Extending and Managing Software Reflexion Models

Gail C. Murphy
Dept. of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver B.C. Canada V6T 1Z4
1.604.822.5169
murphy@cs.ubc.ca

David Notkin
Dept. of Comp. Science & Eng.
University of Washington
Box 352350
Seattle WA 98195-2350 USA
1.206.685.3798
notkin@cs.washington.edu

Kevin Sullivan
Dept. of Computer Science
University of Virginia
Charlottesville, VA
22903-2442 USA
1.804.982.2206
sullivan@cs.virginia.edu

University of British Columbia, Department of Computer Science, Technical Report TR-97-15

ABSTRACT
The artifacts comprising a software system often “drift” apart over time. Design documents and source code are a good example. The software reflexion model technique was developed to help engineers exploit—rather than remove—this drift to help them perform various software engineering tasks. 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 use of the technique to support a variety of tasks—including the successful use of the technique to support an experimental reengineering of a system comprising a million lines-of-code—identified a number of shortcomings. In this paper, we present two categories of extensions to the technique. The first category concerns the typing of software reflexion models to allow different kinds of interactions to be distinguished. The second category concerns techniques to ease the investigation of reflexion models. These extensions are aimed at making the engineer more effective in performing various tasks by improving the management and understanding of the inconsistencies—the drift—between artifacts.

Keywords
Software structure, design conformance, source analysis, program understanding, inconsistency management

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 of this. 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 and difficult to maintain logical but implicit relationships among documents.

The software reflexion model technique was developed to help people perform various software engineering tasks by exploiting—rather than removing—this drift [Mur96][MNS95]. 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 helps an engineer detect, assess, and manage inconsistencies from a task-specific viewpoint.

The initial uses of reflexion models included design conformance tasks, assessing software structure prior to an implementation change [Mur96], and an experimental reengineering of the Excel code base by a Microsoft engineer [MN97]. The users for these tasks identified a set of shortcomings of the basic tools and techniques. In this paper, we present a set of improvements to the reflexion model technique, each of which is aimed at making the engineer more effective in performing various tasks by improving the management and understanding of the inconsistencies, the drift, between artifacts.

The extensions fall into two basic categories. The first category concerns the *typing* of software reflexion models to allow different kinds of interactions to be distinguished (such as separating calls from references to global variables). Section 3 addresses this category, considering typed source models (information directly or indirectly extracted from the software artifacts) as well as typed high-level models (task-specific descriptions of the software). The second category concerns techniques that allow the engineer to better manage the investigation and iteration of reflexion models. Section 4 addresses this category, considering *tagging*, which allows an engineer to eliminate information from a reflexion model until and unless it changes in subsequent iterations, and *annotations*, which helps the engineer track which elements in a reflexion model have been explored.

Section 5 presents a discussion of the design rationale behind these extensions, while Section 6 details some related work, focusing on other approaches that handle source code, one of the major sources of inconsistency in software systems.

2 THE SOFTWARE REFLEXION MODEL TECHNIQUE

To illustrate the key features of the software reflexion model technique, we present a sketch of its use to aid an engineer with a representative change task. The engineer's task is to assess the feasibility of reusing the source for the back-end of the GNU `gcc` compiler with an existing graphical front-end development environment.

The engineer has available the source for the system and a set of textual documentation files describing the design of the system. However, since the `gcc` system is comprised of over 210,000 lines of source code,[1] it is difficult for the engineer to gain an understanding of the system's structure directly from the source code. Although the design documentation provides a high-level view of the system, the engineer is quite reasonably unsure of whether the documentation adequately represents the implementation from which the back-end components must be extracted.

To apply the software reflexion model technique to understand the inconsistencies between the design and the implementation that are relevant to the task, the engineer iteratively applies five basic steps.

First, the engineer selects a high-level structural model suitable for reasoning about the task. For the assessment task on the `gcc` system, the engineer selects the model shown in Figure 1a that consists of modules representing the stages of the compiling process and data interactions between those modules.

Next, the engineer extracts structural information—called a *source model*—from the artifacts of the system. A source model may be produced either by statically analyzing the system's source or by collecting information during the system's execution (Section 5). In this example, the engineer uses the C Information Abstractor tool from AT&T [CNR90] to extract a source model from the `gcc` source code comprising about 32,098 calls between functions and references of functions to global variables. Several extracted tuples are shown in Figure 2.

In the third step, the engineer describes a mapping between the extracted source model and the stated high-level structural model. One of the 39 map entries created for `gcc`—based on the design documentation, which includes a description of the source files implementing various features—is:

```
[ file=^loop\.c mapTo=Optimization ]
```

This entry associates all of the functions and global variables in the file `loop.c` with the `Optimization` module. In addition to referring to files, the engineer may also may map entities in the source model by referring to the names of directories, functions, or variables. The engineer may easily tailor the mapping language to match the implementation environment of the system under study. For instance, the mapping language may be tailored to refer to classes and objects if working with a system implemented in an object-oriented language.

---

[1] Version 2.7.2 of `gcc` was used in this study.

Given these three inputs, the engineer, 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 induced arcs are then compared with the interactions stated in the high-level model to produce a reflexion model.

Figure 1b shows the reflexion model computed for `gcc`. The solid lines in the reflexion model, called *convergences*, indicate source interactions that were expected by the engineer. Dashed lines, called *divergences*, indicate source interactions that were not expected by the engineer. Dotted lines, called *absences*, indicate interactions that were expected but not found. The number attached to each arc indicates the number of calls and global variable references in the source associated with the interaction. The software reflexion model in Figure 1b summarizes 10,583 calls and variable references extracted from the `gcc` source. It takes less than 10 seconds to compute this reflexion model on a 150 MHz Pentium.

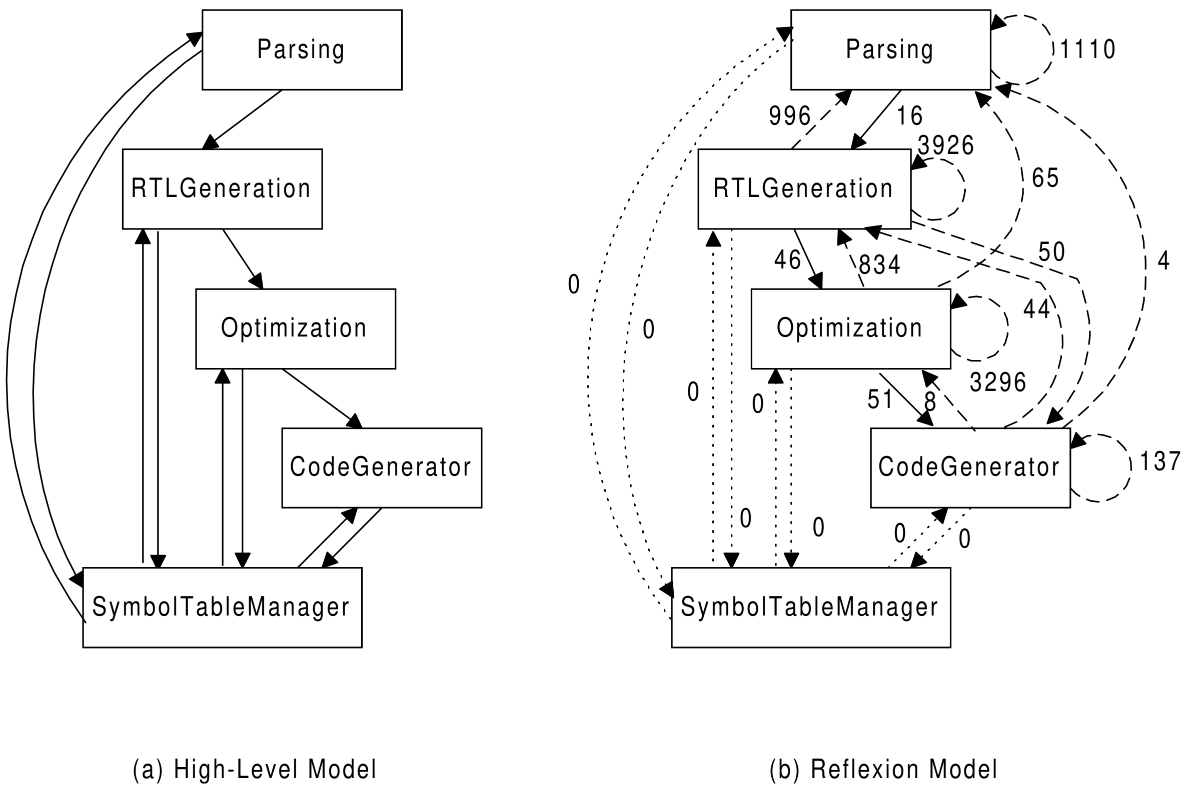

Figure 1. A High-level Model and a Software Reflexion Model for `gcc`.

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, an engineer may investigate the software reflexion model for `gcc` by viewing all calls associated with the arc from `CodeGenerator` to `Parsing`. 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 may iteratively recompute a reflexion model. The last call shown in Figure 2 from `flow_analysis` to `oballoc`, for instance, may cause the engineer to refine the high-level model to include a `Utils` entity representing utility functions and to refine the map to associate some functions in tree.c, such as `oballoc`, with the `Utils` entity.

```
simplify_shift_const       in combine.c    calls  mode_for_size  in  stor_layout.c
function_cannot_inline_p   in integrate.c  calls  tree_last      in  tree.c
flow_analysis              in flow.c       calls  oballoc        in  tree.c
```

Figure 2. A Sample of Source Model Values

This process is repeated as needed until the engineer has sufficient information to perform the desired task. In the experimental reengineering of Excel, the Microsoft engineer iterated the reflexion model over more than four weeks, changing the high-level model slightly and expanding the map to over 1600 entries. A significant amount of uninvestigated inconsistency remained in the reflexion model after this period, almost exclusively in the portions of the system that were immaterial to the engineer's task (which was to identify and extract a component from the system).

By permitting inconsistencies to remain, the reflexion model technique allows engineers to use their time effectively, focusing only on information of interest. In addition to permitting iteration appropriate to the task, initial software reflexion models are quite simple and fast to produce. Often, an engineer has been able to specify the inputs and compute one or more software reflexion models in about an hour. The inconsistencies that appear in the initial reflexion model help guide the remainder of the process.

3 TYPED SOFTWARE REFLEXION MODELS

By computing and investigating a series of reflexion models, a software engineer gains an understanding of the relationships between the source and high-level models. An engineer applies this iterative process until a suitable level of understanding has been gained for the task at hand. The comparison information—the convergences, divergences, and absences—embodied in a reflexion model can help an engineer interpret the correspondence between the two models. For example, an absence in a reflexion model can indicate to the engineer parts of the source that are not currently being included in the computation. Parts of the source may be excluded from a computation because they are immaterial or because a mapping entry was omitted; the engineer, not the tool, is best suited to make this distinction.

In any case, the comparison information eases the investigation of a reflexion model because it makes certain parts of the model correspondence directly visible. To interpret much of the correspondence, however, the engineer must still view the source model values contributing to a reflexion model convergence or divergence. Often, an engineer is more interested in the kinds of source model values contributing to an arc than the details of the particular values. In the case of `gcc`, for instance, an engineer may be more interested in convergences or divergences arising from data interactions than from call interactions because the data interactions may be more difficult to reengineer.

To further aid a software engineer in this kind of investigation and interpretation, we have extended the technique to support the computation of *typed* software reflexion models. In contrast to an untyped reflexion model, as described above in Section 2, where all source model entries are commingled during the computation, in a typed software reflexion model, the distinction between various kinds of relations in a source model is maintained. In the `gcc` example, for instance, if different estimating functions were used for call interactions versus data interactions, the engineer might want to maintain a separation between the two relations. To allow the separation of relations, when desired by the engineer, we have extended the software reflexion model with types. An engineer may choose to describe types in only the source model (Section 3.1) or in both the source and the high-level models (Section 3.2).

3.1 Typed Source Model

In a typed source model, each interaction between source model entities has an associated type. The type indicates the relation to which an interaction belongs. A typed source model for `gcc`, for instance, might associate a "calls" type with each of the function-to-function call interactions, and might associate a "refers-to" type with each of the references from functions to global data variables.[2]

---

[2] To be precise, the relations comprising a source model are bags that include a count of the number of times each tuple appears in the relation. Since the distinction between bags and relations is not critical in this discussion, the term relation is used for ease of presentation.

A typed software reflexion model may be computed from a typed source model and an untyped high-level model. In this case, the types associated with values in the source model induce types on the arcs of the reflexion model. The arcs in the reflexion model computed by pushing source model values through the map are assigned the same type as that associated with the source model value contributing to the arc. Each computed arc is then compared with the high-level model. If the arc matches an interaction in the high-level model, a convergence is created of the type of the computed arc. If the arc does not match, a divergence is created of the type of the computed arc. More than one type of arc between two high-level model entities may result in the reflexion model. Interactions remaining unmatched in the high-level model after the comparison with all the computed arcs are absences; naturally, absences remain untyped.

Figure 3b shows a typed reflexion model for the estimation task on `gcc`. This reflexion model was computed from the typed source model used above and the untyped high-level model presented in Figure 3a. The high-level model incorporates the `Utils` entity identified in the reflexion model described in Section 2. The `SymbolTableManager` has been removed from the reflexion model because the engineer determined the entity was not pertinent to the retargeting task. Different levels of grey are used to represent the type of an arc. Grey lines represent arcs of the type “refers-to”, and black lines represent the “calls” type. As before, the line attributes, solid, dashed, and dotted, represent convergences, divergences, and absences, respectively. Thus, the grey dashed line—a “refers-to” divergence—from the `Optimization` module to the `Parsing` module represents unexpected references from functions mapped to the `Optimization` module to global data defined in the `Parsing` module. As another example, the black solid line—a “calls” convergence—from the `Parsing` module to the `RTLGeneration` module indicates expected call interactions between the two modules.

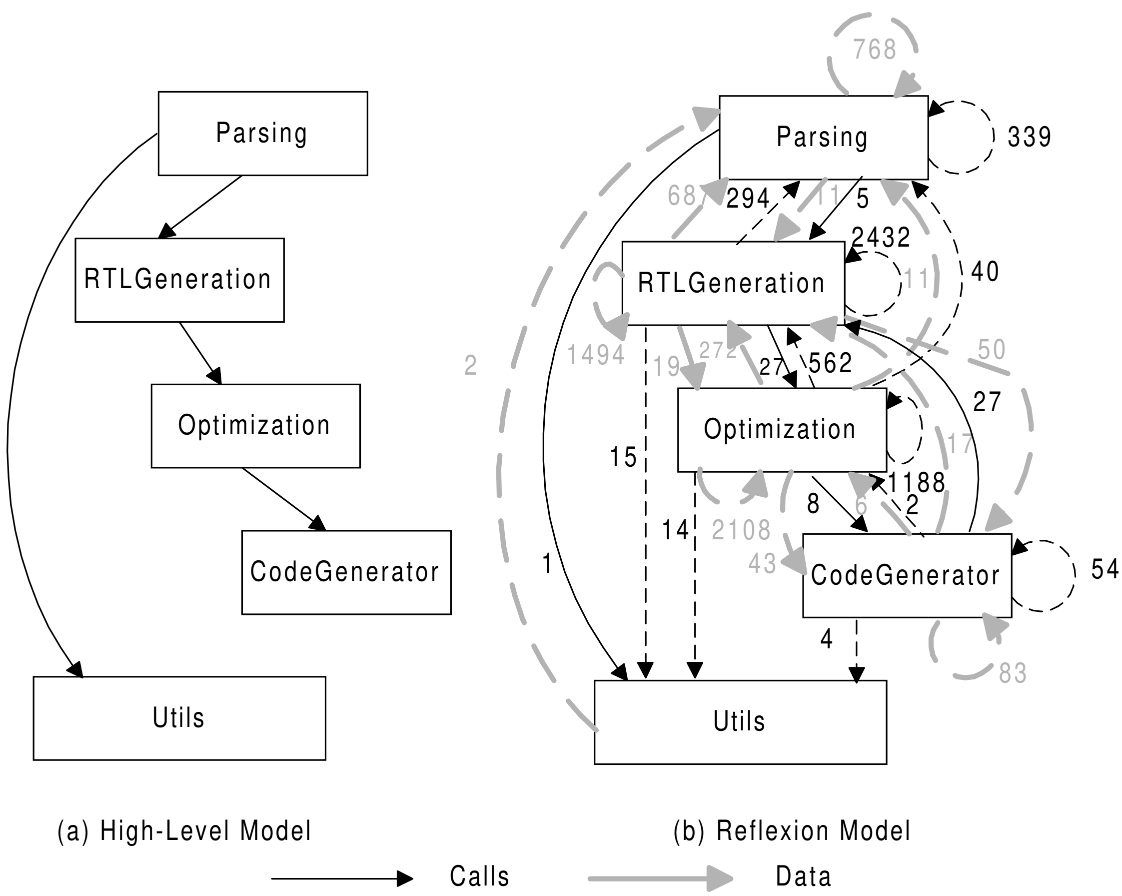

Figure 3. Comparing a Typed Source Model and an Untyped High-level Model

Separating the different kinds of source model relations with types and then computing reflexion models using the typed information can provide useful information to the engineer viewing the reflexion model. For instance, in Figure 3b, two divergences appear between the `Optimization` and `RTLGeneration` modules: one representing source model values of type “call”; and the other representing source model values of type “refers-to”. In comparison, only one divergence appears when the source model is untyped (Figure 1b). From the large number of occurrences of both types of divergences between entities in the reflexion model of Figure 1b, the engineer can quickly and easily determine that many interactions of both source model types are present.

3.2 Typed High-level Model

In a typed high-level model, an interaction between high-level model entities may have an associated type. Similar to a typed source model, the type of a high-level model interaction indicates the relation to which the interaction belongs. In contrast to a typed source model in which all source model values must be typed, an interaction in a typed high-level model may (under most conditions) remain untyped.

A fully-typed high-level model for `gcc`, in which each interaction has an associated type, is shown in Figure 4a. In this high-level model, the engineer anticipates that all interactions are of the “calls” type. Figure 4b shows a reflexion model computed with this high-level model. The reflexion model computation considers the types of the high-level model when performing the comparison between the mapped source model and the high-level model.

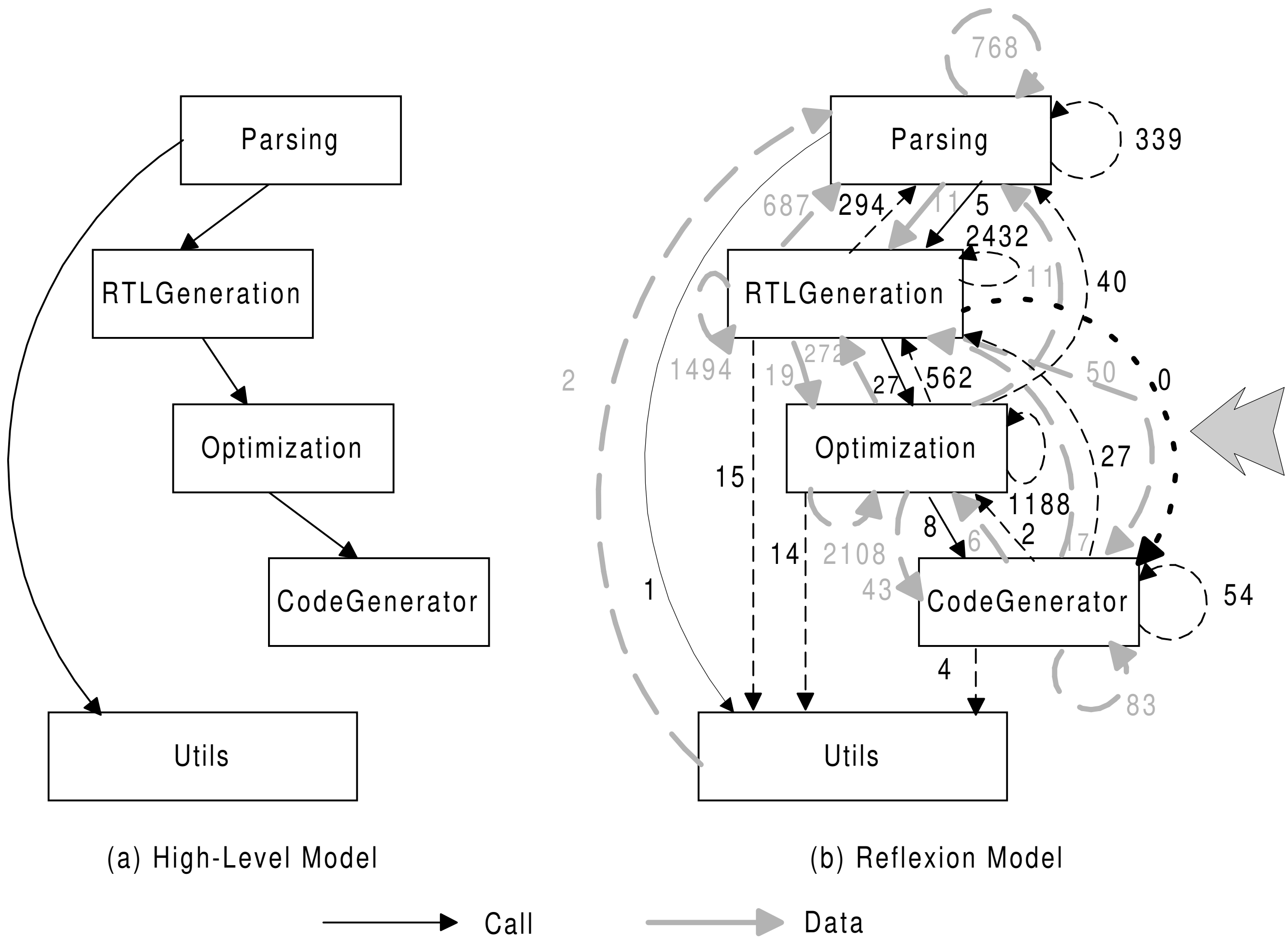

Figure 4. Comparing a Typed Source Model and Typed High-level Model

Compared to the reflexion model computed using the typed source model and untyped high-level model in Figure 3, typing arcs in the high-level model leads to an absent arc of type "calls" from `RTLGeneration` to `CodeGenerator`. This arc is highlighted (by the big grey arrow) in Figure 4b. Thus, as was the case with typing the source model, typing the high-level model can provide more information to the engineer at the level of the reflexion model without requiring a detailed investigation of the source model values associated with a given reflexion model arc.

A high-level model is considered partially-typed if some interactions do not have associated types. A partially-typed high-level model must satisfy the constraint that an interaction may be untyped only if it is between two high-level model entities for which no defined typed interactions are specified. This form of computation is useful when an engineer does not require equal levels of detail in all parts of the reflexion model. Partially-typed high-level models support the goal of a lightweight technique by reducing the burden on the engineer to define a type for each high-level model interaction; an engineer may focus on those parts of the model where typing will provide the most benefit.

A formal description of the computations of all the variants of typed reflexion models can be found elsewhere [Mur96].

4 MANAGING SOFTWARE REFLEXON MODEL INVESTIGATIONS

For large source models, it can still be difficult to effectively manage the investigation of a software reflexion model. Two features have been added to help manage the process: tagging and annotations.

4.1 Tagging

When investigating a reflexion model, the software engineer may wish to elide convergent or divergent arcs from view. The need for elision arises for different reasons. Sometimes, elision is needed to simplify the view during investigation. In the case of `gcc`, for instance, which has a large number of arcs in the reflexion model, an engineer may want to temporarily remove arcs that are not directly related to the task, such as the arcs from `Parsing` to `RTLGeneration`. At other times, elision is used to indicate that an arc has been fully investigated and has been determined to be outside the task.

Elision of reflexion model arcs is supported through a *tagging* operation. When applied to a reflexion model arc, this operation hides the arc from view until one of the following occurs:

- the engineer chooses to again view the interaction,
- the interaction changes designation—for example, from a convergence to a divergence— when a succeeding reflexion model is computed, or
- during the computation of a succeeding reflexion model, the source model values contributing to the interaction change.

The intent is to ensure that a tagged arc is elided only as long as it has the same essential properties as when the tagging operation was applied to the arc.

In some cases, an engineer could also simplify the reflexion model prior to investigation by redefining the high-level model to remove entities not of interest for the task. A drawback of this approach, however, is that the source model values associated with the arc are removed from the summarization. As a result, the engineer will not be notified if the values associated with the arc change. Losing this information may lessen the engineer's confidence in reasoning about the task at hand in terms of the model; the reason is because it increases the chance the engineer will be surprised later on in the investigation. In addition, experiences with reflexion models show that the sizes of the high-level models are modest, rarely consisting of more than a dozen or two high-level entities. Since the reflexion model is summarized in the context of a relatively small high-level model, the benefits of this approach seemed to be far outweighed by the costs.

4.2 Annotations

When investigating reflexion models that summarize many structural interactions, it is also difficult for an engineer to track which interactions in the reflexion model, and more specifically, to track which of the contributing source model values to a reflexion model arc, have been investigated. To help an engineer perform this tracking, we introduced an *annotation* operation. This operation allows an engineer to record arcs that have been investigated, along with notes describing the relevance of the arc to the task. A visual indication of the effect of annotations is provided to the engineer in a displayed reflexion model by labeling any annotated reflexion model arc with the number of annotated source model values. The annotation numeric value is provided in addition to the number of contributing source model values typically shown on a reflexion model.

For example, an engineer investigating `gcc` might determine that seven of the 11 source model values contributing to the divergence of type "refers-to" between `Optimization` and `Parsing` result from handling the inlining of function calls during compilation. To annotate this information using the reflexion model tools, the engineer would record a regular expression describing the divergent arc and the source values of interest.[3] For

[3] The regular expression may refer either to the high-level model entities, to the source model values mapped to the high-level model interaction, or both.

instance, the engineer may state the following regular expression:

```
Optimization Parsing .*_inline_function.*
```

As documentation, the engineer may also record, with this regular expression, text describing that these values refer to inlining. A snippet of the display of a reflexion model with this annotation applied is shown in Figure 5. This display includes numeric values in parentheses that indicate how many of the source model values are annotated. The arc between Optimization and Parsing is annotated with a "7" to indicate the seven values resulting from the handling of inline functions.

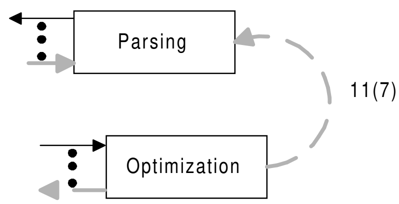

Figure 5. A Snippet of an Annotated Reflexion Model

The annotation operation was developed as a generalization of a tracking method used by the Microsoft engineer who applied the software reflexion model technique to the experimental reengineering of the Excel spreadsheet product [MN97].

5 DESIGN RATIONALE

All of the extensions we have introduced to the software reflexion model technique operate on the *edges* of the graphs representing the source and high-level models. Types are applied to edges as opposed to nodes; the tagging operation elides edges; the annotations operation tracks the investigation of interactions between nodes in a source model. We could have also chosen to apply some of these operations to the graph nodes. For example, it could also be advantageous to define an elision operation in terms of nodes. One advantage of applying these operations to edges is that the map used in the computations, which refers only to nodes, can remain invariant as the operations are applied. In addition, because the maps refer directly to entities (nodes) but only indirectly to edges, it is easier to use the same maps, or iterative improvements to maps, to compute successive reflexion models.

The reflexion model computations, whether untyped or typed, are based on the specification of the maps—the correspondences between the nodes of the source model and the nodes of the high-level model. Comparisons are then made between the projected edges of the source model and the edges of the high-level model. Other choices for specifying the correspondence between the models are also possible. For instance, an engineer may desire to map entities in a source model to edges within a high-level model. This situation can arise when design-level interactions between entities are implemented through mediators [Sul94]. Currently, an engineer must make one of three choices to compute a reflexion model involving a mediator: a mediator entity may be introduced into the high-level model; the source-level mediator entity may be associated with one or both of the high-level model entities involved in the mediated relationship; or the source model may be pre-processed to compute the appropriate interpretation of interactions between the source model entities. More research is needed to understand when different forms for the map, such as enabling the association of a mediator with an high-level model edge, are appropriate and the consequences of the desired forms on a reflexion model computation.

Another design choice we made was to use a minimal representation of types: types are simply names. This choice leads to a straightforward mechanism to specify types. An interaction between entities in a high-level model is typed by specifying a type name for the interaction. This approach is reasonable because the number of interactions in the high-level model is typically small. The approach also makes it easy to form a partially typed high-level model; an engineer simply does not specify a type for an arc. Since the number of source model values (arcs) is usually large, we support a different specification approach, allowing the engineer to specify types on a per file basis where each file contains a set of source model entries.

A simple textual representation was also chosen for the entries comprising a source model. An entry simply consists of a pair of source model entity descriptions. Each description consists of a set of values identifying an entity; the interpretation of the values is dependent upon a naming tree that specifies part of the mapping language to be used in a reflexion model computation. The naming tree used in the `gcc` example, for instance, is a singly-rooted tree consisting of a *directory* that serves as a parent for a set of *files*, each of which serves as a parent for a number of *functions* and *global variables*. Naming trees may also have multiple roots. This representation for the source model has two benefits. First, by changing or augmenting the naming tree, a software engineer may tailor the reflexion model tools to support the investigation of source written in a variety of programming languages. When working with Ada [GPO93] code, for instance, an engineer may use a naming tree that introduces keywords such as package and procedure. Second, the flexible format of the source model entity descriptions makes it easier to produce a source model from a number of existing tools. In general, a small script written in Perl [WS91] or awk [AKW79] or a similar language is sufficient to convert the output of an existing source analysis tool, such as CIA or Field [Rei95], for use in a reflexion model computation.

In fact, a software engineer need not be limited to computing reflexion models based on static analyses of the source code. In a number of cases, it is useful to compute a reflexion model based on a source model collected during the dynamic execution of the system. For instance, an engineer trying to identify performance problems in a system may find it useful to compute a reflexion model based on profiling information. Dynamic information may also be used to either augment or interpret a source model based on static information. Figure 6 shows part of a typed reflexion model for `gcc` that includes both calls extracted statically from the source (the black lines) and calls gathered by gprof [GKM82] as the system executed (the grey lines).[4] An investigation of this reflexion model suggests the some divergences, such as the divergence from `CodeGenerator` to `RTLGeneration`, may not significantly increase the difficulty of the extraction task. The `CodeGenerator` to `RTLGeneration` divergence based on the dynamic information, for instances, results from only two call sites in functions mapped to `CodeGenerator` calling the same function mapped to `RTLGeneration`. (In the case of the source model information gathered by profiling, the numbers on the arcs indicate the number of actual call occurrences made between two functions mapped to each high-level model entity.) The degree to which the engineer may have confidence in this reasoning is, of course, dependent upon the test data used to generate the profile information

[4] Version 6.8.1 of the adventure game was used to collect the profiling information.

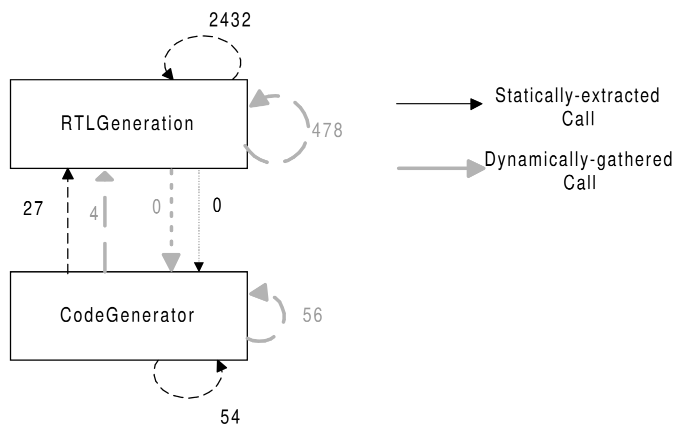

Figure 6: Static and Dynamic Calls in a Reflexion Model

Dynamic information has also been useful to include information in a source model not extracted by a static analysis tool, such as augmenting a statically analyzed C call graph with information about calls through function pointers. Typed reflexion model computations are often used when including dynamic source model information because the separation of the types helps the engineer reason consistently about the different kinds of information represented.

To date, all uses of the reflexion model technique have involved the comparison of information extracted from the source code of the system to different kinds of design models. There is nothing inherent in the technique limiting its use to these kinds of software system artifacts. An engineer could also apply the technique to compare a design model with an analysis model. A strength of the technique is its ability to bridge artifacts representing varying levels of detail. Further research is needed to understand the consequences of handling artifacts at the same level of abstraction; for example, the design of the mapping language is in large part based on the need to compare abstractions of different sizes, as discussed in Section 6.3.

6 RELATED WORK

Software reflexion models touch on a number of different areas of related work. A discussion of the full scope of related work is found elsewhere [Mur96]. In this paper, because inconsistencies in software reflexion models arise from comparing source models to high-level models, we focus on work that considers source code.

6.1 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 [Oss84]. The Software Landscape environment supports similar functionality through a visual module interconnection language [Pen93]. 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 [SSC96]. 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, the second set defines violations that indicate non-compliance. An engineer may then compare structural information statically extracted from C++ source code with the pre-defined 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 heavier-weight description affects the scalability of the Pattern-Lint system. Third, with the reflexion model technique, structural 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.

6.2 Reverse Engineering

Reverse engineering is the process of creating higher-level abstractions from source code [CC90]. 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 twelve reverse engineering techniques based on clustering according to, amongst other things, the level of automation and the nature of the source information supported by the technique, such as control-flow graphs, data-flow graphs, etc. [Lak97]. Of the twelve techniques surveyed by Lakhotia, the Rigi system [MK88] is the sole technique to operate, semi-automatically, on a generic set of source model relations similar to the software reflexion model technique. We thus focus on a comparison to Rigi and to a similar system, called Mercury, built at Microsoft Research.

The Rigi system "supports a method for identifying, building, and documenting layered subsystem hierarchies" [p. 47, WTMS95]. 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 [p. 51, WTMS95].

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.[5] 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 development of the Mercury tool was directly motivated by the use of the reflexion model tools on Excel and was specifically designed to handle the logical remodularization operations performed repeatedly by the engineer during the task. 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.

6.3 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, it had to be language-independent; 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 [Jak95]. 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 behaviour 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

[5] Daniel Weise, Microsoft Research, personal communication, April 1996.

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 bi-directional; 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 uni-directionally 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 bi-directional 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 [HW94]. 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 behaviour between two versions of a procedure [JL94]. 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.

7 CONCLUSION

Many software engineering tasks can be assessed, planned, and performed more effectively when a software engineer has knowledge of inconsistencies that exist between various artifacts associated with the development of a system. The software reflexion model technique helps engineers detect, assess, and manage structural inconsistencies that arise as development artifacts, such as design models and source, drift apart over time.

A key characteristic of our approach is its flexibility: the reflexion model technique places few constraints on the kinds of artifacts compared, on the meaning of inconsistencies, and on the resolution of inconsistencies. Immediately after the initial conception of software reflexion models we took an aggressive stance on tool development: we would only consider adding features that were requested by actual users of the tools. Time and time again, colleagues who considered our research—and indeed, we ourselves—suggested ideas and features that seemed like they would improve reflexion models in some way. But we found that these suggestions rarely coincided with the suggestions we received from our users. The introduction of types and of management techniques show how relatively small changes can collectively improve the effectiveness of software reflexion models.

ACKNOWLEDGMENTS

This research was funded from a number of sources, including the NSF grants CCR-8858804, CCR-9506779, CCR-9502029, CCR-9506779, a Canadian NSERC post-graduate scholarship and research grant, a University of Washington Department of Computer Science & Engineering Educator's Fellowship, the University of British Columbia, DARPA/Rome Labs F-30603-96-1-0314, and a grant from Lockheed-Martin. Equipment was also provided by Microsoft Corporation.

REFERENCES

[AKW79] Aho, A.V., Kernighan, B.W. and Weinberger, P.J. Awk—A pattern scanning and processing language. *Software—Practice and Experience 9*, 4, p. 267-280, 1979.

[CC90] Chikofsky, E.J. and Cross II, J.H. Reverse engineering and design recovery: A taxonomy. *IEEE Software 7*, 1, pp. 13-17, 1990.

[CNR90] Chen, Y.-F., Nishimoto, M.Y. and Ramamoorthy, C.V. The C information abstraction system. *IEEE Transactions on Software Engineering SE-16*, 3, pp. 325-334, 1990.

[GKM82] Graham, S.L., Kessler, P.B. and McKusick, M.K. Gprof: A call graph execution profiler. In Proceedings of the SIGPLAN '92 Symposium on Compiler Construction, pp. 120-126, 1982.

[GPO83] United States Government Printing Office. Reference manual for Ada programming language, MIL-STD-1815A, 1983.

[HW94] Howden, W.E. and Wieand, B. QDA—A method for systematic informal program analysis. *IEEE Transactions on Software Engineering 20*, 6, pp. 445-462, 1994.

[Jak95] Jackson, D. Aspect: Detecting bugs with abstract dependences. *ACM Transactions on Software Engineering and Methodology 4*, 2, pp. 109-145, 1995.

[JL94] Jackson, D. and Ladd, D.A. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*, September 1994.

[Lak97] Lakhotia, A. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software 36*, 3, pp. 211-231, 1997.

[MK88] Müller, H.A. and Klashinsky, K. A system for programming-in-the-large. In Proceedings of the 10th International Conference on Software Engineering, pp. 80-86, 1988.

[MN97] Murphy, G.C. and Notkin, D. Reengineering with Reflexion Models: A Case Study. *Computer 30*, 8, pp. 29-36, August 1997.

[MNS95] Murphy, G.C., Notkin D. and Sullivan, K. Software reflexion models: Bridging the gap between source and high-level models. *In Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 18-28, October 1995.

[Mur96] Murphy, G.C. Lightweight structural summarization as an aid to software evolution. Ph.D. dissertation, University of Washington, Seattle WA, 1996.

[Pen93] Penny, D.A. The software landscape: A visual formalism for programming-in-the-large. Ph.D. dissertation, University of Toronto, Toronto ON, 1993.

[Oss84] Ossher, H. A new program structuring mechanism based on layered graphs. Ph.D. dissertation, Stanford University, Palo Alto CA, 1984.

[Rei95] Reiss, S. The Field programming environment: A friendly integrated environment for learning and development. Kluwer Academic Publishers, Amsterdam Netherlands, 1995.

[SSC96] Sefika, M., Sane, A. and Campbell, R.H. Monitoring compliance of a software system with its high-level design models. In Proceedings of the 18th International Conference on Software Engineering,, pp. 387-396, 1996.

[Sul94] Sullivan, K.J. Mediators: Easing the design and evolution of integrated software systems. Ph.D. dissertation, University of Washington, Seattle WA, 1994.

[WS91] Wall, L. and Schwartz, R.L. Programming Perl. O'Reilly and Associates, 1991.

[WTMS95] Wong, K., Tilley, S.R., Müller, H.A. and Storey, M.D. Structural redocumentation: A case study. *IEEE Software 12*, 1, pp. 46-54, 1995.