

A Software Architecture Reconstruction Method

George Yanbing Guo¹, Joanne M. Atlee¹ & Rick Kazman²

¹*Department of Computer Science, University of Waterloo, Waterloo, ON, Canada*

²*Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, U.S.A*

Keywords: Design recovery, Reverse engineering, Software Architecture Analysis, Design Patterns, Pattern Recognition

Abstract: Changes to a software system during implementation and maintenance can cause the architecture of a system to deviate from its documented architecture. If design documents are to be useful, maintenance programmers must be able to easily evaluate how closely the documents conform to the code they are meant to describe. Software architecture recovery, which deals with the extraction and analysis of a system's architecture, has gained more tool support in the past few years. However, there is little research on developing effective and efficient architectural conformance methods. In particular, given the increasing emphasis on patterns and styles in the software engineering community, a method needs to explicitly aid a user in identifying architectural patterns. This paper presents a semi-automatic method, called ARM (Architecture Reconstruction Method), that guides a user in the reconstruction of software architectures based on the recognition of patterns. Once the system's actual architecture has been reconstructed, we can analyze conformance of the software to the documented design patterns.

1. INTRODUCTION

A *software architecture* is a high-level description of a software system's design, often a model of the software's components (e.g., objects, processes, data repositories, etc.), the externally visible properties of those components, and the relationships among them (Bass, *et al.*, 1998). The concept of software architectures has received considerable attention lately, and developers are starting to document software architectures. However, the living architecture of a software system may drift from the documented

architecture if architecture changes are made during software implementation or maintenance and no similar effort is made to maintain the architecture documents. Although architectural integrity could, in theory, be enforced by a rigorous review process, in practice this is seldom done.

Architecture conformance analysis can be used to evaluate how well the architecture of a software system corresponds to its documentation; it can also assist in keeping the architecture documents up to date. Some progress on this problem has been made at the source file and module levels, where the software's call-graph is extracted from source code and compared with the expected call-graph (Murphy, *et al.*, 1995), (Woods & Yang, 1995). In addition, a number of reverse engineering tools have been developed to automatically extract, manipulate, and query source model information (e.g., REFINE (Reasoning, -), Imagix (Imagix, -), Rigi (Wong, *et al.*, 1994), (Storey, *et al.*, 1996), LSME (Murphy & Notkin, 1996), IAPR (Kazman & Burth, 1998), RMTTool (Murphy, *et al.*, 1995)).

Design patterns are an attempt to codify solutions to recurring problems, to make routine design easier. In an architecture, design patterns prescribe specific abstractions of data, function, and interconnections. Automated conformance analysis of newer software architectures is actually complicated by the use of design patterns and architectural styles in architecture documents. While this statement seems at first to be contradictory to the thesis of this paper, the complication stems from the fact that extraction tools extract *code-level* information, not architectural information. Hence, the analyst needs some way to map from the low-level extracted information up to architectural concepts. To properly analyze the architectures of systems developed using design patterns, we need tools and techniques for recognizing instances of pattern-level abstractions.

This paper shows how code-level extraction can feed into pattern-based architecture conformance analysis. We present a semi-automatic analysis method, called ARM (Architecture Reconstruction Method), that codifies heuristics for applying existing reverse-engineering tools (for reasoning about code-level artifacts) to the problem of recognizing more abstract patterns in the implementation. Once the system's actual architecture has been reconstructed, we can analyze conformance of the software to the documented design patterns.

Following this introduction, Section 2 provides a review of software architecture recovery. Section 3 describes ARM in detail. Evaluation of the method with case studies is presented in Section 4. Finally, Section 5 summarizes this work and proposes future research.

2. SOFTWARE ARCHITECTURE RECOVERY

Software architecture recovery can be divided into two phases:

1. identification and extraction of source code artifacts, including the architectural elements; and
2. analysis of the extracted source artifacts to derive a view of the implemented architecture.

The extracted source artifacts form a *source model*, which comprises a collection of *elements* (e.g., functions, files, variables, objects, etc.), a set of *relations* between the elements (e.g., “function calls function”, “object A has an instance”) and a set of *attributes* of these elements and relations (e.g., “function calls function N times”), to represent the system (Kazman & Carriere, 1998).

2.1 Architecture Recovery Frameworks

There exist many source model extraction tools, such as LSME (Murphy & Notkin, 1996), SniFF+ (SniFF, -), ManSART (Yeh, *et al.*, 1997) and Imagix (Imagix, -), that parse code fragments and extract source model elements, relations and attributes. Tools that use relational algebra to infer new facts from existing facts, such as SQL and Grok (Holt, 1998), can be used to manipulate and analyze source model artifacts. Tools for extracting and analyzing software architectures, such as Rigi (Wong, *et al.*, 1994), CIA (Chen, *et al.*, 1990) and SAAMTool (Kazman, 1996), provide not only visualization but also manipulation mechanisms to help the user simplify and navigate through the visual system representation. However, each individual tool or system has its limitations and restrictions in terms of the architecture recovery phases it covers, its support for applications developed in different programming languages and its flexibility in supporting customized analysis.

A *software architecture framework* integrates and leverages multiple tools in an organized structure to facilitate architecture recovery.

Kontogiannis et al. have developed a toolset, called RevEngE (Reverse Engineering Environment), to integrate heterogeneous tools, such as Ariadne (Kontogiannis, *et al.*, 1994), ART (Johnson, 1993) and Rigi (Wong, *et al.*, 1994) for extracting, manipulating and analyzing system facts, through a common repository specifically designed to support architecture recovery.

The architecture recovery framework of the Software Bookshelf project (Finnigan, *et al.*, 1997) provides access to a variety of extractors, such as C Fact Extractor (CFX) and CIA, for source model extraction. Manipulation and analysis of the source model stored in the repository is possible via tools like grep, sort, or Grok, to emit architectures of the subsystems and of the

system. The architecture that Bookshelf produces is a hierarchical structural decomposition of system in terms of subsystems, files, and functions. The architectures can be visualized using tools such as the Landscape Viewer.

The *Dali* architecture workbench (Kazman & Carriere, 1999), is an infrastructure for the integration of a wide variety of extraction, manipulation, analysis, and presentation tools. The architecture recovery work presented in this paper was performed using Dali.

2.2 The Dali Workbench

Dali's architecture is shown in Figure 1, where the rectangles represent distinct tools and lines represent data flow among them.

Source model extraction can be performed by a variety of lexical-based, parser-based or profiling-based tools that produce static or dynamic *views* of the system under examination. A *view* is a source model extracted by a single extraction tool or technique. A static view contains static source artifacts extracted from source code. A dynamic view contains dynamic elements including dynamic typing information, process spawning and instances of interprocess communication (IPC). These extracted views are stored in a repository, currently a relational database. The various extracted views can be fused together into *fused views* (Kazman & Carriere, 1998).

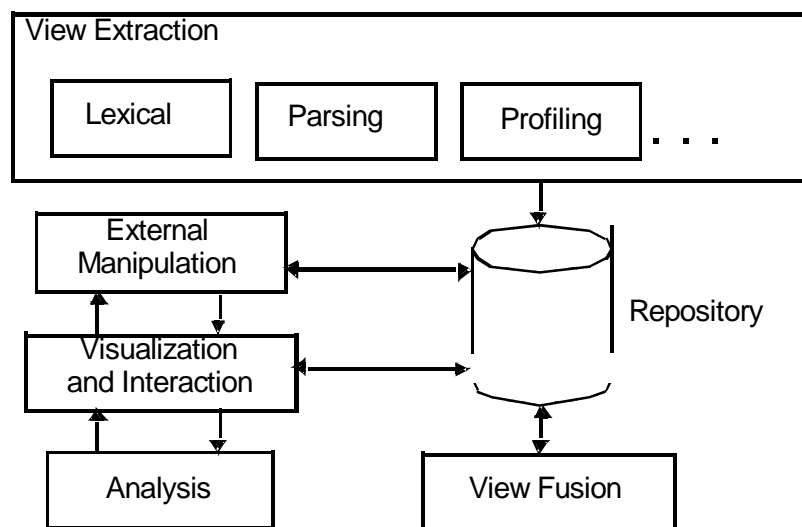


Figure 1: The Dali Workbench

Visualization tools can be deployed in Dali to present the source model and the result of architecture analysis. For example, Rigi is used to present

systems as a graph with nodes denoting the artifacts and arcs representing the relations between them. Dali supports various external manipulation and analysis tools, such as Grok, IAPR (Kazman & Burth, 1998), and RMTTool (Murphy, *et al.*, 1995). The system view can be exported to these tools and the analysis results can be added back to the repository. Using Rigi's command language, new tools can be added in Dali and a software analyst can choose among tools when performing an analysis task. Dali does not rely on having an Abstract Syntax Tree (AST). This allows it to cope with architecture analysis on systems that can not be parsed.

2.3 Architecture Recovery Methods

Automated tools and frameworks can be used to extract and reason about code-level facts. However, human input is needed to extract and infer facts about higher-level abstractions (e.g., design patterns). An *architecture recovery method* defines a series of steps, and the pre/post conditions for each step, to guide an analyst in systematically applying existing reverse engineering tools to recover a system's architecture.

Most current architecture recovery methods are based on a system decomposition hierarchy to reason about software architecture by looking at the relations (calls and uses relations in most cases) between the subsystems, between the files and between the functions (Portable Bookshelf, -). However, it is difficult to use these methods to recover architectures that are designed and implemented with design patterns. As design patterns are described as well-defined structures with constraint rules, a pattern-oriented architecture recovery method must incorporate the design pattern rules as well as structural information such as the system decomposition hierarchy.

Shull *et al.* developed the BACKDOOR analysis method to recognize design patterns in object-oriented systems (Schull, *et al.*, 1996). This method uses a general abstract pattern description, rather than an application-specific pattern instantiation, to guide pattern recognition, and hence could be ineffective in producing accurate results. The pattern definition, detection and evaluation in this method are performed manually, which makes the method primarily applicable to small systems.

3. ARCHITECTURE RECONSTRUCTION METHOD

To assist software architecture recovery of systems designed and developed with patterns, we developed the Architecture Reconstruction Method (ARM) - a semi-automatic analysis method for reconstructing architectures based on the recognition of architectural patterns.

ARM is depicted in Figure 2. As indicated by the dashed boxes in this figure, ARM consists of four major phases:

1. Developing a concrete pattern recognition plan.
2. Extracting a source model.
3. Detecting and evaluating pattern instances.
4. Reconstructing and Analyzing the architecture.

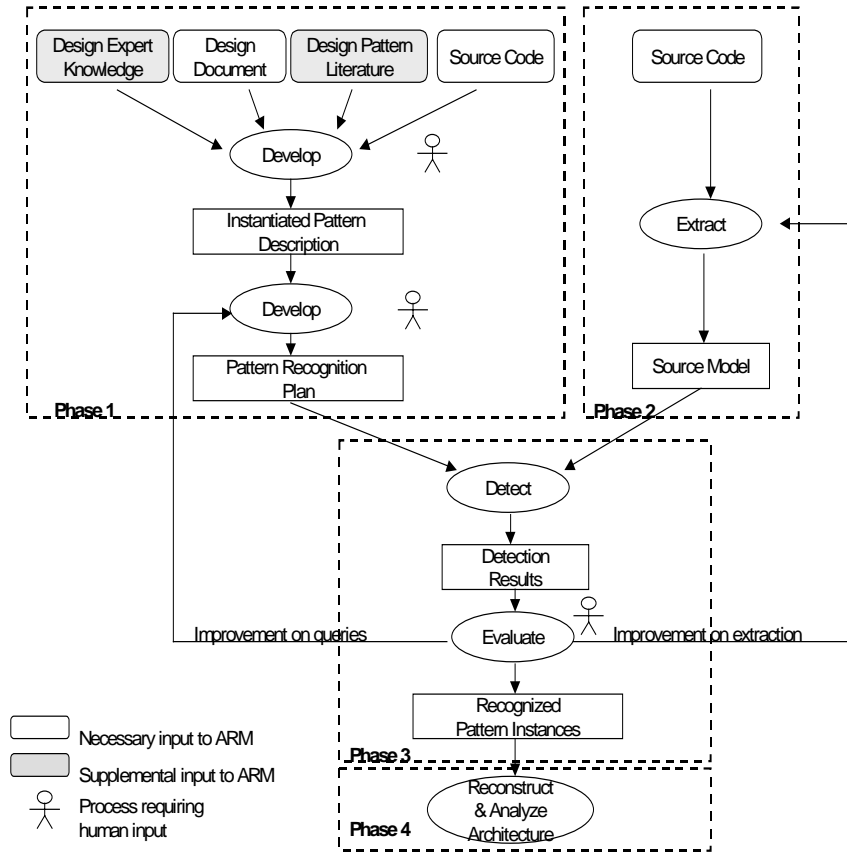


Figure 2: Pattern Recognition Process Flow Chart

3.1 Developing a Concrete Pattern Recognition Plan

Constructing a pattern recognition plan consists of three steps. The first is to develop an instantiated pattern description. By *instantiation*, we mean a concrete pattern description, with all the pattern elements and their relations

described in terms of the constructs available from the chosen implementation language.

Starting with a design document, one can manually determine the patterns used in the design and can extract the *abstract pattern rules* - the design rules that define a pattern's structural and behavioral properties. Pattern descriptions found in the design pattern literature, e.g., (Buschmann, *et al.*, 1996), or obtained from humans who are familiar with the system design can be used to supplement these rules. Using these abstract pattern rules as a guide, one can then examine the source code of several potential pattern instances to derive the corresponding *concrete pattern rules* - the implementation rules that realize abstract pattern rules using data structures, coding conventions, coding methods and algorithms. Such concrete pattern rules can be recognized via syntactic cues, such as naming conventions and programming language keywords, or an analysis of data access and control flow.

An instantiated pattern description is a specification of the concrete pattern rules written in Rigi Standard Format (RSF) (Wong, *et al.*, 1994). A clause in RSF is a tuple (relation, entity1, entity2), which represents the relationship *entity1 relates to entity2*.

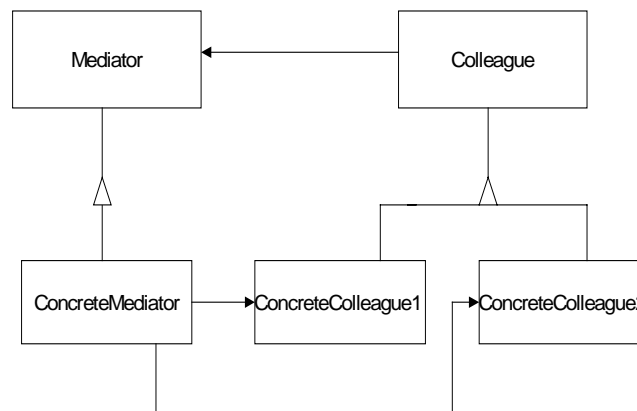


Figure 3: Mediator Design Pattern

For example in the *Mediator design pattern* (see Figure 3), a *Mediator* component serves as the communication hub for all the *Colleague* components. An abstract pattern rule for this pattern is

“The Mediator component mediates communications between colleague components.”

In one of our case studies, the *Mediator pattern* is implemented in a C++ class where mediator and colleague components are member functions.¹ Based on call sequence information (control flow), the following concrete pattern rule is identified to realize the above abstract rule where function B is a Mediator and function A and C are Colleagues.

“Within a class, function A calls function B and function B calls function C, where functions A, B, and C are distinctive.”

Using RSF, this rule can be formally specified as:

```
((calls, Class1:Func1, Class2:Func2) AND
(calls, Class2:Func2, Class3:Func3) AND
(not_equal, Class1:Func1, Class2:Func2) AND
(not_equal, Class1:Func1, Class3:Func3) AND
(equal, Class1, Class2) AND
(equal, Class1, Class3))
```

If an abstract pattern rule can not be mapped to a concrete pattern rule (e.g., the pattern is defined by complex dynamic attributes), one needs to assess whether it is a *necessary* rule for the pattern recognition task in hand. A *necessary* abstract pattern rule specifies a distinct characteristic of the target pattern. A potential pattern instance that is missing such a characteristic would be disqualified from being an actual pattern instance. Based on the assessment, one may decide to proceed to the next step of ARM if the missing abstract pattern rules are *not* necessary, or to terminate the recognition task if *any* necessary abstract pattern rule is missing in the concrete pattern rules.

The second step is to translate the instantiated pattern description into pattern *queries*, written for one of the query and/or analysis tools supported by Dali. If the concrete pattern rules describe specific types of components and connectors, then tools based on a relational algebra such as SQL are suitable because they provide efficient and accurate matching on specific components and relations (connectors). If, on the other hand, the concrete pattern rules do not specify types of components or connectors, but rather allow for a wide range of possible realizations for a pattern, then tools that support more generalized searching criteria, such as the SAAMTool/IAPR toolset, should be used. A user can use the SAAMTool to specify a pattern as a graph and use attributed subgraph isomorphism provided by IAPR to match patterns.

For example, the *Mediator pattern* description can be translated into an SQL query as follows:

¹ This use of the mediator design pattern is an adaptation of what is found in (Gamma, *et al.*, 1994).

```
SELECT DISTINCT c1.tcaller,
  c1.tcallee as mediator, c2.tcallee
INTO TABLE med
FROM calls c1, calls c2
WHERE c1.tcallee = c2.tcaller AND
  c1.tcaller <> c1.tcallee AND
  c1.tcaller <> c2.tcallee AND
  classname(c1.tcaller)=classname(c1.tcallee)
AND
  classname(c1.tcaller)=classname(c2.tcallee);
```

Finally, a concrete pattern recognition *plan* must be developed to specify the “key” component of the pattern that should be recognized first and the order in which the subsequent components should be detected. The queries for a “key” component should *not* depend on detection of other pattern components. The mediator component in the *Mediator pattern*, for example, serves as the communication hub between colleague components and thus is the key to recognizing this pattern.

If part of the target pattern is designed and implemented using other lower-level patterns, it is necessary to develop concrete pattern recognition plans for each pattern component and the compound pattern.

3.2 Extracting a Source Model

The second phase of ARM is to extract a source model that represents a system’s source elements and the relations between them. The output of this phase is a source model that contains the information that is used for detecting *necessary* pattern rules. For example, Table 1 shows some of the relations that Dali currently extracts from C++ programs (Kazman & Carriere, 1999). The relations needed for detecting the necessary pattern rules of the Presentation-Abstraction-Control (PAC) pattern² (Buschmann, *et al.*, 1996) in our case studies are denoted by *.

One complication is that patterns are revealed at different levels of abstraction (e.g., the function level vs. the class level), thus different parts of the recognition plan may need to be applied to a source model at different levels of abstraction. Using abstraction techniques, such as the *aggregation* technique provided by Dali (Kazman & Carriere, 1999), lower level source model elements can be grouped together into a higher level element without loss of information. Thus one can use it to bring the source model to appropriate levels of abstraction for pattern detection and architecture analysis.

² The PAC pattern is described in detail in section 4.1.

Relation	From	To
calls *	function	function
contains	file	function
defines	file	class
has_subclass *	class	class
has_friend	class	class
defines_fn *	class	function
has_member *	class	variable
defines_var *	function	variable
has_instance *	class	variable
defines_global *	file	variable
var_access *	function	variable

Table 1: Typical Set of Source Relations Extracted by Dali.

3.3 Detecting and Evaluating Pattern Instances

Detecting pattern instances using Dali is an automatic process in which one uses query tools to execute a recognition plan with respect to a source model. After running the recognition plan on the source model using the query tools, the detection output consists of all the pattern instance candidates. Human evaluation of these candidates is required to compare them with the designed pattern instances and determine which candidates are intended, which are false positives and false negatives. A false positive is a candidate which is not designed as a pattern instance, but is “detected” falsely as an instance. A false negative is a candidate which is designed as an instance, but is not detected as one.

One can try to improve the results (i.e., remove false positives and negatives) by modifying either the recognition plan or the source model and reiterating through ARM method. To improve the pattern recognition plan, one may choose another component of the pattern as the anchor and reorder the queries to form a new plan, or refine the query constraints for some of the pattern elements. If the source model extraction caused the deficiencies, an analyst needs to try to improve the extraction process by refining the existing extraction tools to catch the defects and/or incorporating other extraction tools to enhance the accuracy of source model, as described in (Kazman & Carriere, 1998).

However, if the source code is incomplete or if the pattern is defined by complex dynamic attributes, it may be impossible for the recognition technique to precisely detect all pattern instances. The evaluation process ends when Dali can detect the maximal set of true pattern instances, and the

human analyst can explain the presence of false positive and the absence of false negative instances. The output is the set of validated pattern instances.

3.4 Reconstructing and Analyzing the Architecture

In the final step, the analyst uses a visualization tool, such as Rigi, to align the recognized architectural pattern instances with the designed pattern instances, organizing the other elements in the source model around the detected instances. The resultant architecture can be analyzed for deviations from the designed architecture.

4. CASE STUDIES

In an attempt to evaluate the applicability and generality of ARM, we applied it to two case studies where the systems were designed and developed with specific architectural patterns in mind. We obtained both source code and design documents for the applications from Informatique et Mathematiques Appliquees de Grenoble (IMAG) Institute in France.

4.1 SupraAnalyse System

The first application is a 25 KLOC system written in C++, called SupraAnalyse, that analyzes experimental data about human subjects' behavior when performing tasks using an interactive system (Lischetti & Coutaz, 1994). SupraAnalyse uses the Presentation-Abstraction-Control (PAC) pattern in its architectural design and implementation. The PAC pattern defines a structure for interactive software systems in the form of a hierarchy of co-operating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control. The *Presentation* component provides the visible interface; the *Abstraction* component maintains and accesses the data model; and the *Control* component manages intra-agent communications between the Presentation and Abstraction components and inter-agent communications with other PAC agents.

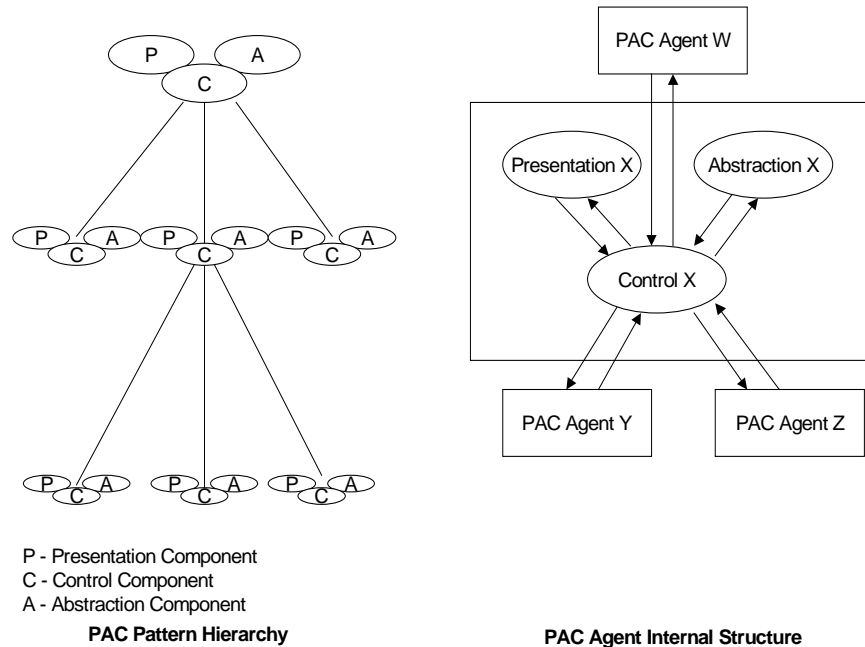


Figure 4: PAC Patterns

Based on the instantiation of PAC patterns in SupraAnalyse, we first developed a recognition plan which consists of a sequence of SQL queries. Because the internal structure of a PAC agent is designed using the Mediator pattern, we iterated the recognition plan development phase to fully specify a sub-plan for recognizing the Mediator pattern. Several extraction tools, including LSME (Murphy & Notkin, 1996), Imagix (Imagix, -) and SNiFF+ (SNiFF+, -), were used to extract a source model that was stored in an SQL database.

Before applying the recognition plan, the source model was simplified to function level and class level abstractions using the aggregation technique. That is, class information such as methods and member variables, was aggregated with class definition; and function information, such as local variable usage, was aggregated with function definitions. PAC pattern components were then detected at function level abstraction, and PAC agents were recognized at the class level.

Evaluation of the detection results was performed to identify false positives and false negatives. For example, the designed PAC agent “Ciment” is identified as a false negative because it can not be aligned to any detected pattern instance candidates. Subsequent iterations of ARM were taken to improve the source model extraction and recognition plan. We ended the iteration process when all false positives and false negatives were

removed or explained by valid causes (such as incompleteness of the source code for the case where some class implementations were missing). For the false positive “Ciment” agent, further study of the source code shows that this designed agent is not implemented.

Finally, we re-constructed the as-implemented architecture (Figure 5) by aligning detected PAC agents with the intended PAC agents in the designed architecture, and grouping the unmatched detected agents together (at the bottom of Figure 5). Architecture conformance was analyzed to identify deviations of the as-implemented architecture from the documented architecture.

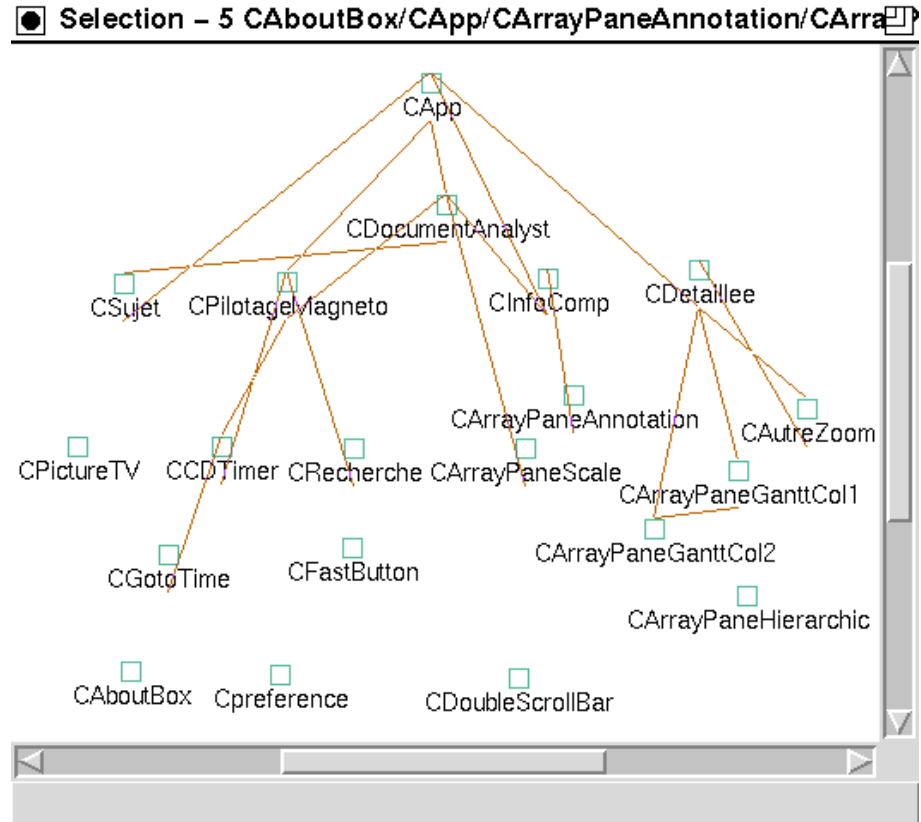


Figure 5: As-implemented Architecture of SupraAnalyse Using PAC Patterns

The as-implemented architecture shows that there are relations that bridge layers of objects and thus violate the design principles of the PAC pattern. For example, agents “CSujet” and “CDetaillee” communicate directly with the top agent “CApp” and thus bridge over the

“CDocumentAnalyst” agent. A further investigation of the layer bridging in the SupraAnalyse system was performed via searching for the *Layer-Bridging* pattern in the PAC agent hierarchy. ARM was applied again for this task. Because a layer may contain *any* type of component and because layer bridging can happen in several types of relations, an SQL pattern recognition plan was deemed inappropriate, since it would have involved listing all possible combinations of component and relation types. Instead we used SAAMTool to construct the Layer-Bridging pattern query as a graph. Nodes in the graph represent *any* type of component and edges represent *any* type of connector. The IAPR tool was then used to process the graphical query on a source model graph---the query posed as a subgraph isomorphism problem (Kazman & Burth, 1998). Three instances of Layer-Bridging pattern were detected. These instances represent problematic areas where the implementation of SupraAnalyse has drifted from the design, when we asked the authors of the system about the layer bridging, they said they were unaware of the presence of the design violations.

4.2 MATIS System

The second case study was conducted on a larger system (77 KLOC) called Multimodal Airline Travel Information System (MATIS): an interactive system which allows the end-user to obtain information about flight schedules using speech, mouse, keyboard, or a combination of these interfaces (Nigay & Coutaz, 1991), (Nigay & Coutaz, 1993). It was implemented in Objective C using the NeXTSTEP Application Development Kit. The primary architectural pattern, the PAC-Amodeus model (see Figure 6) consists of 5 components organized symmetrically around a key component: the *Dialogue Controller (DC)*, which itself is designed using the PAC pattern. The *Functional Core (FC)* maintains domain data and performs domain-related functions. The *Interface with the Functional Core (IFC)* defines a set of interface objects to the Dialogue Controller and maps these interface objects into the formalism of the Functional Core.

The *Low Level Interaction Component (LLIC)* contains the toolkits that implement the physical interface between the user and the application. The *Presentation Techniques Component (PTC)* is a mediator between the Dialogue Controller and the Low Level Interaction Component, and controls the perceivable behavior of the application via input and output commands. The key component *Dialogue Controller* is responsible for task level sequencing, by creating a thread for each request received from PTC and linking the appropriate IFC objects to perform the request. The IFC and PTC components are abstraction layers to enhance portability.

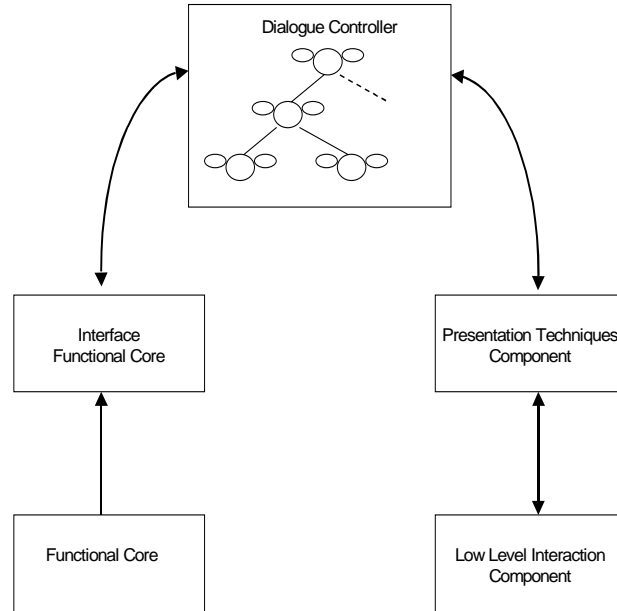


Figure 6: PAC-Amodeus Pattern

Realizing that the DC component is the easiest to recognize as a PAC pattern instance, we formed our recognition plan as follows: first detect the PAC pattern instances and use these to identify the DC; second detect other components and hence the entire PAC-Amodeus pattern using the DC as the “anchor” of the pattern. The PAC pattern queries developed for SupraAnalyse were reused because they were applied to the elements and relations stored in the source model repository and therefore were not dependent on the particular language of implementation. Since the source code contains Objective C files and C++ files, language-specific extractors were developed and used to extract information from the system. A source model was created by combining the extraction results.

Running the PAC pattern queries, we detected 8 PAC agents. Evaluating these PAC agents against the design document shows that the DC is composed of 4 PAC agents; another 4 recognized PAC agents belong to other PAC-Amodeus components. The detected PAC agent information was then added to the repository to enrich the source model. Using the DC as the starting point, other PAC-Amodeus components were subsequently detected by executing the rest of the recognition plan.

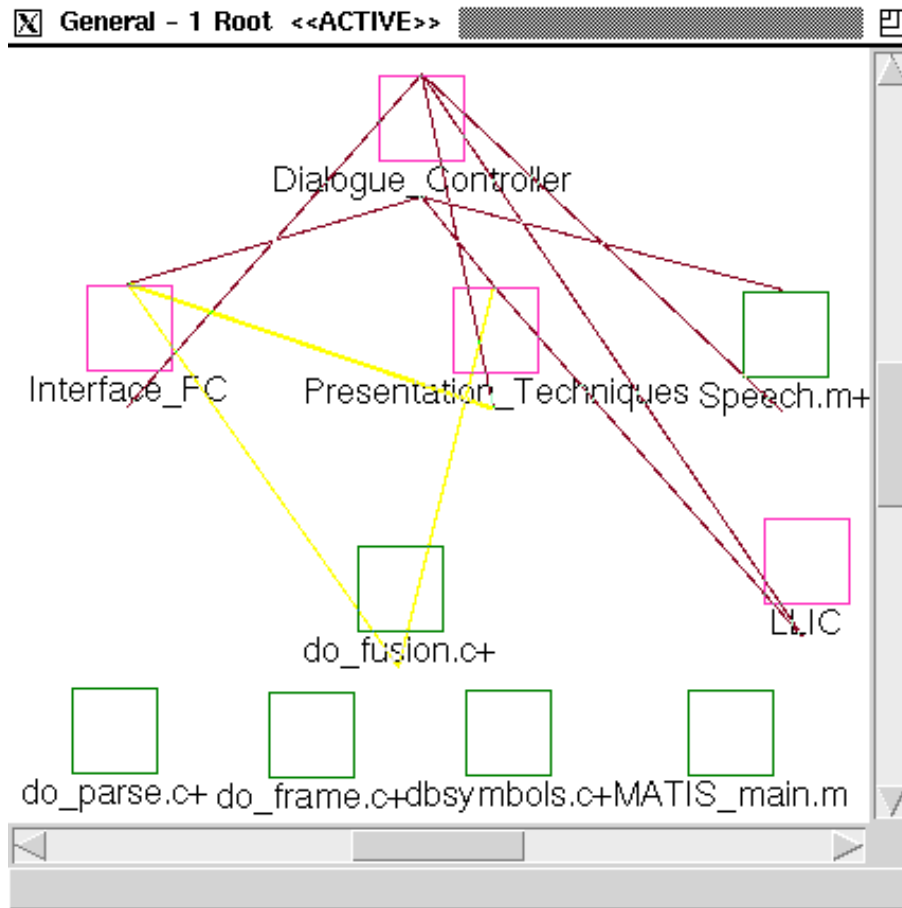


Figure 7: Recognized PAC-Amodeus Pattern in MATIS. Solid lines represent calls relations and shaded lines represent variable access relations

After evaluating the detection results, we reconstructed the implemented architecture of MATIS, shown in Figure 7, using the recognized PAC-Amodeus instance. The PAC-Amodeus structure is evident, but there are several anomalies that need to be investigated. For example, the implemented architecture shows that the FC component, which was designed as an SQL database to process requests sent from the IFC, is missing. Investigation of the source code confirms that these requests are handled by a function in IFC that *simulates* the database processing by returning pre-defined values to certain requests.

As another example, Figure 7 shows that the LLIC calls the DC directly, bridging over the PTC. This clearly violates the design of the PTC component as a layer between the DC and LLIC.

5. LESSONS LEARNED

These case studies both used *patterns* as the primary technique for reconstructing software architectures. The studies demonstrate the usefulness of ARM in assessing, planning and executing pattern recognition tasks. A recognition plan can be laid out to recognize a pattern by a) recognizing nested lower-level patterns first; b) recognizing the pattern's key element; and c) recognizing other elements of the pattern and hence the entire pattern. The pattern matching process is facilitated by using automated query and analysis tools. If an iteration of ARM can not be completed because the exit conditions for a step can not be met, proper assessment of the task should be conducted to identify the causes of detection deficiencies and to provide guidelines for future efforts to improve the pattern detection.

This process is efficient both in terms of the analyst's time and in terms of the amount of processing required to do pattern recognition. Consider, for example, the tools presented here: SQL queries to match patterns are quite efficient (as long as appropriate database indices have been built in advance on the tables of interest), and IAPR pattern-matching, while in principle NP-hard, can be rendered tractable by the judicious use of features that limit the search space, as reported in (Kazman & Burth, 1998).

The time spent in learning and using ARM can be amortized over several architecture reconstruction tasks performed on similar systems (written in the same language and/or using the same design patterns). Queries developed from previous applications of ARM may be reused in executing one or more pattern recognition tasks, as we showed by reusing the PAC pattern queries.

The case studies also provide evidence that static analysis of source code is *not* always sufficient for pattern recognition. Patterns that are implemented using only static mechanisms can be recognized from a source model containing static source artifacts. Patterns whose implementation involves dynamic mechanisms will require extraction of dynamic information, such as process spawning, instances of interprocess communication (IPC), and run-time procedure invocation. In the MATIS implementation, for example, object variables are dynamically typed. That is, an object variable is declared to be a generic type, and assigned specific class types at run time. The best way to solve this problem is to extract the object-type information at run

time. However, due to the lack of access to the NeXT Application Development Kit environment (including its class libraries), we could not execute the system or use dynamic analysis tools to extract the missing object-type information. Fortunately, the object variables were never assigned to more than one type in MATIS. Therefore, we were able to use the static object creation and assignment information to resolve the type of each object. This suggests that extracting dynamic information of a system at run time will sometimes be necessary even in reconstructing a static architecture.

6. CONCLUSIONS

Using design patterns in software design has become a widely used technique for achieving a high quality architecture. Reconstructing architectures of systems that were designed and developed with design patterns has traditionally been approached through manual source code inspections (Schull, *et al.*, 1996). In this paper, we presented ARM - a semi-automatic analysis method - to reconstruct architectures based on recognized instances of design patterns. ARM is an iterative and interpretive process; a human is an integral part of the loop, to evaluate the results and determine what patterns to apply in the subsequent iteration. Two aspects differentiate ARM from other approaches for pattern recognition. One, ARM clearly distinguishes abstract pattern description from concrete pattern instantiation and uses the latter to guide pattern detection. Two, using automated tools to perform pattern matching makes the pattern recognition process less error-prone, compared to manual inspections. Upon the reconstruction of the system's architecture, we can analyze conformance of the software to the documented design patterns.

To further validate the usefulness and applicability of ARM, more case studies need to be conducted on systems in various application domains. Another area of future work is to incorporate approximate pattern matching techniques into ARM. The associated metrics to measure the dissimilarity between the pattern query and the actual pattern instance need to be further studied and established.

Finally, to make ARM still more cost-effective, a pattern knowledge base could be built to provide recognition plans tailored for common instantiations of a given pattern.

REFERENCES

- Bass, L., Clements, P., Kazman, R. (1998), *Software Architecture in Practice*, Addison-Wesley.
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996), *Pattern-Oriented Software Architecture*, Wiley.
- Chen, Y., Nishimoto, M., Ramamoorthy, C. (1990), The C Information Abstraction System, *IEEE Transactions on Software Engineering*, 3, 325-334.
- Finnigan, P., Holt, R. C., et al. (1997), The Software Bookshelf, *IBM Systems Journal*, 36(4), 564-593.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994), *Design Patterns*, Addison Wesley.
- Imagix Corporation, <http://www.imagix.com>
- Portable Bookshelf, <http://turing.toronto.edu/~holt/pbs>
- Johnson, J. (1993), Identifying Redundancy in Source Code Using Fingerprints, *Proceedings of CASCON '93*, 171-183.
- Kazman, R., Abowd, G., Bass, L., Webb, M. (1994), SAAM: A Method for Analyzing the Properties of Software Architectures, *Proceedings of the 16th International Conference on Software Engineering*, 81-90, IEEE Computer Society Press.
- Kazman, R. (1996), Tool Support for Architecture Analysis and Design, *Joint Proceedings of the SIGSOFT '96 Workshops (ISAW-2)*, 94-97, ACM.
- Kazman, R., Burth, M. (1998), Assessing Architectural Complexity, *Proceedings of 2nd Euromicro Working Conference on Software Maintenance And Reengineering (CSMR)*, 104-112, IEEE Computer Society Press.
- Kazman, R., Carrière, S. J. (1998), View Extraction and View Fusion in Architectural Understanding, *Fifth International Conference on Software Reuse*, 290-299.
- Kazman, R., Carrière, S. J. (1999), Playing Detective: Reconstructing Software Architecture from Available Evidence, *Automated Software Engineering*, 6:2, April 1999, to appear.
- Kontogiannis, K., DeMori, R., Bernstein, M., Merlo, E. (1994), Localization of Design Concepts in Legacy Systems, *Proceedings of International Conference on Software maintenance '94*, 414-423.
- Lischetti, N., Coutaz, J. (1994), *Supraanalyse de supratel*. Technical report, Informatique et Mathématiques Appliquées de Grenoble (IMAG).
- Murphy, G., Notkin, D. (1996), Lightweight Lexical Source Model Extraction, *ACM Transactions on Software Engineering and Methodology*, 5(3), 262-292.
- Murphy, G., Notkin, D., Sullivan, K. (1995), Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 18-28, ACM Press.
- Nigay, L., Coutaz, J. (1993), A Design Space for Multimodal Systems: Concurrent Processing and Data Fusion, *Proceedings of InterCHI '93*, ACM Press.
- Nigay, L., Coutaz, J. (1993), Building User Interfaces: Organizing Software Agents, *Proceedings of ESPRIT '91*, 707-719.
- Reasoning Inc., <http://www.reasoning.com>
- SNiFF+, http://www.seed.arch.adelaide.edu.au/docs/sniff_online
- Schull, F., Melo, W., Basili, V. (1996), *An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems*, UMIACS-TR-96-10, University of Maryland.
- Storey, M., Muller, H., Wong, K. (1996) Manipulating and Documenting Software Structures, *Software Visualization*, World Scientific.
- UIMS Tool Developers Workshop (1992), A Metamodel for the Runtime Architecture of an Interactive System, *SIGCHI Bulletin*, 24(1), 32-37.

- Wong, K., Tilley, S., Muller, H., Storey, M. (1994), Programmable Reverse Engineering, *International Journal of Software Engineering and Knowledge Engineering*, 4(4), 501-520.
- Woods, S. G., Yang, Q. (1995), Program Understanding as Constraint Satisfaction, *Proceedings of the IEEE Seventh International Workshop on Computer-Aided Software Engineering (CASE-95)*, IEEE Computer Society Press.
- Yeh, A., Harris, D., Chase, M. (1997), Manipulating Recovered Software Architecture Views, *Proceedings of ICSE 19*, 184-194, ACM Press.