

SPQR: Formalized Design Pattern Detection and Software Architecture Analysis

Jason McC. Smith, David Stotts
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175
{smithja, stotts}@cs.unc.edu

Abstract

We present formal analysis methods and results from SPQR, the System for Pattern Query and Recognition, a toolkit that detects instances of known design patterns directly from object-oriented source code in an automated and flexible manner. Based on previous work in rho-calculus (extended Abadi/Cardelli sigma-calculus) and Pattern/Object Markup Language (POML), the SPQR toolset is easily retargetable to any OO language, though our current results are for C++ programs. In this paper we present an overview of the current SPQR implementation, as well as both positive and negative results from running this tool on production C++ code. We also discuss how the basic formalisms can be applied to other software analyses such as refactoring support and architectural evaluation and comparisons.

1 Overview

Design patterns, first described for OO software by Gamma et al. [11], have become a common vocabulary in which software developers discuss and communicate design and architecture ideas. Identifying instances of design patterns in source code is an important problem which has, until now, been elusive; if solved effectively, it would greatly assist software developers to produce systems which adhere to good design principles. Effectively identifying design patterns in source code would help in maintenance, comprehension, refactoring and design validation during software development.

The abstract and informal nature of design patterns is what makes them both valuable as a design vocabulary (they can succinctly encapsulate a vast number of highly variable architectures and implementations) and difficult to formalize for effective analysis and tool support. Our approach formalizes these concepts by two main strategies: classic divide-and-conquer (we seek easy-to-find small, foundational patterns and use automated inference to compose

them into the larger algorithm- and system-scale structures), and formalized variability (we create within our semantics notions of variance from ideal structure).

SPQR (System for Pattern Query and Recognition)[24, 25, 26] is a toolset that implements our methodology for practical analyses of source code. We use a formal denotational semantics to encode fundamental OO concepts which we term Elemental Design Patterns (EDPs), and a small number of rules which we call reliance operators, for combining these concepts into larger patterns. These reliance operators, when combined with the sigma-calculus[1], provide a formal foundation we call the rho-calculus. In the following sections we will give a brief discussion of the theory behind SPQR and present results gathered from applying the tools to a body of production C++ code.

1.1 Previous Related Research

The design semantics described in this paper are based in denotational object semantics [1], automated reasoning and automated deduction [13], OO design patterns [11], a formal set of flexibility and abstraction operators, and several forms of composition.

The decomposition and analysis of patterns is an established idea, and the concept of creating a hierarchy of related patterns has been in the literature almost as long as patterns themselves [5, 12, 21, 32]. The few researchers who have attempted to provide a formal basis for patterns have most commonly done so from a desire to perform refactoring of existing code, while others have attempted the more pragmatic approach of identifying core components of existing patterns in use. Additionally, there is ongoing philosophical interest in the very nature of coding abstractions, such as patterns and their relationships.

Refactoring approaches. Attempts to formalize refactoring [10] exist, and have met with fairly good success to date [6, 15, 18]. The primary motivation is to facilitate tool support for, and validation of, the transformation of code from one form to another while preserving behaviour. This is an important step in the maintenance and alteration of

existing systems, and patterns are seen as the logical next abstraction upon which they should operate. Such techniques include fragments, as developed by Florijn, Meijers, and van Winsen [9], Eden’s work on LePuS [7], and Ó Cinnéide’s work in transformation and refactoring of patterns in code [17] through the application of minipatterns. These approaches have one missing piece: appropriate flexibility of implementation.

Structural analyses. An analysis of the ‘Gang of Four’ (GoF) patterns [11] reveals many shared structural and behavioural elements, such as the similarities between Composite and Visitor [11]. Relationships between patterns, such as inclusion or similarity, have been investigated by various practitioners, and a number of meaningful examples of underlying structures have been described [3, 5, 21, 30, 31, 32].

Objectifier: The Objectifier pattern [32] is one such example of a core piece of structure and behaviour shared between many more complex patterns. Its Intent is to:

Objectify similar behaviour in additional classes, so that clients can vary such behaviour independently from other behaviour, thus supporting variation-oriented design. Instances from those classes represent behaviour or properties, but not concrete objects from the real world (similar to reification).

Zimmer uses Objectifier as a ‘basic pattern’ in the construction of several other GoF patterns, such as Builder, Observer, Bridge, Strategy, State, Command and Iterator. It is a simple yet elegantly powerful structural concept that is used repeatedly in other patterns.

Object Recursion: Woolf takes Objectifier one step further, adding a behavioural component, and naming it Object Recursion [31]. The class diagram in Figure 2 is extremely similar to Objectifier, with an important difference, namely the behaviour in the leaf subclasses of *Handler*. Exclusive of this method behaviour, however, it seems to be an application of Objectifier in a more specific use. Note that Woolf compares Object Recursion to the relevant GoF patterns and deduces that: Iterator, Composite and Decorator can, in many instances, be seen as containing an instance of Object Recursion; Chain of Responsibility and Interpreter do contain Object Recursion as a primary component.

Conceptual relationships. Taken together, the above instances of analyzed pattern findings comprise two parts of a larger chain: Object Recursion contains an instance of Objectifier, and both, in turn, are used by larger patterns. This indicates that there are meaningful relationships between patterns, yet past work has shown that there are more primary forces at work. Buschmann’s variants [4], Coplien and others’ idioms [2, 5, 14], and Pree’s metapatterns [19] all support this viewpoint. Shull, Melo and

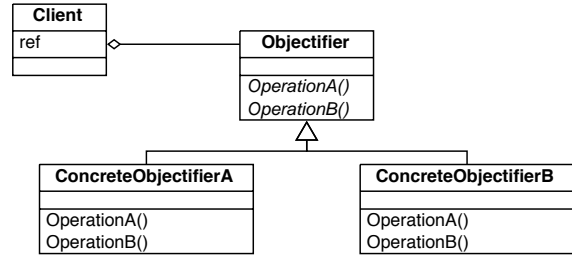


Figure 1. Objectifier

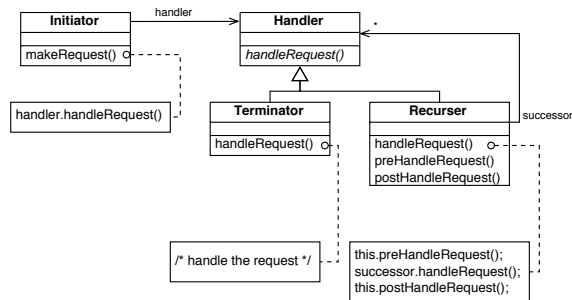


Figure 2. Object Recursion

Basili’s BACKDOOR’s [23] dependency on relationships is exemplary of the normal static treatment that arises. Relationships between *concepts* are vital to the flexibility to the practitioner implementing patterns in design, through constructs we term *isotopes*, which have been discussed in prior publications [26]. A related, though type-based approach that works instead on UML expressed class designs, is Egyed’s UML/Analyzer system [8] which uses abstraction inferences to help guide engineers in code discovery. Reiss’s PEKOE [20], though similar in nature to SPQR, uses, instead, a relational database language for queries and conceptual component definition.

1.2 A Design Calculus: Basic Concepts

OO design patterns[11] have become a common vocabulary in which software developers discuss and communicate design and architecture ideas. Finding design patterns in source code helps in maintenance, comprehension, refactoring and design validation during software development. SPQR (System for Pattern Query and Recognition)[24, 25, 26] is a toolset for the automated discovery of design patterns in source code. SPQR uses a logical inference system to reveal large numbers of patterns and their variations from a small number of definitions. A formal denotational semantics is used to encode fundamental OO concepts (which we term Elemental Design Patterns, or EDPs), and a small

number of rules (which we call reliance operators) for combining these concepts into larger patterns. These reliance operators, when combined with the sigma-calculus[1], provide a formal foundation we call the rho-calculus.

SPQR improves on previous approaches for finding design patterns in source code. Other systems have been limited by the difficulty of converting something as abstract as design patterns into concrete expressions without being overly restrictive. A single design pattern when reduced to concrete code can have myriad realizations, all of which have to be recognized as instances of that one pattern. Other systems have had difficulty due to their reliance on static definitions of patterns and variants. SPQR overcomes this problem by using an inference system based on core concepts and semantic relationships. The formal foundation of SPQR defines base patterns and rules for how variation can occur; the inference engine is then free to apply variation rules in an unbounded manner. A finite number of definitions in SPQR can match an unbounded number of implementation variations.

In our next sections we summarize the formalisms and theory behind SPQR, which are published in full detail in [26]. They are included here for completeness, and the reader may wish to skip to Section , referring to the theoretical discussion as necessary.

1.3 Rho Calculus

The rho calculus consists of two major features: reliance operators, and ubiquitous transitivity. We will give a brief overview of the reliance operators, an example of how their formalization can be used to produce basic programming concepts that are easily found in source code, and a short discussion on the necessity of transitivity.

1.3.1 Reliance operators

We have four forms of reliance operator that we will briefly discuss here; formal treatments can be found in [25, 26]. All four operators have a common notation and basis, however. Given two objects in a codebase, o and p , we can use the sigma calculus notation to indicate a *selection* of a method or field (sigma calculus treats them more or less equally) via the *dot operator*, such as $o.m$ or $p.n$. We use this notation as well.

Our four forms arise from the combination of methods and fields as m and n . m may be a method or field, as can n . If both m and n are methods, we have the method invocation operator, $<_{\mu}$. If m is a method, and n is a field, then the $<_{\phi}$, or field use operator, is used. m a field, and n a method gives us the $<_{\sigma}$ or state change operator, and finally, the cohesion operator, $<_{\kappa}$, indicates a reliance of a field m on another field n .

We described in previous work that an annotation to the operator could be used to indicate a concept we term *similarity*. Two dotright selections are similar if they have the same selector notation. In other words, if two methods have the same selection tag (hereafter called the ‘name’) for their source language, then they are ‘similar’. ‘Similar’ in this context means adherence to the Intention Revealing Selector best practices pattern as defined by Kent Beck[2]. In that, Beck states “Name methods after what they accomplish.” Following this pattern means that if we see two methods with the same name, then we can deduce that for most cases, they are intended to perform similar, if not exactly, the same task; this principle holds for fields as well. We can use this relationship to further deduce quite a number of interesting properties of relationships.

We indicate the two types of similarity, object, and selection, by a dot notation that reflects the $o.m$ notation, to clarify what is being indicated to be similar: $\{+, -, \circ\}.\{+, -, \circ\}$. The \circ indicates a placeholder of indeterminate status of similarity. (If both sides of the notation are indeterminate, it can be eliminated altogether.)

The notation is a direct indication of the relationship between the left hand and right hand sides of the reliance operator: $o.m <_{\mu}^{-,+} p.n$ states that objects o and p are known to be distinct objects, and method m of object o calls a similarly named method n of p . We can deduce that m and n perform similar tasks, and so we can further infer that $o.m$ is redirecting a portion of its intended workload to object p as a subtask.

1.3.2 Method Invocation

Expanding on this idea, we can create a grid of the most meaningful method call relationships (Table 1) and populate it with what each relationship means to the programmer *conceptually*. (Each row is a leftdot (scope) similarity, each column is a dotright (method) similarity.)

An object with a method calling itself, obviously, is the classic Recursion. It is indicated by $o.m <_{\mu}^{+,+} p.n$, where $o = p$ and $m = n$. If the method m calls some other method n in the same object, we call this Conglomeration, $o.m <_{\mu}^{+,-} p.n$.

	+	-
+	Recursion	Conglomeration
-	Redirection	Delegation

Table 1. Basic $<_{\mu}$ similarity concepts

We can extend these concepts to external objects as well, where it is explicitly known that $o \neq p$. In such cases, we term invocation of a similar method to be a Redirection ($o.m <_{\mu}^{-,+} p.n$), (indicating a redirection of a similar

portion of the workload) while a dissimilar method is a Delegation ($o.m \prec_{\mu}^{-} p.n$) (delegating out any subtask).

A similar process can be carried out for the other three reliance operator forms, but the above will serve for the purposes of illustration in this paper. A full treatment can be found in our previous publications.

1.3.3 Transitivity

The above reliance operators are of limited use without the last key piece of the rho-calculus: transitivity. It should be obvious that reliances are transitive: if A relies on B, and B relies on C, then A relies on C as well. More formally: Transitivity is the process by which large chains of reliance can be reduced to simple facts regarding the reliance of widely separated objects in the system. The four forms of relop all work in the same manner in these rules. The similarity trait of the reliance operator is not taken into consideration, and in fact can be discarded during the application of these rules; appropriate traits can be re-derived as needed.

This seems at first to be an obvious extension of the reliance operators, and yet it leads to quite powerful inference capabilities that are not to be found in other systems design research, as we shall see.

1.4 Patterns Catalog

We have previously described, informally, our catalog of Elemental Design Patterns[24], and the manner in which they can be used to compose larger, more complex patterns[26], with the goal of expressing the ubiquitous ‘Gang of Four’ patterns[11]. We will discuss here how the reliance operators conveyed in this paper lead to a much larger number of useful concepts that we use to formally define our EDPs, and thereby define an extremely large number of potential patterns in a flexible and transitive manner. We will from now on refer to the various forms and variants of the reliance operators developed in section 1.3 as EDPs, since they form several of the core patterns in that catalog.

1.4.1 Method Invocations as EDPs

We revisit our method reliances from section 1.3.2, where we described four conceptual variants of the \prec_{μ} operator, based on dual axes of similarity. We now add the concept of typing information to this grid, and derive eight more useful concepts for our definition.

We showed in section 1.3.2 that the $\prec_{\mu}^{+, \circ}$ form was operating on the same object, while $\prec_{\mu}^{-, \circ}$ was an interobject coupling. The former construct can be expanded by type, when it is noted that the self-typing of an object can include superclass scoping of method calls.

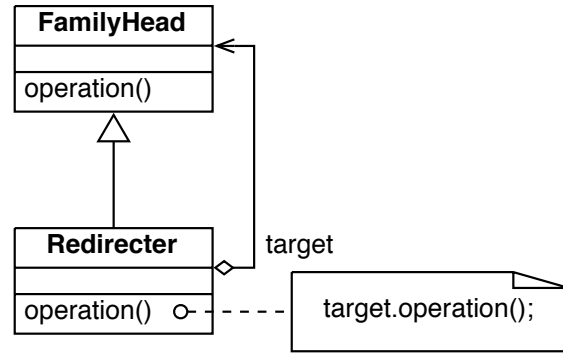


Figure 3. Redirect In Family EDP

We can apply a inheritance typing relation to the Redirect and Delegate EDPs, resulting in another ubiquitous pattern pair: RedirectInFamily and DelegateInFamily. RedirectInFamily is shown in Figure 3, and illustrates that we are imposing a typing relation between two distinct objects, again, such that for object $o : O$ and $p : P$, $O <: P$. DelegateInFamily is a similar construct, with only a variance on the dotright similarity. These two patterns encapsulate the core basis of polymorphism in a family of classes, hence the ‘InFamily’ descriptor, by sending a request to the top of a tree of inherited classes, and allowing polymorphism to select the proper implementation.

Other typing relations have been investigated, such as ‘Other object, Sibling type relation’ and ‘Other object, Self type relation’, leading to a richer suite of EDPs including concepts such as Delegation, RedirectedRecursion (the foundation of the Chain of Responsibility GoF pattern) and several others.

In addition to these method invocation EDPs, we have identified and defined a number of other design and object management issue concepts: CreateObject, AbstractInterface, FulfillMethod, RetrieveShared, and others. Again, these are formally and fully treated in our previous publications.

1.5 Pattern composition

Composition of the EDPs can quickly lead to powerful constructs and abstractions. One such is Singleton, a design pattern difficult to capture using standard static structural techniques, since it uses class-level abstractions, has run-time behaviour that must be described, and can be implemented in many various ways.

We use the technique described in [1] for class-based languages and create a ‘class object’ in the rho calculus that is responsible for creating new objects of a given type, and is a convenient and appropriate place to embed class-level

methods and fields.

From the description given in [11], we know that Singleton: (a) returns a single shared instance of itself; (b) stores that instance privately at a class level; (c) provides a method for retrieving that instance, again at class level. From this, we can almost directly define Singleton in terms of our EDPs and the rho calculus:

$$\begin{array}{c}
 SC \text{ isclassobj for } S \\
 SC.gettor : S \\
 SC.instance : S \\
 \hline
 RetrieveShared(clientmethod.clientsink, \\
 SC.gettor, SC.instance) \\
 \hline
 Singleton(S, SC.gettor, SC.instance)
 \end{array} \quad (1)$$

This definition is simple, direct, and to the point. It is also highly flexible. RetrieveShared is a basic concept that we have defined as a composition of two EDPs: CreateObject and Retrieve. CreateObject and Retrieve are both defined as reliance operator constructs, and the transitivity properties of those provide a high degree of variance of implementation (we term such a variation an *isotope*[25, 26]).

These isotopes are a key element of our approach and allow design patterns to be inferred in a *flexible* manner. We do not require each and every variation of a pattern to be statically encoded, instead the transitivity in the ρ -calculus allows us to simply encode the relationships between elements of the pattern, and an automated theorem prover can infer as many possible situations as the facts of the system provide. In this way a massive search space can be created automatically from a small number of design pattern definitions.

For instance, in section 1.4.1, we introduced the RedirectInFamily EDP. In Figure 4, we illustrate an isotope of that pattern, which at first looks quite different structurally - and yet, by the definition of our reliance operators, we have the same satisfiability of our RedirectInFamily conditions, via transitivity. This variance of implementation without requiring a redefinition in the rho-calculus is the core idea behind an isotope - it allows for a great degree of flexibility in a system's implementation of a design pattern while maintaining the conceptual integrity of that pattern.

From static constructs easily detectable in source code, and three axes of interaction (our four reliance forms, similarity, and type relation) we have derived a suite of simple concepts that can be composed via rules of relationship transivities into a catalog of rich design patterns. Our initial goal is the definition of the Gang of Four catalog in SPQR, but we see this as merely a beginning. The newly emerging system and architecture design patterns [4, 22] are of a scale even larger than those in the GoF, but we are confident that the approach and foundation we have created here will allow for the compositions needed to capture these as well.

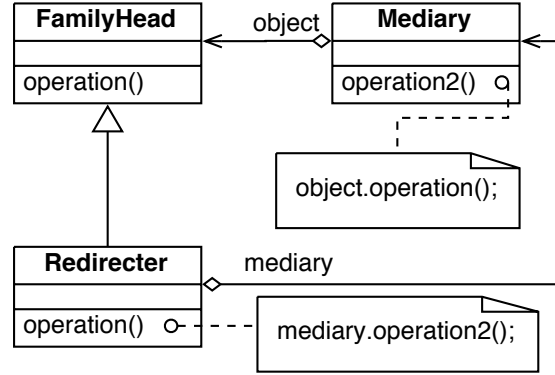


Figure 4. Redirect In Family EDP Isotope

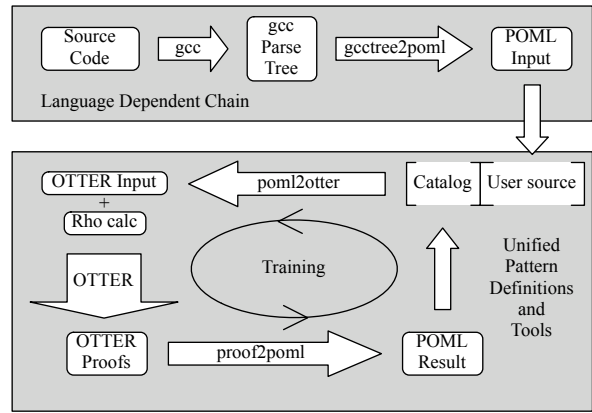


Figure 5. SPQR Toolchain

2 Experiment Setup

2.1 The SPQR Tool chain

We describe here our chain of tools from the viewpoint of a practitioner using them. This toolset, the System for Pattern Query and Recognition, comprises several components, shown in Figure 5. From the engineer's point of view, SPQR is a single tool that performs the analysis from source code and produces a final report. A simple script provides the workflow, by chaining several modular component tools, centered around tasks of *source code feature detection*, *feature-rule description*, *rule inference*, and *query reporting*.

In SPQR, source code is first analyzed for particular syntactic constructs that correspond to the ρ -calculus concepts we are interested in. It turns out that the ubiquitous `gcc` has the ability to emit an abstract syntax tree suitable for such analysis. Our first tool, `gcctree2poml`, reads this tree file and produces an XML description of the object struc-

```

<object>
  <name>r</name>
  <type>Redirecter</type>
  <method>
    <name>operation</name>
    <calls>
      <objectname>fh</objectname>
      <methodname>operation</methodname>
    </calls>
  </method>
</object>
<object>
  <name>fh</name>
  <type>FamilyHead</type>
</object>
<class>
  <name>Redirecter</name>
  <parent>FamilyHead</parent>
</class>

```

Figure 6. RedirectInFamily as POML input

```

all Redirecter FamilyHead r fh operation (
  (Redirecter inh FamilyHead) &
  (r : Redirecter) &
  (fh : FamilyHead) &
  ((r dot operation) mu (fh dot operation)) &
  (r phi fh) ->

  (RedirectInFamily(Redirecter, FamilyHead,
    operation))
).

```

Figure 7. RedirectInFamily as OTTER input

ture features. Figure 6 is an example of the RedirectInFamily pattern definition derived from example code. We chose an intermediary step so that various back ends could be used to input source semantics to SPQR. A second tool, *poml2otter* then reads this Pattern/Object ML (POML) file and produces a feature-rule input file to the automated theorem prover, in the current package we are using Argonne National Laboratory’s OTTER. OTTER finds instances of design patterns by inference based on the rules outlined in this paper. Figure 7 shows the input to OTTER for the RedirectInFamily EDP. Finally, *proof2pattern* analyzes the OTTER proof output and produces a POML pattern description report that can be used for further analysis, such as the production of UML diagrams.

Each stage of SPQR is independent, and was designed to allow other languages, compilers, workflows, inference engines, and report compilation systems to be added. Additionally, as new design patterns are described by the community, perhaps local to a specific institution or workgroup, they can be added to the catalog used for query.

3 Experimental Validation

We are currently running experiments designed to test the limits of SPQR in a real world environment, with code that is legacy derived, refactored, and under active maintenance. It is anticipated that these tests will further prove the practical nature of the SPQR methodology and lead to refinements for scaling to yet larger test cases.

3.1 Test cases

The *TrackerLib* is a research quality framework for video stream real-time object tracking that has been used in several of our published research systems[28, 27, 29]. Several years ago it was initiated from a refactoring of established research code to provide an object-oriented, and more importantly, conceptually clean architecture for future maintenance. To this end, design patterns were used as the main guiding force. This is a reasonable test case for the SPQR formalisms and tools. It is in a target language (C++), it compiles cleanly with a supported compiler (gcc3.3), and it is of sufficient but not unmanageable size (approximately 8kLOC). Design patterns were used in the re-architecting, but have not been validated across the subsequent maintenance and refactorings, archived in a Concurrent Versioning System (CVS) repository. It is of sufficient size and complexity to produce a serious test of the scalability issues under investigation, and yet small enough that hand-checking can be performed on subsets if necessary. One of those subsets, *NotificationCenter*, contains an implementation of a Singleton pattern. We present the analysis of NotificationCenter here in anticipation of leading into a more thorough analysis of TrackerLib including following the evolution of the code through time and observing the shifting pattern instances.

KillerWidget is an example we have used from the early days of SPQR, and it is modeled after a problem encountered by the first author while working at a flight simulator and graphics company[26]. Unfortunately, the original code is not available, but the salient details necessitating a deductive search process are retained. A Decorator pattern exists in the structure, but it does not follow the ‘standard’ form provided in [11]. Instead, there is a level of indirection that static pattern structure detectors would be hard pressed to work through to find the pattern instance. Indeed, it took three engineers familiar with the original code many weeks of analysis to uncover the basic pattern and deduce the behaviour. We expected that SPQR’s use of isotopes to allow flexibility in the pattern instance would be able to find this as easily as a direct example.

Between NotificationCenter and KillerWidget, we have two patterns from the Gang of Four literature (Singleton and Decorator) that provide coverage of the majority of the EDP

catalog and illustrate both structural and behavioural patterns. The test case implementations illustrate both direct and hidden pattern instances.

A third test case is the C++ `std` namespace. A single C++ file containing includes for the entirety of the namespace was compiled and analyzed. While this certainly missed paths of code that would be generated from templates when the namespace elements were actually used, it provides at least a baseline for reasonable expectations.

3.2 Methodology and results

We follow our earlier experiments with SPQR using much the same methodology. In each test case, an existing build system was augmented with one change to the gcc flags: the addition of the `--dump-translation-unit` and `--dump-classes` diagnostic flags. These produce raw gcc dump files, `*.tu` and `*.class` forms respectively. The `gcc2poml` tool was then used to convert these to the Pattern Object Markup Language (POML). In the `KillerWidget` and `NotificationCenter` cases, the `std` namespace code was filtered out, leaving only the code of interest. (Obviously, during the analysis of `std`, this did not occur.) An XSLT transform was then used to convert these descriptions to input to the OTTER automated theorem prover[13]. POML has proven more than adequate to extract the necessary OTTER rules, and simultaneously describe the various patterns to search for.

Table 2 shows some performance and size metrics for the test cases. Code size is measured as strictly the code that was fed directly to gcc and later tools - obviously there is a lot of C++ library code that is being pulled in, particularly in the case of `std`. We omit the `*.class` files since they are in general an order of magnitude (or two) smaller than the corresponding `*.tu` files. POML file size includes debugging information used to map the results back to the original `*.tu` file. Removing these produces on average an 18% file size reduction.

Timings were gathered through timing mechanisms internal to the tools, and averaged over three runs. The test hardware was an Apple 1.25GHZ G4 PowerBook with 512MB of RAM running MacOS X 10.3.9. Timing information for the `spqrsearch` phase is dependent on the search space being traversed. SPQR can be used in a validation manner, or top-down approach, looking for a specific pattern and allowing SPQR to determine the appropriate hierarchy of dependencies and then efficiently search for all, but only those dependencies. Alternately, SPQR can be used in a discovery mode, or bottom-up approach, looking for any and all EDPs that exist, then moving up to the Intermediate patterns, then finally attempting to find whatever Gang of Four patterns might exist in the code. Obviously, this latter method is much more time-consuming. The tim-

	Killer-Widget	Notification-Center	std
<i>File Sizes</i>			
C++			
kB	0.8	28.1	0.5
LOC	48	1083	31
gcc .tu			
kB	202.5	49516.5	14484
# of Nodes	1768	217805	137637
POML			
kB	47	491.4	631.9
classes	8	58	1542
objects	7	65	512
methods	46	904	8404
fields	4	317	2015
OTTER			
rules	171	2227	30322
<i>Timings (sec)</i>			
gcc	0.233	6.9	7.9
gcc2poml	6.51	244.8	262.9
spqrsearch	9.5	29.5	1929.9
Total			

Table 2. Test Case Metrics

ings for `KillerWidget` and `NotificationCenter` are for using SPQR to find the specific pattern assumed to exist in the code, validating its existence. The `std` timing is for the exploratory approach, looking for any and all EDPs. Since no Intermediate or higher patterns could be made from the EDPs found, those runs were not performed.

Analysis proves the existence of a large number of EDPs, as would be expected from their simple nature, yet the number of false positives for the more complex patterns is not reaching the levels that were once expected. Instead, the current code size is such that extraneous inferences that, while correct, are not of particular usefulness, are minimal. It is possible that larger systems will produce logically valid inferences of patterns that are simply accidental, and not relevant to the architecture. In such cases, limits can be imposed on the depth of inference chains traversed by OTTER, a simple change to the OTTER input ruleset.

4 Lessons Learned

SPQR provided affirmation of the existence of the expected patterns from EDPs through particular Gang of Four patterns. In addition, the experiment produced a few surprises.

4.1 Expected Validation

As can be seen in Table 3, SPQR found the expected patterns: KillerWidget contains a non-direct Decorator pattern, requiring inference to deduce the existence through a number of intermediate classes, and NotificationCenter uses a Singleton at its core to ensure single-point access to a global event registration system. The large number of Delegate EDPs in each example is due to how gcc sets up object allocation and memory management through a series of nested calls between the constructors of a class (represented in POML and OTTER as a ClassObject, as indicated by rho-calculus), and potentially several functions that have been represented as methods of the `__GLOBAL__ ClassObject`. Refinement of the POML production tools can remove many of these Delegate hits that while correct, are not of particular interest.

	Killer-Widget	Notification-Center	std
<i>EDPs</i>			
CreateObject	7	15	1009
Inheritance	4	3	208
AbstractInterface	2	-	38
Retrieve	-	6	201
Conglomeration	20	-	204
Delegate	137	302	844
Delegated-Conglomeration	20	52	302
DelegateInFamily	-	-	1127
DelegateIn-LimitedFamily	-	-	25669
Recursion	-	4	64
Redirect	20	14	259
Redirected-Recursion	-	4	64
RedirectInFamily	3	-	-
RedirectIn-LimitedFamily	-	-	-
ExtendMethod	1	-	-
RevertMethod	-	-	-
<i>Intermediate</i>			
FulfillMethod	8	-	76
Objectifier	12	-	16
ObjectRecursion	17	-	-
RetrieveShared	-	6	90
<i>Gang of Four</i>			
Decorator	2	-	-
Singleton	-	1	-

Table 3. SPQR Results

Note that the two codebases were created with only the

highest level patterns (Decorator, Singleton, respectively) in mind, yet a large number of smaller patterns were detected. This is the result we expected, given the "building block" or "isotopic" nature of EDPs.

4.2 Unexpected Results

Of greater importance, perhaps, is an item that appeared while performing the above experiment. While analyzing KillerWidget, the expected Decorator pattern was not found. Appropriate EDPs were being reported, as were the Intermediate level patterns, but the final Decorator was not. This prompted a careful reassessment of the formalisms and relationships of the patterns, but nothing seemed out of place. After much consideration and work, the source code being analyzed was inspected, and a subtle bug was found, where a typo had been calling the wrong method. SPQR was correct, the code was not. This gives us much hope in using SPQR for determining the adherence of source code to an architectural specification.

4.3 Other Lessons

Scalability of formal methods is always a concern, yet we find that for SPQR the issue is nearly non-existent. While the runtime for SPQR is approximately an order of magnitude greater than compilation time (using gcc 3.3) on the same C++ code, the increase with respect to rules added to OTTER does not exhibit the exponential growth that was feared. Instead, we see a nearly linear growth of OTTER input rules with code size, after taking into account redundant code definitions among disparate translation units. OTTER in turn has shown remarkable performance as the number of input rules as increased, an informal analysis of which leads us to conclude that the analysis time will be slightly but not significantly supra-linear with the number of input rules.

The use of POML as a common data format for all incoming code and outgoing results allows us to quickly and easily write post-analysis tools using existing technologies such as XSLT or the various XML parsing and manipulation libraries. These can be used to produce various code metrics in a language-independent manner that previously required language-specific parsers and analyzers.

The ease with which these metrics can be created and gathered signifies that we can start to provide meaningful measures of design pattern coverage of code, using the paths of reliance within the code as a guideline. Given the literature linking maintainability of a system with the comprehensibility of the architecture, and given that design patterns provide a common language for comprehension, these pattern coverage metrics should be indicators with metrics of comprehensibility (within the common language of design patterns), and may provide valuable clues as to the level

of maintainability of a codebase.

4.4 Extension to Architecture

SPQR was initially designed to find instances of design patterns, but we have found that we have the capability to detect a broader range of code relationships and constructs than we first expected. In doing so, we have created a solid basis for the construction of not only definitions of established design patterns, but of any code construct and set of relationships that is desired, and the ability to search for same in source code.

Using the same technique we have developed for the current catalog of design patterns in SPQR, we can further compose existing design patterns into higher level abstractions suitable for the architectural patterns described in the literature such as in [4] and [22]. This hierarchy of abstractions mirrors closely the software design-level model pyramid proposed in [16]: a systematic organization of conceptual pieces starting at the object and method level, interconnected with formalized reliance relationships. As SPQR broadens in scope, higher levels of abstraction should be attainable through the same fundamental methodology we have demonstrated to date:

- 1) Develop a collection of Elemental Architectural Patterns (EAPs), formal expression of fundamental architectural principles from which larger architectural patterns are formed (much as we showed GoF design patterns to be composed of EDPs).

- 2) Develop formal definitions of architectural patterns in terms of the rho-calculus and structural/architectural properties both from EAPs, and from the design pattern level of the factbase.

- 3) Develop formal definitions of any needed new relationships that must exist among design patterns in order for architectural level patterns to exist.

- 4) Develop new variance rules, if any, that express ways in which architectural patterns may deviate from the "ideal" definitions.

- 5) Apply SPQR as for design patterns, but working with architectural level catalogs; the structural facts it will work with are all the base level EDPs (the connection to actual code), and all the inferences made while reasoning about higher-level design patterns. Architectural evaluation and comparisons

Step 5 above is where the SPQR system draws conclusions about architecture-level abstractions based on what it finds in terms of design patterns in the code. However, the information that is inferred and saved about design patterns is now independent of the specific code implementation that was the starting point. Design pattern facts are all code-independent. This means that two different code implementations might both express the same set of design patterns,

arranged in the same relationships to each other, and therefore be judged to be realizations of the same architecture.

We are working on other metrics that can quantify aspects of the architectural factbase that SPQR gathers, thereby giving us some basis for code-independent comparison of different architectures. The ultimate questions to be answered are:

- "Here is codebase A, and there is codebase B... "
- "Which has the better design?"
- "Does architecture of A (or B) follow our practices and standards?"
- "Compare A to B... where are the significant differences?"

5 Conclusion

We have summarized in this paper the theory and methodology of SPQR presented in our previous publications in greater detail, and reported experimental data produced by SPQR when analyzing production C++ code. Further, we have outlined a well-formed path for the extension of these formalisms and tools to the abstractions and relationships of architecture-level patterns and analysis.

References

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., 1996.
- [2] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.
- [3] J. Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 1(2):18–52, may 1998.
- [4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented System Architecture: A System of Patterns*, volume 1 of *Wiley Series in Software Design Patterns*. John Wiley & Sons, 1996.
- [5] J. Coplien. C++ idioms. In *Proceedings of the Third European Conference on Pattern Languages of Programming and Computing*, jul 1998.
- [6] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactoring via change metrics. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 166–177. ACM Press, nov 2000.
- [7] A. H. Eden. *Precise Specification of Design Patterns and Tool Support in their Application*. PhD thesis, Tel Aviv University, Tel Aviv, Israel, 2000.

- [8] A. Egyed. Automated abstraction of class diagrams. *ACM Transactions on Software Engineering and Methodology*, 11(4):449–491, oct 2002.
- [9] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In M. Askit and S. Matsuoka, editors, *Proc. of the 11th European Conf. on Object Oriented Programming - ECOOP'97*. Springer-Verlag, Berlin, 1997.
- [10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.
- [12] B. B. Kristensen. Complex associations: abstractions in object-oriented modeling. In *Proc of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 272–286. ACM Press, 1994.
- [13] W. McCune. Otter 2.0 (theorem prover). In M. E. Stickel, editor, *Proc. of the 10th Intl Conf. on Automated Deduction*, pages 663–664, jul 1990.
- [14] S. Meyers. *Effective C++*. Addison-Wesley, 1992.
- [15] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proc. of the eleventh annual conference on Object-oriented programming systems, languages, and applications*, pages 235–250. ACM Press, 1996.
- [16] T. J. Mowbray and R. C. Malveau. *CORBA Design Patterns*. John Wiley & Sons, 1997.
- [17] M. Ó Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach*. Ph.D. dissertation, University of Dublin, Trinity College, 2001.
- [18] W. F. Opdyke and R. E. Johnson. Creating abstract superclasses by refactoring. In *Proc. of the Conf. on 1993 ACM Computer Science*, page 66, 1993. Feb 16-18, 1993.
- [19] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.
- [20] S. P. Reiss. Working with patterns and code. In *Proc. of the 33rd Hawaii Intl Conf on System Sciences*, jan 2000.
- [21] D. Riehle. Composite design patterns. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 218–228. ACM Press, 1997.
- [22] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented System Architecture: Patterns for Concurrent and Networked Objects*, volume 2 of *Wiley Series in Software Design Patterns*. John Wiley & Sons, 2000.
- [23] F. Shull, W. L. Melo, and V. R. Basili. An inductive method for discovering design patterns from object-oriented software systems. Technical Report CS-TR-3597, University of Maryland, 1996.
- [24] J. M. Smith. An Elemental Design Pattern catalog. Technical Report TR-02-040, Univ. of North Carolina, 2002.
- [25] J. M. Smith and D. Stotts. Elemental Design Patterns: A formal semantics for composition of OO software architecture. In *Proc. of 27th Annual IEEE/NASA Software Engineering Workshop*, pages 183–190, dec 2002.
- [26] J. M. Smith and D. Stotts. SPQR: Flexible automated design pattern extraction from source code. In *18th IEEE Intl Conf on Automated Software Engineering*, pages 215–224, Oct 2003.
- [27] J. M. Smith, D. Stotts, and S.-U. Kum. An orthogonal taxonomy for hyperlink anchor generation in video streams using ovaltine. In *Proc of ACM Hypertext 2000*, pages 11–18, sep 2000.
- [28] D. Stotts and J. M. Smith. Semi-automated hyperlink markup for archived video. In *Proc of ACM Hypertext 2002*, pages 105–106. ACM, jun 2002.
- [29] D. Stotts, J. M. Smith, and D. Jen. The vis-a-vid transparent video facetop. In *Proc of UIST 2003*, pages 57–58 and demo, nov 2003.
- [30] B. Woolf. The abstract class pattern. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 1998.
- [31] B. Woolf. The object recursion pattern. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 1998.
- [32] W. Zimmer. Relationships between design patterns. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 345–364. Addison-Wesley, 1995.