Symphony: View-Driven Software Architecture Reconstruction

Arie van Deursen
CWI & Delft Univ. of Technology
The Netherlands
Arie.van.Deursen@cwi.nl

Christine Hofmeister
Lehigh University
USA
hofmeister@cse.lehigh.edu

Rainer Koschke
University of Stuttgart
Germany
koschke@informatik.uni-stuttgart.de

Leon Moonen
Delft Univ. of Technology & CWI
The Netherlands
Leon.Moonen@computer.org

Claudio Riva
Nokia Research Center
Helsinki, Finland
claudio.riva@nokia.com

Abstract

Authentic descriptions of a software architecture are required as a reliable foundation for any but trivial changes to a system. Far too often, architecture descriptions of existing systems are out of sync with the implementation. If they are, they must be reconstructed.

There are many existing techniques for reconstructing individual architecture views, but no information about how to select views for reconstruction, or about process aspects of architecture reconstruction in general. In this paper we describe view-driven process for reconstructing software architecture that fills this gap. To describe Symphony, we present and compare different case studies, thus serving a secondary goal of sharing real-life reconstruction experience.

The Symphony process incorporates the state of the practice, where reconstruction is problem-driven and uses a rich set of architecture views. Symphony provides a common framework for reporting reconstruction experiences and for comparing reconstruction approaches. Finally, it is a vehicle for exposing and demarcating research problems in software architecture reconstruction.

1. Introduction

Many software engineering tasks are hard to conduct without relevant architectural information. Examples include migrations, auditing, application integration, or impact analysis.

To illustrate the latter, consider the "Basel II" agreement of the Basel Committee on Banking Supervision which regulates financial risk estimation and reporting.[1] Analysts from Forrester research have estimated that migrating to "Basel II" will cost banks such as ING or Deutsche Bank approximately 115 million Euros. 60% of these costs concern changes to the bank's information systems. Such high impact changes cannot be made without a clear picture of the architecture of the underlying information systems.

In an ideal world, the relevant architectural information is documented at the time architectural decisions are made, updated whenever these decisions are revised, and readily available when needed for a particular task. Unfortunately, architectural information, when available at all, is often outdated and incorrect, or inappropriate for the task at hand.

Software architecture reconstruction is the process of obtaining a documented architecture for an existing system. Although such a reconstruction can make use of any possible resource (such as available documentation, stakeholder interviews, domain knowledge), the most reliable source of information is the system itself, either via its source code or via traces obtained from executing the system.

Architecture reconstruction in practice has been predictably ad-hoc, using simple tools and a large amount of manual interpretation. Researchers have been trying to improve the state of the practice primarily by providing better techniques and tools (e.g., cluster or concept analysis, program analysis, and software visualization). The application of these techniques usually involves three steps: extract raw data from the source, apply the appropriate abstraction technique, and present or visualize the information obtained.

Although research papers presenting reconstruction techniques typically describe the steps needed for the successful application of one specific technique, a number of questions remain. What problems require architecture reconstruction? What are typical views that should be recovered? Which techniques are suitable for reconstructing particular views? How can different views be presented so that they actually hepl to deal with the problem at hand? In this paper we propose Symphony, a method that aims at helping reconstruction teams in answering such questions.

Symphony[2] is the result of a systematic analysis of (1) our own experiences in software architecture reconstruction, (2)

[1] See `www.bis.org/bcbs/` and `www.forrester.com`

[2] The name Symphony reflects that a successful reconstruction is the result of the interplay of many different instruments. Moreover, the authors' collaboration in the area of software architecture reconstruction started in the music room of Castle Dagstuhl in Germany.

cases conducted by close colleagues, and (3) the various approaches that have been published in the literature. In particular, the paper integrates four different reconstruction cases carried out by the authors. These cases are used throughout the paper to illustrate each step of Symphony. They are described in more detail in the appendix.

Moreover, the case studies demonstrate the importance of viewpoints in focusing the reconstruction activities to solve a particular problem. Different viewpoints and corresponding techniques were used in all case studies, underlining the need to recognize viewpoints as first-order elements of any architecure reconstruction process.

Having a method like Symphony can help practitioners by giving them guidance in performing an architecture reconstruction. In addition, Symphony provides a good conceptual framework for comparing case studies. It can help researchers by providing a unified approach to reconstruction, with consistent terminology and a basis for improving, refining, quantifying, and comparing reconstruction processes.

Furthermore, the Symphony method is view-based in recognition of the importance of multiple architectural views not only in presenting architecture but more fundamentally in defining the reconstruction activities. Previous research has focused on recovering a single architectural view or a few preselected views. Part of the Symphony process is the discovery of the views that should be reconstructed in order to solve the problem at hand.

This paper is organized as follows. First we summarize related work in Section 2. Then, we define our terminology on architectural views in Section 3. In Section 4 we provide an overview of the Symphony steps, which are then described in Sections 5 and 6. In Section 7 we summarize our contributions and opportunities for future work.

2. Related Work

Software architecture reconstruction is an active area of research, as illustrated by the recent software architecture reconstruction workshops held in conjunction with the Working Conference on Reverse Engineering in 2001, 2002, and in 2003 in Dagstuhl, as well as the workshops organized by the SEI on asset mining for software product lines.

Although there is a substantial body of published work in the area of reverse architecting, we are not aware of other papers addressing the software architecture reconstruction process *per se*. In this section, we summarize those papers that deal with software architecture reconstruction and discuss the process elements covered by them. Note that a significant amount of related work is furthermore discussed in our presentation of the various Symphony steps.

Software architecture reconstruction is a special form of software reverse engineering. Many reverse engineering approaches are based on an extract–abstract–present cycle, in which sources are analyzed in order to populate a repository, which is queried in order to yield abstract system representations, which are then presented in a suitable interactive form to the software engineer. Tilley *et al.* [31] describe the extract–abstract–present approach in more detail, referring to the steps as *data gathering*, *knowledge inference*, and *information presentation*.

A number of reverse engineering activities focus on software architecture reconstruction. Kazman et al. [11] propose an iterative reconstruction process where the historical design decisions are discovered by empirically formulating/validating architectural hypotheses. They also point out the importance of modeling not only system information but also a description of the underlying semantics [11]. Their approach is currently extended to include the reorganization of recovered assets into software product lines [30].

Finnigan et al. [10] propose the Software Bookshelf: a toolkit to generate architecture diagrams from source text.

Ding and Medvidovic describe the Focus approach, which contrasts a *logical* (idealized, high-level) architecture with a *physical* (as implemented, as recovered) one [8]. By applying refinement to the logical and abstraction to the physical architecture, the two are brought together incrementally.

All the previous works differs from Symphony in that they address a determined goal, concrete techniques, and a certain fixed sets of views to be reconstructed, whereas Symphony provides a general reconstruction model.

3. Views in Symphony

Software architectures are generally described by models and their rationales. The goal of Symphony is to reconstruct such models (and their rationales if possible). These models are created using viewpoints and presented using views.

3.1. Views and Viewpoints

A *view* is a representation of a whole system from the perspective of a related set of concerns [15]. While it is now generally accepted that the architecture description should be composed of multiple views, the terminology related to views is not yet widely accepted. In this paper, we refer to the IEEE 1471 standard [15].

In IEEE 1471, a view conforms to a *viewpoint*. While a view describes a particular system, a viewpoint describes the rules and conventions used to create, depict, and analyze a view based on this viewpoint [15]. A viewpoint specifies the kind of information that can be put in a view.

The use of architectural viewpoints and views is a key aspect of Symphony. In forward design, different architectural viewpoints are useful for separating engineering concerns, which reduces the complexity of design activities. When the resulting design is captured in separate views, this separation

Target	Source
layer *uses* layer layer *contains* program layer *contains* copybook	program *uses* program program *copies* copybook file *conforms-to* naming-convention layer *prescribes* naming-convention
table *joined-with* table program *C/R/U/D* table program *enforces* integrity-constraint layer *C/R/U/D* table	program *uses-DB-utility* parameter-list table *has-primary-key* column-list table *has-index* column-list column-list *compared-with* column-list

Figure 1. Some viewpoints for Assessment case.

of concerns helps stakeholders and architects *understand* the architecture.

For architecture reconstruction, multiple viewpoints and views are also beneficial. Different viewpoints help the architect determine what information should be reconstructed in order to solve the problem. The existence of a library of viewpoints found to be generally useful gives the architect a basis for reasoning about how different kinds of architectural information shed light on the problem. Separation of concerns still plays a role, but now in allowing the architect to reason separately about how each viewpoint could contribute to a solution of the problem.

3.2. Source, Target, and Hypothesis

A *source view* is a view of a system that can be extracted from artifacts of that system, such as source code, build files, configuration information, documentation, or traces.

Some source views discussed in this paper are at such a detailed level that they are not generally considered to be architectural views. For instance, the source view may cover abstract syntax trees and control flow graphs.

A *target view* is a view of a software system that describes the as-implemented architecture and contains the information needed to solve the problem/perform the tasks for which the reconstruction process was carried out.

A *hypothetical view* describes the architecture of the system, but perhaps not accurately. It can be a reference or a designed architecture used to check conformance of the implemented architecture to a norm. It can be a postulated architecture, describing the current understanding of the architecture of a system, and used to guide the reconstruction. This view is typically created by interviewing the system experts or by examining the existing documentation.

To illustrate the roles of source, target, and hypothetical views we take a look at a reconstruction conducted as part of a quality assessment of a system written mostly in Cobol.

The *hypothetical view* case consisted of the documentation and presentations offered by the system supplier, who argued that there was no reason for concerns on the quality of the system because of the layering, customization, and data handling mechanisms that were included in the architecture. It was used to guide the design of the target model and for finding potential architectural violations.

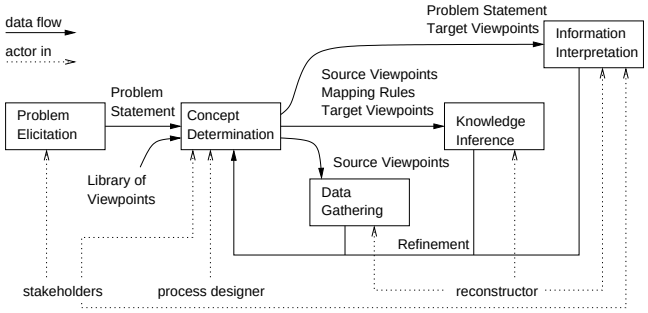

Figure 2. Interaction during reconstruction design.

A selection of the relations contained in the *source* and *target* views is shown in Figure 1. The relations are grouped in a module viewpoint (first row) and a data viewpoint (second row). The target view provides an architectural perspective of the system as implemented, while the source view includes those relations that can be readily derived from the system's source code. As an example, the target model includes CRUD (Create, Read, Update, Delete) information indicating how components manipulate data elements. In some cases, this information may be directly available from the sources (e.g., program file contains SQL statement). In the system at hand, the source model was more complex, since data manipulation was encapsulated in (generated) data utilities, requiring analysis of control (who calls these utilities) and data flow (what parameters are passed to the utility).

The target model recovered helped to identify layering violations, data integrity checks that were bypassed, and ad hoc mixture of custom and product code complicating upgrades to future product releases.

4. Symphony Steps

Symphony has two stages. During *Reconstruction Design*, the problem is analyzed, viewpoints for the target views are selected, source views are defined, and mapping rules from source to target views are designed. The *Reconstruction Execution* analyzes the system, extracts the source views, and applies the mapping rules to populate the target views.

Typically the two stages are iterated: Reconstruction execution reveals new reconstruction opportunities, which lead to a refined understanding of the problem and a refined reconstruction design. The source viewpoints, target viewpoints, and mapping rules evolve throughout the process.

The outcomes of Symphony are twofold: *Reconstruction Design* results in a well-defined procedure for reconstructing the architecture of the system. This procedure may be useful beyond the scope of the current reconstruction: it can play a role in continuous architecture conformance checking and in future reconstructions. *Reconstruction Execution* yields the architecture description needed to solve the problem that triggered the original reconstruction activity.

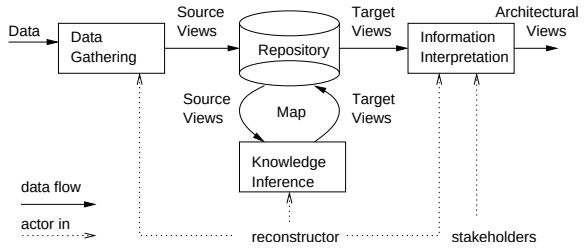

Figure 3. Reconstruction execution interactions.

The various Symphony reconstruction steps are illustrated in Figures 2 and 3. Design steps include *Problem Elicitation* and *Concept Determination*, and are discussed in Section 5. Execution steps include *Data Gathering*, *Knowledge Inference*, and *Information Interpretation*, discussed in Section 6.

5. Reconstruction Design

During reconstruction design we distinguish *problem elicitation* in which the problem triggering the reconstruction is analyzed and discussed with stakeholders, and *concept determination*, in which the architectural concepts relevant to the problem at hand and a recovery strategy are identified.

5.1. Problem Elicitation

Reconstructing architectures requires software architecture experts to study a system and an active involvement of stakeholder representatives, such as testers, developers, management, the business owning the system, and system users. These people are usually in strong demand in other places of the project or the organization. Therefore, there must be a compelling reason to start a reconstruction. Typical reasons include performance problems, high maintenance costs, poor reliability, and considerations concerning system replacement or system extensions. These reasons can typically be collected in a short (one or two page) memorandum offering a management perspective on the problem at hand.

This memorandum forms the starting point for a software reconstruction activity, and the first step is to elaborate this problem statement. This is the purpose of Symphony's *Problem Elicitation* step and requires the involvement of more technical people in the problem analysis.

In our experience, individual technical people involved in system development typically have a fairly good idea of specific technical problems in their area of expertise (e.g., database administration, networking, user interfaces). In the problem elicitation step these different perspectives should be integrated into one overall picture.

There are several techniques that can be used during problem elicitation, such as structured workshops, checklists, role playing, and scenario analysis. As an example, in the assessment case discussed previously, we started with a workshop for which all stakeholders were invited. In this particular case, each participant was asked to report his best and worst experience with the system analyzed.

Outcomes of Symphony's *Problem Elicitation* step include summaries of interviews, workshop sessions, and relevant discussions; summaries of available high-level relevant documentation, if available; an elaboration and refinement of the problem statement based on these summaries; and an initial list of documentation and other resources that can be used during the reconstruction.

Observe that the original memorandum, the collected summaries and the refined problem statement may very well be "architecture-agnostic": they must be expressed in terms familiar to the stakeholders. The translation of the problems-as-perceived to software architecture concepts is the purpose of the "concept determination" step.

The diversity of motivations for architecture reconstruction is exemplified by the four different case studies that lead to the design of Symphony. Two of them, namely, the Assessment and Nokia case, are true industrial cases. The other two were conducted in an academic—nevertheless realistic—setting to better understand architecture reconstruction. As mentioned earlier, a more detailed description of the case studies can be found in the appendix.

Assessment case. The Assessment case (partly described in [6]) involves an assessment of the quality characteristics of a commercial software product written mostly in Cobol that was being customized for a particular client. In the course of the customization process (which took two years) the client grew more and more concerned about the data integrity, reliability, and maintainability. An independent assessment was commissioned which should help to decide whether to continue the project. Source code and documentation were available for use in this assessment.

Nokia case. The products of Nokia are typically organized in product families in order to reduce the development costs and maximize the reuse of the assets. The architects' needs can be summarized as follows: (1) comprehending the as-implemented architecture of the products, (2) managing the organization of components and their logical dependencies in the platform, and (3) enforcing conformance to architectural rules. The main goal is to provide the architects with up-to-date information by reconstructing the same architectural views that they typically use during design.

Compiler case. In this case, the as-built architectures of two large and complex compilers were to be compared against a reference architecture. Although the actual motivation was to evaluate an extension to the original reflexion method by Murphy et al. [20], the case study can indeed be viewed as a realistic task in which an as-built architecture is to be compared against an idealized architecture.

Duke's Bank case. The goal of this reconstruction was to understand Duke's Bank and to determine the abstractions to use in such a system. The motivation was to learn how to

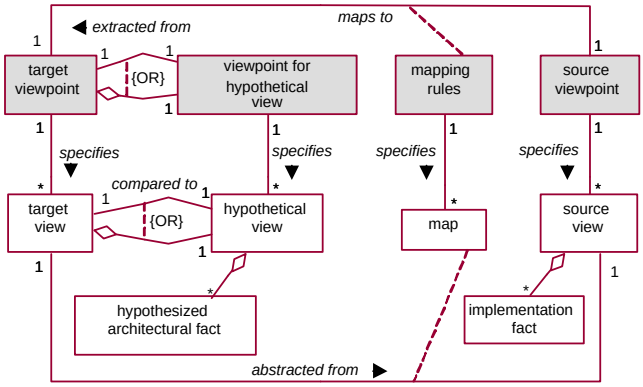

Figure 4. Viewpoints and Views in Symphony

reconstruct so that it can be done more efficiently for future examples. An additional challenge in this case study was to exercise the reconstruction with off-the-shelf tools, such as Rational Rose, grep, emacs, etc.

5.2. Concept Determination

Once the problem is understood, the Concept Determination step is used to determine the architectural information needed to solve the problem and the way to derive this information. In this step, the architect is a process designer, defining the architectural reconstruction that will take place in the final three steps.

There are five outcomes of this step, each of which is described in the remainder of this section. The UML diagram in Figure 4 summarizes the relationships involving the viewpoints and mapping rules produced in this step.

Identify Potentially Useful Viewpoints. The first step towards defining the target viewpoint is to identify a set of viewpoints that contain the information the stakeholders believe will be needed to solve the problem as described in Problem Elicitation. Stakeholders typically know which viewpoints will be useful, or have at least some initial ideas. After getting input from the stakeholders, the architect should review the problems and questions, looking for additional useful viewpoints. Although the architect is responsible for producing the list of viewpoints, ultimately the stakeholders must agree to them.

These viewpoints can come from a library of well-known viewpoints, or a new viewpoint can be created for a specific reconstruction. If the problem is not understood well enough to identify viewpoints of interest, the Problem Elicitation step should be re-applied.

One of the most commonly used viewpoints for architecture reconstruction is the Module viewpoint [13]. It identifies the layers, subsystems, and modules in the system and describes relationships (e.g. usage-dependency and decomposition) among them. Other common viewpoints are the Code architecture viewpoint, which describes directory structure and build relationships, and the Execution viewpoint, which describes the runtime entities and their mapping to physical resources [13]. The Conceptual viewpoint [13], describing the functionality of the system in terms of components and connectors, is less commonly used for reconstruction because it is a more abstract view and is therefore more difficult to reconstruct. (See also [4] for examples of Styles, which is their term for viewpoints.)

The Module viewpoint was also used for all four case studies. The Nokia and Duke's Bank cases used the Code, Execution, and Conceptual viewpoints in addition, whereas the Assessment case study used two other viewpoints (Data and Customization) in addition to the Module.

For reconstruction it may be useful to create new viewpoints, ones which are not used in forward design. An example is the Reflexion Model used by Murphy and Notkin [26]. Their Reflexion Model is based on the usage-dependency relationship in the standard Module viewpoint. It contains three relationships (convergence, divergence, absence) that indicate whether the usage-dependencies reflected in the source code conform to those in the hypothetical view.

Define/Refine Target Viewpoint. As Figure 4 shows, the target viewpoint specifies the target view that will be an output of the reconstruction process. The stakeholders should also agree to the target viewpoint.

One useful approach for creating the target viewpoint is to use the Stakeholder/View tables described in [4], adapted somewhat for reconstruction. In its original form this is a three-step process culminating in a prioritized list of views needed for documenting a software system.

In Symphony, the first step, producing a candidate view list, begins with the potentially useful viewpoints already identified. Each of these should be listed along with the extent to which it is important for solving the problem. The second step is to identify the specific relationships of each viewpoint that are needed. The third step is to prioritize these relationships and eliminate any duplicates. During this process the architect should be thinking about similarities among the relationships, which can be derived from others, which are most critical to solving the overall problem, and should try to consolidate them to arrive at the set of relationships in the target viewpoint.

Not all relationships in the target viewpoint will come from a standard viewpoint. For example, the Duke's Bank case is a J2EE application, where one servlet can "forward" to another, and a jsp can "include" another. These were initially covered by adding a "module forw/incl/etc. module" relationship to the target viewpoint. (In a later refinement this relationship was combined with the usage-dependency relationship, but that determination could not safely be made at the outset.)

As another example, Figure 5 summarizes the Murphy/Notkin Reflexion work in terms of Symphony. Here the target viewpoint contains relations extracted from the Mod-

Starting Viewpoint	Target Viewpoint	Source Viewpoint	Mapping Rules
Reflexion (variant of Module view)	module *convergence* module	dir *contains* dir dir *contains* file func *alloc_to* file func *calls* func	Relation: file *maps_to* module Rules: $(f_i$ *calls* $f_j) \wedge (f_i$ *alloc_to* $file_x) \wedge (file_x$ *maps_to* $m_m)$ $\ldots \Leftrightarrow ((m_m$ *calls*$_a$ $m_n)$ *maps_to* $(f_i$ *calls* $f_j))$
	module *divergence* module		
	module *absence* module		
Module view	module *calls*$_a$ module		$((m_m$ *calls*$_a$ $m_n)$ *maps_to* $(f_i$ *calls* $f_j)) \Rightarrow (m_m$ *calls*$_a$ $m_n)$
	module *calls*$_h$ module		$(m_m$ *convergence* $m_n) \Leftrightarrow (m_m$ *calls*$_a$ $m_n) \wedge (m_m$ *calls*$_h$ $m_n)$
trace target to source view	(module *calls*$_a$ module) *maps_to* (func *calls* func)		$(m_m$ *divergence* $m_n) \Leftrightarrow (m_m$ *calls*$_a$ $m_n) \wedge \neg(m_m$ *calls*$_h$ $m_n)$ $(m_m$ *absence* $m_n) \Leftrightarrow \neg(m_m$ *calls*$_a$ $m_n) \wedge (m_m$ *calls*$_h$ $m_n)$

Figure 5. Viewpoints and Mapping Rules Used in Reflexion Example

ule and Reflexion viewpoints.

The Compiler case study used the Reflexion work as a starting point, so initially its target viewpoint was the same as in Figure 5. However, during the course of the reconstruction it became clear that the target viewpoint needed to be modified to support hierarchies of modules, so in a second iteration the relationship "module contains module" was added.

Define/Refine Source Viewpoint. The source viewpoint specifies the source view. The source view will contain information extracted from the source code and gathered from other sources; the source viewpoint formally describes this information. The challenge in defining a source viewpoint is to determine what information will be needed in order to create the target views. Thus defining the source viewpoint needs to be done in conjunction with defining the mapping from source to target viewpoint.

In the Reflexion example (Figure 5), the source viewpoint contains some architectural and some lower-level information, but all of it can be directly extracted from the code. This was not true for all of our case studies: although automatically-extractable facts formed the basis of the source viewpoint in all, a few relied in addition on relationships that can be populated only by manual interpretation of the sources. For instance, in the Compiler case, we had to inspect the results of an overly conservative automatic pointer analysis to filter out obviously wrong results.

The Assessment case study had a second iteration to refine the source viewpoint. In the first iteration the definition of the source viewpoint was driven by the information existing tools could produce. Since this was inadequate for producing the desired target viewpoint, a second iteration was used with a refined mapping and an expanded source viewpoint.

Define/Refine Mapping Rules. The mapping rules are ideally a formal description of how to derive a target view from a source view. Realistically, parts will often be in the form of heuristics, guidelines, or other informal approaches. If a mapping can be completely formalized, the reconstruction can be fully automated. As said earlier, this is not typically possible for software architecture, thus we expect the mapping to contain both formal and informal parts.

Figure 4 shows that the mapping rules specify the map. The 'mapping rules' entity is an association class connecting the target viewpoint and source viewpoint. Thus it describes the 'maps to' association between these two entities. The map, as the instantiation of the mapping rules, describes how specific implementation facts in the source view are abstracted to architectural facts in the target view.

In the four case studies and the Reflexion example the mappings all contain some informal parts. In the Reflexion example and the Assessment case study, the relation "file $maps\_to$ module" must be manually populated to produce the map. However, the rest of the mapping is a set of formal rules used to compute the target views (Figure 5). Similarly, the mapping in the Nokia case study relies primarily on a series of transformations formalized in relational algebra. At the other extreme, the mapping in the Duke's Bank case study contains a number of rules about how entities in J2EE applications are related, but they provided only partial information for creating the map. Most of the map creation was done manually.

Determine Role and Viewpoint of Hypothetical Views. In addition to the above activities, the stakeholders and architect must determine whether a hypothetical view is needed and what its role will be. This role depends on the purpose of the reconstruction. The most common roles of a hypothetical view are as a guide during the reconstruction activity and as a baseline to compare with the system's current architecture.

When serving as a baseline there are two ways the comparison can be done. One is to create an explicit comparison view, with the comparison embodied in the target view. The Reflexion example and the Compiler case study have such a target view: it identifies modules, usage-dependencies among them, and identifies which of these usage-dependencies match those in the hypothetical view and which do not. In Figure 5 part of the target viewpoint is the $calls_h$ relation, which specifies the hypothetical view (called the 'high-level model' in [26]).

The second way to use a hypothetical view as a baseline is informally. In this case it is used in the last step, Information Interpretation. Typically the architect browses both the target view and hypothetical view, compares them, and based on the results may decide to perform another iteration of the reconstruction process, modifying the target viewpoint, source viewpoint, mapping, or some combination of these.

The Nokia and Assessment case studies used a hypothetical view both for guidance and as a baseline. The hypotheti-

cal view guided the definition of the target viewpoint, helped in populating the map, and served as a baseline during Information Interpretation.

The hypothetical view also has a viewpoint that must be defined. If the hypothetical view is embedded in the target view (as in the Reflexion example) then its viewpoint is defined as part of the target viewpoint. This is shown as the containment relationship between the two viewpoints in Figure 4. If the hypothetical view is not embedded, then typically its viewpoint is very similar to the target viewpoint so that comparison is straightforward. In Figure 4 this is shown as the 'extracted from' relationship between the two viewpoints.

6. Reconstruction Execution

During reconstruction execution, an *extract–abstract–present* approach is used, tailored towards the specific needs of architecture reconstruction. The three steps populate the source view, apply the mapping rules to create the target views, and interpret the results to solve the problem at hand.

6.1. Data Gathering

Intent. The goal of the Data Gathering step is to collect the data that is required to recover selected architectural concepts from a system's artifacts. The motivation is that the truth about the actual (concrete) architecture is in the sources. However, in general, one can look at other artifacts of the system than just its source code. These other artifacts include a system's buildfiles/makefiles, (unit) tests, configuration files, etc. The data gathered are stored in a repository and processed in the Knowledge Inference step.

Examples. The types of data that we have gathered in the case studies are described in Figure 6. These facts are at a low level expressing knowledge in terms of source code elements (hence the term source views). In Knowledge Inference these facts are abstracted (or lifted) to higher levels.

Techniques. Techniques for data gathering can be divided in static and dynamic analyses of the system. Static analyses analyze the system's artifacts to obtain information that is valid for all possible executions (e.g, program structure or potential calls between different modules).

Dynamic analyses collect information about the system as it executes. The results of such an analysis are typically valid for the run in question, but no guarantees can be made for other runs. Dynamic analysis is done by tracing the execution paths/profiles of the code and analyzing them for patterns, sequences, and dependencies. Such traces can be collected using code instrumentation, debugging, and profiling tools, or by connecting to a (prepared) runtime environment.

Note that these kinds of analyses do not necessarily have to be developed by the team that is using them to recover the architecture. Suitable results can be imported from a wide range of reverse engineering tools (such as clustering tools, data flow analysis tools, etc.). In practice, often a pragmatic mix-and-match approach for data gathering is applied, combining the results from various extraction tools using scripting and glueing, for example, based on UNIX utilities such as `join`, `split`, `awk` and `perl`.

Below, we will look a little further into methods for extracting facts from textual artifacts such as program code, buildfiles, etc. since that is the most used technique for data gathering. For a more detailed discussion of various methods for source model extraction, we refer to the related work described in [23].

Manual Inspection. Our experiences show that some of the data needed for a reconstruction project can be easily gathered manually by: examining the directory structure, observing the behavior, or by exploring the source code for beacons that signal aspects of interest [24]. In our cases, this included for example the package structure and build relations for Duke's Bank and the verification of client-server separation in the Assessment case.

Lexical Analysis. Several tools are available that perform lexical analysis of textual files. The most well-known is probably `grep` that searches text for strings matching a regular expression. Tools like `grep` generally give little support to process the matched strings, they just print matching lines. Such support is available in more advanced text processing languages such as `awk`, `perl`, and `lex` that allow one to execute certain actions when a specific expression is matched.

The Lexical Source Model Extractor (LSME) uses a set of hierarchically related regular expressions to describe language constructs that have to be mapped to the source view [25]. Use of hierarchical patterns avoids some of the pitfalls of plain lexical patterns but maintains the flexibility and robustness of that approach.

In our case studies, data gathering based on `grep` and `perl` scripting was used for the Nokia case, parts of the Assessment case and parts of the Duke's Bank case.

Syntactic Analysis. Parser based approaches are used to increase the accuracy and level of detail that can be expressed. These typically create a syntax tree of the input and allow the users to traverse, query, or match the tree to look for certain patterns. This relieves them from having to handle all aspects of a language and focus on interesting parts. The Compiler case study uses syntactical analysis (extended with semantical analysis described below).

Fuzzy parsing. Fuzzy parsers are parsers that are able to discard tokens and recognize only certain parts of a programming language [18]. This can be seen as a hybrid between lexical and syntactical analysis. These fuzzy parsers are hand crafted to perform a specific task. They focus mainly on parsing C and C++ to support program browsing. Typically this involves extracting information regarding references to a symbol, global definitions, functions calls, file includes, etc.

Case	Example Relation	Extraction Technique
Assessment	module containment, copybook usage	lexical analysis using Java regular expression matching
	dynamic program calls	island grammars and data flow analysis
Compiler	variable access	parsing
	dynamic function call	parsing and points-to analysis
Duke's Bank	directory structure, build relationships	manual inspection of directories/buildfiles
	class inheritance and containment	examination using Rational Rose and grep/emacs
Nokia	directory containment, file inclusion, function calls	lexical analysis based on regular expression matching

Figure 6. Some examples of the various data gathering techniques used in the cases.

Island Grammars. Island grammars are a novel technique that can be used to generate *robust parsers* from grammar definitions [23]. Island grammars combine the detailed specification possibilities of grammars with the liberal behavior of lexical approaches. The robust parsers generated from island grammars combine the accuracy of syntactical analysis with the speed, flexibility, and tolerance usually only found in lexical analysis. This makes this approach very suitable for developing source model extractors, even if the resulting extractor is used only for a single project. The DocGen documentation generator used in our Assessment case uses island grammars for data gathering [5].

Semantical Analysis. Additional techniques such as name and type resolution, data flow analysis and points-to analysis can be used to improve the results from other analyses (generally on a syntactical basis). For example, in our Compiler case study, points-to analysis was used to determine more accurate call graphs than could be retrieved from just applying syntactical analysis. In the Assessment case study, a simple form of data flow analysis was used to trace program calls via a dynamic call handler.

Output. The output of the data gathering stage is a populated repository containing the extracted source views.

6.2. Knowledge Inference

Intent. The goal of the Knowledge Inference step is to derive the target view from the source view (typically a large relational data set describing the implementation of the system). The reconstructor creates the target view by condensing the low-level details of the source view and abstracting them into architectural information. The mapping rules and domain knowledge are used to define a map between the source and target view. For example, if the mapping contains a rule about using naming conventions to combine classes into modules, the resulting map lists each class and the module to which it belongs. This activity may require either interviewing the system experts in order to formalize architecturally-relevant aspects not available in the implementation or to iteratively augment the source view by adding new concepts to the source viewpoint.

Depending on the degree of formalization of the mapping, this step can be fully or partly automated. We expect the Knowledge Inference step to be conducted initially in close cooperation with the system experts and, as more domain knowledge becomes formalized, more automation is added. This step can be summarized in the following activities: (1) create the map (containing the domain knowledge), and (2) combine the source view with the map to produce the target view. In practice, the map is often created iteratively, with each iteration refining the map or raising its level of abstraction until it can produce a satisfactory target view.

Techniques. Existing techniques can be categorized as manual, automatic, or semi-automatic. Manual approaches typically use simple, general-purpose tools and manual inspection of the system. While they may use reconstruction-specific tools such as SHRiMP, Rigi, PBS, and Bauhaus to help visualize intermediate results, there is no automated support for the process (see for example [21]).

Semi-automatic approaches help the reconstructor create architectural views in an interactive or formal way. They typically rely on the manual definition of the map. Differences among the approaches concern the expressiveness of the language used for defining the transformations, support for calculating transitive closures of relations, degree of repeatability of the process, amount of interaction required by the user, and the types of architectural views that can be generated.

Relational algebra approaches allow the reconstructor to define a repeatable set of transformations for creating a particular architectural view. In the work of Holt et al. [14] relational algebra is used for creating a hierarchical module view of the source code (by grouping source files into modules and calculating the module dependencies). The reconstructor must manually prepare the containment relations, but new relationships can also be inferred using algebra propositions. Postma [27] uses relational partition algebra (RPA) [9] to calculate module dependencies from dependencies extracted from code. RPA is also used to check the conformance of an extracted target view with a hypothetical view (established in the design phase). The process is repeatable and is part of the build process. Riva has proposed a method for inferring the architectural information based on relational algebra and Prolog [28]. Mens [22] uses logic meta programming (Prolog) for mapping implementation artifacts to high-level design and for checking conformance of architectural rules.

More light-weight examples are the Reflexion

Model [26], Tcl scripts for defining graph transformations in Rigi, SQL queries for defining grouping rules (Dali), or the ad-hoc graph query language (GReQL) of GUPRO.

Fully automatic approaches are based on different kinds of clustering algorithms: coupling, file names, concept analysis, type inference.

All the case studies fall into the category of semi-automated approaches. The map between source view and target view was created manually. The map bridged the gap between conceptually different entities (e.g., source entities versus logical component and connectors in the Duke's bank case) or concrete and hypothesized elements in the source and target views (e.g., the mapping of concrete modules onto hypothesized modules in the reflexion method for the Compiler case). The manual map, then, allowed to propagate and lift relations between source entities to entities in the target view automatically.

For the creation of the map, technological, organizational, and often historical background knowledge as well as domain knowledge is required. For instance, the Duke's Bank case leveraged knowledge of web applications, the J2EE infrastructure, and recommended design patterns. J2EE types provided information about which file executes in which container and which classes are separate components. Design patterns helped identify data-transfer classes and helper classes. The application functionality guided decisions about creating interfaces, combining classes into modules, and determining connectors.

The mapping is often difficult because of hidden dependencies. One interesting experience in the Duke's Bank case, for instance, was the identification of "logical" or "hidden" interfaces. These were not explicitly visible in the source code and were discovered only by studying the control flow of the application and data sharing between classes that had no explicit dependencies. Obviously, the quality of the data gathering is key to a successful knowledge inference. The realization of poor data quality forces us to reiterate the data gathering with different means.

Output. The output is an enriched and structured repository where the source view and the domain knowledge has been combined to create the target view.

6.3. Information Interpretation

Intent. The target views—selected to address a particular problem—are inspected, interpreted, and eventually applied to solve the problem. To these ends, the target views need to be made accessible both physically and mentally to all stakeholders.

Motivation. The views that result from Knowledge Inference are not the answer to the problem but provide a foundation to address the problem. In the Information Interpretation, conclusions are drawn from the reconstructed views. These conclusions then lead to measures to be taken to remedy the problem. (The measures themselves are not part of the reconstruction process.)

Ideally, the viewpoints were selected to allow an immediate use of the views; however, even if the viewpoints are carefully tailored, it might become difficult to get an answer at the level of the target views because they may span a huge information space. In such cases, presentations are required that make this information space amenable to all stakeholders. The presentation must be readable and traceable. Readability relates to the ability to easily find and grasp relevant information in the views; traceability allows us to trace the inferred knowledge back to the original data.

Techniques. The scope of the presentation (i.e., the artifacts and their aspects to be presented) is already given in form of the selected viewpoints and target views. The viewers and task to be achieved are stated in the Problem Elicitation. We focus on presentation and interaction issues here.

Although the selected viewpoints define the vocabulary and semantics for the representation, they do not define *how* to present the information. Information presentation addresses this problem, where we take *presentation* quite liberally: any means to communicate information to a viewer, be it textually, graphically, or through other forms of human perception including any form of interaction with the presentation. Sight is the most often addressed form of human perception by information presentation in the software architecture domain; that is why we are using the narrower term *visualization* instead of *perception* in the following.

Presentation issues have to do with effective visual communication including the visual vocabulary, the use of the specific visual elements to convey particular kinds of information, the organization of visual information, and the order in which material is presented to the viewer. Most application domains have their own conventions and symbology that should be used for the visual vocabulary and elements.

Due to lack of space, we refer the reader to overviews on software visualization in the literature [32, 17, 2]. Yet, at least we want to point out that graphs seem to be a "natural" visualization of architecture elements and their (often binary) relations, as confirmed by independent surveys that indicate their popularity [2, 19] (in the end, class and object diagrams in UML are just graphs with predefined semantics and rendering characteristics). In the Compiler, Assessment, and Nokia case studies, graphs were used to convey the information.

The aspect of interaction refers to the way the visualization is constructed. Visualizations range from "hard-wired", where the viewer has no influence on the presentation, to arbitrary redefinition by the viewer. Visualizations should not be static pictures, but should offer querying, zooming in and out, navigation along cross-references and hierarchies, selective hiding, and gathering of transitive relations.

Some of the case studies used "'standard"' elements, such as hyperlinked HTML or PDF documents with embedded UML diagrams (the Nokia and Assessment cases). UML was also used in the Duke's Bank case, but here the diagrams were crafted manually. Simple types of visualization, namely, textual ones and tables, were also used where appropriate (e.g., the Assessment case used tables for metrics). The Nokia, Assessment, and Compiler cases used navigatable visualizations with zooming and filtering capabilities.

We believe that all case studies could have benefited from more advanced and carefully selected means of visualization. Visualization issues were brought up as an afterthought and, hence, the potential of visualization was only partially leveraged. The reason for this shortcoming is simply that the means of presentation chosen in the case studies were mostly opportunistically selected from available tools. The focus in these cases was to solve the problem quickly with available tools. As the initial processes are repeated more often, we expect that their maturity will improve by a more careful consideration of presentation issues.

A particular problem of software architecture is the need to understand a combination of multiple views, which is further complicated when the views are of conceptually different viewpoints. There have been several suggestions to the "view fusion" problem. If the views overlap in some of their entities, one can use certain inferences to map entities with no immediate correspondence to entities in the other view. For instance, Kazman and Carrière use "lifting" operations along containment relations to fuse views [16]. If the entities may be mapped onto source code, one could leverage overlapping source code regions to identify correspondencies between entities [3]. If there is no such simple correspondence, the mapping is typically manual. Hillard, Rice, and Schwarm [12], for instance, systematically cross reference related entities from distinct views and use Ross's model tie process from Structured Analysis to integrate the views [29]. These cross-references are created as part of Symphony's Knowledge Inference in the form of the maps and stored so that the connection among views is made explicit. The cross-references may be implemented and inserted into the views by available frameworks [1, 7]. Multiple views occured in all case studies (in the Compiler case, the mapping and the dependencies propagated from the source to the target entities were also visualized).

Output. The output of the Information Interpretation is a hyperstructure offering a holistic perspective on the software system as a foundation for investigating the concrete architecture's impact on the problems signaled. This hyperstructure includes traceability links between views and links to other software artifacts, such as the source text, relevant documentation, etc. The ideal hyperstructure allows you to explore the system at various levels of abstraction: it lets you zoom in and zoom out between sources and architecture and navigate between views.

7. Concluding Remarks

In this paper we have presented Symphony, a software architecture reconstruction process that: (1) incorporates the state of the practice, where reconstruction is problem-driven and uses a rich set of architecture views; (2) provides guidance for performing reconstruction, including pointers to applicable technology; (3) allows specific reconstructions to be systematically compared; and (4) allows reconstruction approaches to be systematically compared.

Symphony consists of two stages. The first stage (Problem Elicitation and Concept Determination) produces a repeatable and reusable reconstruction strategy that creates the views necessary to address the original problem. Although not an ultimate goal, the problem-dependent viewpoints created or refined in the Concept Determination phase are another reusable output of this stage.

The second stage of Symphony concerns the execution of the reconstruction strategy. This stage operates only at the level of views constrained by the viewpoints created before. Their outcome is the foundation for addressing the problem for which the particular reconstruction is carried out. A secondary outcome is the sequence of mappings from the source views to the target views. This sequence allows one to trace back the information in the views to the artifacts from which they were derived.

This paper also shares real-life reconstruction experience by presenting and comparing different case studies. Reconstruction in practice is problem-driven, using not a fixed set of views but ones chosen to solve the particular problem. The viewpoints used in practice are not confined to the Module viewpoint typically used in the research literature.

Viewpoint selection and definition is an important part of the Symphony process. Using viewpoints to specify the input and output of an activity allows us to decompose the reconstruction process systematically and to review the outcome of each activity. In addition, we can reuse an activity—once defined and used for a reconstruction process—as a building block to compose new reconstruction processes.

Symphony has been applied in academic and industrial case studies and unifies other existing reconstruction techniques and methods. The process model described in this paper allows readers to leverage from that experience when setting up their own architecture reconstruction efforts. We provide a step-by-step methodology that can be followed and give pointers for the selection of appropriate techniques and methods for each of the phases.

In addition, Symphony provides a common reference framework that can be used when classifying and comparing various techniques and methods described in the literature. Such a common reference also helps people to report

on their own reconstruction efforts in a uniform way so that others can easily understand it.

Last but not least, Symphony is a research tool: it helps us to find and demarcate research problems in software architecture reconstruction. For example, Symphony's viewpoint emphasis calls for a catalog of reconstruction methods, techniques, and experiences organized by viewpoints. Moreover, it raises the question what reconstruction-specific viewpoints exist. Symphony's inclusion of mappings between source and target views suggests finding a systematic way to discover and describe such mappings as a key research question. Problems like these are hard to tackle. Symphony makes it possible to address them on a case-by-case basis, offering its process model as a way to classify and compare results.

Acknowledgements Arie van Deursen and Leon Moonen received partial support from ITEA (Delft University of Technology, project MOOSE, ITEA 01002), and SENTER (CWI, project IDEALS, hosted by the Embedded Systems Institute).

References

[1] K. M. Anderson, R. N. Taylor, and E. J. Whitehead Jr. Chimera: hypertext for heterogeneous software environments. In *Proc. European conference on Hypermedia technology*. ACM, 1994.

[2] S. Bassil and R. K. Keller. Software visualization tools: Survey and analysis. In *Proc. Int. Workshop on Program Comprehension (IWPC)*, pages 7–17. IEEE CS, May 2001.

[3] M. P. Chase, D. Harris, and A. Yeh. Manipulating recovered software architecture views. In *Proc. Int. Conf. on Software Engineering (ICSE)*, pages 184–194. ACM, 1997.

[4] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[5] A. van Deursen and T. Kuipers. Building documentation generators. In *Proc. Int. Conf. on Software Maintenance (ICSM)*, pages 40–49. IEEE CS, 1999.

[6] A. van Deursen and T. Kuipers. Source-based software risk assessment. In *Proc. Int. Conf. on Software Maintenance (ICSM)*. IEEE CS, 2003.

[7] P. Devanbu, R. Chen, E. Gansner, H. Müller, and A. Martin. Chime: Customizable hyperlink insertion and maintenance engine for software engineering environments. In *Proc. Int. Conf. on Software Engineering (ICSE)*. ACM, 1999.

[8] L. Ding and N. Medvidovic. A light-weight, incremental approach to software architecture recovery and evolution. In *Proc. Working Conf. on Software Architecture (WICSA)*, pages 191–200. IEEE CS, 2001.

[9] L. Feijs, R. Krikhaar, and R. van Ommering. A relational approach to support software architecture analysis. *Software Practice and Experience*, 28(4):371–400, 1998.

[10] P. J. Finnigan, R. C. Holt, I. Kalas I, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, Oct. 1997.

[11] G. Y. Guo, J. M. Atlee, and R. Kazman R. A software architecture reconstruction method. In *Proc. Working Conf. on Software Architecture (WICSA)*, pages 15–33, 1999.

[12] R. F. Hillard II, T. B. Rice, and S. C. Schwarm. The architectural metaphor as foundation for system engineering. In *Proc. Ann. Symp. of the Int. Council on Systems Engineering*, 1995.

[13] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Object Technology Series. Addison Wesley, 2000.

[14] R. C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *Proc. Working Conf. on Reverse Engineering (WCRE)*, 1998.

[15] IEEE P1471-2000. IEEE recommended practice for architectural description of software-intensive systems, 2000.

[16] R. Kazman and S.J. Carrière. View extraction and view fusion in architectural understanding. In *Proc. Int. Conf. on Software Reuse (ICSR)*, 1998.

[17] C. Knight and M. Munro. Mediating diverse visualisations for comprehension. In *Proc. Int. Workshop on Program Comprehension (IWPC)*, pages 18–25. IEEE CS, May 2001.

[18] R. Koppler. A systematic approach to fuzzy parsing. *Software Practice and Experience*, 27(6):637–649, 1997.

[19] R. Koschke. Software visualization in software maintenance, reverse engineering, and reengineering: A research survey. *Journal on Software Maintenance and Evolution*, 15(2):87–109, 2003.

[20] R. Koschke and D. Simon. Hierarchical reflexion models. In *Proc. Working Conf. on Reverse Engineering (WCRE)*. IEEE CS, Nov. 2003.

[21] P. K. Laine. The role of sw architectures in solving fundamental problems in object-oriented development of large embedded sw systems. In *Proc. Working Conf. on Software Architecture (WICSA)*, 2001.

[22] K. Mens. *Automating architectural conformance checking by means of logic meta programming*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2000.

[23] L. Moonen. Generating robust parsers using island grammars. In *Proc. Working Conf. on Reverse Engineering (WCRE)*, pages 13–22. IEEE CS, Oct. 2001.

[24] L. Moonen. *Exploring Software Systems*. PhD thesis, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, Dec. 2002.

[25] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, July 1996.

[26] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE CS Transactions on Software Engineering*, 27(4):364–380, Apr. 2001.

[27] A. Postma. A method for module architecture verifi cation and its application on a large component-based system. *Information and Software Technology*, 45:171–194, 2003.

[28] C. Riva. Architecture reconstruction in practice. In *Proc. Working Conf. on Software Architecture (WICSA)*, 2002.

[29] D. T. Ross. Removing the limitations of natural languages (with the principles behind the RSA language). In *Proc. the Software Engineering Workshop*. Academic Press, 1980.

[30] C. Stoermer, L. O'Brien, and C. Verhoef. Practice patterns for architecture reconstruction. In *Proc. Working Conf. on Reverse Engineering (WCRE)*. IEEE CS, 2002.

[31] S. Tilley, S. Paul, and D. B. Smith. Towards a framework for program understanding. In *Proc. Int. Workshop on Program Comprehension (IWPC)*, pages 19–28. IEEE CS, 1996.

[32] M. Wiggins. An overview of program visualization tools and systems. In *Proc. 36th Annual Southeast Regional Conf.*, pages 194–200. ACM, 1998.

A. Case Study Descriptions

In this appendix we provide descriptions of the reconstruction case that formed the starting point for Symphony.

A.1. Compiler Case Study

Problem Elicitation: The purpose of the reconstruction in the compiler case study was to find out how well the actual architectures of existing compilers conform to the canonical architecture of compilers. Two large and complex compilers were analyzed, namely, *sdcc*, a C compiler for microcontrollers (100 KLOC), and *cc1*, the C compiler (400 KLOC) of the Gnu compiler collection (gcc).

Resources: The ressources available in this case study were the source code of the two compilers, a compiler reference architecture as described in textbooks on compiler design, our own background knowledge in building such compilers, and the Bauhaus toolkit for program analysis and architecture reconstruction.

Source and Target Viewpoints and Mapping An extension to the reflexion method was used to perform the comparison between the actual architectures and the reference architecture. The source and target viewpoints are described in Figure 7.

Source Viewpoint	Target Viewpoint
dir *contains* dir	module *convergence* module
dir *contains* module	module *divergence* module
module *contains* declaration	module *convergence* module
declaration *depends-on*$_a$ declaration	module *contains* module
	module *depends-on*$_a$ module
	module *depends-on*$_h$ module

Figure 7. Viewpoints in Compiler Case

The *depends-on*$_a$ denotes the actual references between declarations as derived from the source code. The types of references extracted are summarized in Figure 8.

Iterations: There were two iterations at the design stage of Symphony. In one of them, the target viewpoint was refined in adding hierarchical modules. The original reflexion method by Murphy and colleagues has only non-nested modules in the hypothetical module viewpoint. Typical compiler architectures are described at varying levels of detail, however. For this reason, we extended the target viewpoint by allowing nested modules. The extension required us to adjust the original definitions of convergence, divergence, and absence in the reflexion model. A second refinement was required in the source viewpoint. Both compilers use not just direct calls, but also calls through function pointers. These indirect calls were added to the source viewpoint.

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 8. Extracted reference types.

There were several iterations at the execution stage, in which the mapping from source entities to modules in the hypothesized target view as well as the hypothetical target view were refined.

Data Gathering The data was gathered through parsing and global name and type resolution. To resolve the function pointer calls statically, we used a Steensgaard-based points-to analysis.The analyses were all supported by the Bauhaus toolkit.

Information Interpretation: The resulting reflexion model can be presented naturally as a graph. The Bauhaus toolkit offers a graph visualization and navigation tool that supports nested graphs and source code views. The nodes and edges in the graph are traversable and linked to the source. This way, the resulting reflexion model could be browsed and the absences and divergences investigated easily.

The outcome of this case study was the comparisons of the two actual architectures to the reference architecture as well as the comparison between the two actual architectures themselves. Different architectural patterns and architectural anomalies were found in the actual architectures.

A.2. A Symphony for Nokia

Nokia is a worldwide telecommunication company developing telecommunication equipments and terminals. Products are typically organized in product families in order to reduce the development costs and maximize the reuse of the assets. Over the past years, we have developed an architecture reconstruction process for one Nokia product family (hereafter referred as NPF). In this section we recast the NPF reconstruction process [28] to the Symphony process model. Nokia sensitive details have been omitted as much as possible and the diagrams were simplified where possible.

Problem elicitation. NPF evolved from a small set of products to a software platform for developing tens of products. Its evolution is driven by the consolidation of family assets in the platform and the development of new features for the various products. The architects need to clearly understand and control the assets available in the platform and ensure that architectural rules are respected in the products. Their needs can be summarized as follows: (1) comprehending the as-implemented architecture of the products, (2) man-

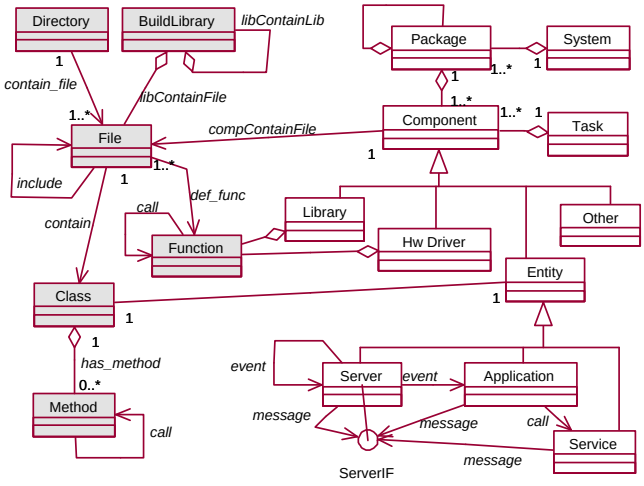

Figure 9. Source and target viewpoints for NPF.

aging the organization of components and their logical dependencies in the platform, and (3) enforcing conformance to architectural rules. The main goal is to provide the architects with up-to-date information by reconstructing the same architectural views that they typically use during design.

Concept determination Discussions with the architects and the current design documentation suggested that the following viewpoints would match most of the architects' needs: *component view* (the logical components, their interfaces and their logical relationships), *task view* (task allocation and inter-task communication), *development view* (organization of the source code and their dependencies), *deployment view* (physical location of components in the processing units), *feature view* (run-time implementation of a feature), *organizational view* (organization of the development activities, such as projects, programs, sites).

The target viewpoint was determined from the reference architecture of NPF (describing architectural styles, rules and conventions for all NPF products) and from discussions with the experts. The target viewpoint for NPF is shown in Figure 9 (gray entities are also in the source viewpoint). This approach guarantees the correct level of abstraction and granularity to satisfy the needs of the architects.

Data gathering. The entities that are directly detectable in the source code (e.g., Directory, File, Function, Server, Application, Service) are recovered by processing the source code either with scripts based on regular expressions or commercial front-ends. The choice of the technology depends on the accuracy we want in the target views. Other entities (e.g., System, Package, Component, HW driver, Library, Other) are recovered by interviewing the architects. For this purpose, we fill a component inventory database with information about components, their type, interfaces (as intended by the architects), ownership, the hierarchical organization of components in packages and the mapping from components to source files. The union of the gathered data and the component inventory represents the source view for NPF.

Knowledge inference The target views are derived from the source view with a series of transformations formalized in relational algebra. In the case of the *component view*, we select the logical components, their interfaces, and their hierarchical organization in packages and calculate their logical dependencies by lifting the low-level dependencies.

Information presentation The views are regularly published in the intranet and are consulted by the development teams in their daily tasks. The views are published in simple and easy-to-interpret formats: hierarchical graphs in Rigi, hyper-linked web pages and UML diagrams in Rational Rose. The diagrams allow the users to browse the various dependencies (high and low level) starting from the top-level packages of NPF.

A.3. Duke's Bank with Symphony

Problem Elicitation: The ultimate goal of the Duke's Bank reconstruction was to determine how best to describe its software architecture and what kind of tools would be useful for reconstructing the architecture of J2EE web appliations. Thus a subgoal was simply to understand its architecture.

Resources: The resources available were an online tutorial (a document) and the source code for Duke's Bank. The Conceptual, Module, Execution, and Code architecture views were used as the starting point for determining the target viewpoints. Only general-purpose tools such as grep, emacs, Rational Rose, etc. were used for this exploratory reconstruction.

Iterations: As shown in Figure 10, the process began with four iterations, each having a different target viewpoint. The fifth iteration served to refine the Module view based on knowledge gained during the previous iteration. Similarly, the sixth iteration served to refine the Conceptual view to reflect the final Module view produced in the previous iteration. In the seventh iteration, both Code and Execution architecture views were refined in order to reflect the modules found in the Module view. (In the first and second iterations, each class was assumed to be a separate module.)

Data Gathering: Data Gathering used general-purpose tools and much manual work.

Code arch view: examine directories.

Execution view: observe executing application; review J2EE reference documents.

Module view: use Rose, grep, emacs, and read source code.

Conceptual view: observe behavior of application; read source code; read diagrams in Duke's Bank tutorial.

	Starting Viewpoint	Target Viewpoint	Source Viewpoint	Mapping
First iteration:	Code arch view	module allocated-to file	directory structure; package structure; file names	class name matches file name; Java rules about packages; treat each class as a module
		module in deployable-file (e.g. .jar file)		
Second iteration:	*cross-view*	module instantiated-in container	build relationships; deployable-file read-by process (JVM)	rules about file types instantiated in container types
	Execution view	container executes-in process; process alloc-to host-machine		
Third iteration:	Module view	module provides interface; module uses interface; module uses module; module contains module	method calls method; class fwd/incl/etc. class; class contains method	rules for combining methods into interfaces and classes into modules; identify "logical" interfaces
	trace target to source	module contains class		
		module fwd/incl/etc. module		
Fourth iteration:	Conceptual view	input-form sent-to component; component displays page; port attached-to role; component has port; connector has role	navigation map; tracing through source code; page contains input-form	identify connectors; rules about the execution behavior of J2EE types (servlet, jsp, JavaBean, etc.)

Figure 10. Viewpoints and Mapping Rules Used in Duke's Bank

Knowledge Inference: Knowledge Inference was done manually and relied heavily on domain knowledge.

Code arch view: straightforward.

Execution view: apply J2EE rules for component types.

Module view: create interfaces from methods; combine classes into modules; identify data-transfer classes and helper classes; refine by combining forw/incl with uses; refine by adding module knows module; refine by adding logical interfaces.

Conceptual view: determine connectors; split pages into input forms; group processing by tier.

Information Interpretation: The following standard techniques were used to document the software architecture of Duke's Bank.

Code arch view: tables.

Execution view: diagram.

Module view: set of diagrams (each focuses on a different part of the system, e.g. Dispatcher and all its relationships is a separate diagram).

Conceptual view: set of diagrams (user navigating to a page and the resulting response, e.g. ATM link, ATM page, ATM response).

In addition to resulting in a documented software architecture, the reconstruction uncovered some open questions that were to be resolved by further reconstructions of other applications. One issue was the small granularity of components in the conceptual view and modules in the module view, which is driven by the different characteristics of different J2EE class types (servlet, jsp, JavaBean, entity bean, etc.). While existing extraction tools could be adapted to extract most of the information for the source view, certain information is implicit and is not amenable to automated extraction. The mapping could be made more formal, although certain part will likely always need manual input (identifying components, connectors, interfaces, and modules).

A.4. Assessment Case

Problem Elicitation The reconstruction was conducted to support the assessment of the quality characteristics of a commercial software product written mostly in Cobol that was being customized for a particular client. Characteristics investigated include reliability, maintainability, and data integrity.

Resources Resources available in the reconstruction included documentation at various levels of abstraction and the full source code. Moreover, selected stakeholders were available for interviews, such as

- the lead architect from the supplier
- representatives from the teams responsible for data migration, system customization, and deployment
- the problem owner / business representatives.

Views The most important views included the module, data, customization, and component-connector views. Selected source and target relations for the data and module view are shown in Figure 1.

The viewpoints were chosen to help address the concerns raised by the client purchasing and customizing the system. The target model recovered helped to identify layering violations, data integrity checks that were bypassed, and ad hoc mixture of custom and product code complicating upgrades to future product releases.

Iterations The model was evolved and populated in a number of iterations:

1. Organize a stakeholder workshop, describe the problems discussed, identify relevant information, get first impression for relevant views.
2. Do automated source code analysis with already existing Cobol redocumentation tools such as DocGen [5].

These tools are based on data gathering using a Java regular expression library, inference using SQL querying and Java programming, and presentation using graph visualization, HTML, tabular information, and metrics.

3. Analyze the layers in the module view. Partition programs and copybooks according to naming conventions.

4. Analyze table usage for the data view. Discovered that the DocGen table analysis is insufficient for the case at hand. Additional information required included joins, indexes, and integrity checks. Extracted and presented using Perl.

5. Analyze the customization code and variation points. Partition system into product and custom code based on naming conventions. In addition, try opportunistic search for customer names / expected customization key words in product to find violations.

6. Client-server protocol analysis. Identify servers and clients based on documentation. Analyze scalebility of underlying communication protocol by reverse engineering this protocol from the C code (by hand). Criticize, and propose alternative (standard) protocols.

The last four steps were not done strictly sequential, but by two different people working in parallel. Each step also included browsing and searching the models obtains thus far, focusing in particular at trends and outliers (file size, revision istories, fan in, fan out, number of table columns).