RC24669 (W0810-084) October 15, 2008
Computer Science

IBM Research Report

Bridging the Gap between Legacy Procedural Code and the Automated Extraction of Design

Jason McC. Smith
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Bridging the Gap Between Legacy Procedural Code and the Automated Extraction of Design

Jason McC. Smith
IBM T. J. Watson Research
Hawthorne, NY 10562
jasonmsm@us.ibm.com

Abstract

Automated design extraction from object-oriented source code has been demonstrated in the System for Pattern Query and Recognition, a static analysis tool and accompanying Elemental Design Pattern definitions that can identify instances of known design patterns in a flexible manner. SPQR concentrates on the concepts embodied in code instead of the constructs that form the code. This paper demonstrates an extension of this concept-based approach to embedded procedural code. Various common idioms in the C language are mapped to object semantics. The resulting model is then analyzed by SPQR. Presented are the results of applying a tool implementing this new concept to a production firmware codebase, with results and new predictive metrics of maintainability of code.

1 Introduction

Program comprehension is a difficult problem that has spawned a wide array of partial solutions. Finding a solution in the domain of embedded firmware is particularly difficult due to several unique constraints. Generally speaking, modern programming languages use semantically-rich abstractions to convey the intent of a developer at a high level. In theory, this increases both the readability and comprehensibility of the code. Embedded systems, however, are generally written in lower-level, earlier-generation languages such as C; they cannot take advantage of recent abstractions to simplify the code or make the developer's intent more explicit. Recently, most software engineering research seeking to improve program comprehension has been inapplicable to systems created using older languages because those research attempts have relied on the high-level abstractions available in modern languages such as Java[12], C++[2], or C#[14].

Techniques aimed at comprehension of legacy systems frequently limit themselves to the semantics available within the implementation languages. These techniques use call graphs, data flow analysis, and other such low-level inspections, but do not address the intent behind these mechanical manipulations in the project's solution space. While providing a fine-grain highly-detailed view of the inner workings of the code, they do not give the developer a faster working knowledge of the system by raising the level of abstraction.

This paper describes a novel approach to forming a bridge between legacy code written in older procedural languages and modern abstraction analysis techniques designed for object-oriented languages. The target technique in this case is embodied in the System for Pattern Query and Recognition (SPQR), a suite of tools designed to identify and document instances of design patterns directly from object-oriented source code in a language- and domain-independent manner[18, 20]. This paper presents the general problem of program comprehension and the specifics of the embedded domain that make the problem more difficult. A short survey of existing techniques for legacy embedded procedural source comprehension is provided, with an overview of SPQR for comparison. Then, the particulars of the analyzed project are presented, followed by the methodology used for the analysis. Next, the contributions of this paper to procedural semantics detection are discussed. The results of the analysis are described, as well as the insights they were able to give the project's team members. This paper concludes with areas identified for further refinement based on team feedback.

2 Problem

Program comprehension is a difficult problem in any sense. Much of the history of programming languages can be viewed as incremental efforts to encapsulate the lessons and best practices of the prior language generation. It is rare, however, to be able to take advantage of these learned

lessons when working with legacy code. In particular, procedural languages are unable to enforce many of the best practice policies that are inherent in object-oriented languages. Abstractions such as containers, decomposable algorithms, and contextual method behavior (polymorphism) are manually implemented with varying degrees of success and clarity. This lack of consistent abstraction clarity leads in many cases to poorly comprehended code. Incomprehensible code is unmaintainable code[3, 23, 24]. At a minimum, poorly-understood code leads to perceived fragility.

An inherent tension between refactoring and stability complicates legacy systems , but there is an additional tension that prevents legacy projects from advancing more rapidly. Much of the current research in software engineering that is abstraction-aware focuses on modern object-oriented languages. It is difficult to argue with this choice. There is a plethora of OO source code to be analyzed, and the nature of OO languages makes them simpler to analyze for key abstractions.

The bulk of code currently in operation, however, is still procedural[7]. Much of our basic infrastructure depends on legacy projects that cannot, for various reasons, be reimplemented in modern languages. There is a chasm between the needs of the legacy practitioner, and the analysis techniques of the modern researcher. A way for existing code to gain benefits from the latest research is necessary.

The situation for embedded and firmware legacy systems is even more dire. Not only are the assumptions of the domain vastly different from those of most software comprehension researchers, there are further constraints in play. These are both technical and management related.

On the technical front, performance and correctness are critical in embedded systems. Updates are frequently very costly. The design methodology is usually much closer to hardware than software, further widening the gap with modern software comprehension research. Non-co-locality of cause and effect by use of hardware-defined control blocks is common. Attempts to apply current 'common wisdom' from mainstream development, such as modularizing a control-block driven system, can make the comprehensibility worse, not better.

From a management perspective, there is a further pressing issue. Legacy projects are frequently staffed by employees who have been around as long as the code. They may be the only members of the company who truly understand the source code, and enough time has passed in our industry that many of them are now retiring. This means that the knowledge that they have in their heads will leave with them. New trainees have been educated using modern languages, design principles, and techniques, and require a large time and training investment before they are able to take over significant duties. The code may not be fragile, but the teams are.

The research described in this paper is intended to bridge the issues facing legacy systems and the most recent software comprehension research, without requiring risky alteration of the existing systems.

3 Prior Research

There have been numerous attempts to improve comprehension of legacy procedural code, but few have attempted to raise the level of abstraction beyond low-level details of execution and memory layout.

Advanced techniques such as slicing analysis[9, 16], and tools which combine static and dynamic analyses in a user-directed fashion[8] are highly useful for tracing through specific code features and performing impact analysis. They do not, however, effectively raise the level of abstraction for the developer. They maintain a close correspondence with the primitives of the languages and domains in which they operate.

At the other end of the spectrum are pattern and abstraction detection systems such as PEC[13], FUJABA[15], and Similarity Scoring[22], which address pattern validation and detection to varying degrees of success. They are all, as with most research in this realm, Java only.

It should be obvious at this point that there is a fundamental mismatch between the two groups of tools, despite their successes in their own realms. The procedural tools provide a wealth of information, but it is at a very low level, inappropriate for trying to raise the level of abstraction and comprehension. The pattern and abstraction detection systems are all Java-specific, and do not work with other languages. Their applicability to the embedded domain is therefore non-existent.

Finally, there is my prior research in program abstraction detection, the System for Pattern Query and Recognition[18, 20, 21]. Three main pieces comprise SPQR: rho- or ρ-calculus, a formal denotational semantics for relationships among the entities of object-oriented languages; a set of tools and a common intermediate representation to perform static analysis around those relationships; a suite of formalized building blocks of implementation and design principles called Elemental Design Patterns.

ρ-calculus is based on sigma- or ς-calculus[1], and extends that body of work to describe the relationships between the fundamental entities of object-oriented languages. ρ-calculus encompasses call graphs, data flow dependencies, state changes, cohesion, coupling, and other such relationships and dependencies within a single overarching concept, *reliances*. The only formal reliance appearing in this paper is the kappa-reliance, indicating that one variable relies on another variable. In the expression `a = b`, where `a` and `b` are variables of compatible types, `a` relies on `b`, and this can be written `a` $<_\kappa$ `b`. ρ-calculus

also defines the various ways in which these reliances can interact through transitivities and reduction rules suitable for inferencing by a deductive database, automated theorem prover, or graph-walking solver. Finally, ρ-calculus includes an open-ended n-tuple composition technique for the expression of n-ary relationships that fall outside of the standard binary relationships of the reliances. These relationships correspond to higher-order abstractions which describe the source, and may be anything from simple programming concepts to design or architectural patterns. As with the reliances, ρ-calculus states how the n-tuples interact to form higher-level abstractions.

The tools of SPQR include front-end parsers to consume source code. They produce an intermediate representation as files in an XML schema named the Pattern Object Markup Language[17]. POML is a practical and concrete representation of ρ-calculus, expressed for human readability and machine manipulation. As indicated by the name, patterns and objects are the focus of POML. After production from the front-end parsers, various transformations for visualization and reporting can be performed on POML files, including the production of many standard software engineering metrics via simple XSLT transformations. For pattern detection analysis, SPQR transforms the POML files into input suitable for an inferencing engine. When combined with the rules of ρ-calculus, the facts extracted from the source code are a rich soup of information that can be quickly queried for the presence of known design patterns or other abstractions.

A suite of low-level abstractions corresponding to the basic elements of software implementation and design was created to make the query process more interactive and parallelizable. I named these Elemental Design Patterns[19]. These are binary relationships defined by the reliances of ρ-calculus, refined by semantic cues in their context within the source code. Creating software without using the EDPs is impossible. As both natural programming concepts and formalisms for automated manipulation, the EDPs bridge the raw reliances and the n-tuple abstractions of ρ-calculus. Additionally, they are the fundamental building blocks of the remainder of the design patterns literature. Through incremental composition, even the largest described patterns are definable with a combination of EDPs and ρ-calculus.

SPQR has been successfully demonstrated to detect high-level abstractions such as the Gang of Four design patterns directly from source code in an automated and practical manner[18]. This paper describes the application of these techniques to procedural code.

4 Methodology

This section presents the analysis methodology, discussing the practical and theoretical steps taken to perform the research. The goal of this experiment was not to find high-level design patterns, such as the Gang of Four literature[10], that SPQR has previously been used for in object-oriented languages. Satisfying such a goal would require that the appropriate patterns already appear in the test code. Because the status of inherent abstractions in an undocumented test project is necessarily unknown, the goal of establishing the presence of the Elemental Design Patterns in the code was chosen. This target is possible to validate manually, raises the level of abstraction for the developers, and provides a platform for further analysis and research with SPQR.

4.1 Test case

The selected test case is part of a product line at IBM, referred to as Project A. As such, it is covered under some information restrictions. Project A is the driver code for an embedded hardware controller that has rigorous requirements for performance and reliability. Updates to the firmware necessitate a dynamic update while the system is running due to contractually-obligated uptime requirements. In short, the presence of a defect is extremely undesirable. These performance and reliability needs coupled with the extreme downside of requiring a patch, lead to a belief that change introduces more risk than it may in most development environments.

Project A is approximately 300kLOC of highly complex but finely tuned C code. Development on Project A has spanned decades. The core team members are long-time employees, who have spent much of their careers maintaining this code. The retirement of these core developers and maintainers is now an imminent event. IBM's goal is to capture as much of the undocumented knowledge of the system as possible for verification before their retirement. Additionally, management faces a situation where new hires brought into the team face a comprehension gap: the semantics and language that recent graduates have been educated in is extremely dissimilar to the semantics and assumptions which the career developers operate under. Training is a long process due to the perceived fragility of the code.

Several attributes of Project A make it a good candidate for this research. First, it is in a procedural language, C. Second, it is in the embedded domain, where core assumptions are far removed from current software engineering research assumptions of abstraction extraction. Third, the team is small enough to be approachable, and is willing to be part of the research process. Finally, it is of sufficient size to give a rich data set, while still being small enough that results could be hand-verified if necessary.

4.2 Approach

This section describes the approach for analyzing Project A. First, the adoption of a new front-end for SPQR is presented, followed by the adaptations of an existing tool to make it appropriate for use with SPQR. Next, the semantic mapping from procedural C code to the conceptual relationship model in SPQR is described. This section concludes with a discussion of the required manipulations of the resulting output to make it suitable for delivery to the development team.

4.2.1 BEAM

SPQR required a new front-end for static analysis of Project A due to legacy restrictions of the prior implementation. The BEAM tool (Bugs, Errors and Mistakes) [4, 5, 6] is a static analysis tool that is currently used in several product groups within IBM for bug detection. BEAM can analyze C, C++, Java, PL/1, PL/X and other languages, and was therefore a natural fit for SPQR's language-independent approach. Because BEAM was already integrated into the workflow of Project A, the transition to the new analysis techniques was smooth.

4.2.2 BEAM Extensions

The extensions to BEAM consist primarily of a library, implemented in C++, that performs the semantic mapping, and a significant amount of glue code to access the internal AST representation in BEAM. Because the AST representation in BEAM is language agnostic, adding support for the additional languages BEAM analyzes, such as C++, is expected to be straightforward. BEAM was also extended to produce POML on request. This combined solution is referred to as BEAM/POML. This means that BEAM is now a producer for the rest of the SPQR toolchain, and projects that have adopted BEAM can get the benefits of SPQR with little overhead.

Several enhancements to the original SPQR analysis methodology were made, including improved temporal ordering validity and loop-conditional propagation. In the prior SPQR incarnation, relationships were created on a per-method basis as a logical set, without regard for the sequential ordering of statements or control flow. Both are now taken into consideration when applicable, producing more refined results.

4.2.3 Procedural Semantic Mapping

The conversion of source code from procedural language semantics to the semantics of ρ-calculus is straightforward, but not trivial. There are two classes of semantic mappings from C to ρ-calculus: language, and idiomatic. Language-based mappings transform language constructs in C to equivalent concepts in an object-oriented design space. Idiom-based mappings transform standard usage patterns of C into OO concepts. Each is addressed in turn next.

Language Features

There are two types of language-based mappings: those dealing primarily with structural language features that can be mapped to object scoping semantics, and those that reflect semantic language features that can be mapped to object abstractions. I will discuss each in turn.

Structural language features dictate how a procedural system is laid out according to file and scoping. For example in C, structs provide a scope for internal variable names, and the static keyword, which indicates linkage, alters visibility semantics. The latter concepts of linkage visibility are naturally considered as additional scoping issues. I will discuss scoping in general before introducing the individual language features.

Scoping is a semantic partitioning of a potential namespace which both prevents clashes from the reuse of names, and provides a hierarchical set of abstractions through which a developer may reason about software. As with most procedural languages, C has primitive scoping through grouping constructs such as struct. Considering procedural code as an object-oriented system, however, requires that a more stringent and rich scoping system be used. This leads to two core mismatches that must be resolved.

The first mismatch is with the concept of *global entities*. In C, any variable or function may be placed in the global scope and given external linkage. Any other appropriately linked-in code may refer to these entities directly. In ρ-calculus, as with other purely object-oriented semantics, raw or floating fields and methods are disallowed. To satisfy this constraint, an `__GLOBAL__` artificial object named `__GLOBAL__` is created. It is provided as a scoping 'home' for raw global entities from the original C code. A function previously called through `myFunction()` would now be called as `__GLOBAL__.myFunction()`. Top-level objects are handled differently in the semantics of ρ-calculus than objects and fields created during the execution of the code, and they map well to compiler- and runtime-actions in traditional procedural systems.

The second mismatch occurs with the mapping of *static file-level entities*. In C, a function or variable may be restricted to access by a particular translation unit's members by way of the *static* keyword. This is one way in which C developers reduce the pollution of and maintain the global namespace. Conceptually, this is a scoping to the file in which the entity is defined. We can emulate this by creating an object much like `__GLOBAL__` for each translation unit. If an entity is given external linkage, it is placed in `__GLOBAL__`. If an entity is given internal linkage and tied to a particular file, it is placed in that translation unit's ob-

ject. A convenient way to add traceability is to name these TLU objects as [source_file_name]_TLU.

There will be one TLU object for each translation unit, but there will be only one __GLOBAL__ object per analyzed codebase. Because each object must have a type, a type is generated for each TLU object, and for __GLOBAL__, that matches each object's semantics.

In C, structs, unions, and enums all map, at a fundamental level, to the concept of a class type in ρ-calculus. This is not surprising, as classes and structs in C++ differ only in default visibility, private or public respectively. A struct is therefore handled as if it were a data-only class. A union poses a slight permutation of this, in that the storage for the elements in a union is unified. Setting one name within a union scope results in altering the value for all names within that scope. This is handled by creating reliances between all names within a particular union. A change to one is therefore propagated as a change to all. The code below demonstrates one conceptual implementation of a two-member union in a C++-like pseudo-language, with set/get accessors.

```
class myUnion {
public:
  int  getFirst(){ return int(*storage); };
  char getSecond(){ return char(*storage); };
  void setFirst(int val) {
    memcpy(storage, val, sizeof(int)); };
  void setSecond(char val) {
    memcpy(storage, val, sizeof(char)); };
private:
  void* storage;
};
...
MyUnion aUnion;
aUnion.setFirst(5);
char c = aUnion.getSecond();
...
```

Were the accessors replaced with generated properties, as in Objective-C or C#, the use of this myUnion class would look and perform identically to a C union. Imitating the properties approach, new reliances can be generated. The resulting semantic is similar to that below, in a pseudo-language implementation.

```
class myUnion {
public:
  property int first;
  property char second;
private:
  void* storage;
  gettor first { return int(storage); };
  gettor second { return char(storage); };
  settor first {
    memcpy(storage, val, sizeof(int)); };
  settor second {
    memcpy(storage, val, sizeof(char)); };
};
...
MyUnion aUnion;
aUnion.first = 5;
char c = aUnion.second;
...
```

This can be further simplified using simple reliances from ρ-calculus, because we are not concerned with the implementation details of memcpy or the casting of return values. We only need to know that when any one name in the union is modified, all are modified. We can demonstrate this refinement, as ρ-calculus, by expressing these cross-reliances as the data-to-data kappa-reliance: $MyUnion = [int first, char second], first <_k second, second <_k first$ Note that the reliances directly generated by this mapping from union to ρ-calculus are all inferable from the reliances in the implementation. The intervening steps have been eliminated as unnecessary implementation details.

C enums are another special case of a class type, in that they provide physical, but not logical, scoping. In a C file, an enum looks much like a struct with default initialization values: it defines a type, and defines members of that type. We can treat it as such in ρ-calculus as well. One difference is that the enum does not provide a name scoping around its members. Instead, they are hoisted up to the surrounding namespace. For instance, if enum { first, second } is declared at global scope, then first and second have global scope.

There are two ways to map this to object semantics. One is to create a number of similarly scoped constant objects of a new type reflecting the enum, and give them static persistent storage. This allows code that uses the enums to remain unaltered but it creates an encapsulation issue. There is nothing barring a developer from using this equivalent implementation to create new instances of the enum type, and assigning them new values, effectively extending the enum type dynamically. While the scoping for this approach is correct, the potential behavior is not.

The second approach, and the one taken in this research, is to change the scoping and restrict this behavior. This is accomplished by considering the enums as the equivalent of a C++ construct: static const members of a class whose constructors and assignment operator are restricted to remove extension of the possible values. A primitive implementation equivalent is shown below.

```
class MyEnum {
public:
    static const MyEnum first = 1;
    static const MyEnum second = 2;
};
```

Reference to the enum members is accomplished by altering the *reference point* of an enum to include the type scoping. What once was a use of `first` becomes a use of `MyEnum::first` in C++, or `MyEnum.first` in ρ-calculus. This preserves typing of the enums and establishes clear encapsulation principles.

Anonymous unions in C are a special case, requiring an alteration of the scoping on a fundamental level. While not supported in ANSI C, they are supported in IBMs *xlc* compiler suite, and were required for this analysis. Following the model of ANSI C++, the members of a union are considered to be members of the encompassing scope, but have shared storage. We can accomplish this by following the example of a standard union above, generating appropriate reliances among the members, but placing those members into the surrounding scope.

Similarly, the use of the *static* keyword forces a change of scoping for entities in C during this mapping. For local variables, which have persistence within a particular function, an equivalent field can be created as a field in the scope that contains the resulting method corresponding to the original function. The most common mapping is that this resulting field will be in the translation unit object holding the mapped method. To help prevent name clashes and to increase traceability, the function name is prepended to the variable name for uniqueness. In the rare cases of clashes after this name manipulation, further steps can be taken to ensure uniqueness. References to the variable within the function are altered to refer to this new name. Below we see this mapping in pseudo-code.

```
In sourcefile.c:
void f() {
  static int a = 0;
  a++;
};

Maps to:
object sourcefile_c_TLU {
int f_a = 0;
void f() {
  f_a++;
  };
};
```

In addition to the structural and scoping mappings, there are several semantic mappings from the C language to the object semantics of ρ-calculus and SPQR. Object initialization, container semantics, and abstract methods all have analogues within C. These concepts go beyond the simple scoping semantics discussed so far, and are not directly expressible within ς- or ρ-calculus. They need to be expressed as higher-level abstractions. The Elemental Design Patterns from SPQR form a base lexicon of such abstractions, and we can map these particular features to EDPs.

Object initialization is a core concept, and considered by some to be the defining concept, of object-oriented languages[ref cardelli]. There is no native initialization language feature for non-primitive types in C. Instead, the dynamic allocation of memory is deferred to a standard library function, `malloc()`. This does not cover all allocation cases, however, as local and const variables with initializers must be handled by the compiler and language runtime. Automatic initialization is addressed first, then the dynamic allocation issue.

In the catalog of semantics and abstractions used in SPQR, object instantiation is encapsulated as the CreateObject Elemental Design Pattern. It encompasses object instantiation in all its forms, and provides a simple unifying abstraction for higher level abstraction manipulation.

Local variables that are not pointers or of a primitive type in C can be considered instances of CreateObject. In C++, a local non-primitively-typed, non-pointer variable is created at runtime initialization by calling the appropriate constructor. This is obviously an instance of CreateObject. In C, a similar action occurs, when local variables are allocated. While a constructor is not called, as would be the case in C++, the allocation should be recognized as such. Because of this, a CreateObject EDP instance is created for each appropriate variable in C. This example shows the various types of C constructs that generate CreateObject instances.

```
typedef struct MyStruct {...} MyStruct;
void f() {
  int a;         // Primitive no CreateObject
  MyStruct s;    // CreateObject for s
  MyStruct *sp; // Pointer, no CreateObject
};
```

Containers are another higher-level abstraction that is increasingly common as a core concept in many languages, and are a fundamental piece of many design abstractions such as design patterns. C has two mechanisms for basic containers. The first, arrays, are a direct expression of a container. The second, pointers, sadly are not, and are used for many purposes, including interaction with arrays in sometimes indirect and odd ways. Each of these mechanisms is discussed in turn.

As a direct expression of a container, arrays give us a simple way to detect a Container EDP and mark a variable as such. On seeing a variable declared as an array of another type, we can immediately create an instance of Container for the corresponding field. The interaction of the contents of that container with other variables and methods, however, is a bit more interesting. Consider the following C code.

```
int a[10];
a[5] = b;
c = a[5];
```

c should rely on b indirectly through the transitivities of ρ-calculus, but there is no way to indicate this without having the reliance pass through the array a first. This intermixing of the reliances between the container and the items it contains, however, can lead to some rather surprising and semantically vague conclusions, as shown next.

```
void g(int arg[]) { arg[0] = arg[9]; };
void f(int arg[]) { g(arg); };
int a[10];
a[5] = b;
f(a);
```

If a relies on b, ($a <_\kappa b$) as required in the first example, then we infer that f() also relies on b through the call to f(a). Inspection of the body of f, however shows no point at which the internals of a are accessed or otherwise relied upon. a should be considered an opaque container, or invalid inferences begin to creep in.

The solution is to create a placeholder for the elements of a container. Full dynamic analysis can deduce many of the runtime interactions and relationships between the items placed within a container, but such an analysis is expensive, and we seek a fast static analysis solution. We can consider the container as a new pseudo-type, and provide a scoped field container_elem as a proxy for all items that may be placed into, or extracted from, the container.

From our example code above, we would have $a.container\_elem <_\kappa b$, and $c <_\kappa a.container\_elem$. Now $c <_\kappa b$ as expected, but we can not infer that f relies on b. a.container_elem does not appear in the facts expressed about f(), and we cannot follow through to b.

This simple distinction between the container and the mocked up entity for the items stored in the container is fast, easy, and satisfies many of the logical inconsistencies that crop up otherwise.

With a model for a Container that includes a distinct container_elem entity, pointers as containers can be addressed. Unlike arrays, we cannot detect container semantics for a pointer at the point of variable declaration. Instead, they must be detected at the point of container behavior, when the pointer is used as either an iterator into a dynamically-allocated memory block, or manipulated and dereferenced in such a way as to imply container semantics. These are not tremendously difficult behaviors to detect, and they help distinguish how a pointer is being used. Below are a few of the relevant constructs and their mappings.

```
int a[10];  // Container
int *b;     // Unknown
int c;
c = (*b);   // b may be Container
c = (*b++); // b may be Container & Iterator
c = a[c];   // c is index, but not Iterator
c = a[c++]; // c may be Iterator
```

In addition to the basic Container semantics, we can also deduce when variables are being used as iterators into a container. In the case of the array above, if we have a variable that is being used as the index into the array, and that variable is being manipulated repeatedly through either loop detection, or by the variable being local-static in a repeatedly called function, then we can state with strong certainty that the variable has Iterator semantics in addition to being used as an index. An instance of an Iterator EDP can then be made.

With pointers, the situation is again a bit more difficult. We can, however, leverage the above Container semantics detection. When a pointer is being incremented or decremented before being dereferenced, it is acting as both the container and the iterator into that container. In such cases, an Iterator EDP would be noted as well as the Container EDP, for the same pointer. Note that if a pointer is not acting as a Container or an Iterator, it is considered a plain variable.

Abstract methods would seem to be a concept that has no analogue within C, but in many ways, function pointers fit well. In object-oriented languages, a pure or virtual abstract method is a promise of a later method body definition. When that method is overridden in a subclass or subtype, that promise is fulfilled. By breaking the concept of an abstract method into these two pieces, it becomes possible not only to more cleanly describe many behaviors in OO systems, but it becomes possible to apply these sub-concepts to C function pointers.

A function pointer is a promise of a later function definition. The assignment of a function to the pointer is the fulfillment of that promise. When a function pointer is declared, or assigned to, we can create instances of Abstract Dynamic Method and Fulfill Dynamic Method EDPs, respectively. These are dynamic twins to the Abstract Method and Fulfill Method EDPs used in SPQR's previous OO analysis. Adding these into the EDP collection enables SPQR to analyze not only code using function pointers, but also dynamic languages with runtime-composable types.

The function pointers themselves pose a problem in ρ-calculus, however, being neither a field nor a method, but something in-between. We solve this by creating a method that corresponds to the function pointer as a callable entity, and a matching field that represents the function pointer in non-call expressions. The two are then tied together with the appropriate reliances, as were the members of a union, at the beginning of this section.

I expect that this function pointer analysis will lead to a more robust way of detecting subtypes within C, as a method for mapping groups of function pointers into hierarchies of pseudo-inheritance. Containers of function pointers

will likely be the key here.

As would be expected, the standard features in a procedural language are not sufficient to capture many object semantics. Object-oriented languages, after all, were intended to codify best practices in procedural languages. We must turn to the practices and idioms of procedural languages for further mappings.

4.2.4 Usage Idioms

In addition to the standard language features, there are several development idioms that map well to object semantics: file layout, including directories, preprocessor macros as functions or variables, and the use of malloc for dynamic memory allocation.

Using directories to partition a conceptual space is an established idiom in most languages and development environments. They are used to encapsulate and contain similar source files into discrete units. In some languages there is a one-to-one mapping of directory layout to language semantics, such as Java's use of directories to delineate packages[12]. This idea can be mimicked to use file location information to create a namespace partitioning in ρ-calculus. Namespaces in ρ-calculus are indicated by an object, of which `__GLOBAL__` and the TLU objects discussed in the previous section are special cases. The resulting scoping is modified at a reference point such as a function call, or variable access, to conform. In C, there is a small issue in that declarations and definitions need not be within the same directory. It is not uncommon to see headers gathered into a single directory for easy access, with corresponding source files scattered in other directories based on behavior. This can cause erroneous scoping, as the uses of the header-declared entities and the definition from the .c file is placed in separate namespaces. This is solved in BEAM/POML by providing a user option at the command line to control namespace partitioning. While a more automatic solution could be created, it would require a re-resolving of the scoping at a large scale on a whole-system basis. While suitable for future work, it is not within the scope of this project.

Preprocessor macros in C are commonly used as stand-ins for functions, in an effort to remove the overhead of a function call for performance reasons, or for read-only constant variables. Embedded systems with direct access to hardware frequently use them for a third reason: flexible mapping of hardware memory locations for direct access, so that it appears as a variable. BEAM/POML treats these macro directives as if they were normal functions or constant variables, placing them in the namespace in which the macro is defined.

Finally, dynamic allocation is handled in C through the use of the library call `malloc()`. These are simple to detect and convert to a CreateObject EDP, when surrounded by an appropriate cast, and used in an assignation to a variable. The idiomatic mappings described in this section are shown below.

```
#include ''subdir/myHeader.h''
                    // declares void g(void);
typedef struct MyStruct { ... } MyStruct;
#define A\_CONST 5; // const int A\_CONST = 5;
void f() {
  g();              // subdir::g();
  sp = (MyStruct*)(malloc(sizeof(MyStruct)));
                    // CreateObject for sp
};
```

4.3 Output Transforms

The individual POML files for each TLU were aggregated using a custom XSLT, into per-directory (and therefore namespace) POML files. These in turn were aggregated into a single, holistic system-wide POML file suitable for analysis of the entire codebase. This approach produces a series of POML files at varying levels of detail, allowing a developer to focus solely on the current area of interest.

Visualization of the data was performed in two stages. First, the POML files from the above aggregation step were manipulated by an XSLT that transforms a POML file to a DOT[11] file suitable for rendering by any number of graph layout tools, such as the GraphViz package from AT&T[11]. The data in the base diagram is the set of all types in the system, partitioned by namespace, and displayed as a UML class diagram, complete with aggregation, dependency, and composition indicators. Several options can be passed to the XSLT, to have it include call graphs, data flow, and pattern instances.

5 Lessons Learned

This section discusses the results and lessons from the analysis of Project A with BEAM/POML and SPQR, which can be summarized in two phases: research-oriented, and project-oriented. The research-oriented results are concerned with the overall scalability and performance of the analysis, while the project-oriented lessons are those that were identified by the development team.

As stated in Section 4.1, Project A is approximately 300kLOC of C in 187 source and 109 header files, which is broken into 12 major modules partitioned by directories, and five minor modules (defined only by the build system's behavior) gathered into one directory. In total, BEAM/POML detected 619 distinct class types, and 96 namespaces. Of those 96 namespaces, 17 were directory-derived namespaces, one was `__GLOBAL__`, and the remaining 78 were the translation unit namespaces for the source

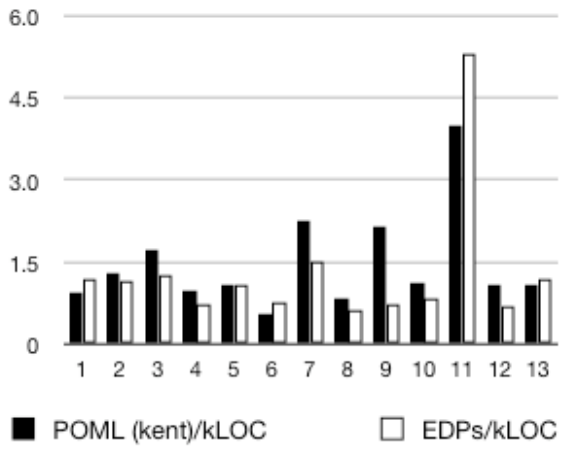

Figure 1. Information and Abstraction Density

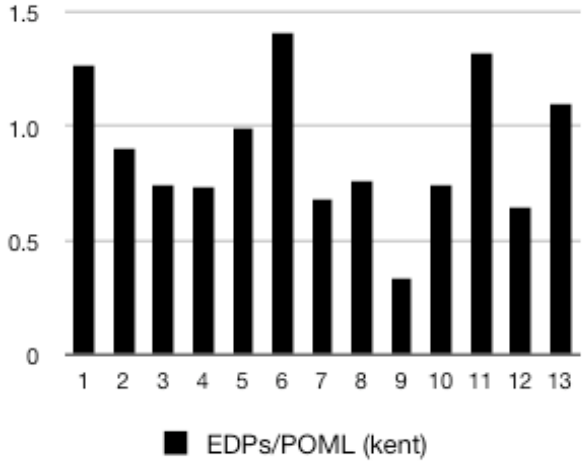

Figure 2. Relative Abstraction Density

files which required them. Total analysis time was 206.8s on a 2.4GHz MacBook Pro, indicating good scalability.

As expected at the outset, the higher-level abstractions corresponding to the standard design patterns literature were not detected. The embedded firmware domain is not heavily represented in mainstream patterns literature. The large number of detected EDPs, however, did provide an interesting view into the inner workings of the code that the team had not considered before. Anecdotally, the patterns of function pointer declaration and assignment were considered 'interesting' and prompted a discussion on a possible refactoring more closely aligned with traditional OO abstract methods and subtyping mechanisms.

The density of concepts extracted from the code was noteworthy. Figure 1 shows the 'conceptual density' of the thirteen modules, measured in two ways: as the number of overall entities detected per kLOC of code, and as the number of EDPs detected per kLOC of code. The first measures the ratio (in thousands) of constructs, relationships, and abstractions statically detected per kLOC. It is a measure of the overall density of facts within the module, an *Information Density*. A high ID indicates code that, without abstractions, will require more work on the part of a developer to comprehend completely due solely to the density of raw information. A low ID indicates code that is simple.

The second measure concerns only the number of EDPs per kLOC, indicating the relative richness of abstraction in the module, or *Abstraction Density*. A high AD indicates code that is more likely to be expressible in higher-level abstractions such as design patterns. A low AD indicates that the code may suffer from a lack of abstraction, or it may be that the proper abstractions to describe that code have not yet been defined. In either case, it points at code that may be more difficult to understand through either inherent complexity that is indescribable in common terms, or a lack of proper working concepts.

If we take the ratio of the above two measures, such that we have the number of EDPs per thousand POML entities detected, as in Figure 2, we arrive at an *Relative Abstraction Density* measure. When the RAD is high, you may expect that the code will be easier to understand, regardless of the ID. A module with a lower ID, such as module 6, will simply have less information to comb through, and is likely to have a relatively high number of abstractions. Code with a high ID but a high AD, such as module 11, should similarly be more maintainable if those abstractions are understood. When the RAD is low, as in the case of module 9, you may expect that comprehension will suffer. The Project A team was able to anecdotally confirm these relative findings in a broad sense. More data will be required before guidelines based on quantitative values can be created.

The results concerning one EDP should be noted. CreateObject is a ubiquitous part of any OO system, yet a brief scan of Project A's analysis results showed not a single CreateObject being triggered from a dynamic allocation. This was considered a bug, until further checking revealed that the source code did not contain even one call to `malloc`. All memory in this hardware controller is by necessity hardwired to specific addresses at compile time, and there is no dynamic allocation. This reveals yet again a fundamental mismatch between embedded software development and the core assumptions of most of the current software engineering research in design abstraction. BEAM/POML along with SPQR is able to bridge this conceptual gap.

The visualizations created were of mixed utility. They did provide an innovative view into the code, as a UML

class diagram of procedural C, but the scope was simply too large for easy consumption. As a raw metric, the base UML diagram, rendered by GraphViz using 10pt font, exceeded 8m by 24m in size. Even this, while poor in scalability and showing only a fraction of the available data, provided the team with several new insights. Multiple cross-module connections were shown on the visualizations that the team was unaware of, but a few minutes of code checking confirmed their existence. The team was able to verify several instances of key types that had a large number of dependency connections from wide-spread code, as 'hot-spots', yet were surprised not to see certain expected connections to particular modules. Again, this was confirmed quickly in the code, validating the data.

6 Validity Threats

This section addresses the limitations of this work with respect to the data set, the methodology from Section 4, and to the interpretation and predictive power of the results.

The data set for this research is currently limited to one codebase, Project A. Asserting unequivocally the generality of the insights gained from this project is difficult. It is not in contention, however, that the research was able to quickly provide the development team new and useful understandings of their own code. From a practical standpoint, these preliminary results showed the success of the research.

Simply put, too much data was extracted by SPQR to present to the development team in an easy to use manner. While manual validation of a random sample of code reliances and abstractions was performed with no false positives, more precise visualization techniques must be developed to quickly convey the larger lessons within the data to developers.

The core abstractions defined in the EDPs are already of too high a level to be properly expressed using standard UML diagrams. While a small number of them can be shown, the larger collections of abstraction patterns are of primary interest. They very often span large regions of the system, tying together disparate portions with simple connections that are lost in a larger diagram when it is displayed as a physical layout such as a UML class notation. While rich data is extractable, it is difficult to sift through at this time.

7 Future Iterations

In this section I will discuss future improvements to the research, based on project partner feedback. BEAM/POML + SPQR in its current state is already producing meaningful results for the development team on Project A. There are, however, several possible enhancements to the workflow. The static analysis in BEAM/POML described in Section 4.2.3 would benefit from some key refinements: first, macro support, second, temporal ordering of statements, and third, subtyping relationship detection. Additionally, the visualization tools and techniques are primitive.

While there is some macro support in BEAM/POML, it has difficulties with deeply nested mock function macros, and extraction of a mock function body from the AST sometimes leaves unresolved address references. This is a known issue, but it means that code with ubiquitous macro replacement of functions that are being deeply nested may lose relevant relationships. Very little such code was detected in Project A, and BEAM/POML was enhanced to accommodate Project A's instances of this behavior. A more robust solution will be required for the general case.

Although the temporal ordering of statements is commonly taken into account when producing reliances, a substantial improvement over the original SPQR implementation, the results are still not as precise as possible. A full implementation of temporal calculus would require substantial reworking of ρ-calculus, and SPQR would lose much of the practical inference speed it currently has. I believe there is a middle ground defined by partitioning method bodies into backward-slices seeded from statements that have external side effects. These slices would then become the input for not only improved reliance inferencing, but would stand ready for assisted refactoring opportunities.

Finally, the lack of subtyping within C and other procedural languages is a significant block to a rich and detailed extraction of higher level abstractions, such as is possible in object-oriented languages. Many of the identified best practices abstractions in OO rely fundamentally on subtyping relationships. There is an interesting possibility, however. In OO, subtyping is used to tie together abstract methods with their points of fulfillment in the child classes. By identifying the Abstract Dynamic Method and Fulfill Dynamic Method EDPs in C's use of function pointers, we can reverse this scenario, and use the presence of the abstract methods and their points of fulfillment as seeds for determining subtypes. This would represent a significant enhancement to SPQR's ability to analyze procedural code at a high level.

It was quickly obvious to both myself and the Project A development team that the visualizations currently used are not up to the task of presenting in a meaningful way the wealth of information SPQR was able to extract. The scope is simply too large to produce a diagram that provides both the desired high-level abstractions, and the fine structure detail demanded by a UML-based approach. We are collectively investigating new methods for data presentation that will allow the development team to scale up or down as needed, and disconnect the semantic map from the physical code, while maintaining strong traceability.

8 Conclusion

This paper described the novel application of the abstraction and design pattern detection of SPQR to procedural code in a production environment. The BEAM tool was selected as a parsing front-end for the C language, and it was modified to emit POML files suitable for SPQR consumption. This BEAM/POML tool was run on approximately 300kLOC of C source from a production embedded firmware system at IBM. SPQR was then used to identify instances of the Elemental Design Patterns catalog from the POML files, as a way of establishing a first benchmark for higher level abstraction detection in procedural code. The results of this effort quickly provided the development team with new insights in the realms of refactoring opportunities, code clarity, and general comprehension of their code, but suffered from a lack of appropriate presentation techniques. We will continue working together to establish a collection of visualization methods applicable to multi-scale abstractions in source code. This work establishes that legacy systems in highly specialized environments can benefit from modern software engineering research, with the proper bridge.

References

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

[2] Accredited Standards Committee X3 (Information Processing Systems) – Working Group 21. Working paper for draft proposed international standard for information systems – programming language C++. Technical Report X3J16/96-0225 WG21/N1043, American National Standards Institute, Dec 1996.

[3] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–94, 1993.

[4] D. Brand. A software falsifier. In *Proceedings of the 11th International Conference on Software Reliability Engineering*, 2000.

[5] D. Brand, M. Buss, and V. C. Sreedhar. Evidence-based analysis and inferring preconditions for bug detection. In *IEEE International Conference on Software Maintenance*, 2007.

[6] M. Buss, D. Brand, and V. C. Sreedhar. Flexible pointer analysis using assign-fetch graphs. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, 2008.

[7] T. Chan, S. L. Chung, and T. H. Ho. An economic model to estimate software rewriting and replacement times. *IEEE Transactions on Software Engineering*, 22(8), 1996.

[8] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 602 – 611, 2001.

[9] R. Ettinger. *Refactoring via Program Slicing and Sliding*. PhD thesis, Oxford University, 2006.

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

[11] E. R. Gansner and S. C. North. An open graph visualization system and its applications. *Software - Practice and Experience*, 30:1203–1233, 1999.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification*. Addison-Wesley Professional, 3rd edition, Jun 2005.

[13] H. C. Lovatt, A. M. Sloane, and D. R. Verity. A Pattern Enforcing Compiler (PEC) for Java: Using the compiler. In S. Hartmann and M. Stumptner, editors, *Conferences in Research and Practice in Information Technology*, volume 43. Appeared at The Second Asia-Pacific Conference on Conceptual Modeling (APCCM2005), 2005.

[14] Microsoft Corporation, editor. *Microsoft Visual C# .NET Language Reference*. Microsoft Press, 2002.

[15] J. Niere, L. Wendehals, and A. Zündorf. An interactive and scalable approach to design pattern recovery. Technical Report tr-ri-03-236, University of Paderborn, Paderborn, Germany, January 2003.

[16] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper. Faster wcet flow analysis by program slicing. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, pages 103–112, New York, NY, USA, 2006. ACM.

[17] J. M. Smith. Pattern/Object Markup Language (POML): A simple XML schema for object oriented code descripton. Technical Report TR04-010, University of North Carolina at Chapel Hill, 2004.

[18] J. M. Smith. *SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns From Source Code*. PhD thesis, University of North Carolina at Chapel Hill, Dec 2005.

[19] 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 Soft. Engineering Workshop*, pages 183–190, Dec 2002.

[20] 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.

[21] J. M. Smith and D. Stotts. *Intent-Oriented Design Pattern Formalization Using SPQR*, chapter 7, pages 123–155. IDEA Group, Inc, 2007.

[22] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, nov 2006.

[23] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8), aug 1995.

[24] A. von Mayrhauser and A. M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–437, jun 1996.