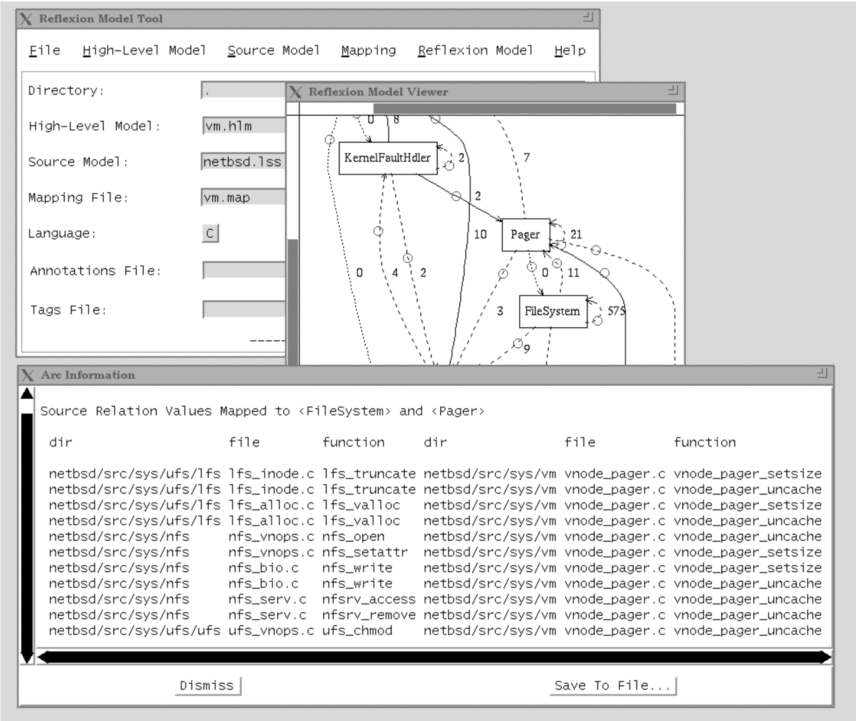


Figure 3: The Reflexion Model Tool User Interface

$[HLMENTITY, SMENTITY]$

$$
\begin{aligned}
&HLMRelation == HLMENTITY \leftrightarrow HLMENTITY\\
&SMRelation == SMENTITY \leftrightarrow SMENTITY\\
&HLMTuple == HLMENTITY \times HLMENTITY\\
&SMTuple == SMENTITY \times SMENTITY
\end{aligned}
$$

ReflexionModel

$$
\begin{aligned}
&convergences : HLMRelation\\
&divergences : HLMRelation\\
&absences : HLMRelation\\
&mappedSourceModel : HLMTuple \leftrightarrow SMTuple\\
\hline
&convergences \cap divergences = \varnothing\\
&divergences \cap absences = \varnothing\\
&convergences \cap absences = \varnothing\\
&\operatorname{dom} mappedSourceModel =\\
&\qquad convergences \cup divergences
\end{aligned}
$$

The variables (above the dividing line) in the schema define the information that must be maintained by a reflexion model system. The relation described by the *convergences* variable defines where a high-level model agrees with a given source model; the relation described by the *divergences* variable describes where a source model differs from a high-level model; and the relation described by the *absences* variable defines where a high-level model differs from a source model. The fourth variable, *mappedSourceModel*, defines a relation that describes which source model values contribute to a convergent or divergent arc in the reflexion model. The information in *mappedSourceModel* is used to support 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 Figure 3.

In addition to declaring the state components of a reflexion model, the static schema includes the definition of constraints (below the dividing line) that must be satisfied by all reflexion model systems. The first three constraints state that the values of the *convergences*, *divergences* and *absences* relations are disjoint. The fourth constraint states that the investigation of the source model values contributing to a reflexion model arc is only meaningful for values in the *convergences* and *divergences* relations; *absences* have no contributing source model values.

3.2 Computing a Reflexion Model

The dynamic schema, *ComputeReflexionModel*, presented below, 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. The high-level model (*hlm*?) is described as a relation over high-level model entities, and the source model (*sm*?) is described as a relation over source model entities. The mapping (*map*?) is an ordered list of map entries. Each map entry—defined by the type *MapEntry*—names zero or more source model entities and associates with these entities, one or more high-level model entities.[3] *SMENTITYDESC* represents the type of a description naming zero or more source model entities (e.g., a regular expression over logical and physical software structure in our tools).

$[\mathit{SMENTITYDESC}]$

$$\begin{aligned}&\mathit{MapEntry} ==\\ &\quad \mathit{SMENTITYDESC} \times (\mathbb{P}\,\mathit{HLMENTITY})\end{aligned}$$

ComputeReflexionModel

$$\begin{aligned}
&\Delta \mathit{ReflexionModel}\\
&\mathit{hlm}? : \mathit{HLMRelation}\\
&\mathit{sm}? : \mathit{SMRelation}\\
&\mathit{map}? : \mathrm{seq}\,\mathit{MapEntry}\\
&\mathit{mapFunc} : (\mathrm{seq}\,\mathit{MapEntry} \times \mathit{SMENTITY}) \rightarrow\\
&\qquad\qquad (\mathbb{P}\,\mathit{HLMENTITY})\\
\hline
&\mathit{mappedSourceModel}' =\\
&\quad \bigcup \{\, t : \mathit{SMTuple} \mid t \in \mathit{sm}? \bullet\\
&\qquad (\, \mathit{mapFunc}\,(\, \mathit{map}?, \mathit{first}\; t\,) \times\\
&\qquad\; \mathit{mapFunc}\,(\, \mathit{map}?, \mathit{second}\; t\,)\,) \times\\
&\qquad \{\, t \}\,\}\\
&\mathit{convergences}' =\\
&\quad \mathit{hlm}? \cap (\mathrm{dom}\; \mathit{mappedSourceModel}')\\
&\mathit{divergences}' =\\
&\quad (\mathrm{dom}\; \mathit{mappedSourceModel}') \setminus \mathit{hlm}?\\
&\mathit{absences}' = \mathit{hlm}? \setminus (\mathrm{dom}\; \mathit{mappedSourceModel}')
\end{aligned}$$

[3] An additional invariant which constrains a map entry to name only high-level model entities specified in *hlm*? has been elided for presentation purposes.

Computing a reflexion model also requires a function (*mapFunc*) that matches entities from the source model to the specified map, producing a set of associated high-level model entities.

The value of *mappedSourceModel* is computed by pushing the elements of each source model tuple through the map, resulting in two sets of high-level model entities. The cross-product of these sets is taken and each element in the resultant set is associated (through another cross-product) with the original source model tuple. Once the *mappedSourceModel* is computed, the values of the *convergences*, *divergences*, and *absences* relations are easily determined through set intersection and set difference operations.

3.3 A Family of Reflexion Models

This Z 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*). Our reflexion model system describes source model entities using a combination of structural information and regular expressions. Our tools support the use of two different mapping functions. The most common mapping function used produces the set of high-level model entities associated with the first match of a given source model entity to an entry in the map. Alternatively, our tools support the use of a mapping function that returns the set of high-level model entities resulting from the union of all matches found in the map. Our tools can thus be configured to provide implementations of two different reflexion model systems.

4 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.

Reengineering A software engineer at Microsoft Corporation applied reflexion models to assess the structure of the Excel spreadsheet product prior to a reengineering activity. The Excel product consists of over one million lines of C source code. 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. A detailed mapping file consisting of almost 1000 entries was produced and is being used to guide the extraction of components from the source. The engineer found the approach valuable for understanding the structure of Excel and planning the reengineering effort.

Design Conformance We used a sequence of reflexion models to compare the layered architectural design of Griswold's program restructuring tool [GN95] with a source model consisting of calls between modules. The reflexion model highlighted, as divergences, a few cases where modules in the source code did not adhere to the layering principles. We are unaware of any other approach that would allow such violations to be found so directly.

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 [Boo91]) prepared prior to implementation. This case was unique in that the reflexion model was fully convergent with the source model.

System Understanding We used reflexion models to try to determine why a compiler used in undergraduate education at the University of Washington was difficult for the students to change. We computed a reflexion model comparing an extracted Ada file imports relation with a conventional model of a compiler [PW92]. The reflexion model contained meaningful divergences between almost all pairs of high-level entities. This high degree of coupling explains, in part, why the students had difficulty changing the system.

We later applied reflexion models to a newer version of the compiler, written in C++. The reflexion models for this compiler were far less cluttered than the Ada version. However, some unexpected interactions were identified using the reflexion model; these may provide the basis for either minor restructuring of the compiler or at least additional warnings to the students.

5 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 the speed of the computation.

The formal characterization presented in Section 3 provides a basis for considering the theoretical complexity of computing a reflexion model. From the dynamic Z schema, *ComputeReflexionModel*, we can see that the time complexity of computing a reflexion model is dependent upon the cost of computing the *mappedSourceModel* relation and the cost of comparing that computed relation to the high-level model. An upper bound, then, on the complexity of computing the *mappedSourceModel* relation is given by:

$$O(\#sm \times \#map \times t_{comparison}) \times O((\#hlm)^2)$$

where $\#sm$ is the cardinality of the source model relation, $\#map$ 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, and $\#hlm$ is the cardinality of the high-level model relation. Since the number of entities in the high-level model is generally small and constant, the $O((\#hlm)^2)$ can, in practice, be ignored, yielding:

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

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 tuples) 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 40 minutes on a Pentium to compute a reflexion model for Excel.

By trading space for time in the implementation of our tools, we have been able to support the computation of reflexion models for large software systems and large maps in a minute or two (see Table 1). Specifically, our tools hash the match of high-level model entities for a given source model entity the first time a source model entity is seen. The additional space requirements depend upon the naming scheme used for source model entities and the number of unique entities in the source model. In the case of Excel and the naming scheme used by our tools, there are 18,118 unique source model entities each requiring on the order of 100 bytes (less than 2 Mb in total). Our tools provide a variety of options to let the engineer determine the appropriate space time tradeoff.

6 Discussion

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 (restructuring, reengineering, or porting, etc.). 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.

Syntactic Models As described in the formal characterization, reflexion models are computed without any knowledge of the intended semantics of the high-level or the source model. A benefit of this syntactic approach is the ability of an engineer to use reflexion models to investigate many different kinds of structural interactions in a software system (calls, data dependences, or event interactions, etc.). It also means, however, that it is the engineer's responsibility to ensure that it makes sense to compare a selected high-level model with an extracted source model. For example, comparing the specified calls diagram of Figure 2a with an extracted calls relation makes sense, but comparing the same high-level model with a source model representing the `#include` structure of NetBSD would probably be meaningless.

In practice, engineers have exploited this flexibility by changing the meaning of their models over time. For example, 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.

To aid the engineer in interpreting reflexion models computed with models containing different kinds of information, we are adding support for typing relations in both source and high-level models.

Maps The declarative maps used in computing a reflexion model enable an engineer to focus on information of interest in the source in two ways. First, an engineer may specify a partial map that contains entries for only those parts of the system relevant to the task at hand. Second, an engineer may iteratively refine a map to the appropriate level of detail necessary for the task being performed.

Generally, an initial reflexion model is computed with a rough and partial map. Then, based on an investigation of the reflexion model, an engineer refines the map in the areas of interest until the necessary information about the system is obtained. Sometimes, as was the case when we applied reflexion models to assess the structure of the compiler implemented in Ada, a

System	Language	Approx. Lines of Code	Source Model (Tuples)	Mapping (Lines)	High-Level Model (Entities)	SPARC 20/50 (min:sec)	486 PC 100 MHz (min:sec)
UW compiler	C++	3,700	607	33	5	:00.5	:01
UW compiler	Ada	4,200	72	9	5	:00.2	:01
Restructuring Tool	CLOS	47,000	5,855	215	9	:04.3	:06
NetBSD (VM)	C	250,000	15,657	7	8	:04.0	:19
Excel	C	1,200,000	119,637	971	15	2:13.0	4:05

Table 1: Performance of Reflexion Model Tools

fairly rough map was sufficient. In contrast, in the case of Excel, a detailed map was desired to plan reengineering activities.

The reflexion model approach enables an engineer to balance the cost of refining the map with the level of detail necessary for performing a particular software engineering task. We plan to track the use of some declarative maps across the evolution of several systems to determine the degree of sensitivity of our mapping language to changes made in the source. This will aid an engineer in judging the amortization costs of creating detailed maps for large systems.

Querying Reflexion Models Reflexion models bridge the gap between an engineer's high-level model and a model of the source. The convergences, 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. In the second kind, an engineer performs queries to determine the source model entities and values that were not included in the reflexion model. This query enables an engineer to assess whether the map is sufficiently complete and to investigate whether absences in the computed reflexion model are the result of incompleteness in the map.

Based on the results of the interpretation, 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. To better support an engineer in the interpretation process, we are currently developing techniques to improve the querying and investigation of a series of computed reflexion models.

7 Related Work

Reverse Engineering The reverse engineering approaches closest to ours use clustering information, which is generally culled from a combination of human input and numerical computation, to create abstract representations for the engineer. Examples of this approach include Rigi [MK89] and Schwanke's statistically-based architectural recovery technique [Sch91].

Reflexion models differ in a number of ways. First, in our approach the engineer specifies the high-level entities explicitly, whereas the architectural recovery systems instead infer them. Second, we focus on comparing high-level and source models, rather than on discovering high-level models. Third, our mappings are declarative, associating source and high-level entities, in contrast to approaches such as Rigi, which uses operational mappings.

Rigi differs from our approach by enabling an engineer to build explicit hierarchical models of the software structure of a system. We instead support the investigation of substructure (i.e., a subsystem) by computing different reflexion models at various levels of abstraction.

Model Comparison Ossher has considered the comparison of relational models of software structure for a fixed type of high-level model, a GRID [Oss84]. The intent of a GRID description of a system is to "specify, represent, document and enforce the structure of large, layered systems" [Oss87, pg. 219]. The GRID mechanism permits an engineer to choose a concise description of the software structure and then to annotate how the actual structure of the system deviates from the description. Our approach is not intended to specify or enforce a desired structure of the system, but rather to compare two models of the structure using an explicit mapping between the two models provided by the engineer.

Jackson's Aspect system [Jac93] compares partial program specifications (high-level models) to data flow models extracted from the source to detect bugs in the source code that cannot be detected using static type checking. Aspect uses dependences between data stores as a model of the behavior of a system and assumes that this model is correct, focusing on how the source model differs from the posited behavior. In contrast, we focus on high-level structural models, and we are interested in both how the source model differs from the high-level model and also how the high-level model differs from the source.

Jackson has also developed a semantic difference approach for comparing the differences in input and output behavior between two versions of a procedure [JL94]. This approach derives an approximate model of the semantic effect of a procedure consisting of a binary relation that summarizes the dependence of variables after executing the procedure upon the value of variables at the entry to the procedure. The semantic differencing tool compares the binary relations resulting from different versions of a procedure. We focus on the comparison of relations at different levels of abstraction through an explicit and declarative mapping.

8 Summary

A reflexion model summarizes information extracted from source code into a high-level model that is sufficiently accurate to support an engineer in performing a software engineering task. The engineer defines three inputs to a reflexion model computation: a high-level model, a source model, and a map. The reflexion model presents the summary information in the context of the high-level model defined by the engineer. The engineer interprets and iteratively computes successive reflexion models until satisfied. The feasibility and flexibility of the approach have been demonstrated through its application in a number of different settings on systems ranging from several thousand to over one million lines of code.

Acknowledgments

Robert Allen, Kingsum Chow, David Garlan, Bill Griswold, Michael Jackson, Kurt Partridge, Bob Schwanke, and Michael VanHilst each provided helpful comments on earlier drafts of the paper. 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. We also thank the anonymous referees for their constructive comments.

References

[AKW79] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. Awk – A Pattern Scanning and Processing Language. *Software – Practice and Experience*, 9(4):267–280, 1979.

[Boo91] G. Booch. *Object-oriented Design with Applications*. Benjamin-Cummings, 1991.

[GN95] W.G. Griswold and D. Notkin. Architectural Tradeoffs for a Meaning-Preserving Program Restructuring Tool. *IEEE Transactions on Software Engineering*, 21(4):275–287, April 1995.

[Jac93] D. Jackson. Abstract Analysis with Aspect. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis*, pages 19–27, 1993.

[JL94] D. Jackson and D.A. Ladd. Semantic Diff: A Tool For Summarizing the Effects of Modifications. In *Proceedings of the International Conference on Software Maintenance*, September 1994.

[KR78] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[MK89] H.A. Müller and K. Klashinsky. A System for Programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86. IEEE Computer Society Press, April 1989.

[Oss84] H.L. Ossher. *A New Program Structuring Mechanism Based on Layered Graphs*. PhD thesis, Stanford University, December 1984.

[Oss87] H. Ossher. A Mechanism for Specifying the Structure of Large, Layered Systems. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 219–252. MIT Press, 1987.

[Ous94] J.K. Ousterhout. *TCL & the TK Toolkit*. Addison-Wesley, 1994.

[PW92] D.E. Perry and A. Wolf. Foundations for the Study of Software Architecture. *ACM Software Engineering Notes*, 17(4):40–52, October 1992.

[Rei95] S.P. Reiss. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers, 1995.

[Sch91] R. Schwanke. An Intelligent Tool for Reengineering Software Modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.

[Smi84] B.C. Smith. Reflection and Semantics in LISP. In *Proceedings of the 1984 ACM Principles of Programming Languages Conference*, pages 23–35. ACM, December 1984.

[Spi92] J.M. Spivey. *The Z Notation*. Prentice Hall, second edition edition, 1992.

[Str86] B. Stroustrup. *C++ Programming Language*. Addison-Wesley, 1986.

[WTMS95] K. Wong, S.R. Tilley, H.A. Müller, and M.D. Storey. Structural Redocumentation: A Case Study. *IEEE Software*, 12(1):46–54, January 1995.