Technical Report TR05-013

Department of Computer Science
Univ. of North Carolina at Chapel Hill

Case Studies in Automated Design Pattern Detection in C++ Code using SPQR

Jason McC. Smith and David Stotts

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

stotts@cs.unc.edu

June 6, 2005

Case Studies in Automated Design Pattern Detection in C++ Code using SPQR

Jason McC. Smith
University of North Carolina at Chapel Hill
Sitterson Hall CB #3175
Chapel Hill, NC 27599-3175
smithja@cs.unc.edu

David Stotts
University of North Carolina at Chapel Hill
Sitterson Hall CB #3175
Chapel Hill, NC 27599-3175
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 some of the more intriguing analyses that were made possible.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*design languages, object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*patterns*; F.4.1 [**Mathematical Logic**]: [lambda calculus and related systems]; D.2.11 [**Software Engineering**]: Software Architecture—*patterns*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, Enhancement—*restructuring, reverse engineering, and reengineering*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; D.3.1 [**Programming Languages**]: Formal Definition and Theory

General Terms

design, languages, measurement, theory

Keywords

design patterns, elemental design patterns, sigma calculus, rho calculus, pattern decomposition, pattern identification, refactoring, education

1. OVERVIEW

Design patterns have been one of the more successful software innovations of the past decade, helping engineers produce high-quality software by describing a common language. They are, however, a one-way technology for all practical purposes. Finding implementations of design patterns in source code has been difficult, but would greatly benefit the comprehension of large systems, training of new engineers, and the validation of design. Current and previous attempts at detecting design patterns have been based on the detection of source code constructs. Design patterns are not properly thought of as constructs however, but as *concepts*. This mismatch has been at the root of the lack of practical pattern detection.

The *System for Pattern Query and Recognition* (SPQR) is a research project that takes a different approach: treating design patterns as the embodiment and encapsulation of concepts, not constructs. This enables the treatment of object-oriented systems as a graph of related concepts of programming. Defining the mapping between the static constructs in source code and the flexible abstract concepts such that they can be reliably detected is a task with several distinct stages.

Analyzing the base literature of design patterns, namely the seminal Design Patterns text[13], reveals many shared concepts between patterns that fulfill very different purposes. By deconstructing the established patterns into subpatterns, we can show well formed relationships between more fundamental concepts of programming. In taking this to its logical conclusion, we have established a set of Elemental Design Patterns (EDPs). These are *binary relationships* between elements found directly from source code: objects, methods, fields, and types. As such, the EDPs are immediately detectable from source code, and can be considered the DNA of programming, but from the viewpoint of concepts, not constructs. They form the core of what we call Concept Oriented Programming. Design patterns are an aspect of COP, and provide a wonderful ready-made validation suite.

Once these fundamental pieces of design patterns have been detected from source code, we build them back into the higher level abstractions for which we are searching. We have extended the sigma-calculus of Abadi and Cardelli[1] to rho-calculus[28], which defines how these conceptual items can interrelate. Rho-calculus also defines these interrelationships as a series of reliances, which we have formalized

as *reliance operators*. Reliance operators describe how the EDPs can be integrated back into larger design patterns in well-formed and precise ways. Transitivity operations allow these integrations to be extremely flexible with respect to the original source code. The same concept can be written in a myriad of ways in source code, which we term *isotopes*[25, 28], but the rho-calculus and EDPs capture them all through formal means.

Formalized EDPs and rho-calculus provide a unique opportunity: using formal theorem proving techniques to deduce relationships of note within a system. These inferences can be performed using an automated theorem prover such as OTTER from Argonne National Laboratories. Using our detected EDPs from a source code base and the rho-calculus as inputs, we can query OTTER about the existence of higher level patterns quickly and methodically. The results are considered potential patterns due to the currently highly conservative nature of SPQR. It is much quicker, however, for an engineer to double-check the existence of a pattern whose location is known, than to find it in the first place.

The results from SPQR can then be used to provide feedback to the architects of a system, establishing the presence (or absence) of patterns they intended to be implemented. It can also, perhaps more interestingly, find unintentional patterns, giving designers an opportunity to make hidden functionality more explicit in the design. To take this further, SPQR can be trained to detect any relationship of concepts that one wishes to find, leading to the possibility of finding pattern fragments. These last two features can be used as the inputs for a refactoring plan, and SPQR can then establish the validity of the final product.

The lingua franca of SPQR is the Pattern Object Markup Language (POML), an XML schema designed specifically to map the salient features of sigma- and rho-calculus into an easily parsed format. Currently, POML is used as the input and output format for SPQR, with a sequence of XSL transforms providing the translations between the necessary forms, including human-ready charts and reports. Other XSLs have been created to quickly produce code metrics on any code base that has been reduced to a POML form. Because of this, source code written in any language that can be mapped to sigma-calculus can be analyzed in a simple and methodical manner. This allows comparison of diverse code bases in differing languages to be performed on an equal footing.

SPQR was designed to be a practical toolset for engineers while providing a robust platform for future research. It has the necessary ease of use and flexibility that real world situations demand, but retains a rich formal basis. The choice of using an automated theorem prover has proven to be important, as it allows for deductive reasoning to be performed efficiently on large systems.

For ease of reading, we will defer a discussion of related research in this area until the end of the paper.

2. SPQR: FORMALISMS

We will not provide a full review of the formal foundations of SPQR here. We instead refer the reader to our previous publications for a more formal treatise[24, 25, 26, 27, 28, 29]. A brief summary is, however, in order.

Two primary elements comprise SPQR's formalisms: a catalog of programming concepts derived from analysis of the design patterns literature, the Elemental Design Patterns (EDPs), and a formal semantics for combining and interrelating those concepts, called the rho-calculus, or ρcalculus.

EDPs are a series of relationships between exactly two entities of object-oriented programming: objects, methods, fields, and types[1]. They indicate the core concepts of how these entities, primarily objects, interact with each other. Method invocation, typing, and field accesses are examples of this interaction. These are extremely simple pieces of programming that practitioners use regularly, usually without much consideration. We have cataloged some of these relationships as EDPs, and have provided extensive descriptions using the format of the design patterns literature. In this manner, we have brought these ubiquitous elements of programming out of the realm of 'unselfconsciousness'[2] and into a common language and terminology.

By creating a well-formed catalog of such concepts which concentrate on binary relationships, finding these EDPs directly and quickly in source code becomes possible. We seek to provide a formal basis for the description of EDPs and to make them more useful in the aggregate.

Rho-calculus, a synthesis of Abadi and Cardelli's sigma-calculus[1], and a small suite of semantics for describing relationships between entities is that formal basis. We call these semantics *reliances* and use them to describe relationships such as dependence, coupling and cohesion. Sigma-calculus allows us to unify the various language features of object-oriented languages, and rho-calculus allows us to treat a multitude of traditional dependency analyses in a similarly unified manner.

Given that under sigma-calculus the only entities that can exist are objects, methods, fields, and types, and that fields are merely scoped objects, we can produce five reliance operators that capture the essence of reliances within sigma-calculus. These are: inheritance (typing relation), mu (one method invokes another method), phi (method uses a field), kappa (a field relies on another field for its value), and sigma (a field relies on another method for its value). These are indicated by the notations: $<:$, $<_\mu$, $<_\phi$, $<_\kappa$ and $<_\sigma$ respectively[28].

Following the model set forth by Abadi and Cardelli, these five reliance operators have their own forms of transitivity and interoperation. These forms are embodied in the *rho fragments*, which, when combined with the fragments of sigma-calculus, form the complete rho-calculus.

These transitivities and interrelations allow us to recombine EDPs into higher level abstractions, such as the Gang of Four patterns, in a precise and meaningful way. By using a dictionary of concepts to describe interactions, we can leverage the transitivities to map to a large number of implementations of these concepts in source code, and search over a large inferable space of relationships to find particu-

lar combinations. That dictionary is our EDP catalog, the conceptual glue the rho-calculus, and the inference engine the automated theorem prover OTTER .

2.1 An Example EDP

As a short example, consider one of the most basic uses of the mu-form reliance operator: $a.m <_{\mu} b.m$. Method m of object a relies on a call to method m of object b. The equality of method names is not accidental, but deliberate. We term this relationship the Redirect EDP. It is directly detectable in source code as a method call from $a.m$ to $b.m$, and should be obviously transitive. Though not entirely useful or insightful on its own, we can apply a inheritance typing relation to Redirect, resulting in a slightly more interesting EDP: RedirectInFamily. RedirectInFamily, shown in Figure 1, illustrates that we are imposing a typing relation between two distinct objects, such that for object $a : A$ and $b : B$, A is a subclass of B, or $A <: B$. This pattern encapsulates 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.

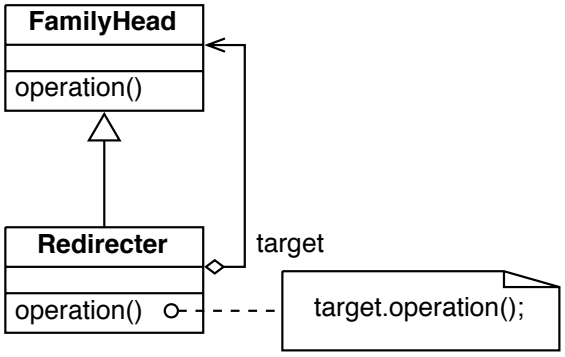

Figure 1: Redirect In Family EDP

This EDP is a component of a number of more useful patterns, such as Decorator[13], but it would be of little utility if we were required to implement it precisely as shown every time. Because we have described RedirectInFamily as a series of reliances, we can also find an instance of it in Figure 2, in what we term an *isotopic form*. In this form, conceptual integrity is retained, but flexibility is allowed and accounted for.

3. SPQR: IMPLEMENTATION

We describe here our chain of tools from the viewpoint of a practitioner using them. SPQR comprises several components, as shown in Figure 3. From the engineer's point of view, SPQR is a single tool that performs the analysis of source code and produces a final report. A simple script provides the workflow by chaining several modular component tools, which are centered around the 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

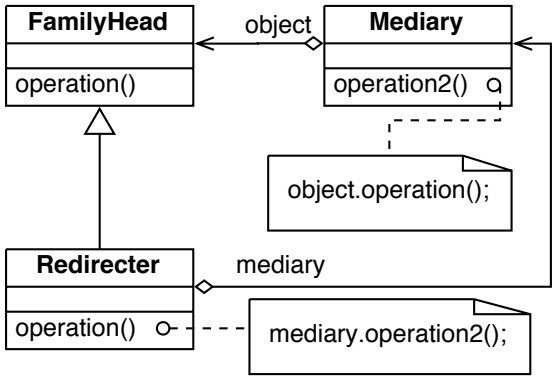

Figure 2: Redirect In Family EDP Isotope

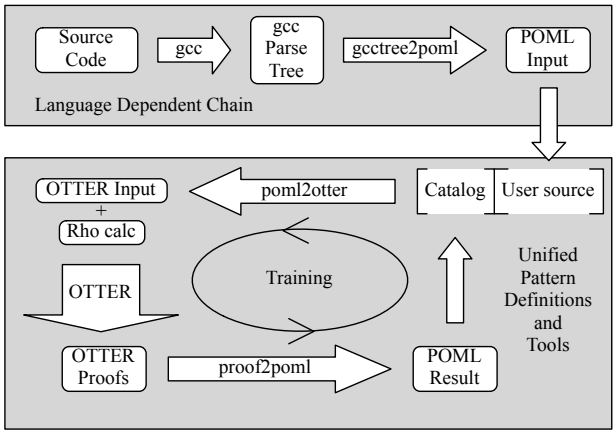

Figure 3: SPQR Toolchain

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 structure features. Figure 4 is an example of how RedirectInFamily might look as rendered from source code. We chose an intermediate step so that various back ends could be used to input source semantics to SPQR. A second tool, *poml2otter* then reads this Pattern/Object Markup Language (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 5 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.

```
<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 4: 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 5: RedirectInFamily as Otter input

The current `gcctree2poml` targets C++ specifically, creating a mapping from the language elements to rho-calculus, and then representing this as POML. C++ has proven to be an interesting challenge, but has ultimately shown to be representable in rho-calculus rather elegantly.

4. 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.

4.1 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[31, 30, 32]. 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[28]. 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 [13]. 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.

4.2 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[15]. POML has proven more than adequate to extract the necessary OTTER rules, and simultaneously describe the various patterns to search for.

Table 1 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.

	Killer-Widget	Notification-Center	std
File Sizes			
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
Timings (sec)			
gcc	0.233	6.9	7.9
gcc2poml	6.51	244.8	262.9
spqrsearch	9.5	29.5	1929.9
Total			

Table 1: Test Case Metrics

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 timings 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, then moving up to the Intermediate patterns, and so on.

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.

5. 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.

5.1 Expected Validation

As can be seen in Table 2, 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.

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.

5.2 Insights From Analysis

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

The `std` namespace provided a couple of interesting results as well. Notice the tremendous number of DelegateInLimitedFamily EDP hits. This was entirely unexpected, and warranted a closer inspection. We performed a lexicographical sorting on the data in order to detect patterns through manual reading. It quickly became obvious that almost every instance occured in the `std::locale`, `std::exception` and `std::stream` subsystems, with six common methods. This began to make sense, once we looked at the implementations of these subsystems. Each has a rich forest of classes that work in concert, and inherit from one or two common base classes. It would be expected to find a significant number of DelegateInLimitedFamily EDPs in such a case. DelegateInLimitedFamily, as the name suggests, is a Delegate EDP that performs its method invocation on an object not of a parent class, as in RedirectInFamily, but rather in a sibling class, hence the 'Limited' nomenclature. This approach is common, and can indicate a tightly coupled module, or can illustrate a potential hot cluster of overactive dependencies. A quick analysis of how many methods in the active classes

which are part of this weave, and the number of interwoven classes and their relationships, should be able to distinguish between the two in all but a handful of border cases.

A second grouping that emerged was repeated DelegateInLimitedFamily occurrences with the same sibling classes, but a different base class. A moment's reflection provided the reason. Take for example the hierarchy in Figure 6, drawn from the `std` example. DelegateInLimitedFamily has criteria that include that the Delegator and Delegatand be sibling classes, or inherited from the same base class. In this case the Delegator is `domain_error`, and the Delegatand is `length_error`. Both inherit off of the `logic_error` class, and this rule is satisfied. Note however that both of these classes *also* inherit from the `exception` class, leading to a second instance of DelegateInLimitedFamily, all else being equal.

	Killer-Widget	Notification-Center	std
EDPs			
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	-	-	24192
Recursion	-	4	64
Redirect	20	14	259
Redirected-Recursion	-	4	64
RedirectInFamily	3	-	-
RedirectIn-LimitedFamily	-	-	-
ExtendMethod	1	-	-
RevertMethod	-	-	-
Intermediate			
FulfillMethod	8	-	76
Objectifier	12	-	16
ObjectRecursion	17	-	-
RetrieveShared	-	6	90
Gang of Four			
Decorator	2	-	-
Singleton	-	1	-

Table 2: SPQR Results

This brings us to an important point: that while many EDP hits will be logically correct, they may not be of particular interest to an engineer attempting to comprehend the system. The `logic_error` based DelegateInLimitedFamily pattern is probably the only one that is necessary to gain insights on the code at hand, yet the `exception` based instance will also be reported. The question then becomes how best to manage a level of detail for the engineers, leading them to conceptual cues without swamping them with correct but essentially duplicate results? We must also balance this desire to cull extraneous results with the need to include such logical inferences in the proofs of much larger searches. This is, after all, the entire reason for the production of SPQR.

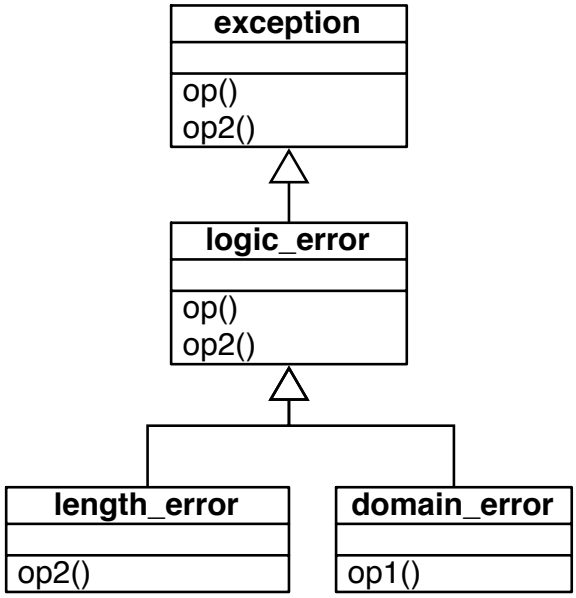

Figure 6: Example DelegateInLimitedFamily Hierarchy

We assume that simpler is better, in that the most refined and basic form of a reported pattern will be the simplest to find and understand in the code. To this end, we use the *ignore* utility of Otter to produce a running weight for inferred rules. The $IGNORE element allows us to tag rules with generated values, in this case the total proof lengths leading to the current inferred rule, and Otter will dutifully ignore that data when making inferences. We can use this data to cull out reported patterns at a later stage, however, setting a threshold as appropriate. Post-analysis tools could report, for instance, only the smallest weighted instance for a particular combination of satisfiers. In the above case, it would be possible to report the `logic_error` based pattern, and cull the `exception` based one. Note that this in no way interferes with Otter deducing much richer and deeper patterns that may require the `exception` based instance as a satisfier, or using that instance to infer yet deeper rules that lead to more complex and insightful design pattern instances.

5.3 Automated Training Support

One concern when dealing with formal methods such as those behind SPQR is that it will require practitioners to become ad hoc theoreticians or formalists. That is not the case with SPQR, as it was designed from the beginning to be a practical tool for the average software engineer.

Trainability of SPQR has also become of practical importance in our research, as we continue to extend the capabilities of the system and the coverage of its search catalog. Again, POML is the key to the practicality of SPQR, and a training cycle has become evident in our work with it.

Assume that a group or developer has defined a new pattern that they wish to have SPQR search for. It would be cumbersome to require users to have a deep knowledge of rho-calculus and the formalisms at the core of SPQR. Instead, SPQR can be used to train itself.

First, a canonical form pattern is written in the source language of choice. This source code should be as basic as possible, capturing only the salient and necessary features needed to define the pattern. This may prove more difficult than at first it seems - distilling a pattern down to the minimal canonical form is more difficult than one would think, yet is the very essence of the art of pattern definition, no matter the form. SPQR's exploratory mode (as seen with the `std` namespace) can assist here, by producing a comprehensive analysis of the exemplar code, and using the abovementioned weighting system to provide a minimal necessary description. Inferred EDPs, sub-patterns, and patterns will be reported, but in the most direct form possible. This gives feedback to the engineer that they are either on the right path, or need to somehow alter their code to fit their expectations.

The minimized code is then run through SPQR in exploratory mode a final time, and the POML results used to form the basis of a new search definition. A new POML element describing the resulting pattern, the roles it defines, and the code features that fulfill those roles, is added to the results file. This process is easy to automate, being a simple text file insertion. This final POML file is then added to the SPQR catalog, and becomes a searchable pattern. This is the very process we use to bootstrap the larger, more abstract design patterns from various sources, and is quite quick. The manual use of rho-calculus, or indeed even POML, is minimized and only used for fine tuning in rare cases. We feel this is a very automatable process, and as the catalog of lower level design patterns increases, it will only become more so.

5.4 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.

6. PREVIOUS RELATED RESEARCH

The design semantics described in this paper are based in denotational object semantics [1], automated reasoning and automated deduction [15], OO design patterns [13], 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 [7, 14, 22, 35]. 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.

Structural analyses. An analysis of the 'Gang of Four' (GoF) patterns [13] reveals many shared structural and behavioural elements, such as the similarities between Composite and Visitor [13]. 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 [5, 7, 22, 33, 34, 35]. These examples have, however, been at a scale just below that of the original design patterns literature, and of a much coarser granularity than the EDPs. It is because of their still too-large size that they require further decomposition to be suitably formalizable.

Objectifier: The Objectifier pattern [35] 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

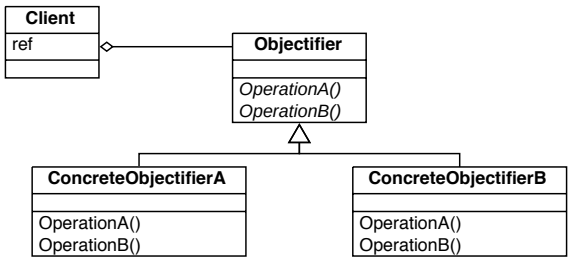

Figure 7: Objectifier

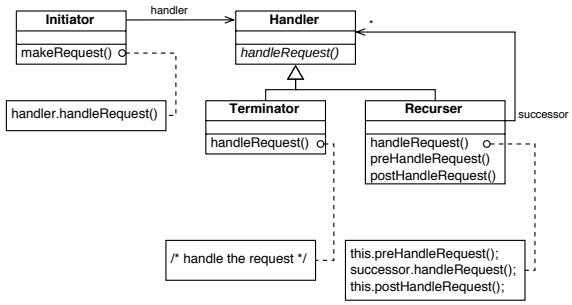

Figure 8: Object Recursion

Recursion [34]. The class diagram in Figure 8 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.

Refactoring approaches. Related to structural analyses is the urge to refactor based on small discrete structural transformations. Attempts to formalize refactoring [12] exist, and have met with fairly good success to date [8, 17, 19]. 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 [11], Eden's work on LePuS [9], and Ó Cinnéide's work in transformation and refactoring of patterns in code [18] through the application of minipatterns. These approaches have one missing piece: appropriate flexibility of implementation. We feel that EDPs and the rho-calculus have much to offer in this area, providing a unifying formalism for conceptual validation under structural transforms.

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 [6], Coplien and others' idioms [3, 7, 16], and Pree's metapatterns [20] all support this viewpoint. Shull, Melo and Basili's BACKDOOR's [23] dependency on relationships is exemplary of the normal static treatment that arises. A related, though type-based, approach that works instead on UML expressed class designs, is Egyed's UML/Analyzer system [10] which uses abstraction inferences to help guide engineers in code discovery. Though Reiss's PEKOE [21] is similar in nature to SPQR, it uses a relational database language for queries and conceptual component definition. Beyer et al's CrocoPat and RML [4] have a conceptual similarity to SPQR and POML, but require specialized tools to manipulate, instead of allowing COTS and freely available tools to do the searches and meta-analyses.

7. 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 illustrate some of the practical considerations moving forward with SPQR as a toolset intended for practitioners, and how SPQR can be used to assist in its own use.

8. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., 1996.

[2] C. W. Alexander. *Notes on the Synthesis of Form*. Oxford Univ Press, 1964. Fifteenth printing, 1999.

[3] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[4] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31(2):137–149, feb 2005.

[5] J. Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 1(2):18–52, may 1998.

[6] 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.

[7] J. Coplien. C++ idioms. In *Proceedings of the Third European Conference on Pattern Languages of Programming and Computing*, jul 1998.

[8] S. Demeyer, S. Ducasse, and O. Nierstrsz. 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.

[9] A. H. Eden. *Precise Specification of Design Patterns and Tool Support in their Application*. PhD thesis, Tel Aviv University, Tel Aviv, Israel, 2000.

[10] A. Egyed. Automated abstraction of class diagrams. *ACM Transactions on Software Engineering and Methodology*, 11(4):449–491, oct 2002.

[11] 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.

[12] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[14] 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.

[15] 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.

[16] S. Meyers. *Effective C++*. Addison-Wesley, 1992.

[17] 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.

[18] M. Ó Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach*. Ph.D. dissertation, University of Dublin, Trinity College, 2001.

[19] 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.

[20] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.

[21] S. P. Reiss. Working with patterns and code. In *Proc. of the 33rd Hawaii Intl Conf on System Sciences*, jan 2000.

[22] 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.

[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. Elemental Design Patterns: A link between architecture and object semantics. Technical Report TR-02-011, Univ. of North Carolina, 2002.

[27] J. M. Smith and D. Stotts. Elemental Design Patterns: A logical inference system and theorem prover support for flexible discovery of design patterns. Technical Report TR-02-038, Univ. of North Carolina, 2002.

[28] 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.

[29] J. M. Smith and D. Stotts. SPQR: Use of a first-order theorem prover for flexibly finding design patterns in source code. Technical Report TR-03-007, Univ. of North Carolina, 2003.

[30] 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.

[31] 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.

[32] 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.

[33] B. Woolf. The abstract class pattern. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 1998.

[34] B. Woolf. The object recursion pattern. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 1998.

[35] 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.