

Hierarchical Reflexion Models

Rainer Koschke, Daniel Simon
University of Stuttgart
Universitätsstr. 38, 70569 Stuttgart, Germany
{koschke, simondl}@informatik.uni-stuttgart.de

Abstract

The reflexion model originally proposed by Murphy and Notkin allows one to structurally validate a descriptive or prescriptive architecture model against a source model. First, the entities in the source model are mapped onto the architectural model, then discrepancies between the architecture model and source model are computed automatically.

The original reflexion model allows an analyst to specify only non-hierarchical architecture models, which is insufficient for larger systems that are decomposed into hierarchical subsystems. This paper extends the original reflexion model to hierarchical architecture models, describes a method to apply this technique, and reports on case studies conducted on two large-scale and complex applications (namely, the C compiler `sdcc` for 8-bit microprocessors and the GNU C compiler `gcc`).

1 Introduction

Among the four viewpoints¹ to describe the architecture of a software system introduced by Hofmeister and colleagues is the module viewpoint. The module viewpoint shows the decomposition of the system and the partitioning of modules into layers [2], pp. 12–13.

Modules constitute work assignments and are used to allocate a project’s labor and other resources during development as well as maintenance. Examples for modules are subsystems, packages, classes, processes; examples for their relations are data and control flow and use dependencies.

By creating the module view during forward engineering, the architect addresses how the conceptual

¹A viewpoint is a specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis [3]. A view is a representation of a whole system from the perspective of a related set of concerns.

solution can be realized with available programming languages, software platforms, and technologies.

Far too often, the module view that was initially designed is not in sync with the real implementation due to changes made in the source without updating the documented module view. Murphy and colleagues [6, 7, 8] have developed the reflexion model technique to reconstruct the mapping from the specified or hypothesized to the concrete module view. The basic idea of the reflexion model is to create a hypothesized view from existing documentation or interviews with architects and then to map elements in the concrete module view to the hypothesized view. A tool then computes resemblances and differences between the two views.

The technique was successfully used in several case studies. The most interesting case study—reported by Murphy and Notkin [6]—is the analysis of Microsoft Excel, which consists of about 1.2 MLOC of C code.

The original reflexion model does not allow an analyst to structure the hypothesized modules hierarchically. Yet, large systems are typically decomposed into modules with submodules. Knodel reports that practitioners noted that the lack of means to further decompose the hypothesized view is a major shortcoming in the practical use of the technique when he tried the original reflexion model in an industrial case study at a system of 1.5 MLOC [4].

Contributions. This paper extends the original reflexion model to hierarchical hypothesized module viewpoints (Section 3). The extension requires to define consistency constraints for the specification of hypothesized views. Because the hierarchical reflexion model offers multiple levels where an analyst can start, this paper provides a method for an incremental application of the hierarchical reflexion model (Section 4). We applied the hierarchical reflexion model on two C compilers (Section 5). A comprehensive discussion on related research on component recovery has been published elsewhere [5].

our terminology	Murphy et al.
hypothesized (module) view	high-level model
concrete (module) view	source model
conceptual entity	high-level model entity
concrete entity	source model entity

Figure 1. Terminology translation.

2 Original Reflexion Model

This section describes the original reflexion model as proposed by Murphy and colleagues [6, 7, 8]. Our extensions to the technique are described in Section 3.

In the following, we align our terminology with the IEEE standard [3] and the terminology used by Hofmeister and colleagues [2] and therefore diverge from the terminology by Murphy and her colleagues. A mapping of our terminology and the original terminology used by Murphy is provided in Figure 1.

The hypothesized view of the original reflexion model is depicted in Figure 2 using the UML notation. (Note that module viewpoints by Hofmeister and colleagues [2] are semantically richer.) **Concrete entities** are entities that are present in the module view—in other words—are derived from the source code. **Concrete physical entities** are those that can be immediately derived from the sources (through parsing) or from the file system. Examples are functions, variables, types, classes as well as files and directories. **Concrete logical entities** are coherent concepts, such as modules and subsystems, that may not be specified explicitly by means of the programming language. They may be derived by an intermediate step by means of (semi-)automatic or purely manual component recovery techniques (for an overview on component recovery techniques, see [5]). Examples for concrete logical entities are logical modules and subsystems. For the purpose of this paper, the distinction between concrete physical and logical entities can be neglected. We mention these two types of concrete entities here just to point out that the reflexion model allows an analyst either to start immediately with the files and directories (if they represent coherent modules and subsystems) or to derive the coherent modules and subsystems first (if the existing system decomposition into files and directories does not match the logical organization of the system).

The reflexion model allows to describe the concrete module view at different levels of granularity. For instance, a system may be organized in global declarations, modules, and subsystems. A part-of relationship is used to specify the composite entity to which

an entity belongs. The part-of relationship is reflexive, asymmetric, and transitive, and each entity has at most one composite to which it belongs.

Conceptual entities are entities of the hypothesized view. They represent coherent conceptual modules, that is, key abstractions of the problem or solution domain that are expected to be present in the system. In the original reflexion model, conceptual entities are not hierarchical (in Section 3, we will extend the original reflexion model to cope with hierarchical conceptual entities).

In the following, we are using the convention that capital letters denote conceptual entities and lower-case letters denote concrete entities. Graphically, we are using the UML notation with stereotypes for relations and entities; we render concrete entities as circles and conceptual entities as boxes. Hypothesized views are drawn above concrete views and the two views are separated by a dotted line.

Concrete entities may reference other concrete entities, and conceptual entities may reference other conceptual entities. The *reference* relationship is an abstract dependency relation. In case of concrete entities, the reference relationship is derived from source code. Concrete references are, for instance, calls, variable accesses, and type dependencies. We will use *ref* to denote the reference relationship in the following.

As an extension to the original reflexion model, we may arrange types of concrete references in a type hierarchy; the root of this type hierarchy is the reference relationship. For instance, variable accesses may be either *reads* or *writes* of a variable. The advantage of type hierarchies for references is that we can refer to *variable accesses* when we mean both *reads* and *writes* of variables. For instance, an analyst can then specify a general reference relationship in the hypothesized view—when she does not really care about the exact type of dependency—or a more specific type, such as call, if she expects only this type but no other. In the following, we do not make a distinction between different types of references; but it should be obvious to the reader how such a distinction could be made.

Generally, facts on the concrete view are derived automatically from source code, whereas the hypothesized view is created by an analyst manually. The mapping of the concrete to the conceptual entities—denoted by *maps-to*—is also specified by the analyst. This mapping may be partial (because not all concrete entities are relevant) and should be injective to obtain results that can be interpreted easily.

Based on the hypothesized and concrete module views and the mapping between these two views, resemblances and differences can be specified as follows

concrete module viewpoint hypothesized module viewpoint

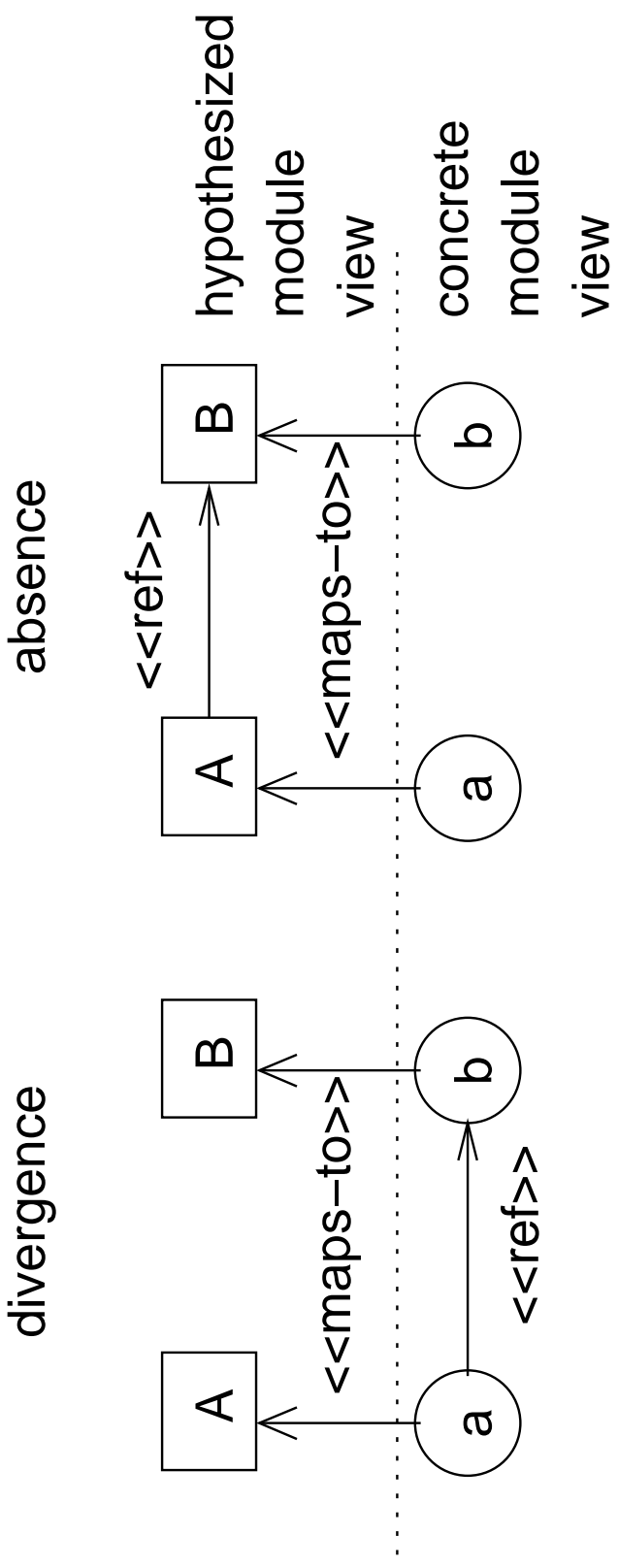
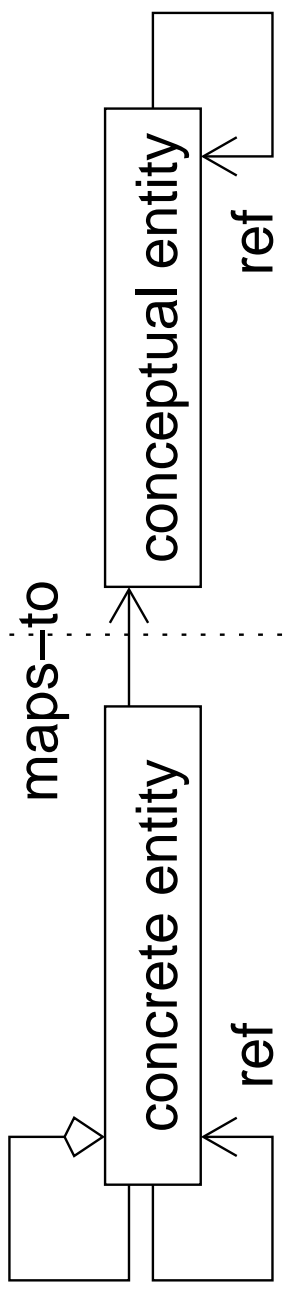


Figure 2. Concepts in the original reflexion model.

(cf. Figure 3):

Convergences are references in the hypothesized view also present in the concrete view.

Divergences are references in the concrete view for which no reference in the hypothesized view exists.

Absences are references in the hypothesized view not present in the concrete view.

In the following, we refer to these relationships as **comparison relations**. The definitions of the comparison relations can be formalized as follows, given a hypothesized view, C , and a concrete module view, M , ($A, B \in C$; \widetilde{ref} describes the actual reference between conceptual entities according to the concrete module view):

$$\begin{aligned} \widetilde{ref}(A, B) &\Leftrightarrow \exists(a, b \in M) : (ref(a, b) \\ &\quad \wedge maps\text{-}to(a) = A \\ &\quad \wedge maps\text{-}to(b) = B) \end{aligned}$$

$$convergence(A, B) \Leftrightarrow ref(A, B) \wedge \widetilde{ref}(A, B)$$

$$divergence(A, B) \Leftrightarrow \neg ref(A, B) \wedge \widetilde{ref}(A, B)$$

$$absence(A, B) \Leftrightarrow ref(A, B) \wedge \neg \widetilde{ref}(A, B)$$

3 Improved Reflexion Model

This section discusses problems of the original reflexion model and shows how to overcome these.

3.1 Mapping

If not only leaf nodes but also composite nodes in the module view are mapped, an injective mapping may be difficult to ensure. Consider, for instance, the mapping in Figure 4: If a is mapped onto A , we would assume that all its parts are likewise mapped to A . Yet, one of its parts, namely, b , is mapped to B . This situation raises the question what the semantics of mappings for composite entities should be. Murphy and colleagues circumvent this problem by defining a precedence for the mappings rules: Technically, the mapping is achieved through a specification file that contains mapping rules. Each rule consists of a regular expression and a target conceptual entity. The first regular expression that matches the name of a concrete entity determines the target conceptual entity. But what if the mapping is established through a click-and-drop user interface? Should the precedence be determined through the order in which mappings were added? We think this is inappropriate because the order is determined just by the analyst's preference for either a top-down or bottom-up approach (or a mixture of both).

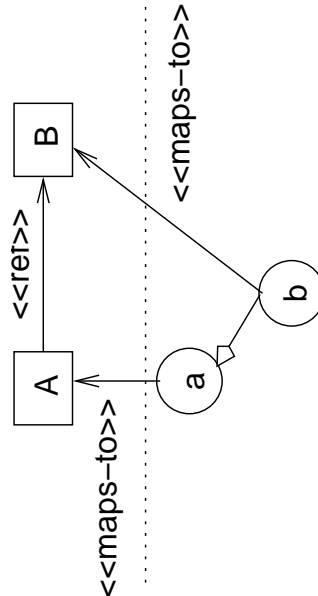


Figure 4. Inconsistent mapping?

Drawing any conclusion for the semantics of the mapping specification is usually inadequate.

Because mappings at the lower levels are more specific than mappings at a higher level and are rather used to specify exceptions to larger-grained mappings, we let lower-level mappings override higher-level mappings (a is in concrete view; C denotes the hypothesized view):

$$\begin{aligned} maps\text{-}to(a) &= \begin{cases} A & : maps\text{-}to(a, A) \\ maps\text{-}to(parent(a)) & : \neg \exists(A \in C) maps\text{-}to(a, A) \\ undefined & : otherwise \end{cases} \\ parent(a) &= \begin{cases} a' & : partof(a, a') \\ undefined & : otherwise \end{cases} \end{aligned}$$

3.2 Hierarchical Reflexion

A shortcoming of the reflexion model is that it does not allow an analyst to further decompose the hypothesized view. For the architecture of large systems, it is necessary to be able to describe the hypothesized view at varying levels of details. This section extends the original model to a hierarchical model.

An extension to the hypothesized viewpoint using a part-of relationship (cf. Figure 5) requires to refine the definitions of \widetilde{ref} and the comparison relations. This refinement should be consistent with the original re-

flexion model. For this reason, let us review the original semantics of the reference relationship in the hypothesized view implied by the original definitions of convergence and divergence:

1. presence of a reference relationship from A to B in the hypothesized view implies that every concrete entity that is mapped onto B is visible to every concrete entity that is mapped onto A
2. presence of a reference relationship in the hypothesized view implies that there must be at least one such corresponding reference in the concrete view
3. absence of a reference relationship from A to B in the hypothesized view implies that none of the concrete entities mapped onto B is visible to any of the concrete entities mapped onto A

To transfer these ideas to a hierarchical hypothesized view, it is straightforward to introduce a "lifted" reference relationship: In case of hierarchical views, we can specify that a composite entity, A , references another entity, B , if at least one of the parts of A references B or at least one of the parts of B . In other words, we can lift the reference relationships from parts to composites.

The adjustments to the definitions of the comparison relations are more complicated. We need to consider the influence of a reference relationship between two conceptual entities view both on these entities and on their containing entities. Let us first look at an example in which we have a reference relation at a lower level, as sketched in Figure 6. According to the reference relation between A'' and B' , the reference between a' and b is a convergence. On the other hand, a naive definition of visibility could cause the absence of a reference between A and B to suggest a divergence.

This problem cannot be solved simply by lifting allowable references from parts to their composites. If the analyst lifted references from parts to composites, she would allow any other part of the composite to reference the entity referenced by the part, too. For instance, in Figure 6, u may then also reference b , which is obviously not intended.

A reasonable interpretation of Figure 6 is to say that the reference at a lower level is an exception to a must-not-reference (i.e., absence of a reference relation) at a higher level. This way the analyst can express: "All parts of A —except A'' and its parts—must not reference anything of B ." In other words, the more specific specification (the one at the lower level) overrides the more general specification.

The second issue with the definition of the comparison relations we need to look at is the influence of

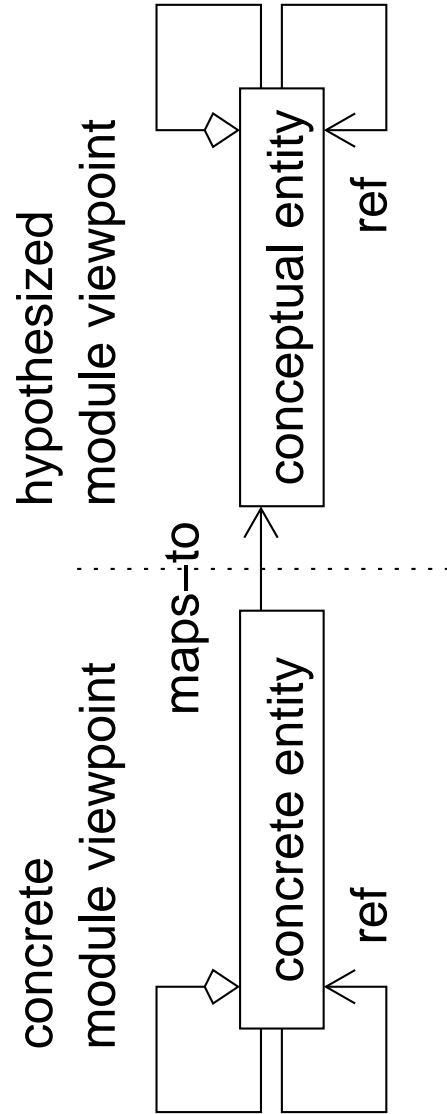


Figure 5. Concepts in the hierarchical reflexion model.

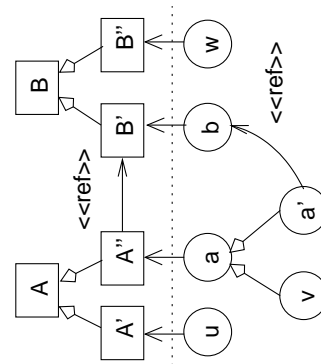


Figure 6. Consistent hierarchical view?

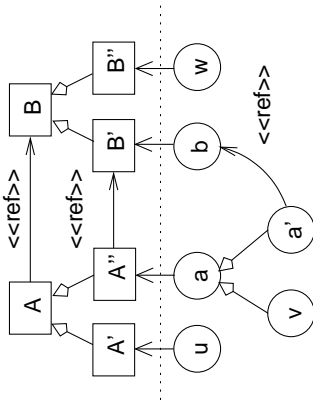


Figure 7. Inconsistent hierarchical view?

a reference relationship at a higher level on reference relationships at lower levels. Figure 7 serves as an example. The reference between A and B allows all parts of a to reference b and would also allow u to reference b . At the lower level, we find a reference between A'' and B' . On one hand, this reference is subsumed by the reference between A and B : It allows all parts of a to reference b . On the other hand, it is also more specific to the reference between A and B : It does not allow u to reference b . To decide how this case should be handled, we need to investigate how this situation can arise in the first place. We assume a scenario in which the analyst specifies the hypothesized view interactively and incrementally. We do not know whether she builds the hypothesized view bottom-up or top-down. If she worked top-down, it could be that she wants to refine a reference relationship after she has gained more knowledge. If she works bottom-up, she might want to generalize a reference relationship found at the lower level. Which of these two possibilities lead to the situation is hard to judge. The order in which the relations were defined is a hint, but we should not rely on that. What we can offer instead is a check for consistency before the hypothesized and concrete module view are contrasted. Pragmatically, the check may be used to just emit warnings rather than to disallow to compute comparison relations altogether, which is useful for an incremental application of the reflexion model where temporary inconsistencies are acceptable in the course of the analysis. On the analyst's wish, consistency can be turned into a hard precondition. Consistency can be defined as follows (non-hierarchical views

are always consistent according to this definition):

$$\begin{aligned}
 (*) \quad & \forall(A, B, A', B') : \\
 & (ref(A, B) \wedge ref(A', B') \wedge \\
 & partof^*(A', A) \wedge partof^*(B', B)) \\
 & \Rightarrow A = A' \wedge B = B'
 \end{aligned}$$

Let us summarize our consequences for the definitions for *ref* and *visibles*: References at a lower level of the hypothesized view override *absent references* at higher levels; references at lower levels are superfluous if they are covered by references at higher levels; that is, they do not imply that siblings of their sources must not reference the target. Such lower level references can be removed for computing divergences altogether. Consistency of the hypothesized view—as defined by $(*)$ —may be checked on demand. The definitions of the comparison relations, however, do not enforce $(*)$. Now we are in the position to define *ref* and *visibles* appropriately (*partof** is the transitive closure of *partof*; \widetilde{ref}^\uparrow is the lifted actual reference relationship as derived from the concrete view M ; A, B in hypothesized view C):

$$\begin{aligned}
 \widetilde{ref}^\uparrow(A, B) & \Leftrightarrow \exists(a, b \in M) : (ref(a, b) \\
 & \wedge partof^*(maps\text{-}to(a), A) \\
 & \wedge partof^*(maps\text{-}to(b), B)) \\
 convergence(A, B) & \Leftrightarrow ref(A, B) \wedge \widetilde{ref}^\uparrow(A, B) \\
 absence(A, B) & \Leftrightarrow ref(A, B) \wedge \neg \widetilde{ref}^\uparrow(A, B) \\
 divergence(A, B) & \Leftrightarrow \neg \exists(A', B') : (partof^*(A, A') \\
 & \wedge partof^*(B, B') \\
 & \wedge ref(A', B')) \wedge \widetilde{ref}^\uparrow(A, B)
 \end{aligned}$$

Because of *partof**(X, X) holds for all X , the preceding definitions are consistent with the original definition by Murphy and colleagues.

Convergence and absence classify existing reference relations in the hypothesized view, whereas divergences result from non-existing reference relations in the hypothesized view. Thus, divergence is a predicate on unexpected actual references immediately derived from the concrete view. That is why we are using the non-lifted reference relationship in the definition of *divergence*. All three relations may be lifted in a hierarchical hypothesized view—like references—as an information to an analyst who may want to inspect only one level of the hierarchical hypothesized view at a time. The lifting, however, gives only informative annotations because it may result in divergences, absences, and convergences between the same two composite nodes; for instance, in Figure 8, $ref(a, b)$ and $ref(A', B')$ imply a lifted convergence between A and B ; $ref(a, b')$ implies a lifted divergence between A and B ; $ref(A', B')$ without corresponding reference in the concrete view implies a lifted absence between A and B .

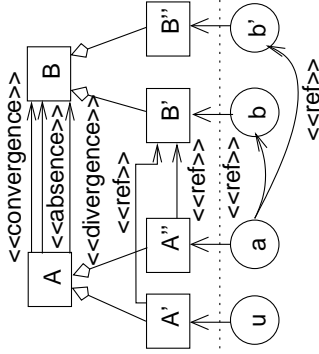


Figure 8. Lifted comparison relations.

4 Method

This section describes a method to apply the hierarchical reflexion model. It consists of the following highly iterative steps:

1. Define the scope for the reflexion model in terms of the hypothesized as well as the concrete module view.
2. From available sources of information (documents, architects, textbooks, directory structures, make-files, etc.), create a first version of the hypothesized view. Focus on key abstractions that form larger building blocks. These may be later refined; yet in the beginning, you should identify what is in the scope and what may be neglected.
3. Identify initial modules for the concrete view that implement key abstractions of the application domain. In the first iteration, directories may be considered subsystems and source files may be considered modules. If the code structure may not be trusted, automatic and semi-automatic component recovery techniques may assist in this step [5]. The key abstractions form a seed for the next steps. You should start with coarse-grained modules (modules that may themselves contain modules) to help with the initial scoping.
4. Map the modules onto the conceptual entities. Analyze the neighboring modules of the seed modules to check whether they should be mapped to the same conceptual entity.
5. Compute the reflexion model.
6. Resolve divergences and absences by adjusting the mapping (you need to analyze the details now) and the hypothesized view. You may refine the scope as well as the concrete and hypothesized view.

7. Continue with step 4 until concrete and hypothesized views are reasonably similar for the task at hand and the scope chosen.

5. Case Studies

This section reports on two case studies conducted to investigate the usefulness and feasibility of the approach in realistic large and complex applications. To obtain comparable results, both applications analyzed are similar tools, namely, C compilers. Both compilers are written in C. The entire source code for both compilers is distributed under GPL.

The first C compiler analyzed is *sdcc*, a free-ware, retargettable, optimizing ANSI-C compiler by Sandeep Dutta designed for 8-bit microprocessors [9]. The current version 2.1.8a targets Intel-MCS51-based Microprocessors (8051, 8052, etc.), Zilog-Z80-based MCUs, and the Dallas DS80C390 variant. It can be retargetted for other microprocessors, support for PIC, AVR and 186 is under development. SDCC has extensive language extensions suitable for utilizing various microcontrollers and underlying hardware effectively.

The second analyzed application is the well-known C compiler *cc1* (version 3.1), part of the GNU C compiler collection (*cc1* is typically called through the tool *gcc*, which acts as a dispatcher to the various compilers: *gcc* calls the appropriate compiler based on the file suffix —*gcc* itself is not a compiler).

We first introduce our hypothesized architecture for these two compilers and then describe how the architectures as-built of the two compilers are mapped on this reference architecture.

5.1 Hypothesized Compiler Architecture

Our background knowledge in compiler construction and published compiler architectures, such as the one by Shaw and Garlan [10], led us to the hypothesized architecture shown in Figure 9. Note that this architecture is large and complex and, hence, requires hierarchical modules. The architecture differs considerably from the architecture described by Shaw and Garlan both in terms of details—our architecture has more details—and the relations in the view: The architecture by Shaw and Garlan is more abstract using logical control and data flow relations, whereas we consider reference relations that can be directly derived from the source. To make the difference clearer, from the point of control flow successorship, it does not matter whether module *A* calls *B* or whether another module *D* calls first *A* and then *B*. In both cases, *B* is a control flow successor of *A*. In a view that has only

references, the fact that A references B is completely different from D referencing first A and then B . Another difference is that the view based on references does not imply any order in which the references occur, whereas control flow is necessarily at least a partial order. It should also be noted that in principal the reflexion model also works for more abstract control and data flow relations; only the analysis to generate these flow relations is more difficult.

The basic building blocks of the compiler are the front, middle, and back end and two additional data structures that represent the input program at different levels of details and degrees of source and target language independence. A source-language specific abstract syntax tree (AST) is used to represent the syntactic nesting of the input program and is later transformed to a more primitive intermediate language (IL) on which the code optimizations are performed. IL is highly independent from the source and target language. To perform the optimizations, the IL nodes are embedded in a control flow graph (CFG).

The front end is in charge of analyzing the source program and is a source-language specific part of the compiler. The front end generates an AST and a symbol table. The latter allows immediate access to AST nodes that represent declarations. Some C compilers integrate a preprocessor module to handle the preprocessor directives. Other C compilers read only preprocessed code and require an external preprocessor. That is why the module *Preprocessor* is optional. Another particularity of C is the ambiguity of identifiers representing types defined by `typedef` and other identifiers in the C grammar. One solution to this problem is to let the *Lexer* access the symbol table to decide whether it should return a token representing an identifier associated with a `typedef` or an ordinary identifier. In the architecture, this solution is anticipated by the reference from the *Lexer* to the *Symbol Table*.

The middle end implements the optimization based on IL. This part may be reused for different source and target languages. The back end performs additional target-language specific optimizations and eventually generates code.

Control is the driver of the compiler. It initiates the initialization of all modules by way of module *Initialization*, sets the command line switches contained in and offered to all other modules by *Global Configuration*, and then calls the *Parser*, the *Semantic Analysis*, the mapping of the AST to IL, the various code optimizers, and eventually the code generation. An additional module offers utilities and error handling to most other modules.

reference type	description
static call	statically bound call of function
dynamic call	call through function pointer
access	use, set, or address-taken of a variable or record component
r-access	address-taken of a function
signature	type occurs in function signature
of-type	type of a variable or record component
local-var-of-type	function has local variable of type
based-on-type	one type uses another type for its declaration

Figure 10. Extracted reference types.

5.2 Extraction

For extracting the facts for both compilers, we used the Bauhaus toolkit [1], which offers an extractor for C, a points-to analysis based on Steensgaard’s algorithm [11] to resolve function pointers for a conservative call graph, computation of the hierarchical reflexion model, and support to navigate and visualize the reflexion model.

Figure 10 summarizes the extracted reference types. Note that we—unlike the original application of the reflexion model by Murphy and colleagues—are also using type dependencies. If a module of one system were to be used in a different system, the other system would need to offer a type comparable to the one referenced by the reused module. Moreover, the type dependency gives important information on how the individual data structures are related. That is why we are also interested in type relationships.

As it turned out in the case studies, the points-to analysis was a particularly helpful part of the fact extraction because both systems use numerous function pointers. Otherwise we would have missed important dependencies.

5.3 Compiler *sdcc*

The compiler *sdcc* is a relatively new and still ongoing development. The compiler is being extended to support other target languages. As opposed to *cc1*, *sdcc* supports only one source language. It has one principal author, but many people are currently contributing to its extensions. The sources consist of about 100 KLOC of C code (including comments and blank lines; counted with the Unix tool *wc*). The 49 source units that make up the system contain about 2.360 global declarations in total (header `file.h` and code

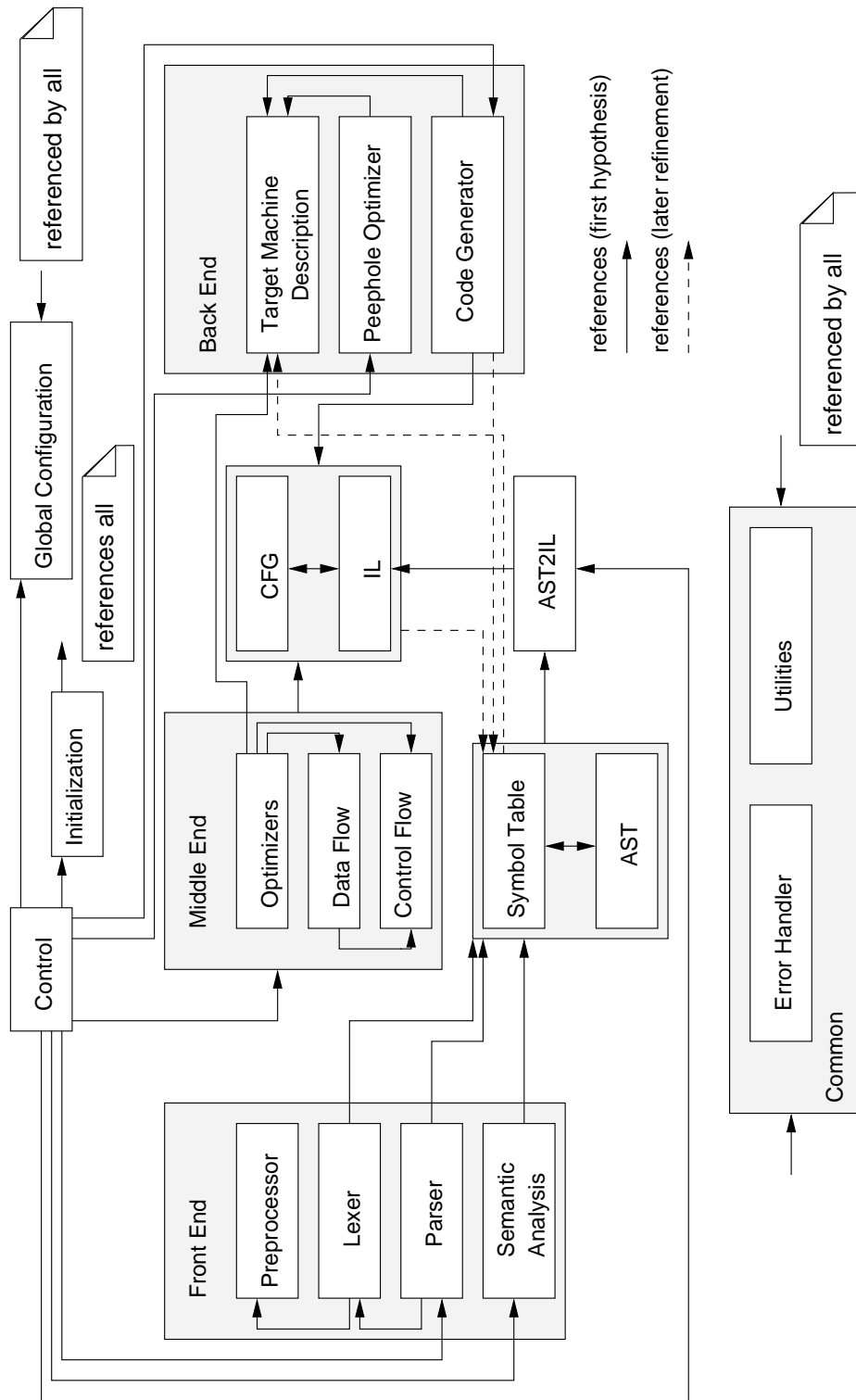


Figure 9. Hypothesized compiler architecture (the dashed dependencies were not in the original hypothesized architecture but added later during the analysis of the reflexion model).

file `file.c` together comprise a source unit `file`) and about 15.300 references in between.

The modules of *sdcc* map easily to the hypothesized architecture in Figure 9. We needed six iterations to establish the mapping. Each iteration refined the mapping and the hypothesized view further. Altogether, the whole architecture reconstruction process took us about six hours.

We started with mapping source units and directories. All the references in Figure 9 could be confirmed (*sdcc* does not have a module for preprocessing). However, there were many divergences. As a matter of fact, the dependency graph for *sdcc* is a highly connected graph at the source file level. We then refined the mapping at the level of global declarations after investigating the divergences. We mapped 45 global declarations to conceptual nodes different from those to which the source units were mapped that contain these global declarations. These global declarations were mostly mapped on *Global Configuration*, which contains information about the currently processed compilation unit, function, output filename, etc. In one notable case, we mapped a character buffer declared but not referenced in the *AST* module to module *Utilities*. We also refined our hypothesized view: We overlooked the dependencies from *IL* and *Code Generator* to the *Symbol Table*. Both the basic blocks in the CFG and the operands in the IL refer to symbols stored in the *Symbol Table*. This information is also needed for code generation. Another dependency we added later was the one between *Symbol Table* and *Target Machine Description* because symbols in *Symbol Table* also refer to register descriptions.

Still, even after the refined mapping, there are still many divergences. The reasons for these divergences are that later compilation stages require information from earlier stages that are not transmitted through the AST or IL modules. *Symbol Table* refers to variables of *Parser* to obtain the declaration block number and line number in which a symbol was declared when a new symbol is created by function `newSymbol`. This dependency seems to be questionable because the current block and line number could also be transferred as a parameter to `newSymbol`. Comparable functions in *Symbol Table* actually obtain these values as parameters. Similarly questionable dependencies exist from *AST* to *Parser*.

There are accesses from the *Back End* to the *Parser*, which are most questionable, too: The *Parser* declares and initializes (statically) two variables that administer stack sizes for activation records. The *Parser* does not use or manipulate these variables otherwise. These two variables are referenced only in later compilation

stages.

The optimizations are actually triggered by the *Back End* and not—as we assumed—by *Control* directly. However, *Control* calls a module of the *Back End* which in turn starts the optimizations.

Other unexpected divergences are caused by *local-var-of-type* and *signature* dependencies. They are harmless from an information hiding point of view, yet do represent a static dependency.

Another kind of interesting unexpected dependencies are dynamically bound calls through function pointers. Some of these divergences may be caused by the overly conservative points-to analysis we used (the analysis is both context and flow-insensitive). Nevertheless, these dynamic calls let us uncover an interesting architectural pattern in *sdcc* used to parameterize the compiler for the various supported target platforms. Certain optimizations require additional information on the target platform, such as number and types of register and so on. The target platform is an exchangeable part of the compiler so that the optimizers cannot reference a certain back end directly. Instead they obtain the necessary information through a data structure representing the target machine. There is one instance of this data structure for each target machine. Each instance is configured for the specific target machine with appropriate function pointer values that yield the machine-specific information upon call. This way, *sdcc* does not need to be recompiled for each platform; the same executable of *sdcc* may generate code for all supported platforms. By similar means, IL obtains register names from the back end.

This kind of parameterization is very different from other cross-compilers such as *cc1*. As we will see in the next section, *cc1* is parameterized statically through code generation and may be configured for only one target platform at a time.

Another dependency through dynamic calls exists from the *Parser* to the *Back End*. This dependency is due to passing `pragma` directives that are specific to a platform directly to the *Back End*. Finally, there is another dependency through dynamic calls from *Utilities* to the *Code Generator*: *Utilities* offers an abstract data type *set*. This *set* type offers an access function that allows one to apply a function passed as parameter to all members of the set.

5.4 GNU Compiler *cc1*

The first stable version of the C compiler *cc1* was released in 1990. Since then it has evolved and many programmers have contributed to it. Today, *cc1* consists of about 500 KLOC (including comments and blank

lines; counted with the Unix tool *wc*).

The compiler *cc1* supports code generation for various platforms, similarly to *sdcc*. However, the compiler can be configured for only one platform at a time. The configuration of *cc1* we analyzed generates code for Linux/Intel 686.

As opposed to *sdcc*, the overall architecture of the GNU compiler collection supports multiple source languages. The compilers for these source languages share a common middle and back end.

The build process for *cc1* creates three static libraries that are linked to the sources of the language-specific front end. One of these libraries, namely *libiberty*, offers re-usable utilities. Because this library does not really belong to the core of *cc1*, we excluded it from the analysis. The other libraries, *libcpp* and *libbackend*, implement the C preprocessor and the language-independent optimizations and code generation of the GNU compiler collection, respectively. The latter is reused for other compilers in this collection. The name of this library is somewhat misleading as it constitutes both the middle end (platform-independent control and data flow analyses and optimizations) and platform-specific code generation (register allocation, assembler generation, and peephole optimizations); but from the perspective of the language-specific front ends, this distinction does not matter.

The two selected libraries and the front end consist of 156 source units. Some of these source units are automatically generated (namely, the scanner and parser from a lex/yacc specification and the platform-dependent code generator from a target-machine description). The sources contain about 13.600 global declarations (variables, constants, functions, types) and 104.000 references between these global declarations. Because of this huge amount of global declarations, we mapped only source units onto conceptual entities and did not further refine the mapping at the level of global declarations.

We mapped the source units onto the architecture described in Figure 9 where we refined this architecture by additional modules as part of the existing ones. The mapping took about eight hours.

The language-specific part of the compiler *cc1* preprocesses the sources (by means of an integrated preprocessor), analyzes C (lexically, syntactically, and semantically), and creates an annotated abstract syntax tree (AST). The AST is transformed into the register transfer language (RTL), a language and platform-independent intermediate language, on which control and data flow analyses and optimizations operate. Eventually, the optimized RTL is transformed to assembly code.

It was straightforward to identify and structure the *Front End*. Also the basic division of the modules into middle and back end was relatively easy because the target-specific part consists of only a few source units, one of which automatically generated. Likewise, it was relatively easy to assign meaning to the modules in the middle end. Yet, the number of divergences was enormous. Figure 11 shows the references between the source units of *cc1*: The modules in the middle and back end are highly coupled such that a hypothesized view without divergences would resemble a fully connected graph.

On the other hand, the *Front End* is relatively loosely coupled to the other subsystems. The only references from the other subsystems to *Front End* are initializing calls and the call to the parser. As one would expect, there is no reference from *Control* to the preprocessor, so it is hidden from *Control* whether the internal or an external preprocessor is used.

One particularly strange divergence at first sight was a reference from the *Middle End* to the *Preprocessor*. As it turned out, this dependency is caused by a hash table implemented for the preprocessor and used by the middle end. For programs linked with *libbackend*, this requires to link *libcpp* as well. To resolve the divergence, the hash table may be moved to *Utilities* conceptually. Interestingly enough, there are at least three different hash table implementations in *gcc*.

A more interesting divergence is a coupling between the code for profiling and the *Front End*. The profiling instrumentation injects own functions being called at the entry of a user-defined function. To generate these artificial functions, a compilation of a function is faked: The body is created by AST constructors and then the semantic analysis is started to eventually generate code. For a correct semantic analysis, the profiling instrumentation calls functions to enter and exit scopes declared in *Front End*.

6. Conclusions

This paper extends the reflexion model to hierarchical hypothesized views and describes a method to apply the technique. The method was applied to two large and non-trivial systems. The size of both systems called for hierarchical hypothesized views. The analysis of the resulting reflexion model led to adjustments in all three constituting parts: the hypothesized view, the concrete view, and the mapping.

Although some parts of the methods may be automated (extraction, computation of the reflexion model, visualization and support for navigation), the process is mainly manual. Considerable domain knowledge is

Figure 11. Module dependencies for *cc1*.

required for the hypothesized view and the mapping. Although we were not familiar with the implementation of the two systems we analyzed, we succeeded in the mapping in relatively short time. A critical factor for this success was the relatively high coherence of the concrete modules which allowed us to map whole source units with only minor adjustments. If the concrete module view needed to be recovered before, the whole process would have taken much longer.

One shortcoming of our current implementation of the reflexion model is that it is based on references only (although we do extract a rich set of references including calls through function pointers). More abstract relations are required to specify hypothesized views that abstract from irrelevant details. In particular, we need abstract transitive control and data flow relations.

An overall observation of the two case studies is that architecture reconstruction is a difficult task, highly iterative and manual to a large extent due to the complexity of the systems. The highly idealized architectures one finds in the literature turn out to be very complex in real implementations. There are typically many more references as suggested by idealized architecture descriptions. These references are partly a consequence of software aging and ad-hoc changes, but they are also due to the inherent complexity of the application. Even

the still idealized compiler architecture in Figure 9 has many dependencies, for which there are good reasons.

Acknowledgment

We want to thank Gerd Bleher, Andreas Christl, Jens Knodel, and practitioners from our industrial partner who tried the original reflexion model in an industrial setting for their insightful feedback.

References

- [1] Bauhaus. <http://www.bauhaus-stuttgart.de/>, June 2003.
- [2] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Object Technology Series. Addison Wesley, 2000.
- [3] IEEE Standards Board. IEEE recommended practice for architectural description of software-intensive systems—std. 1471-2000, 2000.
- [4] J. Knodel. Process models for the reconstruction of software architecture views. Diploma thesis no. 1987, University of Stuttgart, Computer Science, July 2002.
- [5] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Institute for Computer Science, University of Stuttgart, <http://www.iste.uni-stuttgart.de/ps/rainer/thesis>, 2000.
- [6] G. C. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 30(8):29–36, Aug. 1997. Reprinted in *Nikkei Computer*, 19, January 1998, p. 161-169.
- [7] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proc. of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, New York, NY, 1995. ACM Press.
- [8] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE TSE*, 27(4):364–380, Apr. 2001.
- [9] *sdcc*, a c compiler for small devices. <http://sourceforge.net/projects/sdcc/>, June 2003.
- [10] M. Shaw and D. Garlan. *Advances in Software Engineering and Knowledge Engineering*, chapter An Introduction to Software Architecture. World Scientific Publishing Company, River Edge, NJ, 1993.
- [11] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.