**RC23343 (W0409-135) September 21, 2004**
**Computer Science**

# IBM Research Report

## Pervasive Query Support in the Concern Manipulation Environment

**Peri Tarr, William Harrison, Harold Ossher**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

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

# Pervasive Query Support in the Concern Manipulation Environment

Peri Tarr, William Harrison, Harold Ossher
*IBM Thomas J. Watson Research Center*
*P.O. Box 704*
*Yorktown Heights, NY 10598, USA*
*+1 914 784 7278*
*{harrisn,ossher,tarr}@watson.ibm.com*

## Abstract

*Queries play a fundamental role in aspect-oriented software development (AOSD). They are used to find and define concerns, and to specify how concerns are to be woven together. The Concern Manipulation Environment (CME) is an open-source environment intended to support AOSD across the software lifecycle and to provide an integrated development environment for developers using AOSD This paper describes the pervasive query support CME provides for use by all components and tools within the environment, though both its query language (Panther) and its query implementation framework (PUMA).*

## 1. Introduction

Aspect-oriented software development (AOSD) relies on flexible specification of modules and their interconnections. The static definitions found in conventional programming and architectural description languages are inadequate: flexible description of modules and their interconnections intrinsically requires queries.

The queries are used to describe points of interest in software. These include *join points*[1] and *concerns*[2], which may be crosscutting. Queries form part of the definition of aspect-oriented software, in much the same way that methods and classes form part of the definition of object-oriented software. Queries must therefore be elevated from a support capability for developers, who ask questions about the software, to a first-class part of the software description itself.

Queries are needed, in fact, in several different contexts in AOSD. Perhaps the best-known is in the expression of weaving. *Pointcuts* in AspectJ [10] and related languages, for example, are queries over runtime events at which advice can be applied, such as object creation, method call or execution, or field access. There are related queries that find corresponding elements in the program itself. Similarly, some composition relationships in Hyper/J [11] use patterns that are queries over static software elements, such as classes and their members, and Hyper/J's default *mergeByName* involves a query that finds tuples of entities with matching names.

In this context, the use of queries instead of simple references is essential for two reasons: *multiplicity* and *intensional specification*. A single weaving specification often involves multiple elements, such as applying a single piece of advice to multiple joinpoints, or specifying a multiplicity of class compositions through a single *mergeByName.* It would be prohibitively cumbersome to enumerate all of these. But far more important, such an *extensional* specification would fall out-of-date when additions were made to the software. Using queries as an *intensional* specification ideally allows the developer to express the semantic essence of the desired weaving, which can be automatically updated when the software changes. The correctness of the update does depend, of course, on the nature of the query, making *semantic queries* (such as finding all public classes or all methods reachable from a starting point) attractive. Intensional specifications can also help to address the well-known problem of enforcing properties consistently across the different artifacts of the software lifecycle. For example, when writing requirements or use cases, a developer may want to state properties expected of, and relationships to, artifacts—like design, code, and test cases—that do not yet exist. Intensional specifications facilitate such definitions.

AOSD also involves a variety of activities other than weaving. There has been growing interest in identifying, modeling, extracting and reengineering concerns that occur in existing software, though not as separate modules. The process of finding such con-

[1] The locations at which aspects are integrated.

[2] Collections of software artifacts or artifact fragments that have some common semantic meaning or purpose, whether localized features or systemic behavior.

cerns, and those parts of the software elements that pertain to them, involves analysis, understanding and navigation of the software. Queries, whether semantic, name-based, or text-based, play a prominent role here. Tools such as FEAT [14] and JQuery [6] provide a variety of query capabilities to facilitate this task.

Once exploration has led to the identification of concerns, it is valuable to record that knowledge in a concern model, as allowed by FEAT, for example. A concern can be defined *extensionally*, as precisely the current results of the query; *intensionally*, as the query itself; or both. As in the case of weaving, intensional is generally far preferable, as such queries provide a characterization of the concern that can be re-evaluated if the underlying software changes.

Queries are thus necessary in at least three contexts in AOSD: identifying concerns, modeling (or defining) concerns and other artifacts, and weaving (or composing) concerns. In an environment that supports AOSD as a whole, it is therefore clearly valuable to have uniform, shared support for queries throughout. This provides a consistent user model, where, for example, queries used for exploration can later be used for weaving, or as part of the definition of artifacts (e.g., as pointcuts), or, conversely, queries in an artifact can be used for exploration. It also provides reuse, consistency, and integration across AOSD technology implementations, where different components and tools in the environment all use the same query support.

The Concern Manipulation Environment (CME) is just such an environment, intended to support AOSD across the software lifecycle. It contains a growing set of tools for specific software-engineering tasks, including identification, modeling and composition of concerns, with concern extraction to come shortly. It is built on a set of flexible, extensible components, intended as a platform on which AOSD tools can be built and integrated. The CME is available as an Eclipse open-source technology project [4].

This paper describes the pervasive query support the CME provides for use by all components and tools within the environment. One of the goals of the CME is to support experimentation with and interoperation of multiple AOSD languages and approaches. Since each of these might involve its own pattern or query language, the CME's generic query support component (called *PUMA*), is flexible and extensible, and can accommodate multiple query languages. A second goal of the CME is to provide an integrated development environment for developers using AOSD. Towards this goal, we have also developed a specific query language, called *Panther*, built on PUMA, which we believe provides a powerful, general means of querying software structure and relationships. It also includes the static part of the AspectJ pointcut language, making it natural for AspectJ developers.

Section 2 describes a model of software--the domain of queries--used by the CME query support. Section 3 defines concepts underlying the query languages it supports, illustrated with examples expressed in Panther. Section 4 shows Eclipse views for entering queries and examining their results. Section 5 discusses the architecture of the query support, including the support for multiple query languages. The remaining sections discuss experience, related work, and conclusions.

## 2. The Software Model

To be able to serve as a platform for AOSD across the lifecycle, the CME must be able to handle many kinds of artifacts, and to accommodate new kinds of artifacts easily. The central CME support for queries, concern modeling, extraction and composition therefore uses a general model of software that is independent of artifact details. Specific kinds of artifacts are supported by *artifact plugins* that know the details of their artifacts and can read the artifacts' representations and populate the general model. The plugins can also perform some artifact-specific tasks at the request of the CME. This section describes the general model of the software being searched.

### 2.1. Entities

An *entity* is any named software element. Some entities can be *containers*, whose *elements* are other entities. We distinguish several kinds of names in the CME, differing in their qualification. In this paper we use "name" to mean *self-identifying name*, a name that is complete enough to identify the entity within a working context. This is typically a qualified name or path name with name components for each container surrounding the entity. Names can include *signatures*, as in the case of methods in languages with overloading.

Most entities have other properties of interest too, such as location and content. Their nature is, of course, dependent on the nature of the entities, but the CME requires a way to deal with them that is not. The following subsections describe how the CME deals with four kinds of properties in an artifact-independent way.

2.1.1. **Modifiers**. *Modifiers* are tokens or keywords that convey some information about an entity, such as "interface" or "public" in Java. Entities can have arbitrary modifiers, and queries can refer to them. The query support treats them simply as strings.

In many languages, the absence of a modifier essentially implies a default modifier. For example, a Java class with no visibility modifier implies "package" visibility. Artifact plugins synthesize actual modifiers for these cases to enable them to be handled uniformly and to allow them to be differentiated from *any* in queries.

2.1.2. **Classifiers**. Some modifiers are used to categorize entities. For example, "interface" in Java is used to distinguish interfaces from classes. The CME provides *classifiers* for expressing this categorization in queries. For example, interfaces and classes in Java, aspects in AspectJ and classifiers in UML are all "type" constructs. In some situations, one might want to simply ask for all types, rather than having to identify (or even know) the different kinds of types.

Classifiers are also treated as simple tokens, like "type," arranged in a classification hierarchy. The hierarchy is prepopulated with some common classifiers, but artifact plugins supporting artifacts with new kinds of entities can register new classifiers and specify their place in the hierarchy. They can also register relationships between modifiers and classifiers, specifying that an entity with a particular modifier classifies as the specified classifier. For example, the modifiers "interface" and "class" in Java, "aspect" in AspectJ and "classifier" in UML all classify as "type."

2.1.3. **Attributes**. As simple tokens, modifiers and classifiers convey limited information. Entities can also have *attributes*, which are name-value pairs. The values can be simple, such as a location attribute with a string value, or collections or other more complex structures.

2.1.4. **Methoids**. The most complex information associated with an entity is usually its contents, such as the text of a document (or section) or the body of a method. We deal with searches inside content in an artifact-independent fashion, with a construct called a *methoid*. A *methoid characterization* is a pattern that can be used to find material within an entity, such as "all gets of variable *v*." Each piece of matching material found is called a *methoid occurrence*, which may be materialized as an entity itself. The pattern matching is artifact-specific, and must be performed by the artifact plugins. Each artifact plugin registers the methoid characterizations it can deal with, so that the central CME support can use methoids in a uniform manner.

### 2.2. Concern Model

Although the CME support for concern modeling is the topic of a separate paper [6], this section gives a brief outline of the concepts that are needed to understand queries involving concern models.

A *concern* is a first-class software element (entity) that represents a particular concept and that brings together other entities that are related to that concept. A concern is not itself a container, in the sense of a package that "owns" the entities, preserving their existence. It is more akin to an indexing structure that brings together entities that have their own independent existence. Entities are known to belong to a concern because they are listed explicitly or because they satisfy the expression of some query. Concerns can be nested, and the same entity or concern can be in multiple concerns. It is clear that certain more-specialized constructs in various AOSD approaches, like the pointcuts of AspectJ, can also be seen as concerns, defined by queries.

A *concern space* models the entities making up a body of software. It provides flexible means of modeling *concerns* and which entities pertain to them. The concern space also models *relationships* among entities, such as reference, inheritance, dependency and traceability. Artifact plugins, separate analyzers and other tools can populate the concern model with relationships they know about. Relationships, like other kinds of entities, are subject to classification (Section 2.1.2).

Some concerns can be designated as *contexts*, and relationships can be scoped by them. This allows for multiple views of software and context-dependent manipulation.

## 3. Query Concepts

Two key goals of the CME's query support are to enable queries to become first-class elements of the definition of aspect-oriented software, and to serve as a platform for different AOSD artifacts, methodologies, and activities. To accomplish this, it must both provide a powerful set of generally-applicable and extensible query concepts, and accommodate specializations and new query capabilities readily. It should do so in such a way that the new capabilities are integrated seamlessly with existing concepts and artifact types. The query language concepts apply to any kind of artifact. To enable special-purpose support for particular artifacts, relationships, or tasks, a number of language constructs are tied to the extension and plugin points provided by the search engine for use by artifact plugins.

CME's query concepts can be grouped roughly into four categories: *what* can be sought, *where* to look, and *how* to express the query and the desired result. This section's subsections address each of these key issues.

They also address how the language is extended for multiple types of artifacts, paradigms, tasks, and processes, exploring tradeoffs that AOSD tool and technology providers must address when supporting queries across different types of artifacts.

The performance of query evaluation is always an important issue, and addressing it adequately requires accommodation even at the conceptual level. Therefore, a number of query language concepts specifically enable a wide variety of optimization strategies. Optimization is discussed in more detail in Section 5.

### 3.1. What can be Sought: Selectables

CME queries search for *selectables*, which are elements or points of interest in software. Different paradigms and formalisms support different kinds of selectables, but some common examples are classes, methods, and fields in Java, *joinpoints* (such as sets and gets of field values) in AspectJ, task and dependency definitions in Ant, tags and attributes in XML, and use cases, types, attributes, constraints, and associations in UML diagrams. Artifact plugins must identify and classify (see Section 2.1.2) the selectables relevant to their artifact types. Queries can then be formed over them, and existing types of queries can immediately be applied to them.

Selectables can be *characterized* (or referred to) in a number of ways. Two very common ones are by name (e.g., "everything named foo*) and by classification e.g., "all methods"), and these are often combined (e.g., "all methods named foo*"). Other important characterizations include by property (e.g., all entities with a given attribute value or member), by structure (e.g., all methods in the control flow of a given method), and by relationship (e.g., all subtypes of a given type, or all entities that refer to a given entity). These kinds of characterizations are supported in CME's query facility, and the facility is open to others.

The wide variety of artifacts in the software development process, and of the kinds of selectables within them, might seem to impose significant conceptual overhead on developers. Fortunately, as noted in Section 2.1.2, many different, artifact-specific kinds of selectables are classified into common categories, which represent concepts developers use every day. The CME query facility specifically supports multiple classification of selectables, and queries involving both classifications and specific kinds of selectables. This reduces the complexity for end users, and it also provides for an extremely powerful, extensible mechanism for supporting queries involving multiple artifact types.

Different tools' use of the query facility can limit the kinds of selectables to be sought to a subset of those introduced by the artifact plugins. For example, the selectables in AspectJ's use are limited to the join points that are used for purposes of weaving. This limitation is appropriate in the context of AspectJ weaving, though other selectables might be of interest in other contexts. Clearly, the more a tool constrains the kinds of selectables it addresses, the more limited that tool will be with respect to supporting different AOSD usage scenarios across the software lifecycle.

### 3.2. Where to Look: Query Contexts

*Query contexts* define the universe or scope over which a given query operates. The results of a given query are bounded by the definition of the context—results are either elements of the context or are derivable algorithmically from elements of the context. Query contexts are important for a number of reasons, not the least of which is as a means of reduction of complexity for developers. A developer can iteratively reduce the set of entities s/he sees by applying the same query to more and more constrained query contexts, and consequently, s/he ends up "drilling down" into a smaller, less complex set of entities. Query contexts include:

- An *input collection*, which contains the set of elements to be searched, which may be the universe.
- A *universe* which is the largest set of elements that may be examined during any search. For example, consider a query that computes the set of subtypes of a given set of types, *t*. In this case, the input collection is *t*, and the set of potential subtypes that may be considered would be included within the universe, perhaps along with other elements.
- Any *auxiliary information* for use when computing the query. For example, this may include previously-computed structures, such as control-flow graphs or indices, or collections other than the input collection, that may be considered when evaluating a query. Auxiliary information is self-describing, to facilitate its use.
- *Variables* that can be used and set in the query.

### 3.3. Queries

Queries are specified using some notation, and the CME specifically makes provision for multiple query notations to accommodate different activities and domains. This section describes the set of generic query concepts defined by CME. We illustrate some of these concepts with examples from the Panther query lan-

guage, a query notation we have developed to support AOSD in the CME. To lower the entry barrier for the many people who are familiar with AspectJ, Panther's syntax is intended to embed some of the AspectJ queries (called *pointcuts* in AspectJ). But Panther also incorporates a richer set of queries, required to support AOSD across the lifecycle and to permit the use of queries in defining artifacts.

The fundamental concept underlying the realization of queries is the *query operator*, which is simply a function that takes zero or more parameters and return values or collections of values. Query operators can be arbitrarily nested (i.e., the result of one is passed as a parameter to another) and connected (using set operators, described subsequently). The following subsections describe different kinds of operators.

We note that, although "wild cards" are not a mandatory part of query notations, they are very common, especially in AOSD languages. At present, Panther supports the AspectJ wild cards: "*", which matches any sequence of identifier characters (or, when used alone, any complete name) and "..", which matches any number of name or signature elements.

3.3.1. **Predicates.** Predicates produce a boolean value indicating the truth or falsity of a proposition about an object (which may be an element in the input collection or query context, or any other object). Predicates may examine an element's name, signature, modifiers, classifiers, or other attributes. Two common predicates in PUMA are *isEqual* and *isNull*.

Predicates may also contain wild cards to permit parts of the query to be unspecified and the values that would be required of them to make the predicate true become part of the query result. For example, the predicate "* *register(..)*" is true of methods whose name is "register," irrespective of their return type or parameters, and the predicate "*call(* register(..))*" is true of calls of such methods.

3.3.2. **Selectors.** Selectors produce collections of selectables that are derived from an input collection. The selector is governed by a predicate that must hold for the results, and the selector examines all candidates, selecting the ones for which the predicate is true. Selectors may implicitly select from the input collection in the input context. For example, the selector "* *register(..)*" returns all methods in the input collection that return any type, take any parameters, and whose name is "register". Different kinds of selectors include:

**Declaration selectors**, like *decl* and *relationship*, are used to choose selectables that are declarations of entities, of which relationships are a special case. For example, *decl(type C*)* selects the declarations of all entities whose name starts with C and that classify as *types*. Relationships are selected based on the relationship's name and signature (a description of the relationship's endpoints). For example, *relationship implements(*, interface I)* selects all "implements" relationships between any entity and any interface named I. Wild cards can be used—for example, *relationship implements(..)* selects all "implements" relationships, regardless of the endpoints, and *relationship *(..)* selects all relationships.

**Methoid selectors** choose selectables that are methoid occurrences. In Panther, these include the AspectJ pointcut designators that do not require runtime tests: *get* and *set* (read and write of fields, respectively), *execution* (the execution of any entity that classifies as a method), *adviceexecution* (the execution of any entity that classifies as advice), *call* (a call to an operation), *staticinitialization* (the execution of a static initializer), and *handler* (the execution of an exception handler). Panther will include *throw* (the site where an exception is thrown) and *label* (a named point) soon.

**Containment selectors.** The *in* selector returns selectables that occur within a given entity. For example, *in(class C)* returns all selectables (methoid occurrences or declarations) that occur within class *C*. An important consequence of the openness of selectables in the CME is that the kinds of selectables returned depend on the artifact plugins in use and what they register. This is true of many other queries also, and supports the seamless introduction of new kinds of artifacts. However, in some contexts, such as when working with AspectJ pointcuts expecting AspectJ semantics, this is not appropriate. The user can restrict the kinds of selectables sought using classifier-based queries, or can use more specific selectors as shortcuts. For example, Panther also supports the AspectJ pointcut designators *within* and *withincode*, picking those selectables that occur within a given type or method (respectively), but only those that correspond to AspectJ join points. Thus, *within(class C)* returns any *get*, *set*, *execution*, *call*, *staticinitialization*, and *handler* methoid occurrences in class C, in accordance with AspectJ's semantics.

Observe that selectors can represent shortcuts for commonly used compound queries involving other operators. For example, the *containingType* and *in* selectors are shorthands for queries involving the "contains" relationship. The definition of such shortcuts is left as a design activity for query language designers.

3.3.3. **Navigators** produce an entity found by following a "navigation direction" from a given object to another object. The most ubiquitous example of a naviga-

tor is an attribute accessor, which takes an object and attribute name and returns the attribute value. At present, Panther navigators include *elements* (returns the members of one or more input aggregates or collections), *parents*, and *containingType* (for collections of declarations that are members of types, returns the types that defines them). Relationship navigators include *endpointsof*, *sourceof*, *targetof*, and *expand*. The latter applies to summary relationships between coarse-grained entities, such as concerns or classes, and expands them into relationships between their elements.

3.3.4. **Set operators** including *union*, *intersection*, and *difference*, and **logical operators**, including *and*, *or*, and *not*, are supported in Panther.

3.3.5. **Transitive closures** of relationships allow the results of a relationship query to be followed without knowing how many applications of the query would be needed for all results to be found. For example, the Panther "+" query (analogous to AspectJ) traverses the "extends" and "implements" relationships transitively to find all subtypes of its argument type.

3.3.6. **Variables and unification**: In addition to operating on the collection from which they select elements, selectors are parameterized by other values, such as the parts of names, modifiers, or relationships governing the selection. As an alternative to specifying them literally, these values can be specified using *variables*, whose values are set and stored in the query context, or using *unification variables*. Unification variables are variables whose value is instantiated each time a different value for it can be found that makes a predicate true. So, for example, a planned Panther query such as "*call(* register(<x>,<x>))*"[3] identifies all calls to any method that returns any type, whose name is "register," and which takes two parameters, *both with the same type*. For each such call found, the unification variable, *x*, is bound to the parameter type.

3.3.7. **Panther example**. Suppose a Java system includes an open capability for displaying data structures. Supporting a particular data structure involves providing a suitable implementation of the *Printer* interface that must register itself by calling a static *register* method provided by the *Registrar* class. The query

```
<printers =
   sourceof(relationship
     implements((class || aspect) *, interface Printer))>
```

finds all classes or aspects that implement Printer and records them in the *printers* variable. The query

```
<regcalls = in(<printers>)  &&
            call(static * Registrar.register(..))>
```

assigns all registration calls to *regcalls*. A common error in these situations is forgetting the registration:

```
<printers> - containingType(<regcalls>)
```

finds any printers that do not contain registration calls.

### 3.4. Query Results

Executing a query can both produce a result and cause a change to the query context. The query context may be changed to record or provide access to useful computed intermediate results or other information, such as analysis results (e.g., control-flow graphs, dependence graphs) or indices, or to set new values for variables. The query result itself has a structure that is potentially complex, reflecting the effect of unification variables. Rather than being simply sets of values, the results of a query need to be viewed as a multiplicity of sets of values. Each set of values in this multiplicity has associated with it the *bindings* indicating the value of each unification variable which causes the set of values to be a valid result.

The CME query infrastructure permits the person or software that issued a query to specify the structure of both the output collection and its elements. For example, the desired result may be a collection of tuples <*x*, *y*> where *x* is a concern named C*, and *y* is a concern named C*Extension (where the value of "*" is the same in all elements of any tuple). Such tuples are used, for example, as *correspondences*, specifying entities to be composed with one another [7]. Particularly in an AOSD context, where queries are used both to identify concerns in existing software artifacts *and* to define parts or all of software artifacts, this ability to generate new values as results, rather than simply selecting a subset of input values, is critical.

## 4. Query User Interface

The CME includes tools for developers provided as Eclipse plugins. The query capability is supported by the "CME Search" view for entering queries, and the "CME Search Results" and "Visualizer" views for displaying the results. They support queries against a concern model, which is viewed and manipulated by means of the "Concern Explorer" view [6]. This section gives an outline of the search views, illustrated in Figure 1.

The search view allows textual entry of queries and saving of queries for later use. It also includes a query wizard that guides users through construction of differ-

[3] PUMA provides basic support for unification, and Panther will expose it in future.

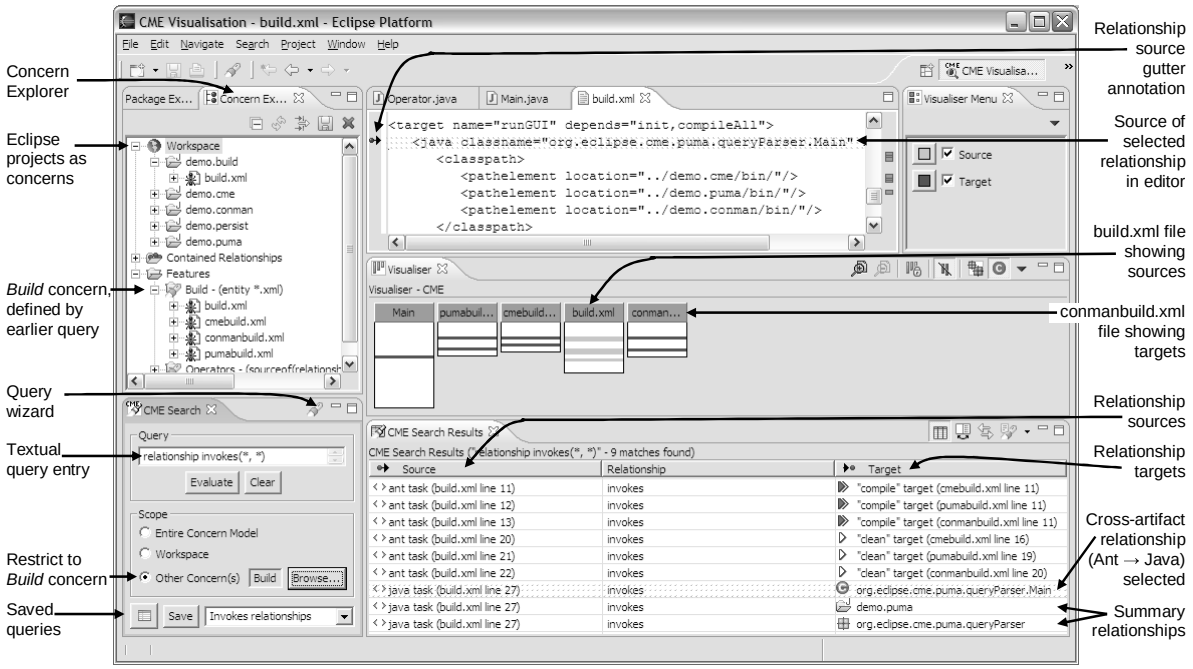


**Figure 1 – CME Visualization Perspective in Eclipse**

ent sorts of queries, showing them the available options at all stages. A third way of issuing queries is to select an element in the Concern Explorer, right click, and select "search for." This will list some useful queries applicable in the context of the selected element.

The search results view displays the query results in tabular form. If the results are relationships, multiple columns are used to display the relationship names and endpoints. Users can sort on any column, and can double-click any element to show it in an editor. Users can also select one or more relationship results and use the context-sensitive menu to run "expand" query on them.

The visualizer was generalized from the one developed for AJDT [1] (with a venerable history, back to SeeSOFT [5]). In the CME, it is configured to display query results, showing how they are distributed across groups (such as packages) or members of those groups (such as files or classes). For relationships, all endpoints are shown, with sources and targets in different colors in the case of directed relationships.

## 5. Search Engine Architecture

As noted earlier, the CME architecture is designed to support multiple query languages. It does so though a central query engine, called *PUMA*, which supports optimization and execution of *query graphs*[4]. A query in a CME query language must be compiled to a query graph. PUMA and the Panther compiler are outlined in this section; details are beyond the scope of this paper.

### 5.1. PUMA

PUMA is a general query engine that can be customized to search arbitrary data structures. This is necessary to support the CME's ability to handle arbitrary artifacts. It accomplishes this by allowing for registration of two kinds of plugins, operators and attribute accessors, for use in query graphs. An *attribute accessor* is associated with a particular type of data, and is capable of determining the value of a particular attribute of that data, such as the name or modifiers of a UML Classifier. An *operator* is capable of performing a particular query operation on a data structure (or collection of data structures), such as finding all calls within a method. The CME provides a set of standard operators and attribute accessors, to deal with collections and material in concern models. Since arbitrary artifacts can be represented in the concern model, these are often sufficient. Artifact plugins can, however, provide additional operators and attribute accessors suitable for querying their artifacts.

PUMA provides flexible opportunities for optimization, based on query-graph transformation, leveraging of available auxiliary information, such as indices or domain knowledge acquired from a particular usage context, and other techniques. For example, queries

[4] Directed-acyclic graphs can describe expressions with common subexpressions.

involving multiple uses of a unification variable are converted to *fronts*, each responsible for a single use, and each consisting of multiple *stages*. Execution of the stages across multiple fronts is then ordered dynamically based on collection sizes, so as to bind the variable in the smallest collection and then use the bindings in the larger ones. Extensive exploitation of the optimization capabilities remains future work.

### 5.2. The Panther Query Compiler

The Panther compiler parses the query language into an abstract syntax tree (AST), then uses pattern-matching on the AST to select plugins to process it. Each plugin handles a root node and, optionally, descendants, down to but excluding specified *fringe* nodes; other plugins are found to process the fringe nodes and their descendants. A plugin can transform the AST and/or generate a portion of the PUMA query graph needed for query execution.

Panther plugins are chosen based on the form of the AST regions they apply to, the types of results they expect for fringe nodes, and the information they expect to be available at execution time, such the type of collection being searched and what indices are available. Plugins can thus be written to handle, and optimize, a variety of specific situations. Such plugins are registered with Panther, describing their requirements. Panther finds all applicable plugins in a particular context, asks each for its cost in that context, and selects the lowest-cost plugin. This flexible plugin architecture makes the panther compiler a suitable basis for implementing other query languages also.

## 6. Experience

Juri Memmert reported to us some experience with using the Panther query language in his consulting practice [11]. He needed to identify the characteristics of a specific component within a code base of about 1000 classes to prepare it for extension. The component used the *command* pattern. It was important to identify the callers of the command's *execute* method as a new concern, and to characterize the calls with respect to the other concerns they belonged to. These calls cut across 4 different applications and more than 12 previously-identified concerns.

Memmert began defining the concern with a query to find all calls of the execute method, then added qualifications based on the other concerns they belonged to. This revealed some calls from within the component implementing the commands themselves. This was a violation of the intended architecture, which assumes that all commands are atomic in nature. Memmert then formulated desired architectural constraints as further queries. These revealed a number of violations, leading to significant reengineering. After that, queries correctly characterized the new concern.

Memmert reported being hampered by lack of documentation of the query language, but, nonetheless, the ability to work with the concern as a first-class entity made it easier to accomplish the original task, and the reengineering resulted in valuable improvements to the system. Commands are now guaranteed to be atomic, enhancing the predictability of the system and removing a long-standing bug. The queries also revealed that the commands used fell into two categories: *core commands* common to all applications and *extended commands* used in only some applications. It became obvious that the concerns of the core commands always appeared together, while the extended commands and their concerns were not similarly distributed. So, instead of treating the use of commands as one concern, they were better modeled as one concern for all core commands and one concern per extended command. This insight led to a plug-in architecture that furthers the extensibility of the component.

During the course of CME evolution to date. PUMA has been used to implement three query languages: a low-level language that allows PUMA operators to be called directly, used for testing and debugging; a subset of Panther that supports name matching with wildcards and unification variables, currently used in the Concern Composition Component [7]; and Panther itself, exposed via Eclipse views, as noted above, and used to implement *intensional concerns* in concern models [6]. The query support has thus been used for three key AOSD activities: concern identification, modeling and composition. It has also proven flexible enough to address multiple types of artifacts (Java, AspectJ and Ant to date). The Ant artifact plugin was developed after Panther was mostly implemented, and it immediately enabled queries against Ant artifacts.

A forerunner of the Panther query language, implemented as an API on the concern model, was used in a significant refactoring project [2]. The task was to separate support for Enterprise JavaBeans™ (EJBs), from the rest of a large application server. The server consisted of some 15,000 Java classes, separated into over 250 components. The EJB support and its use cut broadly across these components.

A simple concern model was constructed, consisting of some 80 top-level concerns, into which the components were grouped, and a new EJB support concern to contain all the EJB support. Initially, the EJB support concern was set to contain those components wholly

devoted to its implementation. Then an iterative process was followed to refine this concern. Each iteration began with determining all links from the other concerns to the EJB support concern, using our queries. Then some of the links were selected for removal during the current iteration, and the removal was accomplished by a variety of refactorings. At peak, the number of links reported was about 1000. Despite the size of the system, each report of all links was generated in under 5 minutes.

## 7. Related Work

The problem of being able to ask questions about interrelated software artifacts is not a new one, and it has been explored at length in the software engineering environments literature over the past two decades. Numerous approaches have been employed for representing and querying software artifacts and their interrelationships, including relational databases, object-oriented databases and database programming languages, rule-based languages, predicate calculus, and functional languages; more recent efforts, such as XLinkit [12] and InfiniTe [2], have used combinations of XML, XLink, and/or OCL. Our work specifically takes into account some of the results and experiences produced by these efforts. While all of these efforts provided a subset of the capabilities of the CME's query facility, we are not familiar with any that included all of the capabilities required to support queries in the AOSD context, in large part because of the difference in purpose and software model.

Several efforts in the AOSD area have produced tools that support flexible queries over particular types of artifacts. One of these is JQuery [9], which supports queries over Java code. It includes a customizable GUI, which enables users to define query-based views of a working set of Java code, a capability that is not yet present in CME. JQuery is implemented on a Prolog-like logic query language, which gives it considerable expressive power, including unification, but its queries are limited to a single type of artifact. Another, related effort [8] has used XML to represent Java class files, and implemented a subset of AspectJ's pointcut queries using XQuery [16] (a functional programming language with specialized support for querying XML structures). Unlike JQuery, this approach does not support unification, but it is not clear that either approach is more expressive than the other. The authors argue for the extensibility of the set of queries based on the computational completeness of XQuery, an argument that can also be made for JQuery. Neither of these approaches have addressed the non-trivial issues involved in supporting queries for a wide range of purposes and across multiple types of highly interrelated artifacts, as this work has, and both approaches are vulnerable to representation sensitivity. FEAT [14] supports the identification and modeling of concerns in Java software. It supports relational queries and permits the definition of concerns based on queries, which is an important use case for us as well. FEAT's query language is deliberately not as expressive as ours—e.g., it excludes transitivity for simplicity [15], but we include it because it is essential for a variety of AOSD tools—and it does not address multiple artifacts.

OCL specifies a notation for constraints over UML entities. As a specification, it does not identify or address issues required to realize the specification (such as performance and extensibility), as we have. Nonetheless, OCL is interesting in that it uses many of the same query concepts provided by the CME. OCL cannot be used to define artifacts, and it is restricted to UML artifacts, making it inherently less broadly applicable than the CME's query facility, but we considered OCL as a possible end-user query language, as it is a standardized, relevant notation, and we believe the CME's query framework would support it well.

We observe that OMG has recognized the need for capabilities similar to those in the CME's query facility, and, as part of its Model-Driven Architecture effort (to which has strong relevance), issued a request for proposals for QVT, a query, view, and transformation mechanism that will work on MOF,making it applicable to a wider range of artifacts. A QVT specification does not yet exist for comparison.

## 8. Conclusions and Future Work

The aspect-oriented paradigm fundamentally changes the way software is defined, modularized, viewed, and interconnected by expanding the ways in which entities can be modularized and the interrelationships among modules can be specified. The successful definition and use of aspects in software mandates inclusion of queries as a first-class notion. Queries are used both to enable developers to locate areas of interest in software that are not modularized, and to form part of the definitions of concerns and their interrelationships. Moreover, queries must apply across the software lifecycle: both to the interrelated artifacts and different activities in a software process, including concern identification, extraction, and integration.

To address these significant requirements and constraints, the CME provides pervasive query support, realized by an open, extensible, artifact-neutral query component, for use by all components and tools in the

environment. The query infrastructure provides, as part of its fundamental concepts, a set of key open points. These include selectables, modifiers, classifiers, relationships, and query operators. It also provides core support for wide range of optimization strategies--a critical issue that cannot be addressed as an "add-on."

Some initial validation for the claims of broad applicability and utility exist. We have used the CME's query facility in the implementation of components that support three different but ubiquitous activities in an AOSD process--concern identification (the Panther language), (re)modularization and modeling, and composition. We have also used it successfully to express queries over three types of artifacts (Java, AspectJ, and Ant) and their interrelationships. Most of the open points have been used, and a number of important design and implementation issues that are faced by AOSD technology providers have been identified. An independent consultant has also applied the Panther query language successfully to a project.

The use of a CME-like query capability, while mandatory in AOSD, is useful in any software engineering process, to help developers locate concerns of interest. We note that the earliest version of the query facility was used with a strictly Java software base, and the developers involved reported significant benefit from using queries to identify latent, unmodularized concerns and to explore interrelationships among those concerns in their software. For this reason, the CME's query capability represents an important path for incremental adoption of AOSD.

Much work remains for future, most notably in the area of fully exploiting some of the open points. We have not yet used the optimization infrastructure to its full capacity. Exploring the issue of incremental query (re)evaluation will likely be critical to scalability in an IDE, such as Eclipse. We would also like to exploit existing open points to integrate relevant existing capabilities, such as analysis (e.g., as provided by FEAT) and data mining, to enable a wider range of semantic queries based on such information.

## 9. Acknowledgements

Matt Chapman, Andy Clement, Helen Hawkins and Sian January developed the CME Eclipse views and provided valuable input and feedback on the CME query facility. Andy and Matt developed CME support for AspectJ and Ant artifacts. Andy and Adrian Colyer provided recommendations regarding the query language and insights from their early use of the CME. We are grateful to Juri Memmert for being an adventurous and understanding user, for his insights into what is needed, and for sharing his experience. Stan Sutton provided helpful input during the design of Panther and on an earlier version of the paper.

## 10. References

[1] AJDT: AspectJ Development Tools Eclipse Technology Project. http://www.eclipse.org/ajdt.

[2] K.M. Anderson, S.A. Sherba and W.V. Lepthien. "Towards Large-Scale Information Integration." In *Proc. 24th International Conference on Software Engineering*. May 2002.

[3] A. Colyer and A. Clement. "Large-scale AOSD for Middleware." In *Proc. 3rd International Conference on Aspect-Oriented Software Development*, March 2004.

[4] Concern Manipulation Environment Eclipse Technology Project. http://www.eclipse.org/cme/.

[5] S.G. Eick, J.L. Steffen, E.E. Sumner, Jr. "*SeeSoft - A Tool for Visualizing Line Oriented Software Statistics*", IEEE Transactions on Software Engineering 18(11), pp. 957—968, November 1992.

[6] W. Harrison, H. Ossher, S.M. Sutton, Jr. and P. Tarr. "Concern Modeling in the Concern Manipulation Environment." Submitted for publication.

[7] W. Harrison, H. Ossher and P. Tarr. "Concepts for Describing Composition of Software Artifacts." Submitted for publication.

[8] M. Eichberg, M. Mezini, and K. Ostermann. "Pointcuts as Functional Queries." In Proc. 2nd Asian Symposium on Programming Languages and Systems, 2004.

[9] D. Janzen and K. De Volder. "Navigating and Querying Code Without Getting Lost." In *Proc. 2nd International Conference on Aspect-Oriented Software Development*, March 2003.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold. "An Overview of AspectJ." In *Proc. 15th European Conference on Object-Oriented Programming*, June 2001.

[11] J. Memmert, JPM Design. Personal communication.

[12] C. Nentwich, L. Capra, W. Emmerich and A. Finkelstein (2002). *xlinkit: A Consistency Checking and Smart Link Generation Service.* ACM Transactions on Internet Technology, 2(2):151-185.

[13] H. Ossher and P. Tarr. "Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software." *CACM* 44(10): 43–50, October 2001.

[14] M.P. Robillard and G.C. Murphy. "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies." In *Proc. 24th International Conference of on Software Engineering* , May 2002.

[15] M.P. Robillard and G.C. Murphy. "Capturing Concern Descriptions During Program Navigation." OOPSLA 2002 Workshop on Tool Support for Aspect Oriented Software Development, October 2002.

[16] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery.