

Toward Query-based Constraints

(Position Paper)

Elissa Newman and William L. Scherlis
School of Computer Science
Carnegie Mellon University
{elissa | scherlis}@cs.cmu.edu

1. Introduction

The fragility of syntactic pointcuts leads to evolvability problems in current AOSD languages because they make the aspects dependent on names and structure in the code base. Composition is difficult because meanings are not explicit, and so tools cannot be used to assist in assessing the extent of interaction among aspects. We propose a new concept that we call query-based constraints to help address these issues of evolvability, composability, and understandability that afflict the current popular aspect languages. Our approach enables the programmer to express low-level design models, to assure consistency between the models and his code, to encapsulate and define concerns, and to assure critical properties of his program as it evolves. We use an iterative code querying process to elicit and express the understanding required to develop low-level design models.

The use of constraints trades programmer effort in expressing design intent for evolvability and composability. The principle of early gratification (elaborated below) states the process of expressing the latent design information in code via formal program annotations should result in concrete near term benefit to the programmer in the form of improved consistency with design intent (analogous to type checking) and increased program understanding (for example, to provide assurance regarding critical properties). Our approach is designed with incremental benefits and adoptability in mind.

2. Properties of AOSD Systems

Composability and evolvability of concerns are important challenges for current AOSD technologies. Past AOSD-related workshops at ECOOP 2000 [9] and ECOOP 2001 [10] identified several mechanisms for increasing the reusability and evolvability of concerns, including semantic joinpoints and concern specifications. The idea of semantic joinpoints, or more aptly semantically specified pointcuts, is to specify pointcuts with minimal coupling to the code base. Coupling is high, for example, when specific class, method, and field names are used, or when strong assumptions are made regarding code structure such as hierarchy and visibility, or when wildcarding is used, say, on the names of class members (i.e., methods and fields) to specify sets. The vision is that coupling is reduced when pointcuts are specified in terms of their *characteristics* rather than the associated names in the code base. In addition, more semantically focused pointcut specifications could help in addressing the difficulty of composition of concerns, because more information regarding intent and meaning is available to guide composition strategies and to enable static analyses to facilitate detection of interactions. Several approaches for accomplishing the decoupling of pointcut specifications from the code base are suggested in [9], including specifying pointcuts using semantic program information such as that culled by static analyses and the use of semantic categories of joinpoints.

The authors of Section 3.3 of [10] argue that an important feature in the reusability of AOSD systems is the ability for concern providers (in the case of reusable concerns) to specify properties of their concerns, such as constraints. The authors stress the importance of having both intra-concern and inter-concern constraints.

Various forms of the concern composition problem, also known as the aspect interaction problem, inhibit composability. Composition interaction can be found in composition-based strategies [17], such as with Hyper/J [5,16]. So-called semantic mismatches occur when two or more concerns do not produce a behaviorally correct program when composed [10]. These may be intentional, or intended by the developer (e.g., if only one of a set of concerns may be composed with a system at any given time), or unintentional. An approach to define annotations to relay information such as mutual exclusion can be found in [4].

3. The Well-formed Code Base

Research into improving the current AOSD techniques in this respect by enabling semantically specified pointcuts and semantically correct concern composition is ongoing. We propose here an alternative, more ambitious approach. We conjecture that the promise implied by the use of the word "semantic" in the phrase "semantic pointcuts" can, in fact, be fulfilled more directly if the AOSD system has available to it certain *semantic* information. Historically, there has been resistance to movement in the direction of using deep information about the meanings of programs, and for good reason: There are huge practical difficulties in capturing, expressing, and reasoning about specifications and assertions regarding program *functionality*.

But there are many categories of semantic information, and the extent of this challenge varies considerably among these categories. We hypothesize, in particular, that there can be great value in expressing and exploiting design intent regarding "mechanical" properties of programs—models and assertions that relate to significant non-local properties. We have explored properties including effects [2], unique references, aliasing [2], use of locks in

multi-threaded programs [3], policy restrictions on subclassing [1], concurrency policy [3], and relationships among code segments, threads, and shared data [15].

We consider a well-formed code base to include three sets of elements. The first is a collection of source code, which may include multiple variants that are behaviorally equivalent (according to some appropriate notion of external visible behavioral equivalence). The second is a set of low-level design models that address semantic program properties significant to the "mechanical" attributes of code listed above. We focus on these attributes because they contribute most significantly to establishing preconditions for the creation of new code variants, for the successful execution of model-based queries, and for transformations (see Conclusion). The models can be diagrammatic and linked with code via anchors and annotations. Our approach includes composable static analyses that use the annotations as cutpoints. The third set of elements is a linkage of the code base with the models, which provides an assurance of their overall consistency. This implies, in particular, that the models have precise semantics. It is important to note that, in the case of the mechanical properties mentioned above, this is not a major challenge. It is more challenging, however, to construct the evidence of consistency in a scalable way, though there is a body of work related to software engineering program analysis and assurance that strongly suggests feasibility. The assembly of code variants, models, and evidence of consistency in a well-formed code base can provide a powerful basis for diverse views into a system [7]. Transformation to support the evolution of a well-formed code base thus act on the diversity of entities. If a programmer acts manually on an element of a well-formed code base, tools can be used to restore an overall consistency or, if expedient, to drop particular models and variants that become obsolete or unrecoverable.

The adoptability of our approach is premised on the *principle of early gratification*: increments of annotation effort should yield increments of benefits. That is, the programmer's efforts in recording design knowledge, linking it with code, and identifying concerns are immediately rewarded via improved analysis results, greater program understanding, etc. The more effort that is exerted, the more benefit he shall receive. We posit that the resistance to capturing more design knowledge during development or during program understanding and reverse engineering activities is due to the lack of immediate or clear long-term benefits for the expenditure of the *expression cost*—the conceptual work to express and make explicit the design knowledge already in a programmer's head. The programmer takes a series of small steps involving querying code and expressing design knowledge as annotations. Outcomes of this stepwise action include (1) identification and localization of concerns via concern manifestations, (2) creation of constraints on the code base in order to express low-level design models, and (3) assurance that intentions expressed by the constraints continue to be met through the evolution of the code base. Our hypothesis is that, specifically with respect to the low-level mechanical design knowledge described above, it is possible to comply with this principle of early gratification.

4. Query-based Constraints

Constraints may be based on the low-level design models in the code base, or they may be related to concern composition. A model-based constraint is a restriction on code expressed with respect to one of the low-level design models that limits the set of options available to a programmer. In other words, a constraint restricts the possible variants of source code to those that can fulfill its properties. Here are some examples (more are in Section 5):

- This class may have only these two subclasses
- Methods wishing to call this particular method must hold this lock before access
- The client must assure data integrity before calling this method (with an explicit check or through an assertion)

We consider a *model-based constraint* to express a design commitment in the language of a particular model. By providing this capability, design commitments not directly expressible in code (e.g., because they have a fundamentally non-local character) can nonetheless be made explicit and be supported by a tool. This idea may appear to conflict with the principle stated earlier regarding adoptability. Our hypothesis is that the low-level "mechanical" models we propose: (1) can capture a wide range of design elements directly useful to programmers, and (2) this can be done in a way that is compatible with their present state of knowledge and practice.

Concern-composition constraints express inter-concern restrictions, such as intended mutual exclusion. These constraints can then be used to detect intentional forms of semantic mismatch, therefore solving this portion of the concern interaction problem.

When multiple constraints are applied to code simultaneously, they implicitly define a set of code variants that respect all of the constraints of all the models. It is possible that the set will be empty; in which case the constraints are unsatisfiable (or not feasibly satisfied) and one or more must be removed or altered. Checking the feasibility of simultaneous application of constraints to the code base may allow detection and resolution of certain concern interaction issues. Constraints may also be used to specify pre- and post-conditions for transformations (see Conclusion).

5. Querying Code

In our approach, we use code querying as a form of interaction with the code base that includes low-level models. This supports program understanding and the recovery of lost design information. More significantly for us,

it is a basis for the identification of concerns and the codification of those concerns in entities called *concern manifestations*.

Our approach is to incrementally query code and visualize the querying process. Query trails (history of queries) are recorded, and backtracking in the query process is allowed. Individual queries can be named and can be marked for inclusion in a concern manifestation. Queries are used to define pointcuts for the application of transformations that compose concerns into a system (see Conclusion). Queries can also be used to express both model-based and concern-composition constraints. A constraint can be expressed as a requirement on the outcome of a particular query.

Code querying is an iterative process that begins with a new *query* or a refinement of a previous query. A *response* to the query is provided in the form of a code view [7,13,14], elided code, diagram, visualization, or yes/no answer. This may prompt the programmer to produce further queries, or he may choose to codify his new understanding as a record in the form of formal program annotations, informal remarks, diagrams, the query trail itself, or a concern manifestation.

Short examples of expressible code queries include:

- What lock protects the **value** variable?
- Does the **getResult()** method acquire any locks?
- What is the lock order among this set of locks?
- What locks may be held when method **y()** is called from this code base?
- Is it possible that the variable **x** is aliased? What are the possibilities?
- What is the set of known subclasses of this class?

A more in-depth query example depicting concern identification is depicted in section 6. Here are some examples of how the non-local properties captured by these queries can be expressed as model-based constraints:

- The lock named **ValueLock** referenced by **this** should be used to protect the **value** variable
- The **getResult()** method acquires the lock named **MyLock**
- The class **List** should have exactly two subclasses: **Empty** and **Cell**

In each of these cases, the constraint can be expressed as a requirement on the outcome of a particular query. This can then be directly treated as a kind of low-level semantic model because of the semantic nature of the queries. Thus, as the code base evolves, the model-based constraint expressed by the query can be monitored for compliance (a process analogous to regression testing).

6. Concern Manifestations

In our approach, concern manifestations are used for two main purposes: 1) as a form of record in the code querying process and 2) as a means of localizing a constraint. Concern manifestations themselves can be made comprehensive, although they are not required to be such, out of respect for the principle of early gratification.

Our concern manifestations are based on code queries and are similar to those in FEAT [11]. FEAT is a tool used by the programmer to iteratively examine an existing code base via predefined queries based on static analyses and to identify relevant parts of code. The result of this process is a Concern Graph that contains the program pieces that make up a concern and the basic relationships among those pieces. In FEAT, the program pieces may be packages, classes, methods, or fields. Each program piece is either completely or partially within a concern, so each may be marked as “all-of” or “part-of” the concern, respectively.

Our approach to iterative code querying creates concern manifestations (i.e., query-based definitions of code) similar to the approach described in [12]. The programmer uses a general-purpose language of (semantic and syntactic) queries, with tool support, to identify those portions of programs that apply to a given concern. This query language is currently under development. The resulting *comprehensive* concern manifestations can then be used as a scoping mechanism by the constraints and transformations (see Conclusion).

In an ideal world, all concerns expressed are comprehensive. In the reality of engineering practice, concerns and queries can both be approximate. If there is success in achieving the early gratification principle, then there is incentive to migrate to comprehensive concern manifestations. How can this be achieved on an incremental basis? One mechanism is to provide the programmer with a way to improve the rough results of queries that are assembled to form concerns. For program properties that can be expressed formally (e.g., locking discipline), tools can assist. For more intrinsically informal program properties, such as those related to system features, this goal may be infeasible. Regardless, we have considered how programmers can be provided with ways to take incremental steps.

One way to provide this incrementality is through the use of concern manifestations with several parts: the definition resulting from via the query process, and programmer-specified *additions* and *exclusions* to the definition. The addition and exclusion constructs are similar to those in intentional views [6]. We can also specify portions of a concern as *contained* within the concern, which means that they are referenced only by code elements appearing elsewhere within the concern, and not by code outside the concern. This is analogous to the “all-of” construct in FEAT.

Concern manifestations are stored along with the query trail used to capture them. The query trail can then be referred to as a form of rationale, and can be used to recompute the definition of the concern manifestation if the code base has changed. In the case of comprehensive concern manifestations, which are completely query-based, in

a well-formed code base, the concern manifestations will remain consistent after code changes. This may require the application of a structural transformation (see Conclusion). For other concerns, the programmer-specified sections will need to be revisited by the programmer using the query trail as a guide.

The following is an example query trail used to create a concern manifestation related to copying and aliasing data. In `java.lang.StringBuffer`, there is a private boolean member variable named `shared`. If the `shared` flag is false, then the `value` array containing the characters of the string is unaliased. If the flag is true, then there is a possible alias—the array could also be referenced in a separate (immutable) `String` object. The flag enables deferral of array copy operations otherwise necessary every time a `StringBuffer` is constructed from a `String` object. With `shared`, the copying can be deferred until the moment prior to a mutation operation. The `String` constructor uses a special package-private method `getValue()` to access the `value` array. It uses `setShared()` to denote that it is aliasing the `value` array.¹

1. Create concern **SB_shared**

2. Query [**shared_defs**]: Where is `shared` set in `java.lang.StringBuffer`?

Declaration:

- private boolean shared;

Methods:

+ public StringBuffer(int)
+ private final void copy()
+ private void expandCapacity(int)
+ public synchronized void setLength(int)
+ private synchronized void readObject(java.io.ObjectInputStream)
+ final void setShared()

3. Place **shared_defs** in concern **SB_shared**

4. Query [**shared_uses**]: Where is `shared` used in `java.lang.StringBuffer`?

+ public synchronized void setLength(int)
+ public synchronized void setCharAt(int,char)
... 8 more public methods

5. Place **shared_uses** in concern **SB_shared**

In this process, we are iterating towards comprehensiveness in the concern manifestations that result from queries. In order to form an initial approximation to the intended concern manifestation, we start with explicitly identified **additions**, **exclusions**, and **contained** sections of the concern. The declaration of `shared` belongs in the **contained** section of the **SB_shared** concern because both its definitions (**shared_defs**) and uses (**shared_uses**) are in the concern.

6. Place declaration(**shared**) in concern **SB_shared.contained**

We decide to look at the call sites of the non-public methods in `StringBuffer`, because public methods that define or use the field `shared` will not be in the contained section for this concern because they may be called outside the scope of the `java.lang` package.

7. Query [**nonpublic_calls**]: Where are the non-public methods of `java.lang.StringBuffer` called?

Calls to private final void copy():

+ public synchronized void setLength(int)
+ public synchronized void setCharAt(int,char)
... 8 more public methods

Calls to private void expandCapacity(int):

+ public synchronized void setLength(int)
+ public synchronized StringBuffer replace(int, int, String)
... 10 more public methods

¹ We make use of informal structured natural language in lieu of any specific query syntax. We expect to propose a query syntax in future publications.

Calls to final void setShared():

Calls to final char[] getValue():

In java.lang.String:

```
- public String (StringBuffer buffer)
  { synchronized(buffer) { buffer.setShared(); this.value = buffer.getValue(); ... } }
```

The method `getValue()` is only called when `setShared()` is called. The method `setShared()` is already in the concern because it is in the result set of the query `shared_defs`, and is only called when this particular `String` constructor is called. Therefore, both the `String` constructor and `getValue()` are in the concern.

8. Place `final char[] getValue()` and `public String (StringBuffer buffer)` in `SB_shared.additions`

Because the `String` constructor is a public method, it cannot be contained within the concern. We will now see what methods `setShared()` and `getValue()` call. If they do not make methods calls outside the concern then they are in the `contained` section of the concern.

9. Query []: Show me calls made by `final void setShared():`

None

10. Query []: Show me calls made by `final char[] getValue():`

None

11. Place `final void setShared()` and `final char[] getValue()` in `SB_shared.contained`

If `expandCapacity(int)` is contained within the `SB_shared` concern, then all of the methods that call it will be within the sets `shared_defs` and `shared_uses`.

12. Query []: Show me calls to `expandCapacity` omitting all definitions and uses of `shared`:

```
+ public synchronized void ensureCapacity(int)
+ public synchronized StringBuffer append(String)
... 3 more methods
```

The previous query result determined that `expandCapacity(int)` does not belong in the `contained` portion of the concern. By the same reasoning, if `copy()` is contained within the concern, then all of the methods that call it will be within the sets `shared_defs` and `shared_uses`.

13. Query []: Show me calls to `copy` omitting all definitions and uses of `shared`:

Empty

14. Place `copy` in `SB_shared.contained`

15. Show me the concern manifestation for `SB_shared`:

concern SB_shared:

```
definition: { union (shared_definitions, shared_uses) }
additions: { final char[] java.lang.Stringbuffer.getValue(),
               public String java.lang.String(StringBuffer buffer) }
exclusions: { }
contained: { declaration(shared),
               private final void java.lang.StringBuffer.copy(),
               final void java.lang.StringBuffer.setShared(),
               final char[] java.lang.StringBuffer.getValue() }
```

We now can achieve comprehensiveness for `SB_shared` by encompassing all aliases. (We make some favorable assumptions regarding the combination of alias analysis and alias-related program annotation in order to accomplish this.)

Further, more ambitious development could address some of the invariants pertinent to the concern, such as the relationship between the value of `shared` and the aliasing status of the private instance variable `String.value`.

7. Conclusion

We have introduced several ideas related to query-based constraints and their role in aspect technologies, particularly with respect to admitting "semantic" notions into the definition of concerns. The key ideas are (1) the notion of model-based concern manifestations, (2) the use of semantics-based queries to define these concerns, and (3) the coupling of queries with specific concern manifestations to define model-based constraints and concern-composition constraints.

We are also pursuing two related ideas involving transformations. The first idea is to define a set of meaning-preserving transformations (akin to refactoring) that alter code structure (or produce additional variants of code structure) without fundamentally altering externally visible behavior. These structural transformations involve code, models, query trails, and concern manifestations. As models evolve, transformations can be used to shift among variants in order to manage tradeoffs in the space of non-functional requirements. The second idea is to provide transformations that support the evolution of system functionality through composition of concerns while retaining an overall consistency of code and models, including, for example, respect for model-based constraints and relevant concern-composition constraints.

Our approach provides a way to manage programs and associated models in a well-formed code base by exploiting information about critical mechanical properties. Iteration in the code querying process allows developers to recover and express design information as a basis for defining concerns and constraints.

Our approach requires some increments of work by the programmer to identify and express the various low-level design models, but preliminary case studies suggest that it may indeed be possible to respect the principle of early gratification. Most importantly, by using our approach, programmers gain an ability to more easily understand and express critical properties, to provide concrete assurance of critical mechanical properties, to monitor the status of those properties as a system evolves, and to facilitate analyses pertaining to the interactions among multiple concerns.

8. Bibliography

- [1] E.C. Chan, J. T. Boyland, and W.L. Scherlis. *Promises: Limited Specifications for Analysis and Manipulation*. In ICSE'98.
- [2] A. Greenhouse and J. Boyland. *An Object-Oriented Effects System*. In ECOOP'99.
- [3] A. Greenhouse and W.L. Scherlis. *Assuring and Evolving Concurrent Programs: Annotations and Policy*. In ICSE 2002, 453-463, May 2002.
- [4] S. Hanenberg and R. Unland. *Specifying Aspect-Oriented Design Constraints in AspectJ*. In OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development.
- [5] Hyper/J™ Web site: www.research.ibm.com/hyperspace/HyperJ/HyperJ.html
- [6] K. Mens, T. Mens, and M. Wermelinger. *Supporting unanticipated software evolution through intentional software views*. In USE 2002 at ECOOP 2002.
- [7] E. Newman. *Localizing Views for Separation of Concerns*. In ICSE 2001 Workshop on Advanced Separation of Concerns.
- [8] E. Newman, A. Greenhouse, and W. L. Scherlis. *Annotation-Based Diagrams for Shared-Data Concurrency*. In Workshop on Concurrency Issues in UML at UML 2001. Available at: <http://wooddes.intranet.gr/uml2001/Home.htm>
- [9] Proceedings of Workshop on Aspects and Dimensions of Concern: Requirements on, and Challenge Problems For, Advanced Separation of Concerns. Eds. P. Tarr, M. D'Hondt, L. Bergmans, and C.V. Lopes. In ECOOP 2000 Workshop Reader. LNCS 1964, 2000.
- [10] Proceedings of the Workshop on Advanced Separation of Concerns at ECOOP 2001. Available at <http://trese.cs.utwente.nl/Workshops/ecoop01asoc/ws.pdf>
- [11] M. Robillard and G.C. Murphy. *Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies*. In ICSE 2002.
- [12] M. Robillard and G.C. Murphy. *Capturing Concern Descriptions During Program Navigation*. In OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development.
- [13] W.L. Scherlis. *Analytic Views: Embedded Components and Managed Change*. Workshop on Software Evolution at ICSE'98.
- [14] W.L. Scherlis. *Structural Views, Structural Evolution, and Product Families*. From ARES Workshop on Product Families and Software Evolution, Springer-Verlag, 1998.
- [15] D. F. Sutherland, A. Greenhouse, and W.L. Scherlis. *The Code of Many Colors: Relating Threads to Code and Shared State*. In Workshop on Program Analysis for Software Tools and Engineering (PASTE'02), at FSE-10, November 2002.
- [16] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. *N Degrees of Separation: Multi-dimensional separation of concerns*. In Proceedings of the 21st International Conference on Software Engineering, 107-119, May 1999.
- [17] P. Tarr, H. Ossher, and J. Henkel. *Visualization as an Aid to Compositional Software Engineering*. In Workshop on Advanced Separation of Concerns at OOPSLA 2001.