To appear in Proceedings of 2nd Euromicro Working Conference on Software Maintenance And Reengineering (CSMR 98), IEEE Computer Society Press, 1998.

Assessing Architectural Complexity

Rick Kazman
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA 15213-3890
+1 412 268-1588
kazman@sei.cmu.edu

Marcus Burth
Department of Mathematics
and Computer Science
University of Mannheim
Mannheim, Germany
burth@pips01.informatik.uni-mannheim.de

ABSTRACT
While it is widely agreed that architectural simplicity is a key factor to the success of large software systems, it is not obvious how to measure architectural complexity. Our approach to measuring complexity is based on observation that large systems with a regular substructure are simple to create and maintain, whereas even relatively small systems created in an *ad hoc* fashion quickly become unmaintainable. This paper describes a system, called IAPR, that aids in architectural exploration and measurement by attempting to match patterns to an architecture. To do this, IAPR implements a heuristic form of sub-graph isomorphism—an NP-hard problem—using the Constraint Satisfaction paradigm to limit the complexity of the problem space.

Keywords
Knowledge-based Software Engineering, Software Architecture, Patterns

INTRODUCTION

It has been well established that large software systems with a coherent, documented, and regular structure are easier to create, maintain, communicate, and modify. Recent work in structural modeling [1.], language design [10.], user interface programming [3.], and design patterns ([2.], [4.]) as well as more general work in design complexity [12.] shows that being able to analyze and build a system with a regular set of architectural building blocks provides substantial benefits. These benefits have been shown to aid human comprehension of complex systems, and hence aid both development and maintenance.

However, even though building systems by using patterns as building blocks seems like an intuitively obvious idea, most developers do not have the luxury of being able to redesign existing systems so that they take advantage of this insight.

As a consequence, it is important to be able to identify *existing* patterns within a code base (as Shaw and Garlan do, for example, in [11.]), so that these may be documented and, just as important, not inadvertently changed. It is also useful to identify areas in an architecture that are similar to known patterns, as these structures will be obvious targets for re-engineering efforts.

This paper introduces the IAPR (Interactive Architecture Pattern Recognition) system. IAPR is a system for locating user-specified at varying levels of abstraction patterns within a software architecture. IAPR can be used in two related ways: as a diagnostic tool or as an exploring tool. As an exploring tool, IAPR will match user-specified patterns to an architecture, and can locate areas of the architecture that are "near" matches. As a diagnostic tool, IAPR provides a significant new architectural complexity metric for designing systems. A system can be measured according to its pattern coverage: the *proportion* of an architecture that can be covered by patterns and the *number* of patterns it takes to cover an architecture. These are complementary measures of the system's regularity, and hence its architectural complexity.

Currently there are few architectural metrics. The ones that do exist measure some form of coupling and cohesion ([6.], [8.]), or fan-in and fan-out [5.]. These measures do not reliably correlate with architectural complexity for two reasons. First, counter-examples abound: for instance typical utility routines and controllers have high fan-in and fan-out respectively, but may be otherwise unproblematic. Second, and more important, is that measures such as coupling/cohesion and fan-in/fan-out are not truly architectural metrics. These measure the complexity of individual parts of an architecture, but give no indication of the architecture's overall complexity. IAPR, on the other hand, measures architectural *regularity*, which, we will argue, directly corresponds to architectural complexity.

As a side note, it is important to make clear that when we speak of patterns we are *not* restricting ourselves to object-oriented design patterns. The patterns that we use can be in any architectural view and may be *any* connected set of components and connectors: processes, objects, filters, databases, procedure calls, sockets, RPC, and so forth. In addition, IAPR patterns can be arbitrarily hierarchically nested, so that, for example, the database of a system might be represented—and matched—as a single architectural element (if it was not important for the specific analysis) whereas other sub-systems might be described down to the level of processes or even procedures.

OVERVIEW OF THE IAPR SYSTEM

IAPR analyzes a software architecture with respect to a set of patterns. Both patterns and architectures are represented in IAPR as annotated graphs. Thus the pattern matching problem is a sub-graph isomorphism problem, which is known to be NP-hard. To address the complexity of this problem, heuristic methods are vital. We have thus cast IAPR as a Constraint Satisfaction problem, following Woods and Yang [12.]. In fact, the entire problem of matching architectural patterns to an architecture is nothing more than a reverse engineering—or program understanding—problem at a higher level of abstraction.

Woods and Yang's PU-CSP (Program Understanding Con-

straint Satisfaction Problem) algorithm is based on modeling program plans as templates and then matching these templates to some repository of legacy source code. Although this problem is NP-hard, PU-CSP has been shown to be computationally tractable in large programs. This tractability is achieved through dramatic reductions in the size of the search space. These reductions are achieved by the enforcement of *structural* and *knowledge* constraints. Structural constraints are things like scope, control, precedence, and data-sharing relations found within the legacy code base. Knowledge constraints come from the program plan library itself, and refer to relationships among plans.

Similarly, IAPR, as will be shown, has successfully analyzed architectures of up to 1,000 elements (components and connectors).[1] It does this by restricting the size of the search space through *attribute* and *inter-element* constraints, as will be discussed in the next section.

The IAPR system offers a *user-friendly* method of matching a set of architectural patterns against a software architecture. Patterns and architectures both are created and represented through interaction with a GUI (graphical user interface). Through the GUI users "draw" the architecture, and specify the features of architectural elements. The GUI also supports the grouping of elements into sub-systems. These sub-systems can be shown fully expanded, or can be collapsed into a single node in the graphical representation, for ease of analysis and overall system comprehension.

The IAPR system was developed for two classes of use:

- *As a Diagnostic Tool*: Using the tool for analyzing an existing software architecture presupposes that it has been built with structural patterns in mind. This is often true for large scale projects, where a well structured design is necessary for successful development. The tool matches the set of specified patterns against the architecture and generates a measure of the architecture's complexity. The complexity of an architecture is specified in terms of the proportion of the architecture covered by some patterns (i.e. the regularity of architectural substructure) and the number of patterns used in the architecture (i.e. the pattern complexity of the architecture).
- *As an Exploring Tool*: Frequently, a programmer is asked to modify an architecture that they did not design and which they are unfamiliar with. Such an architecture must first be understood before it can be modified. To support this process, IAPR aids in user-driven exploration of an architecture. The user can search for patterns with a particular topology and with particular element properties: connection properties such as fan-in and fan-out, element names (as regular expressions), and other element features such as whether they store data, transform data, block, etc. IAPR helps the user to interactively explore architectural patterns. In particular, this is useful for detecting design problems such as "layer bridging", as will be shown.

Each of these uses of IAPR will be discussed in more detail and demonstrated in the section on Case Studies.

1.We have chosen the term "element" consciously, to be agnostic with respect to whether we are dealing with a component or connector.

System Description

The IAPR System has four functions, as shown in Figure 1: software architecture modeling, pattern modeling, pattern matching, and visualization of the results.

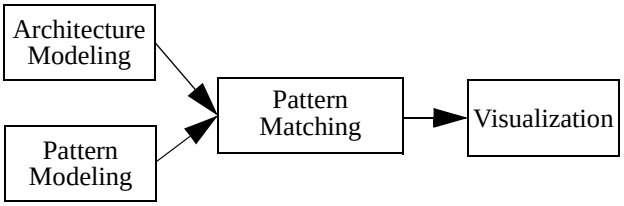


Figure 1: IAPR Functional Architecture

Architecture Modeler

This function, called SAAMtool [9.], enables the user to model the software architecture in an intuitive graphical representation—a components and connectors representation. The components and connectors may be atomic primitives—the primitives that designers commonly work with, such as object, procedure, shared memory, pipe, RPC, procedure call, etc.—or they may be composite: some collection of any of the primitive types grouped into a subsystem. For example, in Figure 2, all list and hash table manipulation routines are grouped together into a single composite node called "Utility Functions". This composite node appears in blue at the bottom of Figure 2.

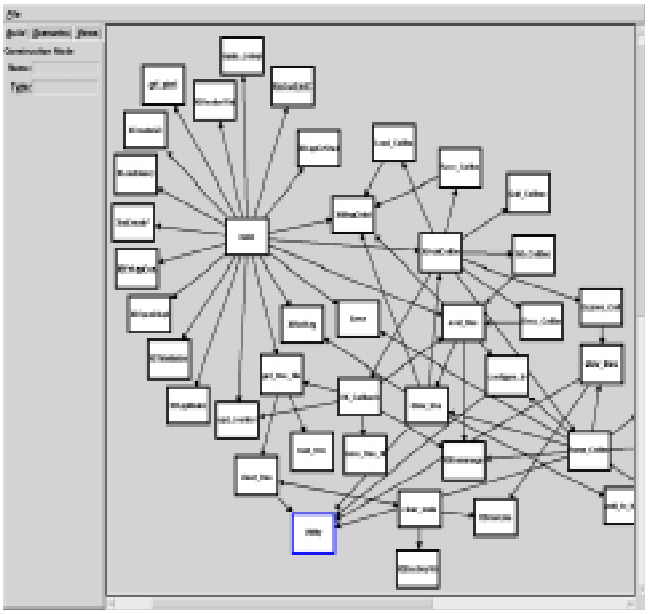


Figure 2: A Portion of a Sample Modeled Architecture

Representing a software architecture requires focusing on multiple aspects of the system. Designers regularly use multiple views of a system—such as a source view, an object (or procedural) view, a dynamic view, and a functional view—to describe a system's architecture [11.]. Since a complete software system incorporates all of these aspects, our underlying representation of a software architecture combines the information requirements of these various

views.

We model an architecture as a hierarchically nested graph of architectural *elements* which are described by a number of attributes and inter-element relationships [7.]. Each element is comprised of *static* and *temporal* features, which describe the element's capabilities and properties from a time-invariant static perspective, and from an execution-time behavioral perspective. These features can be recursively computed (that is, the features of a sub-system can be computed by looking at the collection of features of its constituent elements). For example, a sub-system that consists of a filter—which transforms data—and some shared memory—which stores data—would be represented as an element that both stores and transforms data. Thus we can formally characterize *any* architectural element, whether it is atomic or composite, using a single representation scheme. This is crucial to our ability to be able to match patterns at any level of resolution.

As static features we consider attributes such as data scope (the largest scope across which data can be passed by the element), control scope (the largest scope across which control can be passed by the element), ports (places where this element can bind with some other element), element binding times, times of both data and control acceptance and relinquishment, and whether the element transforms data.

For temporal features we consider properties such as the times at which data and control may be accepted, whether the element "forks" (causes a new thread of control to be created), whether it retains state, whether it blocks, and whether it relinquishes control. These features are sufficient to describe a space of architectural elements from the simplest (procedures, files, shared memory, etc.) to the most complex (processes, semaphores, threads, RPC, etc.).

The features may be *attributive* (those that have a set of enumerable values) or *relational* (those that only have a meaning with respect to some other architectural element). For example, the data scope feature is attributive. It can take on the values: V (data can come from and go to only other elements in the same *virtual* address space), P (data can come from and go to other elements in the same *physical* computer space, but different virtual spaces), D (data can come from and go to *different* physical machines, across a network), or n/a (this element never receives or transmits data).

As we have already mentioned, users typically (and informally) categorize architectural elements as either components and connectors. However, there is no *principled* distinction between these two categories: connectors involve computation (sometimes a considerable amount), and components move information from one place to another. Both need to have their ports bound to other elements to do meaningful work. Therefore, we do not distinguish between components and connectors in our model, since their meaning is derived only intuitively by the user depending on the specific context of the system under investigation. What is a connector to one developer or architect (say, the user of RPC) is a component to another (say, the developer of an RPC service). We model every element solely as a set of static and temporal features, and those features are as relevant to so-called components as they are to so-called connectors.

However, when modeling the architecture we want a user to be able to transfer the intuitive notions of components and connections into a graphic representation. This means that putative component-like elements are represented as "boxes" in our graphical user interface, and elements which are understood to be connectors are shown as lines between the boxes. Internally, however, there exists no criterion which qualifies one as a component or a connection and so they are processed indistinguishably. This uniform representation for architectural elements means that we can compare *any* architectural element with any other in IAPR. Without this representation, it is unclear what it would mean to compute "near" architectural matches. For example, an object (and all its methods) specified by a user in a pattern should be considered "near" to abstract data type (and all its procedures) in an architecture. A CORBA-enabled object, servicing requests on various machines across a network should be considered a "near" match to a client-server pattern. And so forth. Without our abstract formal, feature-based representation of architectural elements, the computation of such near matches would not be possible in any rigorous way.

Relationships among elements, in this view, are expressed solely through the *relational* features of the participating elements. For example, consider a trivial example with two processes, A and B, where A sends data to B through a socket. A user would typically express this system, through the GUI, as two boxes representing Process A and Process B connected by a line, representing the socket connection. *Internally*, however, we would represent the graph of this architecture as three connected data structures:

where the boxes are data structures for independent architectural elements and the lines are relational features linking the elements' representations. The philosophy behind this representation is to simultaneously support the user's intuitive understanding of the software architecture while providing an internal format that we can analyze and reason about.

Pattern Modeler

The Pattern Modeler allows a user to create, save, view, and modify architectural patterns through a GUI that is almost identical to that of the architectural modeler (these two functions are simply different operating modes of SAAMtool). An architectural pattern describes a constellation of elements. This constellation is intended to represent a typical and/or critical substructure of the software architecture under investigation. Our goal, as described in more detail below, is to match user specified patterns against an architecture, to filter meaningful information out of the architecture.

Patterns use the same formal representation techniques discussed for the Architecture Modeler. We use all the static and temporal features with their attributive or relational characteristics to describe pattern elements. However, in addition to the architecture modeling functionality, SAAMtool provides

a comprehensive set of *pattern-specific* specification mechanisms, to create sophisticated patterns that allow us to investigate a software architecture effectively and efficiently.

The pattern-specific facilities are as follows:

- *Specifying Ranges*: to leave some features undecided or to prescribe them to be within a certain range rather than giving them a precise value. For instance, the permitted number of input ports of an element could be restricted to be greater than zero and less than ten.
- *Element names specified as regular expressions*: for example, in one of our case studies, we take advantage of the fact that all Xlib functions begin with the letter "X", while all X toolkit functions begin with "Xt".
- *Cardinality of relationships between elements*: the user can model compositions of one-to-one, one-to-many, or many-to-many relationships. For example, $1 \rightarrow M$ matches the pattern where one element is related to *M* (one or more) elements, where the *M* elements are of the same type (a server-client relationship, for example).
- *Excluding Features*: just as a user can specify the presence of element features or inter-relationships, it is also possible to explicitly exclude the existence of some features or feature values.
- *Hard/Soft Features*: Often pattern elements must satisfy a minimum set of properties to meet the basic characterization of the pattern. This set we call the *hard* features of the pattern, since without their satisfaction there is no match. On the other hand there exist features that typically belong to the pattern, but are not absolutely necessary to characterize its fundamental structure and behavior. These pattern features—attributive as well as relational ones— can be "softened" by explicitly specifying them as *soft* features. The use of soft features enables us to detect "near" pattern matches.

For example, it is a common practise in legacy software, when modifying a sub-system to take a complete copy of the sub-system and modify it, because the maintainer is afraid of the unknown consequences of changing the existing sub-system. This results, over time, in software "bloat", where the system grows increasingly larger, but where a substantial part of this bulk is near-identical copies of the same software. Such a system is an ideal candidate for re-engineering. But first the nearly identical sub-systems must be identified. This is a clear case where near pattern matching is crucial.

Thus near matches could be the starting point for the analyst who wants to propose modifications on the architecture to gain better uniformity (and hence easier maintenance). By re-engineering the architecture, its complexity can be reduced—a higher proportion of the architecture's elements will be matched by a single pattern. The modified architecture possesses more regular substructure according to the matched pattern and promises to be more manageable, understandable, and maintainable [1.].

Pattern Matching

The Pattern Matcher function performs the core computation in the IAPR system. It performs pattern matching based on the Constraint Satisfaction Paradigm (CSP). This section will show first how the CSP is applied to the IAPR domain and second why the CSP is appropriate.

The input for the Pattern Matcher is the representation of the architecture created by SAAMtool's Architecture Modeler and a set of patterns created by the Pattern Modeler function. The Pattern Matcher outputs the complete set of full and near pattern matches in the architecture, and provides these to the Visualization component for user interaction.

Constraint Satisfaction problems consist of three components: a *set of variables*, a *set of values*, and a *set of constraints*. The goal is to find all legal assignments of the values to the variables such that all constraints are satisfied. For our domain, the pattern elements represent the variables and the architecture elements denote the values. The constraints are stated in the pattern and are directly derived from the specification of features in the Pattern Modeler. We are interested in finding all possible instances of a pattern in the architecture. A pattern is detected when all pattern element variables can be assigned architecture element values, such that all feature constraints can be satisfied.

We categorize constraints as *hard* and *soft*, which correspond to the meanings of hard and soft features in the pattern specification. As before we group these subdivisions into attributive and relational constraints. So, we must process four types of constraints: hard attributive constraints and hard relational constraints, which represent the "must-have" features, and the soft attributive and relational constraints, which denote relaxed matching requirements.

The algorithm we are employing, based on [12.], applies a backtracking search strategy coupled with mechanisms that attempt to propagate consistency throughout the problem space before and during search. For instance, we make use of arc-consistency and other "look-ahead" strategies to limit the search space before and during search as far as possible. In addition we employ domain-dependent heuristics such as reordering of variables through which we attempt to rule out dead search paths at the earliest possible time. IAPR's execution phases are described next.

In the pre-search phase the variable domains are determined by checking the consistency of variables (pattern elements) and values (architectural elements) based on the hard attributive constraints. These constraints can be checked without knowing future inter-element relationships. Only consistent values according to the hard attributive features are considered for matching to the variables. These sets of legal values for each variable we call the *variable domains*. The variable domains determine the upper bound of the search space, so that reducing the domains at this stage has enormous effects on the overall problem complexity.

The second phase is the backtracking search process traversing the problem space by extending the variable set under consideration step by step. For each variable assignment the hard relational constraints are tested. As soon as all applicable hard relational constraints are satisfied, the next variable and its domain are taken in account. A solution to the problem—a pattern match—is found, if all variables are instantiated such that no hard constraints are violated. The earlier a

violation is found the earlier a dead path is pruned which greatly saves computational resources.

Because hard constraints can cause a dramatic reduction in the size of the search space, we want to check those constraints first. Consequently, the ordering of variables and constraints is crucial to limit the search space. Our strategy is to put interrelated pattern elements (variables) close together in the variable set. So, we attempt to ensure that when instantiating the re-ordered variable set hard relational constraints are tested first. This re-ordering alone caused a reduction of the search time by a factor of 500 in our experiments. By the end of the second phase, all matches that satisfy the hard constraints are determined.

During the third phase, all the pattern matches are inspected for violated soft constraints and are ranked according to the number of violated constraints. Full matches satisfy all the soft constraints, whereas "near" matches contain some features that could not agree with the corresponding architecture element specification. The Visualizer will present these "near" matches to the user, prioritizing those patterns with the fewest violated soft constraints first.

Visualization of Results
The main purpose of this module is visualizing the results (the computed pattern matches) interactively to the user and providing metrics of an architecture's pattern coverage as indicators of its complexity.

The pattern matches can be presented to the user all at once or, more typically, one by one, so that the user is able to examine each pattern's validity in the specific architectural context. The order of presentation to the user is determined by the quality of a match, which means full matches (all hard and soft constraints satisfied) are shown first, followed by near matches, which contain one ore more violated soft constraints. The matches are shown graphically overlaid on the original architecture. The user is led through these matches by the system. This interaction is performed to arrive at a complexity measure that only refers to true, semantically and syntactically valid matches. As a result, the user receives two complementary indicators of an architecture's complexity:

First, the percentage of architecture elements that could be covered by the pattern set. This number indicates the degree of *regular* structure is the architecture. If the percentage of matches is low, the user might reconsider the overall design of the architecture, might make increasing the structural regularity a goal of a re-engineering process, or at least keep in mind that potential problems regarding modifiability and maintainability that might arise in future work with the software under investigation. This indicator can be split up into per pattern contributions, to get a measure of the effectiveness of individual patterns. All other things being equal (and they seldom are), we would favor a pattern which covers a large part of the architecture over patterns which are only able to cover a tiny part of the architecture.

The second indicator is the number of *distinct* patterns needed to cover the architecture. An architecture matched by only one pattern completely is less complex than an architecture that contains a multitude of different patterns (in the pathological case, there might be as many patterns as elements). In such a situation the designer or maintainer would have to reason about a large variety of heterogeneous substructures during the developing and maintenance phase. In the case where the architecture is matched by a few patterns, we could focus our attention on a small number of structural primitives. For example, the PAC paradigm for user interface construction advocates creating entire interactive systems from a single structural pattern [3.]. The structural modeling paradigm for flight simulators uses only seven different patterns.

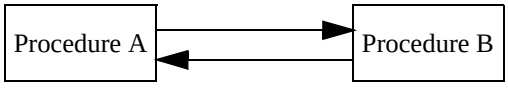
CASE STUDIES
We will now look at two case studies that illustrate the functionality, performance, and limits of IAPR.

tree_builder
This case study analyzes an interactive program for creating, annotating, and laying out two-dimensional hierarchies, called tree_builder. This system makes extensive use of Xlib and the X toolkit for its user interface. The tree_builder case study illustrates how the IAPR tool can aid in analyzing and understanding a system that was built in an *ad hoc* fashion, with no attention given to the overall system architecture. The tool guides the analyst through the architecture by locating and displaying user-specified patterns.

The representation of the tree_builder shows 42 component-like elements and 61 connection-like elements. There are only two architectural primitives used: procedures (represented as boxes in Figure 2) and procedure calls—both standard calls and callbacks—represented as connectors.

The patterns
In this case study we concentrate on patterns that might illuminate critical design aspects (and potential design flaws) in the tree_builder system. In the following we introduce five example patterns for this purpose.

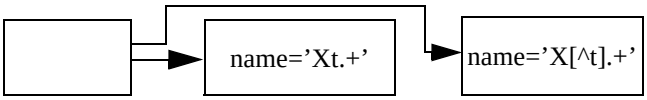
1. Mutual Recursion
This pattern addresses the case when two procedures call each other. This form of recursion typically involves complex dependencies making subsequent modifications difficult. Thus, an analyst might want to put extra effort into documenting and testing this part of the architecture. Otherwise, downstream maintenance may require more resources than expected. This pattern is represented as follows:



It took the Pattern Matcher 12 seconds of CPU time on a Sparc-20 to search the entire tree_builder architecture for this pattern; only one occurrence was found.

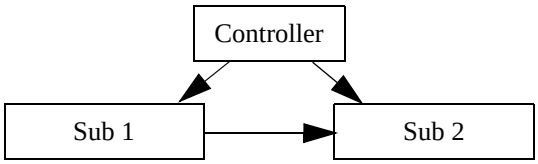
2. Layer Bridging
The tree_builder system utilizes a variety of the X toolkit functions for creating interaction widgets. The Xt intrinsics is, in turn, built on top of Xlib (the X library), which provides a portable abstraction of a workstation's graphical capabilities. It is not only interesting for an analyst to know where these Xlib and Xt calls are located in the architecture (because these functions would need to be ported if the

application was moved to, say, a Windows-based platform). Even more important, however, is to know where Xt and Xlib functions called by the *same* component. Since the X toolkit is layered on top of Xlib, this situation is a form of layer bridging. Layer bridging presents a substantial impediment to the future modifiability and portability of a system. The graphical representation of the Layer Bridging pattern used in this case study is as follows:



where *name* is specified as a regular expression. The remaining features of the elements are set to the architectural primitives of procedure and procedure call. Running IAPR on this pattern resulted in 37 matches in the tree_builder architecture in 7 seconds of CPU time.

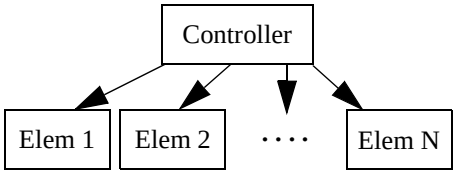
3. Crosstalk
The Crosstalk pattern, shown below, involves a controller and its subordinates, where the subordinates communicate with each other (and not solely with the controller). This construction violates the hierarchical composition of the system, and results in added complexity.



The Crosstalk pattern was matched 3 times, taking 350 CPU seconds. The CPU time was higher for this pattern because it contained no attribute constraints and so the Pattern Matcher could only use structural constraints in matching this pattern to the architecture. As a consequence of the absence of attribute constraints the search space was at its upper bound: $O(D_p^3 * D_c^3)$, where D_p is the number of components (in this case 42 procedures) and D_c is the number of connections (in this case 61 procedure calls). This results in an upper bound of approximately 17 billion possibilities to check for in the relatively modest tree_builder architecture.

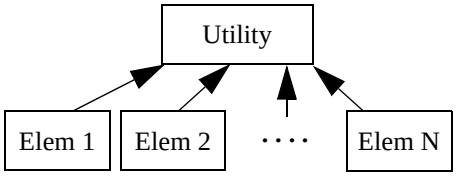
The next two patterns are less domain dependent and so could be applied to almost any architecture.

4. Controller
The Controller pattern describes modules that have many outgoing connections to other elements of the same type. A controller thus has high fan-out. This pattern highlights areas of potential performance bottlenecks in the system. The Controller pattern is specified as follows:



The results of matching the controller pattern to tree_builder are: 15 matches with fan-outs up to 19 connections. These matches can be investigated separately, so that the analyst gets a better understanding of the main control modules of the system. For example, the routine "main" has the highest fan-out with 19 connections as it would be expected for a main control module. Beside "main" there are two other modules with fan-outs of 8. The analyst should be aware of those subsystems and might scrutinize them more closely.

5. Utility
As counterpart to Controller, Utility is a pattern where one component is controlled by many others (and so this component has high fan-in) as follows:



A high fan-in of an architectural element can be desirable since it identifies reusable components (utilities) or at least opportunities for reuse. Utilities are typically highly cohesive and are minimally coupled with the rest of the architecture. The Pattern Matcher identified elements in tree_builder with a fan-in of up to 5 in-coming connections. These elements should be closely scrutinized since, being highly used, they should be efficient and robust.

To conclude this case study, we note that we have illustrated some useful patterns as starting points for understanding the tree_builder system. Some of these patterns were domain dependent, such as the "layer bridging" pattern, and others seem to be applicable to a broad range of architectures (e.g. Controller pattern). Filing and cataloguing the latter group of patterns is a further step towards an efficient approach to unfamiliar architectures. The Pattern Modeler provides a tool for creating and managing a library of patterns.

This case study described patterns used to explore the architecture, looking for areas of high risk, high potential complexity, or opportunities for reuse. The next case study concentrates on the scalability of the IAPR approach.

Synthetic System
As already mentioned, the subgraph isomorphism problem is NP-hard and requires heuristic approaches to limit the size of the problem space for it to be tractable. Our goal is to have the IAPR system provide reasonable results within acceptable time bounds for problems of industrial size. The limiting factor of the IAPR system in this respect is the algorithm implemented in the Pattern Matcher.

This case study presents empirical evidence of IAPR's efficiency on architectures of varying sizes, and patterns of varying complexity. For this purpose, we generated large scale synthetic architectures of sizes 100, 200, 500 and 1000 elements, where "components" and "connectors" are distributed with a ratio of 2:3. Each element feature takes one of three possible values which are uniformly distributed through the architecture. This means that when specifying a hard feature of a pattern element, we reduce its variable

domain to 1/3 of its original size. When specifying a second feature for the same element we reduce the domain to 1/9 of the original domain size, and so forth.

We designed three patterns with 3, 6 and 11 elements. The first one represents a directed connection between two elements (such as two procedures connected by a procedure call or two filters connected by a pipe); the second one contains a cycle of connections within three components (i.e. three components that are in a mutual recursion relationship); and the third one specifies a version of the Controller pattern, as described above, where one component controls 5 other components that do not have to be identically specified.

Table 1 shows the response time results, in CPU seconds. Execution was halted at 7200 seconds (2 hours), indicated by ">". For each pattern, the size and number of specified features is shown. These results show that the response times for all patterns are in an acceptable range for the architecture with 100 elements, when specifying *no* features. This is already a relatively large high-level architecture, as industrial experience has shown.

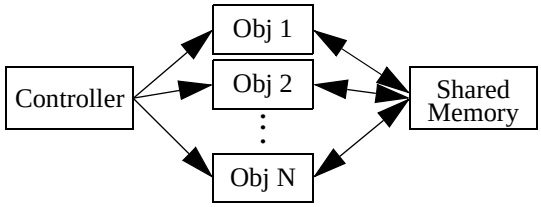
Pattern		Architecture Size			
Type	# features	100	200	500	1000
Simple Connection	0	4	92	>	>
	3	1	4	279	6977
	6	1	1	2	19
Cycle	0	45	1400	>	>
	3	1	21	1648	>
	6	1	2	97	2503
	12	1	1	3	20
Controller	0	178	>	>	>
	6	1	10	1096	>
	11	1	8	451	>

Table 1: Pattern Matching Response Times

However, this case is improbable: the user will always have some knowledge about the elements and will be able to determine a few features. The effects of specifying even a small number of features are dramatic: we get reasonable response times for architectures up to 500 elements for all patterns, and acceptable response times for the 1000 element architecture for the patterns containing up to 6 elements and some features specified. Note also that this system was written in Lisp. A reimplementation in C or C++ would give us at least an order of magnitude performance improveent.

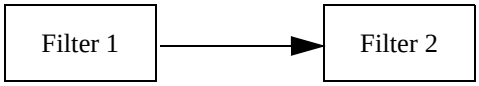
Synthetic Pattern-based System

Our final case study illustrates how IAPR can be used to assess the overall *architectural complexity* of a system, based upon pattern coverage. For this purpose we generated a synthetic system composed of 130 elements and built with 2 structural patterns in mind. The first pattern consists of a controller element that controls a group of objects that all access shared memory. This pattern further has the restriction that the objects must not be connected to any elements other than the controlling module and the shared memory. For example, they may not call each other directly.



This pattern exhibits the use of IAPR features such as 1-M relationships, and the explicit *exclusion* of additional connections by restricting the objects to have only two incoming connections and one outgoing. The fan-in/fan-out restriction is specified as a soft feature since its satisfaction is desirable rather than a strict necessity. The elements used for creating this pattern are Objects, Shared memory, Procedure Calls, shared memory accesses (both reading and writing), and a controller.

The second pattern is a simple filter-pipe-filter connection as shown below. Each of the filters runs as a distinct process and the pipe, provides a pure data (i.e. no control) connection between them.



Our synthetic system is built both from these two patterns, and from some additional, non-pattern related elements as "glue". Applying the IAPR system to this problem domain yields following results:

Pattern	Violations	Matches	Coverage
1	0	6	54%
	2	2	13%
2	0	1	2%
Total		9	70%

Table 2: Pattern Matches on the Synthetic Architecture

In Table 2, each pattern is listed separately and shows the individual contributions to the total number of matches and coverage of architecture elements. The user interface showing some of the matched elements in given in Figure 3. The quality of each match is ranked in terms of the number of violated soft constraints. Table 2 shows that 70% of the architecture can be covered by using only two patterns. However, more interesting in this case is the distribution of the coverage proportion to the two patterns. Pattern 2 (shown in pink in Figure 3) in fact covers only 2% of the architecture through a single match, whereas pattern 1 matches 54% of the architecture in six full matches (shown in yellow) and 13% in two near matches (shown in orange) and so covers a total of 68% of the architecture's elements. Thus, an analyst

would be well advised to concentrate on the first pattern and might question the second one as being "representative" of the architecture.

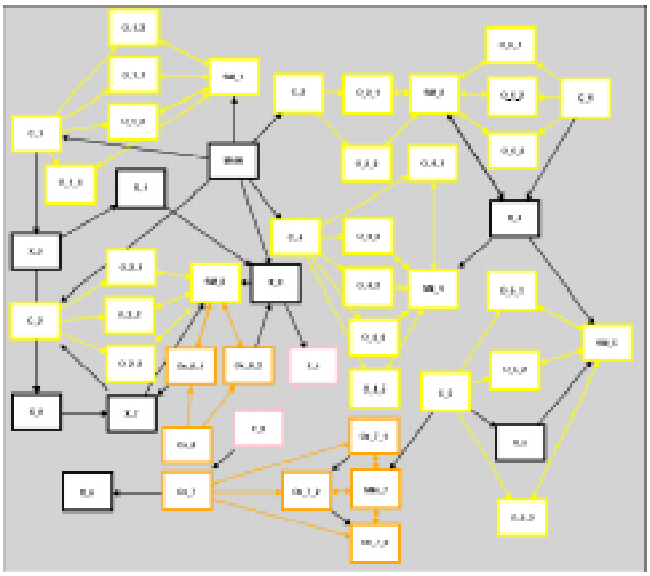


Figure 3: Matched patterns visualized on the architecture

The near matches of pattern 1 did not match exactly because they contain connections that were not specified in the pattern. For example, there are connections between objects Ox_7_1, Ox_7_2, and Ox_7_3. The pattern specifies that objects should only be connected with the controller and the shared memory, and not with each other. Upon seeing near matches, the user has three possibilities: change the architecture to conform with the patterns; create a new sub-pattern that covers this case; or accept the violation as is. In each case, the user's understanding of the regularity and complexity of the architecture is made explicit.

If the user chooses to re-engineer the architecture to conform to the patterns, fewer distinct primitive structures are required. This increases the regularity of the architecture, promotes the reusability of these structures (i.e. developers will be more willing to make generic versions of these structures that are robust, efficient, and flexible), and causes a reduction in the *idiosyncratic* code and structural complexity of the system, thus reducing the complexity of the system.

CONCLUSIONS

In this paper we have introduced a system for discovering patterns within software architectures. We have shown that this task, although NP-hard, can be made tractable for problems of real-world size by employing the Constraint Satisfaction Paradigm. However, one might ask why we are doing this at all. After all, humans are, by nature, excellent pattern recognizers and matchers. So, why are we trying to automate pattern matching in architectures? There are several reasons:

- *architectural matching employs abstract features*: far too many of them to visualize simultaneously. However, an automated algorithm can check any number of features. In fact, the large number of features is to the algorithm's advantage, as these constrain the search space;
- *the visualization space is large*: even moderately complex architectures, such as that shown in Figure 1, are large enough, and have such complex (often non-local) interconnections, that all parts of a pattern will not always be in the user's current view;
- *an architecture may be visualized at differing resolutions*: for example, one sub-system may be displayed as a single node whereas another one might be completely expanded. So, once again, the user may not be able to view all relevant information at once;
- *there are exponentially many possibilities to check*: a human will get bored or tired before checking them all.

The IAPR system can help to alleviate these problems, and introduces a new design criterion of *architectural simplicity.* In addition, the IAPR system can aid in the architectural design and maintenance process by helping an analyst understand an architecture. As an exploring tool, IAPR can help a new maintainer in orienteering through the architecture, and can identify potential problem areas. As a diagnostic tool, IAPR graphically demonstrates the pattern complexity and pattern coverage of a software architecture. Furthermore, in either case, the locating of near matches highlights potentially ripe areas for re-engineering.

REFERENCES

1. G. Abowd, L. Bass, L. Howard, L. Northrop, "Structural Modeling: an Application Framework and Development Process for Flight Simulators", SEI Technical Report, CMU-SEI-93-14, 1993.
2. K. Beck, *et al*, "Industrial Experience with Design Patterns", *Proceedings of ICSE 18*, 1996, 103-114.
3. J. Coutaz, "PAC, An Implementation Model for Dialog Design", *Proceedings of Interact '87*, 1987, 431-436.
4. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns—Miroarchitectures for Reusable Object-Oriented Software*, Addison-Wesley, 1994.
5. S. Henry, D. Kafura, "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering*, SE-7(5), 1981.
6. G. Heyliger, "Coupling", *Encyclopedia of Software Engineering*, J. Marciniak (ed.), 220-228.
7. R. Kazman, P. Clements, L. Bass, G. Abowd, "Classifying Architectural Elements as a Foundation for Mechanism Matching", *Proceedings of COMPSAC 1997*, 1997, 14-17.
8. R. Kazman, G. Abowd, L. Bass, P. Clements, "Scenario-Based Analysis of Software Architecture", *IEEE Software*, November, 1996, 47-55.
9. R. Kazman, "Tool Support for Architecture Analysis and Design", *Joint Proceedings of the SIGSOFT '96 Workshops (ISAW-2)*, 1996, 94-97.
10. G. Krasner, S. Pope, "A Cookbook for Using Model-View-Controller User Interface Paradigm in Smalltalk-80". *JOOP*, August/September 1988, 26-49.
11. M. Shaw, D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
12. H. Simon, *The Sciences of the Artificial*, MIT Press, 1981.
13. S. Woods, Q. Yang, "The Program Understanding Problem: Analysis and a Heuristic Approach", *Proceedings of ICSE 18*, 1996, 6-15.