Technical Report

Software Architecture Recovery

Kamran Sartipi       CS
Professor Kostas Kontogiannis    E&CE
Professor Farhad Mavaddat    CS

May 1999

Software Engineering Group
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1
Contents

1 Software architecture recovery 1
  1.1 Introduction .......................................................... 1
  1.2 Structured query-based software architecture recovery ........ 1
    1.2.1 Choice of the analysis framework ............................ 3
    1.2.2 Modeling the source program ................................. 4
    1.2.3 Incorporating software architecture recovery with reverse engineering paradigm ...................................... 4
    1.2.4 Modeling the high-level specification ....................... 5
    1.2.5 Validating the recovered architecture ....................... 6
    1.2.6 Managing the recovery process .............................. 6
    1.2.7 Relaxing the complexity of the recovery algorithm ....... 7
  1.3 Software architecture features ...................................... 7
  1.4 Expected contribution ................................................ 12
  1.5 Related work .......................................................... 13
    1.5.1 Recognizing architectural aspects .......................... 13
    1.5.2 Approximate matching ......................................... 15
    1.5.3 Behavioral analysis ............................................ 16
  1.6 On-going work .......................................................... 16

2 Architectural design recovery using Data Mining techniques 17
  2.1 Architectural design recovery ........................................ 18
    2.1.1 Data Mining ...................................................... 20
    2.1.2 Frequent itemsets ................................................ 20
    2.1.3 Data representation ............................................. 22
  2.2 A system level query language ....................................... 22
  2.3 Search and control .................................................... 24
    2.3.1 Branch and bound ............................................... 24
    2.3.2 Score valuation .................................................. 27
  2.4 Experimental results ................................................ 28
Appendix: A ................................................................. 33
Chapter 1

Software architecture recovery

1.1 Introduction

Software architectural recovery encompasses various methods for extracting architectural information from some lower level representations of a software system such as source code. Architectural recovery is a two-phase process namely extraction, and analysis [64]. The existing approaches focus on recovering restricted information about the system based on: relational queries [66, 87], interactions between files [33], visualization of the system’s structure, and architectural styles [48, 34]. Despite several attempts for automating the architectural recovery process (i.e., clustering), still few approaches and tools exist [89]. It is rather impossible to define the architecture of a large system at once, hence, the architectural recovery is a gradual process. Software systems usually consist of common patterns in their design which are the basis for the recovery process. Most recovery processes focus on the structural properties of a system, ignoring other important aspects of a software such as high-level behavioral.

Therefore, the architectural recovery problem deals with: devising the necessary methodology and tools for incrementally recovering a system’s high-level properties from its lower level representations using domain knowledge and pattern matching.

1.2 Structured query-based software architecture recovery

We aim at designing the necessary processes, tools, and methods, that enable us to analyze a large software system in order to recover and document the system’s structural and behavioral views. We specially focus on devising methods that allow us for multiple-view user query and approximate pattern matching for the recovery of control and data flow dependencies.

Figure 1.1 illustrates the adopted approach of a cooperative architectural recovery environment. A brief description of the recovery process is provided in the caption of the figure.
Figure 1.1: A cooperative environment for structured query-based software architecture recovery. Based on domain knowledge, available documents, and architectural patterns, a user composes a query specification written in an architectural query language (AQL) which is then translated into a high-level schema. On the other hand, the target system is first parsed and then translated into a source model. The analysis engine then tries to find a matching between the high-level schema and source model by searching and comparison between two models. The analysis engine provides an actual architecture with statistical metrics. Based on this result, the user adjusts and augments the query specification in AQL to capture a larger portion of the target system’s architecture, and then performs another analysis task. This iterative process is the basis for incrementally building the system’s architecture.
Below, different components of the architecture recovery framework in Figure 1.1 are introduced. In each item, a pointer links that component to a more comprehensive section in which first a research issue in software architecture recovery is elaborated, and then the specification of the solution provided by our approach is discussed.

- The environment provides tools to parse and analyze the target system, which is a large software written in one of the environment supported languages. The results are different program representations such as AST and flow graphs (section 1.2.1).

- We devise a source model, as a common schema, and a corresponding method for converting the programming concepts of different languages to the source model representation. This source model provides a convenient search space for the searching algorithm (section 1.2.2).

- We design an architectural query language to capture the user's high-level mental model of the target system about the architectural properties of a software to be extracted. An environment generated parser parses the specification in AQL and generates an AST (section 1.2.3).

- We design a high-level schema and a translation mechanism to map the AQL AST to this schema. A pre-processor semantically analyzes the schema for some properties among the components of the query specification (section 1.2.4).

- We design a mapping mechanism between the architectural concepts and source model concepts of a program as a means for validation testing of the recovery process (1.2.5).

- We devise efficient search and comparison algorithms, as the analysis engine. The algorithm receives the high-level schema and source model as inputs, and returns an actual architecture along with some metrics. An API maps the result of the analysis algorithm to an available visualization tool (section 1.2.6).

- A user, familiar with the domain and AQL, uses the available documents to compose a conceptual architectural query specification, and interprets the resultant actual architecture and associated metrics through a visualization tool (section 1.2.7).

1.2.1 Choice of the analysis framework

The choice of a supporting environment greatly affects a development process in various aspects. An analysis-application development environment encompasses supporting tools for designing, developing, and testing a software processor. Typical facilities of such environments include: parser generator to parse the target system, mechanisms for storing and manipulating the internal representation of the target system, a language for developing the application, and reporting or visualizing facilities. Refine [75], Genova [28], DECODE [25] are examples. Some frameworks provide a tool integration infra-structure for communicating among tools of different vendors, e.g., Dali [53]. Integration of the visualization facilities of
other reverse engineering tools can greatly improve the usability and expressiveness of the developed application tool. For example, PBS [33] or Rigi [5], can be utilized to browse the expected architectural elements, or the actual recovered architecture. Kontogiannis has conducted a survey on experimental and industrial reverse-engineering and re-engineering tools and environments [56].

In our approach, we use the Software Refinery environment [75] with the features that suites our needs for architectural recovery. Software Refinery is an integrated environment for developing software analysis and transformation tools that operate on the database representation of a software. Its major components include: a parser generator, the Refine system, an X11-based user interface toolkit, and an object-oriented database. The database contains the program’s code-objects with the structure of an annotated AST. We also integrate the visualization facilities of Rigi or PBS tool with the Refine environment.

1.2.2 Modeling the source program

In software architecture recovery, an appropriate choice of a source model is central in: recovering desired properties from a program; matching algorithm efficiency; and programming language independence. In general, the preserved information in a program representation and the ease of accessing these information, are the important considerations in selecting a program representation.

In representing a source model, recent trends are towards using AST for preserving all useful information and annotating other information to its nodes [28, 34, 48, 43]. An approach may provide links among the AST nodes and their corresponding parts of data and/or control flow graphs as a convenient and multi purpose search space.

In our approach, the target program, i.e., data for the analysis engine, is a large program in one of the Refine supported languages. Using a pre-defined grammar, Refine parses the program and populates an object database from the language-specific constructs. These objects constitute an annotated AST whose root is the program definition. Other more specific program representations such as flow graphs are also extracted by the Refine system as the explicit views of program.

For each programming language, a pre-processing phase extracts architectural features of the target system, as a subset of the information represented by the AST and flow graphs, and save them in a source model. Therefore, a portion of the whole analysis task is assigned to the pre-processing phase in trade of achieving a language independent algorithm. The abstraction level provided by this source model schema is higher than that of an AST and each concept in this schema abstracts the identical concepts in the source languages.

1.2.3 Incorporating software architecture recovery with reverse engineering paradigm

Despite variants in its definition, software architecture includes some consensused concepts such as elements (i.e., component and connector), views (e.g., data, function), and styles (e.g., pipe/filter, blackboard). A legitimate software architecture recovery process and its
tools must incorporate these notions to some extent. Considering the challenging aspects of a reverse engineering task, the recovery processes focus on a subset of the above notions.

Extracting view oriented information: View-based high-level design of a software system is increasingly important in a development process, section A.2.3. The inherit complexity of dealing with non-structural views of a software system, has practically restricted the architectural recovery activity to structural recovery, e.g., boxes and lines, [62, 5]. Discovering other views are also targeted by some domain-specific approaches, e.g. behavior and process recovery in distributed environment [26, 58]. In this relation, different techniques of static analysis (e.g., clustering, query-based, and knowledge-based analysis), or dynamic analysis (based on run-time data acquisition), may be employed. Research and commercial ADLs describe a software through different sets of architectural features, section A.2.1. A proper collection of these features can be selected to recover a particular view.

Extracting architectural patterns (styles): The term “pattern” emphasizes on the design level aspects of a system, not the implementation specific aspects of a plan. These patterns refer to some high-level relations among implementation constructs in a program which are not easily recovered. These relations contrary to plan representations are not hard coded. In one approach, a query describes a pattern of linked modules, with placeholders defining the modules’ structures and the in between links. A search engine then tries to find matches in the source code. In another approach using recognizers, a small query program, i.e., the recognizer, with a specific mission is delegated for detecting a pattern in a source model [34, 48]. Each recognizer usually embodies the knowledge of several algorithmic concepts. The capability and accuracy of the recognizers in finding high-level relations have been improved using flow analysis and program slicing techniques. For example, binding two instances of client and server object together using matching the port address deduced from the data-flow analysis.

In our approach, we attempt to recover the structural and behavioral views of the system, using a query language and pattern matching techniques.

1.2.4 Modeling the high-level specification

A number of software modeling techniques are used to capture a system’s specification including: entity relation, object modeling, structured modeling, module interconnection, flow graphs, and state-based diagrams [63].

In our approach, the high-level specification is defined as a query in the AQL which is then converted into a high-level schema\(^1\). A query is the specification of the expected (conceptual) architectural structure and behavior in the target system. The AQL syntax and semantics are means for composing a structured and consistent specification which is inspired from the ADLs in section A.2.1. The queries can incrementally build the whole architecture consisting of components (modules) and connectors. Each query specification is separately

\(^1\)The schema has the form of a module interconnection language.
formulated and parsed according to the AQL grammar. An AQL query specification is defined using:

- **Architectural notations**, e.g., component, module, and import/export. The notions such as entities, connectivities, and scope are used for structural view recovery. Temporal operators are used to specify the expected behavioral view of the system. The notions of event, event pattern, and event tracing, are intended for behavioral recovery.

- **Target system signatures**, e.g., MaxArraySize, foo(int, bar): array[real], and a set of wild cards as placeholders, e.g., Max!, "?", "$".

- **Directives** that lead the analysis algorithm to a specific architectural analysis task.

The high-level schema provides ease of search and means for validation testing against the source model. Hence, the schema is an abstraction of the source model.

### 1.2.5 Validating the recovered architecture

Similar to a validation test in forward engineering, a reverse engineering process is expected to prove the conformance of the recovered high-level specification with the actual architecture of the target system. To prove this, a dependency association between the abstract concepts and the source model semantics is needed. In its primitive form, this association is a physical link between a high-level concept and a code fragment in the source program and validation checking is by user’s inspection. A formal approach would define each concept in the high-level specification using the low-level semantics in the source model and try to verify the form of concept decomposition into the source model semantics.

### 1.2.6 Managing the recovery process

Searching for a particular property or groups of related properties in a large database is a cpu intensive process. In some cases, the search algorithms are intractable for a large number of inputs, e.g., finding a subgraph pattern in a graph representation of an architecture [52]. Efficient techniques and heuristics are essential in managing the inherent complexity of architectural recovery tasks. In most cases, proposing heuristics for controlling the computational complexity of the search and comparison algorithms is the major contribution of the approaches in this area [52, 94].

Woods [94] has attempted to generalize the problem of program understanding as an instance of the constraint satisfaction problem. This approach can generalize a majority of the search strategies, and suggests a general form of restricting the search space. The elements of this problem consist of: a set of variables, a finite set of values as a domain of each variable, a set of constraints among the variables restricting the assignments of domain values to variables. A solution instance would be a set of assignments of values to variables satisfying all inter-variable constraints. A library of templates captures the problem domain knowledge. A pattern recovery problem in this context is defined as: given a legacy system, find the mappings between each pattern template and pieces of code. In this context, each
element of a template is a variable, and the set of all source model constructs are the common values for all variables. Two categories of constraints, relating to source model and templates, are considered.

In our approach, we use AI search algorithms which are based on the knowledge extracted from the target system. The recovery algorithm provides approximate matches to the query specification and a similarity measure, computed from a similarity function. This information is investigated by the user, who composes and launches a new query.

The structured nature and the semantics of the AQL queries enables our tool to verify the integrity and consistency of the composed queries such as: i) correct references of the AQL queries to the target system's identifiers; ii) consistency between different parts of a composed query, e.g., disjoint sets of the entities in modules.

1.2.7 Relaxing the complexity of the recovery algorithm

The role of user, as an integral part of an architectural recovery process, is increasing. In fact, the ambitious goal of automating program understanding process is no more supported. Instead, a cooperative environment of human and tool is the most promising solution for relaxing the recovery complexity [25, 33, 5]. This trend necessitates that the domain knowledge be migrated from the library of plans (in traditional program understanding) to the mind of the user, i.e., employing domain experts. With this migration, the mission of the tools have also been shifted from cpu-intensive search heuristics to more information based strategies for providing a variety of information needed by the domain expert [33, 25]. The new terms such as librarian and patron [33], referring to the domain knowledge accumulation for human usage, indicate the importance of a domain expert in the recovery process. The techniques for search and comparison of a pattern with code fragments are still a crucial part in a recovery process.

In our approach, the prospective user is a domain expert. The user should be familiar with: the families of applications in that domain, i.e., reference architecture; the details of the programming language of the target system; and obviously, the AQL and its cooperative environment. With respect to the user, the ADL like query specification has two advantages: i) The queries help the user to directly create a mental model of the expected architecture; ii) the structured syntax and language semantic assist the user in composing relevant and correct queries.

1.3 Software architecture features

In an attempt to specify the design requirements of our AQL, specifically to obtain:

- an optimum list of features to specify a software system architecture,
- a useful and feasible set of architectural views, and
- the form of AQL queries at architectural level,
Figure 1.2: Classification of software features that are used to describe software architectures.
we conducted a survey on a number of experimental and industrial ADLs and reverse engineering tools including: Unicon [81], Rapide [60, 61], ACME [39], ARDEC DSSA [86], Wright [17], ControlH [21], PBS [33, 6], Polylith [73], Genoa [28], and DECODE [74]. As a result, we came up with a classification of essential features used for describing software architecture and a set of architectural views.

In Figure 1.2, four architectural views have been specified, namely structure, behavior, environment, and domain-specific. The notion of views has been defined in section A.2.3. The chosen views are orthogonal and carry most of the important information to be recovered. The features and views in this classification are general properties of programs and encompass the systems in different domains such as information, concurrent, reactive, distributed, command and control, etc. The last view, i.e., domain-specific, is intended to cover particularity of each specific domain, hence it is open.

Structure view

The structural view covers all building blocks and interconnections (glues) that statically describe the architecture of a software system.

The structural view consists of two parts:

• The static features are the property of the source code, hence, can be extracted by static analyzing the source program.

• the snapshot features change over time, hence, represent dynamic aspects of a program. These features can be detected statically by interrupting a running program and registering the program’s context and state. Spawned concurrent processes, class instances (objects), etc. are typical information to be discovered.

The static features are divided into three parts:

• An entity refers to a basic block that constitutes in building a software’s structure. Four entities are distinguishable:

  – A compilation unit is a composite entity corresponding to a particular programming language, e.g., module in Modula-3; file in C, C++, Java; package in Ada. Common software architecture terms such as architecture, component, system, and subsystem, can be defined as aggregation of compilation units.

  – Other entities are simple and relate to the definition and use of functions, aggregate types, and global variables.

• A connectivity refers to an interconnection between two entities. A data flow connection can be established either by referencing, or by passing.

\[\text{Interested readers on view discussion can refer to our draft paper on views [8].}\]
In a reference type of connection, an entity directly references another entity. Examples: importing/exporting types, data, or functions between two modules; binding types, data, or functions to import/export parts of a module; binding formal/actual parameters of a function definition and a function call within a module; accessing a function to types or data. In this context, access by value or reference are treated equally.

In a passing type of connection, binding of formal/actual parameters at call site is performed through sockets (stream or datagram). A mediator such as operating system or stub functions is responsible for message passing.

A control flow connection is done by passing the thread of control from one function (process) to another. This can happen through function calls (local, remote, recursive) or other means for control coupling such as interrupt, exception, synchronization, blocking, etc.

- **Scope** refers to the maximum range that a definition is effective:
  - **Visibility** denotes to the scope hierarchy that is caused through nesting block notations such as: (), {}, begin end. The combination of nested and adjacent blocks creates a variety of visibility relations such as local, global, external, parent, and sibling.
  - **Encapsulation** refers to different access privileges imposed by language constructs for information hiding such as public, private, and protected.

In different programming languages, the combination of visibility and encapsulation of resources causes variances of cases with respect to validity of resource importing or exporting among two modules and their internal nested blocks [7].

**Behavior view**

The behavioral view of a system refers to the services that a system provides to its environment through its interactions with the environment.

The behavioral view can be regarded from two orthogonal aspects of program properties:

- In one aspect, the system’s behavior can be expressed by event traces and pre/post-conditions, which is more appropriate for inspection by an analyst:
  - An event tracing mechanism assumes a (concurrent) state transition model of a system which reacts to the environment’s stimuli and changes its state accordingly. The analyst provides a set of carefully chosen inputs, also known as a *scenario*, to the system and observes the system’s reaction. The system reacts by a sequence of function calls (events) in its call-graph that change the state of the system (transformation of the program’s data). The analyst observes the state change
chain and based on other knowledge about the system, e.g., domain, documents, tools, infers the system’s behavior. This technique is specially useful in dealing with reactive systems [45, 9].

The state changing of a system, as the result of events, may be determined from its output. For example, in telephony systems for each change of state, the system informs the user by a different tone. In some cases a profiling tool, e.g., gprof from Gnu, based on a particular input provides a set of statistics about the time percentages that a system spends in each function along with a trace of function calls. Rapide uses events to simulate a system’s behavior and matches patterns of events among different architectures [60]. One approach uses event traces and compares it with the system specification for detection of software failure [80].

- Pre/post-condition predicates specify the input/output constraints of a function. Some formal specification methods such as Larch are used to verify a system’s specification. One approach uses Larch to find components in a library according to their functionality which is specified through pre/post-condition predicates [96]. The approach does not scale to a large library and complex functionality. Informal information such as texts as pre/post-conditions and comments can help the analyst to understand the functionality of larger components by inspection and composition of the smaller components’ functions. The method can hardly be automated.

- In another aspect, the system’s behavior can be expressed by its temporal and functional properties which are difficult to be analyzed and recovered.

  - The temporal properties\(^3\) of the system’s behavior refer to time-dependent characteristics such as: i) concurrency issues, i.e., process instantiation, synchronization and communication, e.g., rendezvous, resource allocation and deadlock; ii) dynamic allocations, i.e., memory allocation, dynamic scoping, stack of run-time records, dynamic connections between components; etc. These properties are too complex to be recovered.

  - The functional properties of the system’s behavior refers to data transformation characteristics. The way data are manipulated and transported between different locations are interested. The algorithmic aspects of transformation such as computational complexity and memory usage are important issues.

**Environment view**

The environment view of a software system (application) refers to all supporting facilities, including hardware/software and interface, that encompass the system and enable it to operate and provide its services to the environment.

The environment view consists of the following parts:

\(^3\)Also known as dynamic, run-time, and execution
- The platform features describe the hardware requirements of the application for a satisfactorily operation, e.g., cpu type and speed, main/cache memory size and speed, number and quality of the terminals, communication network bandwidth.

- The operating system provides a safe basement for the application to both reside and use the services of the underlying platform via operating system features. Efficiency and confidence are the main characteristics of the operating system’s features. These feature are divided as:
  
  - Kernel: creates a multiuser/multitasking environment for different applications to share and run concurrently; organizes the software’s access to network through sockets; provides resource usage efficiency through buffering mechanism for file system and caching mechanism for memory; provides means for applications to handle concurrency issues and use external devices; etc.
  
  - File system: organizes the creation, references, and deletion of files and their contents; creates a directory/file hierarchy with access privileges; provides a logical view of the physical file system for the applications; etc.
  
  - Window system: provides primitive facilities for constructing flexible, distributed, window-based, mouse-driven, and menu-driven interfaces for applications. Unix X11 and Windows NT are examples. A Window manager, built on top of a window system, provides user-friendly interface to the user. Motif and Open Window are examples.

- The devices support an application program by providing printing, modem connection, and multi-media facilities.

Domain-specific view

The domain-specific view of a software system refers to the features that describe particular properties of a family of systems in that domain.

The features in this view mostly describe the reference architecture in a domain, and common domain knowledge that experts use for better communication among themselves. Section A.2.2 defines DSSA as part of architectural styles.

1.4 Expected contribution

The major contributions of our approach are envisioned to be:

- Designing and developing a query language to compose structured architectural queries, as opposed to the current unstructured queries offered by other approaches. The approaches that we are aware of and were discussed as related works (section 1.5), use queries which by themselves do not impose any structural semantics to the query. De-}
semantic graph, Genova [28]; Harris [48, 47] and Fütem [34, 35], use a Refine-based
language to compose a program like hierarchical query to search the code's annotated
AST. Practical systems such as PBS [33] and Rigi [5] use query programs to search
a relational database of the program facts. Kontogiannis's abstract concept language
uses constraints and placeholders to formulate queries for localizing design level con-
cepts [55]. The plan-based queries, like the last example, are structured, but they
describe a plan concept not an architecture of a system.

- **Incorporating behavioral recovery in a software architecture recovery environment.** The
current methods and tools for architectural recovery focus on the structural aspects
of an architecture, and do not pay attention to its behavior. We intend to use the
event concept, as the encapsulation of all identification and invocation information of
a function call. Then devise a method of discovering a trace of events, and interpret
the system's behavior according to the event traces and patterns. Process recovery is
a closely related subject that addresses a general process recovery using event-traces
[26].

- **Devising heuristics or employing techniques from other disciplines for our searching
and pattern matching problems.** Our approach requires efficient searching algorithms
for the analysis and recovery engine. Heuristics based on dynamic programming [55] or
constraint satisfaction [94, 52] techniques reduce the search space. Search techniques
from other disciplines such as graph theory and data mining would be an important
asset.

*Data mining* is a new and rich domain, encompassing a variety of techniques to discover
useful information for applications such as prediction, estimation, and decision-making
in very large data [31, 10]. There are close similarities between the objectives of
data mining and reverse engineering of large systems; both aim at discovering hidden
properties in a huge amount of data that has some defined structure.

## 1.5 Related work

In this section, we discuss architectural recovery approaches that to some extend relate to
our approach or address some of our problems.

### 1.5.1 Recognizing architectural aspects

A number of approaches view the architectural recovery as the style recognition problem:

**Style recognizers**

Harris tries to find architectural styles (about nine styles) in source code [48]. The method
uses an annotated AST source model as the search domain and an architectural recovery
query language. The query language is built on the top of the Refine language with a minor
change to codify the desired architectural styles. The approach has a pattern recognition flavor, where a number of style recognition queries (around 60) constitute the base of the recognition process. The user's mental model of the architecture is the basis for composing queries. A visualization technique, used to view the file structure and their components, assists the user in creating a first mental model of the architecture. Next, a specialized query is composed to search for specific style related properties in the source model. This query triggers a set of more specific style queries as sub goals, and then reports on the degree of success in recognizing that style. Program slicing and code coverage analysis are used to retrieve the recovered program fragments that implement the desired style, and provide statistics about the level of coverage.

The approach is similar to ours in some aspects such as the use of queries to recognize architectural aspects such as styles. However the difference are: the queries partially and implicitly detect the structure of the system, and moreover, we aim at revealing the system's behavior as a more ambitious view of the system.

Views and styles

Fiutem [34, 35], like Harris, uses recognizers and flow analysis techniques in architectural recovery. His expected architecture has a hierarchical model with components and connectors at different levels, e.g., system, program, and module, and different views at each level, e.g., system, module, task, code, data/call graphs. The system view represents concurrent communicating processes, i.e., instances of programs, as a style. A task view represents the style of threads of controls and their spawning relation. A code view represents source files, directories, libraries, etc. A number of recognizers have been defined using Refine reengineering environment to find preliminary architectural properties which are codified as plans. Higher-level concepts, such as styles, are recovered using lower-level ones. The recovery process, having a hierarchical characteristic, is controlled and visualized via a graphical user interface provided by Refine tool.

This approach is interesting in attempting to illustrate a multiple view of a system. Again, the use of recognizers is similar to our approach.

Layered style

Another approach by Girard [43] effectively recovers the layered architecture of a candidate system (if any). The technique consists of three steps in order to collapse the system's functions, global types and variables into layers. First, using transformations on call graph, mutual-recursive nodes and atomic components, e.g., recognized ADT, are collapsed into single nodes. A dominance graph analysis is then converts the call graph into a dominance tree. A series of interpretation guidelines are then applied to convert the dominance tree into different layers of architecture. The method has shown good results in three case studies.
Cooperative environment

Murphy and Notkin have proposed the software reflexion model to assist the user in testing his/her mental model of the system [66]. The user uses a textual declarative form to define his/her high-level model of the system, and link this model to the source model. The source model is a call graph or an inheritance hierarchy. A software reflexion model is then computed to determine where the user’s high-level model conforms with the source model and where does not conform. The user interprets the reflexion model and defines new relations based upon the results. Regular expressions are used in the forms to facilitate the link of a group of source model entities to a high-level entity.

The reflexion model is similar to our approach in aspects such as the role of user in guiding the recovery process, and composing a query to be matched against the actual architecture. However, this approach only checks the validation of the facts in the user’s model, lacking a discovery nature. The forms of queries are also unstructured.

1.5.2 Approximate matching

In a program understanding approach by Kontogiannis [55], solutions to the problems of code-to-code matching and concept-to-code pattern matching have been proposed. Code-to-code matching is mainly used for code clone detection and similarity distance between two code fragments. The user choice of program features as the matching criteria affects the quality of the matching result. Some of the selected features include: software quality and complexity metrics, global/local variable sets, accessed files, keywords, formal parameters bindings. The matching process uses dynamic programming to calculate the cost of matching between the selected pieces of code, in terms of the required insertions, deletions, and substitutions in order that a match occurs. In concept-to-code pattern matching, a concept language captures the abstract properties of a desired code fragment. The pattern matching process is based on the Markov model: the similarity measure between an abstract pattern and a piece of code is defined in terms of the probability that the abstract pattern can generate that piece of code. Dynamic programming has also been used to reduce the required computations. This approach is based on the fact that plans can not cover all implementation variances of a concept, hence, they miss many cases. However, an approximation matching based on a similarity measure recovers most of the implementation variances of a plan.

This approach is similar to our approach in using queries and approximate matching strategy. The use of dynamic programming technique to reduce the search space is a novel and interesting method.

Kazman and Burth introduce an interactive architecture pattern recognition to recover user defined patterns of architectural elements in a system [52]. The system is modeled as a graph of architectural elements, i.e., components and connectors. Each component or connector is defined using common features, namely static and temporal features, causing the elements to be treated in the same way [54]. The user defines an architectural pattern or style as a graph of elements. The tool then searches to find instances of that graph in
the target system model. The tool uses constraint satisfaction paradigm [94] to restrict the search space. The hard/soft features of the elements allow relaxing the exact matching, i.e., approximate matching. The approach provides some statistic about the regularity of a system in terms of its coverage by a particular pattern.

This approach is interesting in addressing the search and comparison problems in large target systems, as well as providing a cooperative architectural recovery environment.

1.5.3 Behavioral analysis

Savor uses a finite-state-machine model of the system’s specification to supervise a system’s behavior at run time. In this approach, a supervisor dynamically collects information form input and output of a system and determines the system’s current state accordingly. The comparison of the system’s current state and the specification detects possible failure of the system’s behavior [80]. Jonathan and Wolf describe a process model discovery technique based on a set of events that are captured from an executing system. The technique inferences the system’s behavior from event-data that encapsulate the dynamic behavior of a process in terms of identifiable, instantaneous actions such as invoking a process or deciding upon the next activity to perform [26]. The paper also evaluates three other methods for process model recovery based on Markov model, algorithmic grammar inference, and neural networks.

1.6 On-going work

We obtained sufficient insight into the problem through an initial prototype (Chapter 3) which then followed by designing an architecture recovery tool based on data mining techniques and a modified version of branch and bound search algorithm [11]. Our next research steps would be as follows:

- testing our new tool using several legacy software systems and enhancing its precision in recovering high portion of the system’s architectural entities,

- devising a formal model for the structural recovery process, and

- studying the application of techniques in data mining, process control, and reactive systems, in recovering behavioral view of a legacy system.

As a regular theme of our work, we would implement other prototypes and test our ideas through experiment. The next chapter discusses the details of the first prototype using the Software Refinery tool as a proof of concept.
Chapter 2

Architectural design recovery using Data Mining techniques

In accordance with the architectural recovery proposal addressed in chapter 1, we implemented a prototype tool. The user specifies an incomplete architectural design model of the system into structured forms imposed by the AQL syntax. A pattern matching engine then finds the closest match between the query-based specification and a collection of source code components. In the proposed approach, the architectural design recovery process consists of two phases. In the first phase, the source code is represented at a higher level of abstraction using Abstract Syntax Trees and entity-relationship tuples (similar to RSF facts). RSF facts are presented in the form of tuples such as, function calls function, function references variable, file includes library [5].

In the second phase, user-defined queries provide a hypothesis about the system's assumed architecture. A pattern matching engine is used for finding the optimal variable instantiation for a given AQL query, generating thus a concrete architecture. Therefore, the query represents, in an abstract way, design concepts corresponding to a “macroscopic design pattern”. For example, a query may focus on obtaining an optimal arrangement of the functions, types, and variables in the modules that conform to the user specified descriptions. The optimal arrangement is obtained with respect to the evidences gathered from the source code using Data Mining techniques.

The proposed approach focuses on facilitating partial matching, a situation that is frequent in practice and has been addressed in a framework of uncertainty reasoning. It differs from other approaches in the area of constraint satisfaction [93], in the sense that the matching process is guided by the association properties of the data that represent the subject system, rather than satisfying constraints. Therefore, instantiating AQL query variables becomes an optimization problem of maximizing association values as opposed to satisfying constraints.

Considering the size of the search space for the pattern matching engine when a large system is involved, the scalability of the approach is a fundamental requirement. In order to limit the search space and speed-up the matching process, we use Data Mining techniques and a variation of the branch and bound search algorithm. In general, we assume that
the user who is familiar with the problem domain, uses domain knowledge to compose queries (conceptual architecture). The pattern matching engine then provides (generates) a substitution $\sigma$ (i.e., a set of bindings of the type ?var/value) for all the variables that appear in the user defined AQL query. These bindings provide the intended, and sought, concrete architecture.

### 2.1 Architectural design recovery

We consider four fundamental views for software architecture namely, structure, behavior, environment, and domain-specific (section 1.3).

The notion of views has been discussed extensively in the literature [95, 57, 84, 70] (also see section A.2.3). In a broad sense, views are the result of applying separation of concerns on a design in order to classify the related knowledge about the design into more understandable and manageable forms. The notion of views is widely used in different aspects of the software development.

We focus on the structural view of a software architecture\(^1\). In particular, given a legacy application which is represented as an unstructured or poorly structured (due to prolonged maintenance and evolution) collection of libraries, files, functions, data type declarations, etc. We are interested in obtaining a decomposition of the legacy application into a set of structured modules.

Within this context, a module is defined as a conceptual and arbitrary large collection of consecutive source code fragments that can be accessed using an aggregate name [41]. A module $M$ is considered to be a collection of libraries, functions, data types, and variables that conform with a set of relations such as:

- Module Imports Library
- Module Imports Function
- Module Imports DataType
- Module Imports Variable
- Module Exports Function
- Module Exports DataType
- Module Exports Variable
- Module Contains Function
- Module Contains DataType
- Module Contains Variable
- Module References DataType
- Module References Variable

The facts obtained by a legacy system conforming to the relations above can be used to uncover hidden associations using Data Mining techniques.

\(^1\)The structural view covers all building blocks and interconnections that statically describe the architecture of a software system. It consists of static features (i.e., information that can be extracted by statically analyzing the source program), and snapshot features (i.e., information that can be detected statically by interrupting a running program and registering the program's context and state).
More formally, let define the sets: \( \mathcal{L} \) (all libraries), \( \mathcal{F} \) (all functions), \( \mathcal{T} \) (all data types), and \( \mathcal{V} \) (all global variables), that appear in a given legacy application. We consider a module as a quadruple \( M = \langle L, F, T, V \rangle \) where:

\[
L = \{l|Library(l), Imports(M, l)\} \subseteq \mathcal{L},
\]

\[
F = \{f|Function(f), Contains(M, f) \oplus Imports(M, f), Calls(f, f') \lor \text{Exports}(M, f)\} \subseteq \mathcal{F},
\]

\[
T = \{t|DataType(t), Contains(M, t) \oplus Imports(M, t), References(M, t) \lor \text{Exports}(M, t)\} \subseteq \mathcal{T}, \text{ and}
\]

\[
V = \{v|\text{Variable}(v), Contains(M, v) \oplus Imports(M, v), References(M, v) \lor \text{Modifies}(M, v) \lor \text{Exports}(M, v)\} \subseteq \mathcal{V}.
\]

The semantics of the predicates \( \text{Imports, Exports, Contains, References, Modifies} \) are those presented in the standard Software Engineering literature [41, 71]. The \( \text{Contains} \) and \( \text{Imports} \) predicates are module constructors, whereas, the others define the module’s properties.

Having defined the notion of a module we can define an architecture to be a decomposition of a system into \( n \) modules \( M_i = \langle L_i, F_i, T_i, V_i \rangle, i \in \{1..n\} \), which interact with each other through function calls, parameter passing, and global variable access. The constructor predicates impose constraints on the structure of the composed architecture:

- The sets generated by the “Contains” predicate are disjoint among different modules, i.e., \( F_i-\text{Cont} \cap F_j-\text{Cont} = \emptyset, T_i-\text{Cont} \cap T_j-\text{Cont} = \emptyset, V_i-\text{Cont} \cap V_j-\text{Cont} = \emptyset \) for all \( i \neq j \), \( i, j \in \{1..n\} \).

- The sets generated by the “Contains” and “Imports” predicates within a module are disjoint, i.e., \( F_i-\text{Cont} \cap F_i-\text{Imp} = \emptyset, T_i-\text{Cont} \cap T_i-\text{Imp} = \emptyset, \) and \( V_i-\text{Cont} \cap V_i-\text{Imp} = \emptyset, i \in \{1..n\} \).

However, the collection of entities contained in the modules may not cover all the entities of the legacy system, as some entities cannot be grouped into an identifiable module [27]. This issue leads us to the problem of orphan adoption [88]. In an architectural recovery framework, the important issues to be addressed include:

- The formalism to represent information (facts) for a given legacy system.
- The formalism to represent an abstract architectural design in the from of queries.
- A tractable search and approximate pattern matching engine that allows a legacy system to be decomposed based on a given user defined query (pattern)
- An efficient way to present results to a user

The above topics will be discussed in more detail in the following sections.
2.1.1 Data Mining

Data Mining or Knowledge Discovery in Databases (KDD) refers to a collection of algorithms for discovering interesting and non-trivial relations among data in a large database [31]. Most Data Mining algorithms are based on database transactions for market baskets. A transaction contains different kinds of items. In this context, the quantity of items of the same kind in a transaction is not considered. Interesting relations may be discovered among groups of items in transactions (association rules) [15], among sequences of groups of items in transactions (sequential patterns) [16], among time of occurrences of transactions (time-series clustering) [14], etc.

We are interested in the association rules among entities of a software system. Agrawal defines the problem of finding association rules as follows [15]: Let $D = \{d_1, d_2, ..d_k\}$ be a set of data items in a repository. Let $P$ be a set of transactions containing the items in $D$. An association rule is an implication $X \rightarrow Y$, where $X \subset D$, and $Y \subset D$ and $X \cap Y = \emptyset$. The association rule holds with confidence $c$ if $c\%$ of the transactions which involve $X$ also involve $Y$. In addition, if $s\%$ of the transactions in $P$ contain $X \cup Y$, then the association rule $X \rightarrow Y$ has support $s$. The problem is to find those transactions and association rules that have $c \geq \text{min-confidence}$ and $s \geq \text{min-support}$, where min-confidence, and min-support are user defined quantities.

2.1.2 Frequent itemsets

The association rules are based on frequent itemsets generated by the Apriori algorithm [15]. An itemset is a non-empty set of items, and a $k$-itemset is an itemset with size $k > 0$. A frequent itemset is an itemset whose elements are contained in each member of a group of transactions. The size of this group of transactions, i.e., supporting transactions, should be more than the min-support. The algorithm for finding frequent itemsets in a database is as follows: A frequent $k$-itemsets ($L_k$) is obtained by $k$ passes over the database. In the first pass, the algorithm only counts the occurrences of each item in all transactions to determine the frequent 1-itemsets. In each subsequent pass, say pass $k$, the frequent $k$-itemsets are resulted using the frequent $k$-1-itemsets obtained in the previous pass.

More precisely, $L_k = \{C \mid C \in C_k, |C| = k, C.\text{count} \geq \text{min-support} \}$, where $C_k$ is the set of candidate itemsets obtained from frequent $(k-1)$-itemsets. Each itemset in $C_k$ has two characteristics: i) it is obtained by combining two itemsets in frequent $(k-1)$-itemsets whose lexicographically ordered first $k$-2 items are the same; ii) all its $(k-1)$-subsets are elements of frequent $(k-1)$-itemsets. $C.\text{count}$ (support of $C$) is the number of transactions each of which containing all items in $C$. For example, if $L_3 = \{\{1,2,3\}, \{1,2,4\}, \{1,3,4\}, \{1,3,5\}, \{2,3,4\}\}$, then $C_4 = \{1,2,3,4\}$.

The key point in using the rich asset of Data Mining techniques in a different domain such as reverse engineering is to map the entities and relationships of the target domain into the database transaction concept. In reverse engineering domain, the example entities are file, function, and data type, and the example relations (transactions) are fetch, store,

\footnote{subsets with size $k$-1}
<table>
<thead>
<tr>
<th>Entity</th>
<th>Relation</th>
<th>Entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data mining</td>
<td>a transaction</td>
<td>consists of</td>
</tr>
<tr>
<td>Reverse engineering</td>
<td>a container</td>
<td>consists of</td>
</tr>
<tr>
<td></td>
<td>file / function</td>
<td>call to</td>
</tr>
<tr>
<td></td>
<td>file / function</td>
<td>use (read / write)</td>
</tr>
<tr>
<td></td>
<td>file / function</td>
<td>write to / read from</td>
</tr>
<tr>
<td></td>
<td>file / function</td>
<td>send to / receive from</td>
</tr>
<tr>
<td></td>
<td>file</td>
<td>import / export</td>
</tr>
<tr>
<td></td>
<td>a containment</td>
<td>consists of</td>
</tr>
<tr>
<td></td>
<td>function</td>
<td>called by</td>
</tr>
<tr>
<td></td>
<td>file / type / var</td>
<td>used (read / written) by</td>
</tr>
</tbody>
</table>

Figure 2.1: The conversion of various relations in reverse engineering paradigm into the containment relation of a database transaction.

define an and, call. Figure 2.1 shows how the containment relation between a transaction and its items in Data Mining paradigm can be extended to the relation between a container and a set of concepts in reverse-engineering paradigm. We define a concept as a collection of an entity and the operation on that entity. This collection can be treated as an item in a transaction. The first example of the table is interpreted as: a function (or a file) consists of a set of call-to-function (or -file).

Frequent itemsets and association rules among a combination of architectural entities such as functions, aggregate data types, and global variables are used in order to find the best binding for a placeholder according to the already instantiated placeholders. Based on the above discussion, we can assume a function (as a transaction) consists of “call-to-function F-xx”, “use-variable V-xx” and “use-type T-xx” (as items of the transaction). The use relation is interpreted as reading or writing to global variables, or passing data types through argument parameters or return type. A collection of frequent i-itemsets \((i \in \{1..k\})\) along with the container functions (transactions) are generated and stored as the database used by the analysis engine.
2.1.3 Data representation

We use the Refine\(^3\) re-engineering tool [75] to parse the target system and populate an object database from the language-specific constructs. These objects constitute an annotated Abstract Syntax Tree (AST). Using the Refine language we navigate the AST and collect the required entities and their relationships.

The following data is part of the frequent itemsets generated by the Aprioni algorithm.

\[
\begin{align*}
\langle [V-3 \ T-42 \ T-44 \ T-58] [F-83 \ F-176 \ F-646 \ F-647] & \rangle \\
\langle [V-3 \ T-43 \ T-44 \ T-58] [F-83 \ F-647] & \rangle \\
\langle [V-3 \ F-478 \ F-649 \ F-719] [F-647 \ F-648] & \rangle \\
\langle [V-4 \ T-39 \ T-44 \ T-58] [F-83 \ F-89] & \rangle \\
\langle [V-4 \ T-41 \ T-42 \ T-43] [F-83 \ F-647 \ F-648] & \rangle \\
\langle [V-4 \ T-41 \ T-42 \ T-44] [F-83 \ F-647 \ F-648] & \rangle \\
\langle [V-4 \ T-41 \ T-43 \ T-44] [F-83 \ F-647 \ F-648] & \rangle \\
\langle [V-4 \ T-41 \ T-42 \ T-58] [F-83 \ F-647] & \rangle \\
\langle [V-4 \ T-41 \ T-43 \ T-58] [F-83 \ F-647] & \rangle \\
\langle [V-4 \ T-42 \ T-39 \ T-44 \ T-58] [F-83 \ F-647] & \rangle \\
\langle [V-4 \ T-42 \ T-44 \ T-58] [F-83 \ F-176 \ F-646 \ F-647] & \rangle \\
\langle [V-30 \ F-475 \ F-478 \ F-552] [F-548 \ F-549] & \rangle \\
\langle [V-30 \ F-475 \ F-478 \ F-567] [F-548 \ F-549] & \rangle \\
\langle [V-30 \ F-475 \ F-552 \ F-567] [F-548 \ F-549] & \rangle \\
\langle [V-30 \ F-478 \ F-552 \ F-567] [F-548 \ F-549] & \rangle \\
\langle [V-30 \ F-552 \ F-567] [F-547 \ F-548] & \rangle \\
\langle [V-31 \ F-475 \ F-531 \ F-547] [F-62 \ F-536 \ F-542 \ F-546] & \rangle \\
\end{align*}
\]

Each line is a record entry consisting of an itemset (left), followed by the transaction items, and the itemset support. The target system's entities have been encoded into an identification letter (e.g. \(V\) for variable, \(T\) for type, \(F\) for function), and an id-number that uniquely identifies the entity. For example the first line of the sample data above is interpreted as: each function \(F-83\), \(F-176\), \(F-646\), \(F-647\) uses all data-types and variables \(V-3, T-42, T-44, T-58\). These records are part of the frequent 4-itemsets (i.e., 4 items in an itemset), with different number of supporting transactions. Some of these itemsets are final result, and the others can be combined to generate a higher level itemset.

2.2 A system level query language

In this section we present an overview of the Architectural Query Language (AQL) that is used to specify the conceptual architecture of a legacy system. AQL is not presented as a specification language but rather as a description language. It is assumed that a design description represented in AQL may match (generate) a number of different design views.

---

\(^3\)Refine is a trademark of Reasoning Systems Inc.
Each design view corresponds to a possible architecture which is represented as a set of modules. The similarity between an AQL description and a recovered architecture is measured by a matching score. Since AQL descriptions are allowed to have placeholders, the matching process yields a substitution $\sigma$ that consists of a set of bindings between AQL placeholders and the matched source code entities (i.e., functions, data types, and global variables).

Data Mining techniques and a modification of the branch and bound algorithm are used to find an optimal substitution. This substitution provides a means for transforming an AQL query (conceptual architecture) to a concrete architectural description.

A typical module in our AQL language is described below.

**MODULE: M1**

**IMPORTS:**

```
FUNCTIONS:  func $IF(0 .. 3)
TYPES:      type $IT(0 .. 2)
VARIABLES:  var $IV(0 .. 2),
            var ?V1
```

**EXPORTS:**

```
FUNCTIONS:  func $EF(0 .. 2),
            func ?F1(),
            func ?F2()
TYPES:      type $ET(0 .. 1),
            type ?T1
VARIABLES:  var $EV(0 .. 2)
```

**CONTAINS:**

```
FUNCTIONS:  func $CF(2 .. 5)
TYPES:      type $CT(0 .. 2)
VARIABLES:  var first_ftest,
            var $CV(1 .. 5)
```

**REFERENCES:**

```
TYPES:      type $RT(1 .. 3)
VARIABLES:  var $RV(1 .. 4)
```

**MODIFIES:**

```
VARIABLES:  var $MV(1 .. 4)
```

The prefixes "$" and "?" represent multiple-valued placeholders and single-valued placeholder respectively. For example $IF(1,3)$ means minimum one and maximum three of imported functions. Variable placeholders with the same name can only be instantiated with the same value. The matching process provides a substitution $\sigma$, which binds these AQL placeholders with actual entities of the legacy system.

When all placeholders in the query have been instantiated, i.e., bound to values (even to a NULL value), we say that we have a concrete system architecture.

We view an AQL pattern as a means for representing the system-level architecture which:

---

4 As opposed to an abstract architectural description defined by the AQL query.
1. decomposes the program representation into conceptual modules and inter- and intra-module relations.

2. allows to represent alternative design views (structural views represented here), and

3. abstracts away the target system's syntactic and implementation variations.

The matching process is discussed in more detail in the following section.

2.3 Search and control

Search and control are important parts of a design recovery system. Search and control deal primarily with the algorithms required for selecting program parts to be compared against the given pattern (query). In our approach, a Data Mining technique assists in grouping correlated system's entities as the search space for the top level search algorithm.

We focus on AI search techniques which are known to perform well in large search spaces. One such technique, branch and bound search algorithm, has been applied with much success in chess playing and in real-time continuous speech recognition systems. This leads to a top-down understanding that seeks single components that satisfy parts of the given query. A matching score is applied every time a binding is selected, based on the correlation information collected by the Data Mining techniques, and the constraints defined by the user or the AQL semantics. The following sections discuss in more detail the search strategy and the score valuation functions for the proposed technique.

2.3.1 Branch and bound

Given an AQL query $Q$ containing a set of placeholders $\mathcal{X} = \{?X_1, ?X_2, \ldots ?X_n\}$ the objective is to generate a substitution

$$\sigma = \{?X_1/value_1, ?X_2/value_2, \ldots ?X_n/value_n\}$$

that binds AQL query placeholders with actual source code entities. Due to exponential complexity, examining all possible combinations of placeholders and values (brute force algorithm) is not feasible. In order to limit the size of our search space, we use the branch and bound search algorithm. Other search algorithms, such as A*, can also be employed. The choice of the branch and bound algorithm is due to its simplicity.

In a branch and bound algorithm a search tree with incomplete paths is built. At each step the algorithm expands an incomplete path with the highest score among all other incomplete paths. Upon expansion, new incomplete paths are generated and added to the previous ones. The procedure continues until a complete path is found which is an optimal solution. A valuation function allocates a score to each node of the branch and bound search tree to guide the search process.

A search tree $T = \langle R, T_1, \ldots T_k \rangle$ is a tree composed of a root $R$ and $k$ distinct subtrees, where $k$ is the number of different values that can be bound to a placeholder $?X_i$ (assuming the same domain for all placeholders), appearing in a query specification. Each
node $N_j$ in $T$ is associated with a set of bindings $\mathcal{B}_j = \{?X_1/val_1, \ldots, ?X_i/val_i\}$ for all query placeholders instantiated so far. A leaf node $N_l$ is associated with a set of bindings $\mathcal{B}_l = \{?X_1/val_1, \ldots, ?X_n/val_n\}$ that provide values for all placeholders appearing in the query. The bindings $\mathcal{B}_j$ and $\mathcal{B}_l$ are known as incomplete path and complete path, respectively. In this context, a search space is defined as: all values that provide a possible binding to the query placeholders. A tractable search technique focuses on navigating the search space and obtaining an optimal solution.

Figure 2.2 demonstrates a sequence of branch and bound search trees that incrementally instantiate a concrete architecture consisting of three modules. Each depth level of the search tree corresponds to a Data Mining frequent itemset whose itemset size is equal to the depth number. This means, the domain of values for placeholders at depth $k$ are provided by the frequent $k$-itemsets. The single node at depth 1 corresponds to the instantiation of a placeholder using the initial fixed entity (atom), assuming only one atom has been defined in each module. The number of arcs from a node in depth $k$ to some nodes in depth $k+1$ is equal to the number of "associated entities" with those entities bound in a path from depth 1 to depth $k$. This path is incomplete if the number of arcs are more than zero. An incomplete path becomes complete, (i.e., ending at a leaf node), when there is no further associated entities to extend that path. At each node the number of bindings is equal to the node's depth number plus the number of transactions (support) of the corresponding itemset. When a path becomes complete, the rest of unbound module's placeholders are bound with NULL entity. In Figure 2.2 a thick line from the root of the first tree to a leaf of the last tree represents an optimal architectural solution.

In a macroscopic view, the architectural pattern of an AQL query is a graph of modules,
Figure 2.3: An architecture solution is obtained incrementally by separate branch and bound searches for individual modules. An intermediate phase provides a number of candidate solutions to improve the probability of finding an optimal solution.

connected via import/export links. A sequence of branch and bound searches is required to provide solutions for individual modules and incrementally build the architecture. A link is a shared entity among two or more modules that is bound to a link placeholder (e.g., ?V1) in the Imports/Exports parts of the linked modules. The early commitment (i.e., instantiating a placeholder with a value at the earliest time) when the other linked module(s) (to be instantiated in the future) are not known yet, eliminates the chance of instantiating the link placeholders that connect the modules together.

The module’s placeholders are instantiated using both a frequent itemset and its supporting transactions. Generally, the number of transactions decreases from frequent 1-itemsets to frequent k-itemsets (i.e., domain of function placeholders in depth 1 to depth k of the search tree). Hence, there is a high chance that the value of an instantiated function placeholder be withdrawn in further steps, which requires an unlinking process. The withdrawal of the value of a function “link placeholder” further complicates the situation.

At each node of a search tree the link placeholders between the current and previous modules should be examined for instantiation. The values at each node, if not used for a link instantiation, are kept in a pool of values to be used for future link instantiations. At each node a score is evaluated (section 2.3.2) based on the assumption that the values have been bound to module’s placeholders. Once all the links of the architecture are examined, a concrete architecture is built by actually binding the modules’ values (entities) to the modules’ placeholders. To ensure the correctness of this method, we should always maintain the whole incomplete architecture consistent with regard to the bound links and initial bound
elements (atoms). By a consistent architecture, we mean that each architectural entity must be contained in only one module.

The overall search control process uses backtracking mechanism to allow alternative solutions for previous modules, should the solution for the current module fail to produce an architectural solution. In Figure 2.2 the paths in solid trees have been examined by the branch and bound search algorithm and some solutions have been found, whereas the dashed trees provide alternative solutions in backtracking phases. In order to enhance the probability of finding a better optimal solution among others, a number of complete paths are collected for each module (candidate solutions). In Figure 2.2 the best solution found for the previous module is the starting point for the solution pertaining to the next module. This link implies that we keep a repository of best solutions for the previous modules. Figure 2.3 represents the implementation view of the architectural recovery search engine. The candidate solutions provide an intermediate repository of optimal solutions for each module. This arrangement facilitates the backtracking implementation and improves the chance of finding an optimal architectural solution.

2.3.2 Score valuation

The evaluated score of a new binding $?X_d/val_d$, occurring in some node $N_j$ at depth $d$, is used to guide the branch and bound algorithm to expand the search tree. The score valuation function takes the form of a probabilistic score that is assigned based on the characteristics of the legacy system entities and by the application of Data Mining techniques. The score that we use is evaluated by the formula:

$$Score = \left( \frac{\text{No. bound links}}{\text{No. all links}} + \frac{\text{No. entities}}{5} \right) \text{ confidence}$$

where, the confidence is approximated as

$$P(?X_d/val_d|?X_1/val_1,?X_2/val_2,\ldots,X_{d-1}/val_{d-1})$$

and the parameters, i.e., the number of bound/all links, and the number of entities, correspond to a module.

$$P(?X_d/val_d|?X_1/val_1,?X_2/val_2,\ldots,X_{d-1}/val_{d-1})$$

is the probability that the AQL placeholder $?X_d$ can be bound to value val_d given that the other placeholders $?X_1,\ldots,X_{d-1}$ in an incomplete path to depth $d-1$ have already been bound to values $val_1,\ldots,val_{d-1}$.

The confidence level $c$ of the association rule $\{val_1,\ldots, val_{d-1}\} \rightarrow \{val_d\}$ is obtained during the Data Mining process (see section 2.1.1). This association implies that $c\%$ of the itemsets that contain $val_1,\ldots,val_{d-1}$ also contain $val_d$.

For example, assume that in the frequent 3-itemset

$$<\text{V-4 T-41 T-42}> [\text{F-83 F-647 F-648}] 3>$$

the entities V-4, T-41, and T-42 (one global variable and two data types) have been bound to placeholders $?X_1,?X_2,$ and $?X_3$ in some node $N_j$ of depth $3$\(^5\). The frequent

\(^5\)The bindings of transaction functions at each node of the search tree does not affect the confidence value of the association rules, hence, we do not consider them.
4-itemsets of section 2.1.3 provides three associated entities T-58, T-44, and T-43 to the entities of node $N_j$ \(^6\). These three entities extend the incomplete path ending to node $N_j$, to three nodes at depth 4. The confidence value of the first associations is:

$$\{V-4, T-41, T-42\} \rightarrow \{T-58\} =$$

$$\frac{\text{support}(V-4, T-41, T-42, T-58)}{\text{support}(V-4, T-41, T-42)} = \frac{2}{3} = 0.66.$$  

The branch and bound search algorithm expands its path to the node with the highest confidence. In this example, to T-43 or T-44 whose confidence is 1. The use of Data Mining frequent itemsets restricts the domain of the module's placeholders to a small set provided by the association rules.

Other criteria related to the domain of interest can also adjust the score valuation of a node to direct the search algorithm. Important criteria in software architecture recovery would be:

- high cohesion among the elements of a module,
- larger group of elements in a module,
- correlation among the group of modules, and
- compliance with the user defined constraints.

The first criterion is ensured by using confidence level of the association rules for the score valuation. The other criteria can be regarded by considering the number of instantiated placeholders and links in a module.

### 2.4 Experimental results

We used the CLIPS system (C-Language Interface Processing System), a medium size (approx. 40 KLOC) rule-based system, as a vehicle to conduct our experiments. CLIPS has a published architecture and that helps evaluate our results against the reference architecture. We focus on two types of experiments. The first experiment investigates the precision and the accuracy of the pattern matching engine. The second experiment focuses on recovering a partial architecture for the subject system.

Our platform consists of a Sun Ultra 1 (160MHZ, 128M memory) and the experiments are performed in a low CPU-load environment. It takes 6 minutes to parse the target system (CLIPS), using a parser written in Refine C, and to construct an annotated AST in the Refine's database. The Apriori algorithm requires approximately 30 minutes to build the frequent itemsets with support 2. The search time of the branch and bound algorithm is sensitive to the formulation of the score function discussed in section 2.3.2. With the presented score function, the search is fast taking less than 10 seconds for the algorithm to come up with an instantiation for a module of an architecture.

\(^6\)The corresponding frequent 4-itemsets in section 2.1.3 have been prefixed by a star (*).
In a typical scenario for recovering partial architecture of a legacy system the user, usually a domain expert, uses available system documents or other sources to formulate architectural queries in AQL. The user focuses on discovering the module diagram of a part of the target system implemented in one or more related files (a cluster). File clustering is achieved through generating the frequent itemsets using the relations file consists-of call-to-files or file consists-of use-files (see Figure 2.1) and conducting an experiment similar to recovering modules of functions, types, and variables. This framework allows to cluster architectural entities at different levels of granularity.

The first experiment investigates how Data Mining results conform to the results obtained from the system's documentation using the precision and accuracy of the result\textsuperscript{7}. Figure 2.4 shows the result of the experiment for recovering four files of the CLIPS system. The first three columns illustrate: i) the number of entities in the documentation, ii) the number of entities found by our method which were in documentation, and iii) the number of entities found by our method which were not in the documentation. The last three columns of the tables represent the percentage amount of the absolute data in the first three columns, such as:

\[
\text{precision} = \frac{\# \text{found entities in CLIPS file}}{\# \text{entities in CLIPS file}},
\]

\[
\text{false positive} = \frac{\# \text{found entities not in CLIPS file}}{\# \text{entities in CLIPS file}}, \text{and}
\]

\[
\text{false negative} = \frac{\# \text{entities in CLIPS file that not found}}{\# \text{entities in CLIPS file}}.
\]

The results indicate that our experiment could find a portion of the entities, not all of them. This is due to the following facts:

- Data Mining algorithms use min-support/min-confidence to group cohesive entities. A number of architectural entities are filtered out since they do not have sufficient links to any group of entities. These entities should be treated separately and be assigned to other groups using some criteria (orphan adoption problem [88])

- The boundary of a file does not necessarily conform with the boundary of a group of modules (or vice versa) in term of entity usage. Therefore, some entities belong to modules in other files.

Currently, we are experimenting with more relations obtained from the source code so that more associations can be obtained.

For the second experiment a partial architecture consisting of four modules is recovered. The AQL representation of the first module has been illustrated in section 2.2. Figure 2.5 illustrates the module interconnection diagram [41] of the recovered partial architecture of the CLIPS system consisting of four modules. In the associated query specification for these modules, the user has defined a fixed entity in the CONTAINS part of each module.

\textsuperscript{7}The queries were conducted independently, without having a knowledge of the ideal architecture.
<table>
<thead>
<tr>
<th>FILE: rulecomp.c</th>
<th>Entities in Clips</th>
<th>Found entities in Clips</th>
<th>Found entities not in Clips</th>
<th>Precision</th>
<th>False positive</th>
<th>False negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>27</td>
<td>15</td>
<td>6</td>
<td>55%</td>
<td>22%</td>
<td>45%</td>
</tr>
<tr>
<td>Type</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>55%</td>
<td>11%</td>
<td>45%</td>
</tr>
<tr>
<td>Variable</td>
<td>41</td>
<td>27</td>
<td>2</td>
<td>66%</td>
<td>5%</td>
<td>34%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FILE: factmngr.c</th>
<th>Entities in Clips</th>
<th>Found entities in Clips</th>
<th>Found entities not in Clips</th>
<th>Precision</th>
<th>False positive</th>
<th>False negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>34</td>
<td>22</td>
<td>18</td>
<td>64%</td>
<td>53%</td>
<td>36%</td>
</tr>
<tr>
<td>Type</td>
<td>13</td>
<td>7</td>
<td>0</td>
<td>54%</td>
<td>0%</td>
<td>46%</td>
</tr>
<tr>
<td>Variable</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>100%</td>
<td>14%</td>
<td>0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FILES: object.c</th>
<th>Entities in Clips</th>
<th>Found entities in Clips</th>
<th>Found entities not in Clips</th>
<th>Precision</th>
<th>False positive</th>
<th>False negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>method.c</td>
<td>68</td>
<td>44</td>
<td>21</td>
<td>65%</td>
<td>30%</td>
<td>35%</td>
</tr>
<tr>
<td>Function</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>91%</td>
<td>0%</td>
<td>9%</td>
</tr>
<tr>
<td>Type</td>
<td>10</td>
<td>9</td>
<td>3</td>
<td>90%</td>
<td>30%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Figure 2.4: The experimental results on four files of the CLIPS system.
Figure 2.5: A concrete architecture of four modules and their associated links, based on an AQL query on CLIPS system. The search engine uses a branch and bound algorithm on Data Mining frequent itemsets. The names of the entities have been coded.
as the constraint for the search algorithm (e.g., variable first_test in Module M1). The
recovery process is performed automatically without any user interaction. The AQL language
also allows the user to specify several constraints in different parts to restrict the search
space. The proposed approach allows the user to easily increase the number of modules and
incrementally recover the design architecture of the target system. Since the source model,
i.e., Data Mining frequent itemsets, are generated and stored in a preprocessing phase, the
search process is fast allowing the approach to scale to larger systems.

Below, the result of the search algorithm for module M1 is shown. A placeholder
is represented using X or Y followed by a number. A multi-value placeholder in a query
specification is instantiated using Xs to represent the minimum requirements and Ys to
represent the optional placeholders. For example, $CF(2 .. 5)$ in the field “contains function”
generates a sequence of placeholders ((X1 -) (X2 -) (Y1 -) (Y2 -) (Y3 -)) with NIL values.
In an instantiated link placeholder, e.g., (?V1 V-54 1), a digit shows the correspondence of
a module to other modules based on this link. For example, in module M2 which exports
variable V-54 to all other modules this digit is 3. In the result, the lines 3-5, 7-9, and 11-14
indicate the instantiated placeholders and links for parts Imports, Exports, and Contains
respectively. In Import/Export parts three links have been instantiated. The lines 17-19
indicate all the entities found by search algorithm from Data Mining results, and lines 21
and 22 indicate the location of the found entities in a CLIPS file(s).

```
(RESULT)  1 (M1 2 1.0 1.8 NIL

(((Y1 -) (Y2 -) (Y3 -))

(((Y1 -) (Y2 -))

(((?V1 V-54 1) (Y1 -) (Y2 -))

(((?F1 F-493 1) (?F2 F-503 1) (Y1 -) (Y2 -))

(((?T1 - 0) (Y1 -))

((Y1 -) (Y2 -))

(((X1 F-493) (X2 F-503) (Y1 -) (Y2 -) (Y3 -))

(((Y1 -) (Y2 -))

(((X1 V-43) (X2 V-44) (Y1 V-45) (Y2 V-61)

(y3 V-62) (Y4 V-63))

-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-

Functions:(F-503 F-493)

Types:NIL

Variables:(V-43 V-54 V-45 V-61 V-62 V-44 V-63)

File: RULECOMP

(F-503 F-493 V-63 V-44 V-62 V-61 V-45 V-54 V-43)
```

32
Appendix A

Reverse engineering and system analysis

A.1 Background

For several decades, we have been witnessing the influence of large and complex software systems into various aspects of our lives. Frequent incidences of unsuccessful projects, known as software crisis [42], are evidences of the humans’ weaknesses in developing and maintaining large software systems composed of millions lines of code. In recent years, engineers have been confronted with the problem of legacy systems, which are old software systems that continue to be used. On average, a software system is operational for 10 to 15 years before it is replaced [91]. These old systems are difficult to maintain, evolve, and integrate with new systems, since they usually lack adequate design documentation. Thus, software designers and maintainers require tools and techniques to adequately understand and document a large software’s design.

In recent years a shift of research interest, from detailed design issues to architectural design aspects of a system, is evident in research avenues. A number of large projects have been launched to generate necessary tools and methodologies for maintaining or re-engineering our valuable software assets.

In software development phases, architecture description languages (ADLs) capture the high-level design and provide analysis capabilities [81, 39, 60, 21]. The notion of different views of an architecture [85, 57, 84], as design decomposition into orthogonal concerns, suggests further clarity and understandability for the design. Direct software development from large pieces of commercial components, known as COTS [90, 38] is another example.

A brief review of software engineering discipline and the related issues on general topic of reverse engineering, demonstrates the extents of our research.

Software engineering is an attempt to apply engineering discipline to the development of high quality software systems. One of the first definitions of software engineering was given by Fritz Bauer in 1969 [68]: The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines. Software engineering is the result of long-term experience of the practitioners, engineering
techniques from the related domains such as hardware and system engineering, and the state of the technology for automating operations. Research in software engineering is partitioned into three subjects [71]:

- **Process**, is responsible for organizing and monitoring different phases of a software development project such as: *project planning and estimation, system and software requirements analysis, design, coding, testing, and maintenance*. Various processes tend to fall into three well-established paradigms: *classic life-cycle or waterfall model, prototyping, and spiral model*.

- **Methods**, provide the required technical guidelines and steps for conducting different activities mentioned in the process phases. Maintenance activities are performed for different objectives such as revealing and correcting errors after the software has been released (*corrective*), porting the software product to a new environment (*adaptive*), improving the product’s performance and augmenting it with new features (*perfective*), or enhancing its structure for future changes (*preventive*). Modifying a program must be controlled in a sequence of planned activities including design of the changes, re-coding, and testing. Different maintenance techniques fall into two broad categories, *reverse-engineering and re-engineering*.

- **Tools**, assist the engineer in automating those parts of the process phases that are repeatable and lack any creativity. The maintenance tools play an important role in providing structural information for legacy systems that lack documentation.

*Reverse-engineering tools* analyze the source code and build high-level design representations such as control-flow and data-flow diagrams and program-structure views. Re-engineering tools are classified as *code restructuring tools* which recode the system into a structured programming concept, and *data restructuring* which are intended to graphically demonstrate the program’s data structures.

### A.1.1 Reverse engineering taxonomy

A substantial research work has been carried out in different aspects of reverse-engineering and re-engineering to elaborate on the problems that software maintainers are facing. Chikofsky and Cross present a taxonomy of reverse-engineering terms which have been adopted by the research community as the standard terminology [24].

In Figure A.1, different abstraction levels (i.e., requirement, design, and implementation), and the direction of arrows signify the processes of refining and abstracting a software document, and represent the scope of each term. The definition of these terms are as follow:

*Forward engineering* aims at producing an implementation document describing the details of the desired high-level concepts of a system [71].

*Reverse-engineering* is the process of examining an existing system’s representation to extract design artifacts that are less implementation dependent [33, 22, 5, 48, 74, 43, 35]. Reverse engineering helps designers to understand the design structure of an existing system,
Figure A.1: Relationship among reverse-engineering terms. The boxes represent different abstraction levels in forward and reverse-engineering (borrowed from [24]).

normally lacking any design document, before making changes to it. Reverse engineering encompasses two areas:

- **Program understanding (or design recovery)** is a process of extracting meaningful abstractions by examining a system’s representation and employing domain knowledge, informal information, and deduction or reasoning. The bulk of the reverse-engineering research activities is focused on this issue [56, 50, 25, 94, 12, 78].

- **Redocumentation** is the exercise of creating a semantically equivalent representation of a system in the same level of abstraction. Creating flow graphs, structure chart, and pretty print of a source code are examples of redocumentation process. Redocumentation is sometimes thought of generating a different view of the system.

Restructuring is the transformation of the representation in the same level of abstraction so that the system functionality is unchanged. Restructuring is performed to augment the system’s physical state as part of the preventive maintenance. For example, changing the system’s platform, code-to-code transformation, or decreasing the dependencies among system’s modules. Griswold introduces a tool and technique for meaning preserving restructuring of the layered systems [44].

Re-engineering, also known as renovation, is performed to augment a system’s functionality or performance in order to meet the new requirements. Reengineering is a combination of reverse-engineering, to get some higher view of the system, and forward-engineering to reimplement a new system with enhanced design and features.

As mentioned above, program understanding is the most challenging part of the reverse engineering and the maintenance activity as a whole. The program understanding problem
has been investigated at two levels: source code analysis (program understanding in-the-small), and system architecture analysis (program understanding in-the-large). This research focuses on the architectural recovery. Architectural recovery is a reverse-engineering activity that focuses on the architectural design aspects.

A.1.2 Program understanding

In general, program understanding involves three types of research: algorithmic approaches, cognitive studies, and knowledge-based approaches.

Algorithmic approaches assist in understanding a program’s functionality by integrating the functionality of the smaller parts of the program. Most of the process is automated. Different methods are used to derive the functionality of small parts of a program. Functional correctness approaches and tools are used for functional specification integration and verification of the derived specification against the program’s functionality [13]. Algorithmic approaches are completely user-dependent, but they assist in providing a formal specification of the system. These approaches are limited in the sense that they lack discovery nature.

Cognitive studies of program understanding are researches about the cause of the humans’ ability to understand a program. These studies show that the expert-programmer’s talent of writing good programs is based on: a series of stereotyped plans he/she has accumulated for attacking small problems; and the rules of programming discourse [83]. The process of understanding a program is bottom-up, in which the analyst uses stereotype solutions to build intelligent high-level abstractions and understanding.

The notion of plan, pattern, or schema is used to capture the pieces of stereotype knowledge buried inside a program.

Knowledge-based approaches, inspired by the cognitive views of program understanding, try to automate this process by using a repository of plans or patterns as basic blocks of knowledge to build a high-level understanding [13]. A plan, in low-level program understanding, is a commonly used idiom or algorithm in a software system that implements a particular programming concept. The notion of concept is used to relate a plan to the program properties [50]. A pattern, in high-level program understanding, is a graph consisting of conceptual program constructs as nodes, and the relations and constraints among nodes as edges.

Program developers, like architects, tend to use stereotyped patterns in their designs to make their artifacts more regular and understandable. This characteristic of programs encourages the analysts to seek standard patterns in a program understanding task, rather than a general reasoning. The study of design patterns and architectural styles (section A.2.2) is central in understanding a program. Design patterns define a collection of guidelines and rules for designing useful models that are repeated in systems [36, 37]. However, design patterns are means for generating regular implementation,
Figure A.2: A program understanding environment. The control part selects patterns from the library and a program part to feed the comparison component. Reverse engineering is a typical application.

whereas, a library of patterns in reverse engineering is a means for detecting these regularities.

As a broad research field, program understanding benefits from techniques in artificial intelligence (AI), software engineering, and programming languages. A typical program understanding environment, in Figure A.2, consists of a program understander (control and comparison) as the engine, a knowledge-base of patterns, a target program as input, and an application, e.g., reverse engineering, reengineering, that uses the result of the program understanding.

A.1.3 Source code representation techniques

As an integral part of abstracting and understanding legacy systems, representation and modeling techniques have received much attention. The representation methods of programs directly affect the complexity and performance of the reverse engineering task.

Each program representation method has strengths and weaknesses and is appropriate for a particular abstraction level. Source code has some useless information from the program understanding point of view, e.g., comments, syntactic sugar, and functionally similar constructs. These extra information unnecessarily increase the search space in pattern instance localization. Therefore, a pre-processing phase is needed to trim these noise information and provide a more abstract view of a program. It should be noted that, increasing the level of abstraction reduces useful information which may or may not be relevant to our program understanding task. For example, abstract syntax tree, control flow model, and data flow model, respectively eliminate variations in format, control construct, and control structure of a program. Each representation in the above sequence covers a broader class of
programs than the previous one. A representation method with a high-level of abstraction is programming-language independent, but still is language-class dependent. Language-classes, e.g., imperative, functional, and logical, have different semantic features that complicate the cross-translation of their features.

Another consideration for selecting a representation model is the ease of searching that it provides for our intended program understanding task. For example, an analysis task based on the control structure of a program needs redundant control path searches in an abstract syntax tree model. Whereas, selecting a control flow model is more appropriate since it is computed once and is used many.

Examples of program representation techniques

Common program representation techniques for program understanding include: AST [51]; code object base, Genoa [28] and Refine [75]; attributed data flow graph, GRASPR [92]; control/data flow graphs, UNPROG [49]; functional languages, TALUS [67].

A.2 Software architecture recovery

In this section, we focus our discussion on the architecture recovery aspects of a software system and address the important related issues. We try to narrow down the general discussion of the software architecture towards the topics that address architectural recovery related researches.

There is no consensus on the software architecture definition. Garlan and Shaw define software architecture as below [82]:

“Software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.”

Terminology

The following terminology seems to have obtained general acceptance for better communication among researchers in software architecture community. A software architecture is identified by its components, connectors, and style.

- A component represents the encapsulation of the system’s computation corresponding to a compilation unit of programming languages. A component’s interface defines the signature and functionality of the services provided by the component. Examples of components include: filter, process, layer, client, server, memory, etc. [82].

- A connector represents the encapsulation of interactions among two components. The connector’s protocol defines the form of communication, the type and order of the transported data, and the way of assuring the protocol execution. Connectors can be viewed as definers of the roles that must be played by some services inside the components. Examples of connectors include: RPC, event broadcast, pipe, etc. A
connector can be implemented in a variety of ways such as: built-in mechanism in programming languages; system calls in operating systems; and calls to library code [82].

- An architectural style captures broad properties, vocabulary, and configuration constraints that apply to all instances of a family of systems [65]. A number of widely used architectural styles have been identified in the existing systems. A collection of these styles, gathered by Garlan and Shaw [40] include: pipe and filter, client and server, implicit invocation, layered, blackboard, object-oriented, interpreter, state transition, and main program and procedure [82] (see section A.2.2).

Software architecture research areas
Software architecture domain consists of several research areas which are actively pursued. Typical problems in this domain include:

- How to obtain the actual architecture of a system? (the subject of architecture design recovery).

- What architectural information should be captured? (the subject of research on architecture description languages)

- How to realize the architectural description of a system? (addressed by different architectural design processes)

- How to keep different architectural representations consistent as the system evolves? (e.g., traceability between source code and software architecture)

We have conducted a survey on software architecture domain and categorized the approaches into six categories: foundation, architecture description languages, architectural design process, architectural styles (reuse), architecture recovery, and architecture development environment [79].

Architecture recovery approaches
The interest towards architectural design recovery of a software is the result of increasing demand on system maintenance activities, i.e., corrective, adaptive, perfective, and preventive.

Architecture recovery encompasses various methods and tools for effectively extracting high level design information from lower level program representations of a legacy system to assist an engineer in making changes or integration. The use of term “architecture” emphasizes on the targeted abstraction level, which is higher than “design” level as the target of the most traditional reverse engineering approaches. Most of the tools and techniques have been inspired from those of traditional reverse engineering.

The whole process is divided into two phases. In the first phase, extraction, a tool automatically builds a more abstract source model out of the program representation. In the second phase, analysis, a human assisted tool constructs a high-level view of the system.
from the source model. A taxonomy of the existing architecture recovery techniques are as follows [64]:

- **Filtering and clustering framework:** a parser extracts a relational source model from source code and stores it in a database. Then the analyst, using a series of cascading operations on database, provides higher levels of information. This is done by clustering components based on some criteria such as low-coupling and high-cohesion among them. The method has a drawback: the conventional parsers can not extract complex relationships, such as IPC and RPC, among the components, hence, they restrict the level of architectural recovery of the tools. Rigi is an example [87].

- **Compliance checking framework:** the extraction phase is identical to that in the above method (with the same limitation). In the analysis phase, the analyst first defines his/her assumed high level model of the software in an appropriate form (e.g., modules and interrelations, an inheritance hierarchy, a design pattern, or an architectural style). The tool then checks the degree of conformance between the proposed model and source model. Software reflexion model tools are examples [66].

- **Analyzer generators framework:** a parser generates an abstract syntax tree, and stores it in a repository. A query language assists the engineer in generating analysis tools, as a series of queries written in the query language that search from the source model. Abstract syntax trees are language dependent, hence, the engineer must be familiar with the language syntax. This method is more flexible than the previous ones and is used in conjunction with other methods which follow. Genoa is an example [28].

- **Program understanding framework:** this method uses the knowledge of the engineer about the system to automatically generate high level view of the system’s functionality. The knowledge of the engineer is captured in a knowledge-base, and the source code is represented in an abstract syntax tree. A recognition engine then searches through the source model and knowledge-base with the goal of finding the matching parts. The result is a hierarchy of recognized patterns which are the user-guided views of the system. DECODE is a tool in this method [74].

- **Architecture recognition framework:** this method employs a series of architectural style recognizers that search the abstract syntax tree (source model) for the style (or styles) of the software. A recognizer is written as a set of queries in a query language. The number of the recognizers indicates the richness of the method [48].

In order to perform an architecture recovery task, the analyst need to know what architectural concepts and patterns to extract (addressed by architectural styles and views), and how to represent them (addressed by architectural description languages). In the following sections these related subjects are discussed.
A.2.1 Architecture description languages

An Architecture Description Language (ADL) is used to document architectural information in order to describe and analyze a system. From the linguistic point of view, an ideal ADL is characterized by six properties [82]:

- **composition / decomposition**: integration of elements into larger subsystems, decomposition of a system into its constituents, and integration of architectural styles.
- **abstraction**: the ability to define abstract views of high-level or low-level design.
- **reusability**: the property of using generic patterns of elements, e.g., element types.
- **configurability**: the capability of changing a software's structure, independent of the components.
- **heterogeneity**: the ability of integrating different styles in one system, or integrating modules written in different languages.
- **analysis**: the ability of reasoning about the system by checking architectural properties, providing metrics, or simulating run-time characteristics.

The high level of abstraction provided by the architectural view of a software system completely discriminates ADLs from conventional programming languages. Conventional programming languages deal with details such as algorithm and data structure design. As a candidate for ADL, these languages have shortcomings in all aspects cited above. The highest abstraction level they provide is module (or object); module interconnections are performed in restricted forms; communication protocols are implemented inside the module; analysis is limited to simple type checking; and they fail to isolate modules from each other, e.g., module interfaces rely on *provide/require* services with explicit name bindings.

Module Interconnection languages (MIL) [72], an attempt to provide glue for integrating separately developed modules in a distributed environment, have had some success in relaxing the configurability and composition of the systems. A variety of MILs exist which differ in aspects such as: type of interfacing, automatic handling of complex data-types, language and implementation, and execution platform [73].

Polythith [73], a new generation of MILs, provides integration of modules in a heterogeneous environment of operating systems and programming languages. Polythith binds the imported/exported services of already designed and implemented system modules. The software *toolbus* of Polythith provides different wrappers around a module for purposes such as high performance or component's interconnections control. The *system packager* of Polythith handles the distributed execution of the system. The MILs have shortcomings to be considered as ADLs, particularly, they are based on name binding and support restricted interactions, i.e., procedure call and shared memory.

A variety of architectural features have been proposed or implemented on ADLs such as: Unicon [81], Rapide [60, 61], ACME [39], ARDEC DSSA [86], Wright [17], ControlH [21], PBS [33, 6], Rigi [5], Polythith [73], DECODE [74], and Kaptur [19]. No single ADL can
handle a good portion of these features. Below, the important features of two interesting ADLs have been investigated.

- Unicon has been designed based on Mary Shaw's informal model and notation for ADLs [81]. Unicon provides a development environment and aims at covering a broad range of systems. Unicon benefits from a rich definition for architectural connectors as defined earlier.

The features of Unicon are as follows:

- component/connector abstraction,
- library of component and connector templates,
- no naming constraints for component services to be connected,
- a variety of available component implementations (e.g., source, object), and
- open structure to integrate with analysis tools.

- Rapide is a concurrent, event based, simulation language for defining and simulating a system's architectural behavior in a distributed environment [60, 61]. Rapide benefits from a number of important architectural concepts, making it an interesting ADL. The features of Rapide are as follows:

- supporting the notions of architecture, interface, system, module, event, event pattern, poset (partially ordered set of events),
- supporting static and dynamic architectures\(^1\),
- automatic instantiation of a module and checking the module's behavior,
- component and connection abstraction,
- hierarchical architecture, and
- architectural simulation.

### A.2.2 Architectural styles

One important cause of the complexity of large software systems is the variety of non-standard building blocks used in the construction of systems. Software standardization, and hence reusability, is a solution to this problem which assists us in constructing reliable, cost efficient, and more understandable systems from off-the-shelf software components. A comprehensive collection of generic software encapsulation (known as software handbook), encourages the practitioners to reuse software. Software architecture, by introducing architectural elements as the encapsulation of standard functionalities, encourages reusability in system construction. Concepts such as design patterns and architectural styles imply standardization and reusability of generic constructs at high levels of abstraction. A set of well known architectural styles are as follows:

\(^1\) A dynamic architecture is defined using *event pattern matching* which allows a communication link to be established among components at run-time.
- **Pipe and filter** style is a combination of filters which are connected through pipes. Filters incrementally transform a stream of data received from a pipe, and deliver it to the next filter via another pipe. In an ideal situation, this style allows concurrency among filters with a distributed control mechanism. However, other characteristics of the pipes and filters may restrict this concurrency. *Pipeline* and *batch sequential* are restricted forms of pipe and filter. The former has a linear topology, and the latter disallows any concurrency among filters. Examples of pipe and filter style include UNIX shell, traditional compiler technology, and signal processing.

- **Client/Server** is a common architectural style in distributed systems, in which a number of clients, using the remote procedure call mechanism, request service from a server. In most cases, the clients acquire the identity and the types of services of a server from a name-server [20].

- **Implicit invocation** allows components to register events for the services they provide. A manager receives request-events from the components which need specific services, and propagates the request-events to the components that have registered those services. This style supports reusability and system evolution, but suffers from poor predictability and testability. Examples include the Field system [77] and HP SoftBench [23].

- **Layered systems** provide hierarchical layers of service, in which a layer uses/provides service from/to its adjacent lower/higher layer. To compensate for the performance inefficiency, bridging over layers is done in some systems. This style supports re-use, abstraction of the system functionality, incremental enhancement of the design, and locality of changes. The style is ideal for systems whose structures are potentially hierarchical such as the ISO/OSI reference model [1].

- **Data repository** style is divided into *database* and *blackboard* systems, depending on the control mechanism used by the repository (knowledge base) to trigger the processes which share and modify the repository. The repository triggers the processes based on the type of transaction (database), or the current state of the repository (blackboard). Examples of blackboard style include modern compiler technology and speech/pattern recognition systems [69],

- **Object-oriented** style encapsulates an object's state and the implementation of operations on that state. It provides a mechanism for isolating object usage from its implementation, and inheritance capabilities, which support hierarchical design. Aesop, an architectural construction environment, is an example of this style [82].

- **Other styles** include: *interpreter* style, which provides an illusion of the underlying machine (virtual machine) to the user, e.g., programming languages. *Main program/subroutine* style, in which the main program (driver) dispatches the sole thread of control to subroutines, in a particular sequence. *State transition* style, in which the
system transitions to different states according to some stimuli, e.g. reactive systems [46].

- **Domain specific software architectures (DSSA)** refer to a number of families of industrial systems that have reached to a level of maturity to possess specific software architectures (known as reference architectures) for those domains. Examples of DSSAs are found in avionics [2], command and control (C2) [3], and vehicle management systems [4]

Some recent approaches attempt to understand an architecture by extracting elements and interactions that constitute a particular architectural style [48, 34].

### A.2.3 Architectural views

The significance of software architecture **views** as a means for separating the designer’s concerns has been addressed in the literature [95, 57, 84, 70].

In a broad sense, a view can be defined as “a projection of a process according to a well-defined characteristic” [18]. Examples of such characteristics of a software development system include: roles (e.g., manager, designer), products (e.g., specification, design document), and activities (e.g., reviews, implementation). In other words, views are the result of applying separation of concerns on a development process in order to classify the related knowledge about that process into more understandable and manageable forms. The notion of views is widely used in different aspects of the software development. Examples of such aspects include: software requirement elicitation [29], software specification [32], domain modeling [30], program development systems [76], software process modeling [18, 59], and software architecture design [95, 57, 84].

The choice of an appropriate set of views is a common concern both in software development and recovery process. The proposed sets of views for developing or specifying a system consists of: data, function, and network [95]; function, process, development, and physical [57]; conceptual, module-interconnection, execution, and code [84].

**Functional view** is shared by all of the above proposals, hence, it seems to be central. **Data view** is important in design of an information system. **Process view** is required in design of concurrent systems. **Development view** is mostly required for task assignment and scheduling in project management. **Network view (physical view)** is essential in designing distributed systems. **User-interface view** is helpful in designing a highly interactive system.

It is ideal to recover the same set of views of a system that is appropriate for its development. The advantages are twofold: the domain specific features of the system can be properly extracted; and the set of views are suitable for re-engineering the system (if needed). Unfortunately, reverse engineering is much difficult than forward engineering. Recovering some aspects, e.g., functionality, of a large and poorly documented legacy system, is a non-trivial task (if not impossible). The program understanding approaches, discussed in section A.1.2 do not scale up to large systems. **Structural view** is the most appropriate architectural view to be recovered and a number of approaches and tools already exist [33, 5]. Other views
to be considered are data and behavior [26]. The selected views affect the choice of program representation as well as the comparison and control strategies.
Appendix B

AQL language design

B.1 Refine analysis environment

We use Software Refinery environment [75] for this research. The important features of this environment include:

- Refine’s high-level and multi-paradigm (rule-based, functional, imperative, and object-oriented) specification language for developing analysis tools;

- an execution platform, based on Common Lisp, to compile and run the analysis programs;

- lexical analyzer and parser generator, based on Lex/Bison, for generating parsers to parse a user-defined context-free language;

- a collection of pre-defined parsers for different languages such as C, Ada, Fortran, PLix, and COBOL, to parse the target programs;

- several databases of code-objects, structured as annotated ASTs, for internal representation of both the target program and the program written in the user-defined language;

- a number of tree traversal, transformation, and matching mechanisms utilities;

- visual facilities for navigation and manipulation of the database objects and their links;

- a variety of more specialized program representations than AST including data/control flow graphs and structure chart; and

- a collection of metrics as means for evaluation and adjustment of the employed algorithm.
B.2 AQL design

We use the Dialect tool from Software Refinery [75], to design the AQL language. The language design consists of two tasks namely domain model design and grammar design.

B.2.1 Domain model

The domain model defines the structure of nodes and edges in abstract syntax trees. Nodes are implemented using objects in the Refine’s object base, and edges are implemented using attributes that reflect the parent/child relationship between nodes. An object class can be a specialization of another object class. This property allows a hierarchy of object classes that inherit the annotation attributes from super classes. Refine contains pre-defined object classes to inherit from, but lacks multiple inheritance. In order to define a domain model, a language designer must:

- design the structure of the desired AST using the knowledge of an object modeling technique and relations such as association, aggregation, and inheritance;
- define node classes and establish tree connections, i.e., association and aggregation, as class attributes. A single tree attribute can link one object to one or more objects; and
- define a set of class attributes as annotation attributes for each node to maintain the node’s properties.

The domain model design directly affects the amount of efforts required for accessing a desired code-object in the object base. Provision of coherency among groups of neighboring objects and open structure are the keys for localizing the change propagation during incremental enhancement of the language features. The above object-oriented design of a domain model facilitates annotating a variety of properties to the nodes for analysis or code generation purposes. The object classes can be instantiated either explicitly using a language construct, or implicitly during the program parsing.

Domain model of the prototype

The object base hierarchy of the prototype AQL is shown in Figure B.1. The main characteristics of this model include:

- supporting future enhancements through a multi-level and coherent structure; and
- generalizing the entities’ common information to prevent redundancy.

B.2.2 Grammar

The grammars are written using a modified version of the BNF model allowing regular expressions in the production rules. The object class attributes are used as the non-terminals and the provisions such as precedence and associativity can prevent generating ambiguous grammars.
Figure B.1: The object inheritance hierarchy of the AQL’s domain model. The ovals represent class objects and the arrows represent the specialization relation where the lower class inherits from the upper class. The top-level User-Object class is provided by Refine.
Bibliography


