Object Specialization through Behavioral Reflection and Crosscutting Annotation

Hisham H. Muhammad Ana Paula Lüdtke Ferreira

Universidade do Vale do Rio dos Sinos
Av. Unisinos, 950
São Leopoldo, RS, Brazil
`{hisham,anapaula}@exatas.unisinos.br`

Abstract. In object-oriented systems, classes and objects often evolve into complex entities, and still, specification of features is still done mostly in a method-by-method basis. This paper presents a novel technique for object specialization. It consists of a series of language constructs which perform reflective transformations in compile-time. These are based on annotations added to method signatures called *signals*, through which sets of methods from an object can be referred to collectively. This way, not only concerns that crosscut several methods from an object can be "factored out", but they can also be specified in a generic way so they can be mixed-in into objects of different inheritance hierarchies.

1 Introduction

Object-oriented programming languages provide mechanisms to ensure application extensibility, allowing a program to be adapted according to new requirements which might appear during the lifespan of a project. Such adaptations, however, often imply in creating a new support layer to the language itself in order to adequate it to the needs of a specific area. Examples of such scenarios are the implementation of object persistence for databases, mobility for distributed systems or paralellism for high performance computing. In such cases, it may be more interesting to extend the *definition* of a language, so that its applicability reaches beyond its original goals. Unfortunately, extensibility of the languages themselves is an often neglected feature. "Feature-oriented" language extensions tend to be *ad hoc*, increasing the complexity of the original language.

Extensiblity mechanisms for programming languages require access and manipulation of information regarding the characteristics and/or the state of a program. This is typically accomplished through computational reflection ([7]), by the establishment of a policy for manipulating not only data and code, but also their constitution and behavior. Computational reflection is a well known concept in the area of programming languages, and by allowing the specification of program operations in terms of the program itself, it makes a series of new idioms available to the programmer. While the theoretical foundations of computational reflection had already been established, how to express those idioms, or, in other words, how to translate the power of reflection (and, more generally, meta-programming) into program constructs, remains an open subject. Only in the last decade techniques to explore its expressiveness benefits were developed for applied fields such as Software Engineering, mainly in the form of Generative Programming ([5]) and related disciplines such as Template Meta-programming and Aspect-Oriented Programming ([8]).

Approaches such as these hint us at one direction where reflection appears as a tool for higher-level programming. This work explores the possibilities of exploring another direction using similar reflection-based techniques: object specialization. Classes and objects often evolve into complex entities, and still, specification of features is done traditionally in a method-by-method basis. For example, when specializing a primitive `List` into a `BufferedList`, a number of methods have to be changed in order to deal with the buffer. In the traditional model, the process of modification of existing methods is done by reimplementing each one of them. At best, the reimplementation includes a call to the original version of the method.

This paper presents a series of object specialization features based on behavioral reflection. Through the proposed model, methods and attributes can be grouped in categories, so that that use crosscutting concerns are specified in a less *ad hoc*, more declarative way. Those features are employed in a new reflective prototype-based programming language called Glass ([13]). Since they are additions to the traditional specialization model of OOP, they could also be added as extensions to existing class-based or prototype-based languages[1].

This paper is organized as follows: Section 2 presents a mechanism for crosscutting annotation, called *signals*, to be used in the reflective features for object specialization. The reflective features themselves are discussed in Section 3, along with examples of their use. Implementation issues are considered in Section 4. Section 5 discusses related work on reflection and aspectual decomposition. Finally, Section 6 concludes the paper.

2 Signals

Glass is an object oriented, prototype-based programming language with high-level constructs for compile time structural and behavioral reflection. It uses a type system inspired by ML-style signatures ([12]), which is beyond the scope of this paper. For all purposes Glass can be regarded as a typical prototype-based language, not unlike Self ([17]).

```
List is new Object as
   store is Vector
   add is method(o is Object) \
   signals Increase as
     ...
   end
   removeFirst is method() signals Decrease as
     ...
   end
   removeLast is method() signals Decrease as
     ...
   end
   copyFrom is method(l is List) \
   signals Increase, Decrease as
     ...
   end
end
```

```
Stack is new Object as
   top is Object
   push is method(o is Object) \
   signals Increase as
     ...
   end
   pop is method()(Object) signals Decrease as
     ...
   end
   peek is method()(Object) as
     ...
   end
   clear is method() signals Decrease as
     ...
   end
end
```

Fig. 1. Two objects, `List` and `Stack`, featuring user-defined signals, `Increase` and `Decrease`.

Besides usual primitive entities such as objects, handles and methods, Glass features an additional entity, called **signal**. Signals are annotations added to the signature of methods, as depicted in Figure 1. In this figure, signals `Increase` and `Decrease` define sets of methods on objects `List` and `Stack`. These signals act as a annotation for further reflective transformations: they are declared so that the code insertion features presented in Section 3 do not need to refer explicitly to method names, but only to signal names, as they represent the set of methods that declare them.

Notice that the scope of reflective transformations are restricted to the object they are applied to. A transformation can, thus, be applied explicitly to either `List.Increase` or `Stack.Increase`. In Glass, reflective transformations are always applied to methods of a specific object, either using the `object.signal` notation, or enclosing the transformation within the object declaration block itself. Still, generic transformations can be declared as a combination of reflective declarations using only a signal name without an object name — the language construct for this will be presented in Section 3. A generic transformation declared in terms of the signal `Increase` could then be applied to both `List` and `Stack`. This way, code can be shared between methods from objects of different inheritance hierarchies in a more flexible way than using multiple inheritance.

Signals are meant to be associated to methods where it is known in compile time that a given condition may be triggered in run time. For example, the chosen semantics for `Increase` and `Decrease` in the example was to specify that the signaled methods *may* increase or decrease the number of elements in the data structure. The code for controlling the "list empty" flag could be implemented only once, and inserted in all methods using the `Decrease` signal.

Operations can be specified by an object not only in terms of signals of the prototype objects it inherits from, but also in terms of signals of the attributes that consitute the object itself. This way, a set of methods where a given attribute triggers a signal (or, more precisely, where methods of a given attribute that happen to declare a

[1] Glass is prototype-based, so the article will refer mainly to object specialization instead of class specialization. Since these models are equivalent ([17]), the terms are interchangeable.

specified signal are called) can be referred through a signal. For example, it is unnecessary to declare a signal in the signature of every method that accesses a database; instead, the SQL library methods can declare signals, through which methods performing database access can be identified.

Object signals become visible only if explicitly exported (through an `export` clause). This restriction assures that an object controls how much of its internal state can be affected by the manipulation of signals in derived objects or in objects that use it as attribute.

Besides signals declared in method signatures, signals related to the behavior of the handles themselves can be referred to. Every handle has `read` and `write` signals, associated to handle access and assignment. These can be understood as signals declared in `get()` and `set()` methods in the reified view of the `handle` metaobject. Therefore `x.read` does not refer to "all methods of the object pointed by the handle `x` which feature signal `read` in its signature", but "all places where the `x` handle is read".

It is worth noticing that pointcut specification through signals is a diametrically opposite approach to that taken in AOP, where there is a clear separation between component and aspect code. In AOP, method names are grouped together within the aspect, along with reflective transformations; in our approach a signal also corresponds to a set of members, but the association between methods and signals is done by the method itself. The goals of each methodology are different and, in our point of view, complementary.

3 Reflective declarations

Glass supports two primary types of operations for behavioral intercession: **intramethod** and **extramethod** modification. To each of them, a specific language construct is built. Those are features for compile-time reflection, implemented as high-level macro mechanism. A third construct encapsulates reflective constructs, in order to provide generic reuse. Each of these constructs is presented below.

3.1 Intramethod Modification: "when"

Intramethod modification consists in inserting code inside a method body, based on the signals exported by method calls found inside it. If the code of these methods is instrumented with exported signals, one can define a rule such as "insert the given code block in every method before occurrences of calls that produce the signal `s`". Intramethod modification is declared using the `when` construct, which receives a signal as a parameter and builds before and/or after blocks which are to be applied on methods where calls that produce the specified signal occur.

```
BufferHandler is new Object as
   buffer is new List
   doSomething is method() signals Update as
    ...
      buffer.add(i)
    ...
      buffer.copyFrom(anotherList)
    ...
   end
   flushToDisk() is method() as
    ...
   end
   when buffer.increase
    before
      if buffer.size > limit then flushToDisk()
   end
end
```

(a)

```
Application is new Object as
   foo is new method() signals State as
    ...
      BufferHandler.doSomething()
    ...
   end
   bar is new method() signals State as
    ...
   end
   redefine where BufferHandler.Update as
    after: updateScreen()
   end
   redefine where State as
    before: storeState()
    after: compareState()
   end
end
```

(b)

Fig. 2. Transformations with the `when` (a) and `redefine` (b) declarations.

Figure 2(a) shows an example of intramethod modification using the `when` construct. In this example, object `BufferHandler` wants to use `List` as a buffer and flush it every time its size reaches a certain limit. `BufferHandler` creates a copy[2] of `List` and assigns it to handle `buffer`. The `when` declaration at the bottom adds an `if` statement

[2] The `new` keyword indicates a creation of a copy of a prototype object into a new one. Modifications in the copied object may be specified in a `as`/`end` block.

performing the flush check right before each call of methods from `buffer` which signal `Increase`, inside every method of `BufferHandler`. For illustration purposes, a simplistic example is shown, where the code insertion is equivalent to adding the portrayed block in the preceding line of each matching method call. Less trivial code insertion can occur, as in the following example. Suppose that `List` features a method `checkInsert` that returns a value and also signals `Increase`. Some method from object `BufferHandler`, then, contains the following statement:

```
if a(b(), buffer.checkInsert(c()))
```

According to the semantics of Glass, the flush check is added before the evaluation of the target method's parameters, in this example, between the calls of `b()` and `c()`. The order of evaluation is the following. First, `b()` gets called. Then, the inlined flush check, followed by `c()`, and finally, by `buffer.checkInsert()`. The method call is "wrapped around". Implementation strategies are discussed in Section 4.

In spite of apparently breaking with the atomic view of methods existing in behavioral reflection, intramethod modification must not be confounded with linguistic reflection. Instrumentation inside a method body is limited to entry and exit points of method calls. It may also seem to be a breach on encapsulation because method transformations are performed based on their contents. This is not the case, because these transformations are to be understood as extensions of the method calls they wrap.

The use of when is also different than simply specializing an object, adding code and a call to the parent method, for two reasons. First, the scope of the modification is limited to an object, so the method from an object can be specialized differently on different contexts. Second, many methods can be specialized at once, avoiding code duplication.

3.2 Extramethod Modification: "redefine"

Extramethod modification consists of defining rules for the replacement or instrumentation of a method by surrounding its code with additional calls, using signals as a criterion, in an equivalent way to `before`- and `after`-methods from CLOS. This is done through the `redefine` declaration.

There are two forms of using the `redefine` block construct: through the specification of an object and a signal, as in `redefine where o.s`, or by a signal only, as in `redefine where s`. In the first case, the `before` and/or `after` blocks will be prepended/appended to all methods containing a call to a method of object `o` which features the signal `s` in its signature. In the latter case, the affected methods will be those containing `s` in their signature.

The example from Figure 2(b) demonstrates the two forms of the `redefine` declaration. An `Application` object features a number of methods (`foo`, `bar`, etc.). Two `redefine` declarations modify the methods of `Application`. The first one appends a method call (`updateScreen()`) to every method from `Application` that contains calls to methods from `BufferHandler` that signal `Update` (in this example, `foo` will be modified because it contains a call to `BufferHandler.doSomething()`). The second one surrounds every method from `Application` that signals `State` (`foo` and `bar`) with a pair of blocks.

It is worth to point out that the examples given above show that `when` and `redefine` are used not only to specialize inherited methods from a parent object, but also to isolate coincidental parts of the methods being defined on the object itself. Through `when` and `redefine`, aspectual decomposition can performed *within* an object.

3.3 Composition of declarations

A third construct, `declaration`, serves for combining declarations. It can be considered a high-level macro processing construct, with roots in Lisp's `defmacro`. A `declaration` block is a top-level program construct, and so it is not contained within an object block. A `declaration` is specified by a signature, containing its name and a series of parameters. Object and method declarations, as well as `when` and `redefine` declarations, can then be listed inside it. A declaration can be applied in an object block definition as if it was an extension of the language syntax.

Method sets represented by signals which are affected by reflective transformations can be specified independently from the inheritance and subtyping trees. Once a reflective transformation is declared for signal `S`, it can be used on any object where methods signalling `S` exist. Furthermore, in a `declaration`, handles can be reified as `handle` objects and signals as `signal` objects, in order to allow them to be used as parameters as well as parameterization of reflective commands enclosed within the `declaration` block.

The following example demonstrates the addition of a new declaration, `synchronize`, which, on its turn, makes use of the `redefine` declaration.

```
synchronize is declaration (A is handle, S is signal) as
   synchronizeMutex is new Mutex
```

```
        redefine where A.S as
            before : synchronizeMutex.lock()
            after : synchronizeMutex.unlock()
        end
    end
```

The `synchronize` declaration adds an attribute of type `Mutex`, and surrounds all methods that write to a specified handle with a lock. For example, A version of `Stack` with mutually exclusive access is produced by specializing it, applying the `synchronize` declaration:

```
    Pool in new Stack as
        synchronize top write
    end
```

This example demonstrates how generality can be achieved by using handle and signal names as parameters of a declaration.

3.4 Exporting signals

A possible limitation on signal-based code maintainability lies in how to control the (possibly far-reaching) impact of modifications produced by reflective commands such as `when` and `redefine` on methods declared on parent objects. The question is aggravated by the computation model based on "side effects", an inherent characteristic of the imperative paradigm. The addition of code to a method causes new side effects and modifies previously existing ones. The state of internal variables of a method cannot be accessed, nor modified, but the state of an object or attribute which it depends on may be. The association of pre- and post-conditions to the exportation of a signal can be a powerful tool in order to guarantee a method's consistency.

The use of pre- and post-conditions can ensure the maintenance of the object contract even with the use of intrusive operations such as the ones presented above. This way, the specification of properties of an object will not be invalidated by specialization of its methods in a derived object. Further, in Glass, while structural intercession features (not discussed in this paper) such as method assignment have a run-time nature, behavioral intercession is restricted to compile-time, and therefore reflective transformations are always performed relative to the object's initial structural state.

4 Implementation

The forms of code manipulation discussed in the previous Section are performed on the abstract syntax tree level, generated once the program source is parsed. In the prototype implementation, the AST is a collection of nodes that correspond to the rules of the language syntax grammar, with auxiliary nodes that served purely syntactical purposes, such as `<End>`, removed. Once the AST is in memory, the next step is to process the reflective commands, which generate the new structures that represent the modified code. What we have here is clearly the application of transformations on a tree. A clear, concise way to represent these transformations is through **tree-transformational grammars**, or TTGs ([14]). A TTG is a context-free grammar where each production features a term on its leftpart, and one or two rightparts, one indicated with a single arrow which represents the sub-tree to be recognized, and another indicated with a double arrow representing the sub-tree to be inserted in place of the one that was recognized. Each reflective command, then, corresponds to a pattern of applications of transformational grammars to be followed.

The specification of the semantics of the reflective commands of Glass was made using **transformational attribute grammars**, or TAGs ([14]). As its name implies, TAG combines a tree transformational grammar to an attribute grammar ([10]). In attribute grammars, nonterminal symbols of the grammar have associated to them attributes, which can be *inherited* or *synthesized*. Computations are then specified declaratively on these data structures described by the attributes. Inherited attributes can be seen as data computed top-down, and synthesized attributes as data computed bottom-up ([4]).

Code instrumentation performed by the `when` command can be seen clearly as a top-down traversal of the abstract syntax tree corresponding to each method of an object. Translation of this traversal to a TAG is straightforward. The four elements given in the construct's syntax—identifier, signal, before-block and after-block—are entered as attributes to the transformational rule. The `redefine` command corresponds to a bottom-up traversal of each method, synthesizing an attribute indicating whether the method matches the transformation pattern or not. Code insertion is then performed top-down, similarly to `when`.

5 Related work

The abstraction power of a language can be largely increased by making the metaclasss hierarchy explicit; however, the simplistic availability of this hierarchy gives little benefit to the development of application software. As techniques for implementation of computational reflection mature, a number of research groups have been dedicated to studying strategies for the efficient exploration of it in a software engineering context.

Aspect-Oriented Programming is gaining increasing acceptance as a design methodology. The most proeminent language that implements AOP is AspectJ ([9]), an extension of Java. The fundamental principle of AOP is based on the fact that certain characteristics are spread over classes that are not related through the inheritance hierarchy. Aspects are constructs that can refer to methods of different classes, instrumenting them with additional features. This pervasive capacity of aspects, however, can be considered a breach in the encapsulation of classes ([11]). Definition of pointcuts inside the aspect is an explicit reference to the class structure. A complex aspect will feature a series of pointcuts with references to several methods; changes in classes will imply in a revision of the aspects.

We've seen that the use of signals declared in method signatures avoids the problem of synchronizing the base code and the reflective code, abstracting away knowledge about the internal constitution of the target object. For this reason, the use of signals as crosscutting annotation for aspects could present a maintenance gain. Parametrization of reflective code as occurs in the `declaration` construct is also a step to bring aspect-orientation and generic programming closer to each other, as an attempt to improve reusability of code-insertion rules.

The approach taken in signal-based object specialization presented in this work is different from AOP in two ways. First, there is no separation between component language and aspect language. Second, our intent is not to address system-wide crosscutting concerns, but to crosscut methods within objects. Because of these difference of goals, we believe that both techniques can be used together, and through unified pointcut specification, could benefit from each other.

Generative programming ([5]) emerged as another programming discipline for higher level development, based on template meta-programming. It explores the code composition facilities provided by C++ templates ([15]) to build complex objects based on compile time layering. By providing generic facilities for declaration of reflective transformations, signal-based object specialization can be used as a tool for generative programming. Default behavior can be coded into an object and a selected group of transformations can configure a derived object, which then serves as a generator prototype. This serves the purposes of code composition expected by the generative model, and avoids the need for a Turing-complete template meta-programming such as the one (accidentally) provided by C++.

OpenC++ ([3]) is a C++ preprocessor that extends C++ by adding compile time reflection to the language. Its implementation can be considered a high level macro processor. OpenC++ adopts a lower-level approach. It allows for arbitrary syntax extensions, through explicit manipulation of abstract syntax tree nodes or insertion of strings of code. This brings greater flexibility at the expense of postponing syntax verification to the final pass of compilation, when the generated C++ is compiled.

The Explicit Programming methodology shares much in intent with the approach taken in our work. In [2], Bryant et al. present this programming model and a tool called ELIDE which provides Java extensions for Explicit Programming. Our work has many similarities with Explicit Programming in that it allows the user to add vocabulary to distinguish parts of the code that share common design concepts, and, more importantly, this vocabulary is added where the concept occurs in the code. It could be said that the reflective features of Glass allow for Explicit Programming. There are, however, many differences from the design adopted in ELIDE. In Glass, the added annotation does not correspond to a specific transformation, instead transformations are specified later on, in terms of this annotation (signals). Signals are relative to the inheritance hierarchy, and can be reused in derived objects. This way, they can be used for different kinds of transformations, depending on how the objects are used. Also, in Glass, a limited number of high-level idioms for transformations is provided, while Elide allows manipulation of Java code as strings, like OpenC++.

6 Conclusion

In recent years, focus on research in computation reflection has directed towards the development of ways to increase its applicability, now that its theoretical foundations is already well understood. This work proposed a model where behavioral reflection is used to perform code composition using language constructs to specify modification of methods and a high-level linguistic feature called signaling is used to indicate points of instrumentation. Through signals, sets of points where a given code modification should be applied are indicated, without interfering with the basic principles of encapsulation defined by the object-oriented paradigm. It is important to note that all those techniques are based in compile-time reflection.

Numerous possibilities are open for future work, ranging from further development of the prototype compiler and its adaptation to generic backends to the formalization of concepts presented in this paper, such as the completion of the tree-transformational attribute grammars that specify the semantics of `when` and `redefine` (presented in [13]) into a full translation grammar.

The availability of crosscutting annotation embedded in the program language also makes it a good candidate for integration with an aspect language. The approach currently under consideration consists of adding a form of `declaration` that is able to apply transformations to methods from a set of objects at once. Adding `signals` to `new` object declarations is being considered as a way to specify this set of objects, but these are still open issues.

A possibility that was not fully explored in this work is the full reification of handles. Reifying a handle, one can see the pointer itself as an object. The pointer's behavior could then be specified in order to configure the management of objects and methods of a program. Examples of such behaviors would be smart pointers, communication with garbage collectors or even handles that act as low-level pointers to memory areas or communication ports.

As new constructs are added to a language, new design possibilities arise. An interesting area of research would be to investigate how these constructs would affect the implementation of existing design patterns, possibly simplifying or generalizing them, and what new design patterns could be thought of, exporing the `declaration` construct as a form to develop reusable patterns of object specialization.

References

1. Aksit, M. et alli. "Abstracting Object Interactions Using Component Filters". In *Object-based Distributed Processing*, R. Guerraoui, O. Niestrasz e M. Riveill (Eds.), LCNS, Springer-Verlag, págs. 152-184, 1993.
2. Bryant, A., Catton, A., De Volder, K., Murphy, G. C. "Explicit Programming". In *AOSD 2002 Conference Proceedings.* Conference on Aspect-Oriented Software Development (AOSD 2002). ACM, 2002.
3. Chiba, S. "A Metaobject Protocol for C++". In *OOPSLA '95 Conference Proceedings.* Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95), ACM. Austin, 1995.
4. Correnson, L., Duris, E., Parigot, D., Roussel, G. "Declarative Program Transformation: A Deforestation case-study" In *Principles and Practices of Declarative Programming*, Vol. 1702 of Lecture Notes in Computer Science, Springer-Verlag, 1999.
5. Czarnecki, K. et al. "Synthesizing Objects", In *Proceedings*, European Conference on Object-Oriented Programming (ECOOP), Portugal, 1999.
6. Kernighan, B., Ritchie, D. "The C Programming Language", 2nd. Edition, Prentice Hall. March 1988
7. Kiczales, G., des Rivières, J., Bobrow, D. *"The Art of the Metaobject Protocol"*. Ed. MIT. Cambridge, 1991.
8. Kiczales, G. "Aspect-Oriented Programming". In *Proceedings*, European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag. 1997.
9. Kiczales, G. "An Overview of AspectJ". In *Proceedings*, European Conference on Object-Oriented Programming (ECOOP), Budapeste, Hungria. 2001.
10. Knuth, D. "Semantics of context-free languages". *Mathematical Systems Theory Journal*, Vol. 2, 1968.
11. Malenfant, J., Cointe, P. "Aspect Orientation versus Reflection", Technical Repot, École de Mines de Nantes, 1996.
12. Milner, R., et alli. *"The Definition of Standard ML"*, MIT Press, 1997.
13. Muhammad, H. H. "Exploração de Reflexão Computacional através de um Modelo de Objetos sem Classes". Term Paper. Universidade do Vale do Rio dos Sinos. November, 2002.
14. Pittman, T., Peters, J. *"The Art Of Compiler Design: Theory And Practice"*, Prentice Hall, Upper Saddle River, 1992.
15. Stroustrup, B. *"The C++ Programming Language"*, Addison-Wesley Pub. Co., Julho, 1997.
16. *"Qt 3.0 Whitepaper"*, Technical Report, Trolltech AS, Oslo, 2002.
17. Ungar, D. et alli. "Organizing Programs Without Classes" In *Lisp and Symbolic Computation* 4(3), Kluwer Academic Publishers, Junho, 1991.