Towards a Standard Design Language for AOSD

Siobhán Clarke
Department of Computer Science
Trinity College
Dublin 2, Republic of Ireland
+353 1 6082224
siobhan.clarke@cs.tcd.ie

Robert J. Walker
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC, Canada V6T 1Z4
walker@cs.ubc.ca

ABSTRACT

For aspect-oriented software development (AOSD) to live up to being a software engineering method, there must be support for the separation of crosscutting concerns across the development lifecycle. Part of this support is traceability from one lifecycle phase to another.

This paper investigates the traceability between one particular AOSD design-level language, Theme/UML, and one particular AOSD implementation-level language, AspectJ. This provides for a means to assess these languages and their incompatibilities, with a view towards eventually developing a standard design language for a broad range of AOSD approaches.

1. INTRODUCTION

Concerns that have a crosscutting impact on software (such as distribution, persistence, etc.) present well-documented difficulties for software development [5, 11, 17, 26]. Since these difficulties are present throughout the development lifecycle, they must be addressed across its entirety.

The separation and encapsulation of crosscutting concerns has been promoted as a means of addressing these difficulties; the standard object-oriented paradigm does not suffice [17]. In order to overcome the difficulties for crosscutting concerns throughout the lifecycle, an approach is required that provides a means to separate and encapsulate both the designs and the code of crosscutting behaviour.

We believe that it is important to work towards a general purpose AOSD design language that meets certain goals, including the following.

- *Implementation language independence*: The ultimate form of aspect-oriented programming (AOP) languages may not be exactly that of any current one. Thus, any design language that simply mimics the constructs of a particular AOP language is liable to fail to achieve implementation language independence.
- *Design-level composability*: Design-level composability is a desirable property for two reasons. First, designers may check the result of composition prior to implementation, for validation purposes. Second, some projects will continue to require the use of a non-aspect-oriented implementation language because of pragmatic constraints, such as the presence of legacy code written in languages without aspect-oriented extensions; these projects could still achieve benefit from separating the design of crosscutting concerns.
- *Compatibility with existing design approaches*: An AOSD design-level language should also build on existing design languages, such as UML [21], to provide a bridge from old techniques to new, so that software engineering realities such as incremental adoption and legacy support are possible.

AspectJ [16, 27] is an AOP language with low-level support for specifying and composing crosscutting code into a core system. At the design level, Theme/UML[1] [3, 4, 5, 6] provides a means for separating the designs of crosscutting requirements into reusable, extensible design models.

At first glance, it is not clear that the approaches of AspectJ and Theme/UML are compatible; thus, the goal of traceability between them is problematic. AspectJ's approach to providing very general, low-level constructs has permitted, and continues to permit, a broad scope of research into aspect-oriented concepts and design guidelines. But it is not necessarily the form that AOP languages will take in the long-term; domain-specific and higher-level languages are possible and already exist. Likewise, Theme/UML has concentrated on particular facets of the problem of crosscutting design, and may not suffice as a general-purpose AOSD design language in the long-term.

As a first step towards such a design language, we examine the traceability between Theme/UML and AspectJ (version 1.0). This provides two contributions: (1) the description of a reusable, traceable, and evolvable mapping between the constructs in the two languages permits Theme/UML designs to be implemented in AspectJ in a well-engineered manner; and (2) the incompatibilities between these languages and their approaches to AOSD provide lessons as to the form and nature of a more general AOP approach, and of a supporting design language that addresses our long-term goals described above.

Section 2 briefly describes Theme/UML, and illustrates the use of its composition pattern construct through a simple example; further details of Theme/UML can be found elsewhere [3, 4, 5, 6]. We describe the mapping process from composition patterns to AspectJ in Section 3, emphasising this particular example; descriptions of AspectJ and its constructs can be found elsewhere [16, 27].

[1] "Theme/UML" is a new term that encompasses subject-oriented design, the composition pattern construct, and additional changes that arise out of this current work.

AOSD 2002, Enschede, The Netherlands

Section 4 considers the support for AspectJ constructs given by Theme/UML, and Section 5 provides recommendations for investigating the evolution of both languages. Related work is described in Section 6, and Section 7 presents conclusions.

2. BACKGROUND: THEME/UML

It is the nature of crosscutting behaviour that it has an impact on multiple, different elements within software. In order to design such behaviour in standard UML [21], it is necessary to explicitly specify crosscutting behaviour for each of the particular elements it affects; the designs of crosscutting behaviour cannot be separated and encapsulated with existing UML constructs. These limitations result in design models with scattering and tangling properties comparable to those in code [5, 3].

Theme/UML mitigates these problems by supporting the design of crosscutting concerns as separated, encapsulated design models. Composition of these separate design models is specified with a *composition relationship*, detailing which elements are to be combined, and how to integrate them. *Merge* is one strategy for integration that includes all the elements from the input design models in the composed design, reconciling conflicts where appropriate.

The *composition pattern* (CP) construct [6] of Theme/UML, based on an extension to UML templates, permits a crosscutting design model to be independent of any base design model, allowing it to be reused. The composition of concrete design models with CPs is based on the semantics of the merge integration strategy.

Template parameters on a composition pattern may represent operations. A key feature of CPs is that they may define *supplementary behaviour* on such template operations. When a template operation with supplementary behaviour is bound to a concrete operation, the supplementary behaviour is merged with the original behaviour of that concrete operation. Any calls to the original operation result in delegation to some ordered combination of the supplementary behaviour and original behaviour, as prescribed within the specification of that CP.

A more complete description of the extensions to the UML metamodel required to support composition semantics and composition patterns can be found elsewhere [3, 4].

We demonstrate the mapping from Theme/UML to AspectJ through the design of a reusable aspect based on the Observer design pattern [9], with a specification of how to compose it with a base design supporting a digital Library. The composition pattern to support this example is more completely illustrated elsewhere [6]. We extract a subset of the pattern to demonstrate our mapping points.

The Observer pattern describes the collaborative behaviour between a subject and multiple observers. Observer objects register an interest in subject objects; when the state of a subject changes, the observers that are registered with it are notified. From a composition pattern perspective, this requires both structural and behavioural template design elements. We define an Observer CP with two *pattern classes* (classes that are templates to be bound to actual classes during composition with a base design). Subject is defined as a pattern class representing the class of objects whose changes in state are of interest to other objects, and Observer is defined as a pattern class representing the class of objects interested in a Subject's change in state (see Figure 1).

A CP may also contain interaction specifications for behaviour that crosscut template operations. For example, Figure 2 illustrates the behaviour required for notifying observers of changes in state. _aStateChange( ) is a template operation whose behaviour is supplemented with notification of all observers. This operation has been prepended with an underscore to denote that supplementary

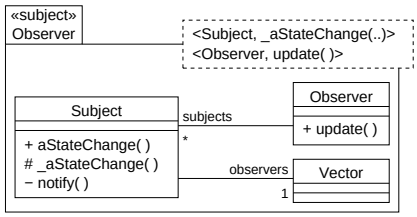

Figure 1: Observer CP Structure

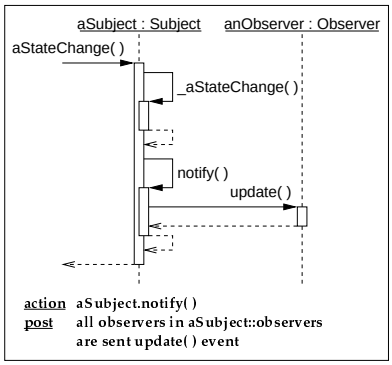

Figure 2: Notifying Observers of State Changes

behaviour is specified, and hence, calls to any concrete operation bound to _aStateChange( ) result in delegation to that operation's original behaviour and to the supplementary behaviour; the interaction specification indicates the order in which the delegation occurs. Since _aStateChange( ) is an operation on Subject, it must be bound to a concrete operation on each concrete class that is bound to Subject. The notify( ) (non-template) operation calls another template operation, update( ), which must be bound to some operation in any class that is bound to Observer.

The composition of a Library design model with the Observer composition pattern is specified by a composition relationship, with a bind[ ] attachment, between the two. The bind[ ] attachment identifies those elements in the base design that will "replace" the template parameters in the composed design. For example, the class(es) acting as subject, and the class(es) acting as observer may

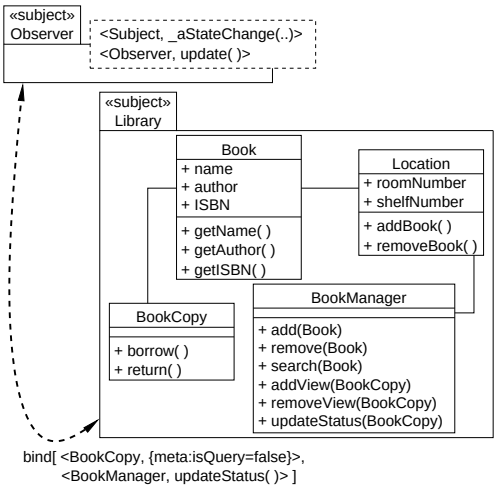

Figure 3: Composing Observer with Library

be defined as bindings to the Subject and Observer pattern classes. In this example, there is only one of each, BookCopy and BookManager, respectively (see Figure 3). Sets of elements may be bound to a template parameter by specifying them within a pair of braces.

For the bindings to the template operations, note how in this example the meta-properties of a design's elements may be queried to assess an element's eligibility to join a set of replacing elements. In this example, the _aStateChange( ) template operation is replaced with all operations within BookCopy that have been defined as being non-query—i.e., those operations that effect a change in state that may be of interest to an observer. The keyword meta within the binding specification denotes that a UML meta-property is tested, and only those operations where isQuery=false holds are bound to _aStateChange( ) for the purposes of the composed library design.

3. MAPPING CPS TO ASPECTJ

The question of how to map composition patterns to AspectJ depends on how faithfully one wishes to represent the design-level entities. There are two chief approaches: (1) represent both a CP and its composition specification as a single aspect, or (2) maintain the separation of a reusable CP from its composition specification.

We briefly demonstrated approach 1 in [6]. This approach results in aspects that lack reusability and evolvability. It requires that the CP be reimplemented for every composition specification that is used; each of these would separately need to be modified should the CP need to evolve.

Approach 2 promises a better solution, and forms the basis for our analysis in this paper. Difficulties with evolving mapped CPs would be reduced if we could produce an implementation-level construct that represented a CP alone, without its composition specification. Then, any changes to this CP would affect only this one construct. This construct should then be more reusable, since it would not be specific to a single composition specification.

Abstract aspects potentially provide such a means of separating the code for crosscutting behaviour in a reusable way. We therefore provide a direct mapping, from uncomposed CPs to abstract aspects, in Section 3.1. The composition with the base library specification is then mapped to concrete aspects that extend these abstract aspects; this is described in Section 3.2.

3.1 Uncomposed CPs Map to Abstract Aspects

Figure 5 illustrates an algorithm to be applied when mapping a reusable composition pattern to AspectJ. It only covers the design model elements illustrated by the Observer composition pattern example; other model elements are discussed briefly in Section 4.

The abstract aspect that results from the Observer CP is shown in Figure 4. We describe the reasoning behind the major features of this mapping below.

Pattern classes become interfaces within the abstract aspect. A pattern class (e.g., Subject) is a "placeholder" for a concrete class. In AspectJ, the only constructs provided that can act as such a placeholder are an interface or a class.

If we use a class, an inheritance relationship will need to defined for the concrete, replacing class. This is because a pattern class may name operations without providing any constraints upon their behaviour, and therefore they need to map to an abstract type in AspectJ. However, a concrete class (e.g., `BookCopy`) that will eventually have to become a subtype of this abstract type may already possess a superclass. It would be necessary to replace any previously defined superclass of the concrete class if we chose to map the pattern class to an abstract class.

On the other hand, mapping a pattern class to an interface allows

```
abstract aspect Observer {
  // --- Type declarations ---
  interface SubjectI {}

  interface ObserverI {
    public void update();
  }

  // --- Introductions ---
  Vector SubjectI.observers;

  private void SubjectI.notify() {
    // Post: all observers in SubjectI.observers
    // are sent update() event
  }

  // --- Pointcuts ---
  abstract pointcut aStateChange(SubjectI s);

  // --- Advice ---
  after(SubjectI s): aStateChange(s) {
    s.notify();
  }
}
```

Figure 4: Aspect mapped from uncomposed Observer CP.

a concrete class to have added to it a declaration that it implement this interface (e.g., `BookCopy implements SubjectI`).

Non-template operations become introduced methods on those interfaces. An interface maps from a pattern class, so that interface must somehow represent the operations required of that pattern class. Since AspectJ permits method introduction on an interface to propagate to all classes implementing that interface, this is a convenient means to add a method to a concrete class whose name is not known prior to binding. For example, Subject::notify( ) becomes `SubjectI.notify()`, which causes any concrete classes implementing `SubjectI` to implicitly have the `notify()` method introduced on them.

Template operations without supplementary behaviour become methods on those interfaces. Once again, an interface mapping from a pattern class must somehow represent the operations required of that pattern class. With a template operation without supplementary behaviour, either explicitly placing the corresponding method within the interface declaration, or introducing an abstract method onto the interface would suffice. We have taken the former route, for the sake of clarity.

Template operations with supplementary behaviour become abstract pointcuts plus advice. To have any supplementary behaviour occur on execution of the concrete method bound to a template operation, advice is an obvious construct. Advice permits behaviour to be specified even on abstract pointcuts. Since the CP is only supplementing the behaviour of this template operation, it need only have an abstract pointcut to advise. Thus, Subject::_aStateChange(..) becomes `aStateChange(SubjectI)` plus **`after`** advice.

3.2 Composition Specifications Map to Concrete Aspects

Now that we have a construct representing uncomposed composition patterns, we need to extend it to represent a particular composition specification. The algorithm for performing this deals with many small details that are conceptually straightforward; thus, we do not give a detailed algorithm, but touch on the highlights of the process.

The elements associated with a bind[ ] specification on a composition relationship are mapped to the appropriate AspectJ con-

Input: A composition pattern *CP*.
Output: An abstract aspect *A* corresponding to *CP*.

- Declare abstract aspect *A*.
- For each pattern class $PClass_i$ in *CP*:
  - Declare interface I_i in *A*.
  - For each template operation with no supplementary behaviour, $\langle op \rangle_{i,j}$, on $PClass_i$, declare corresponding (abstract) method $m_{i,j}$ on I_i.
  - For each template operation with supplementary behaviour[2] $\langle \_op \rangle_{i,k}$, on $PClass_i$:
    - Declare corresponding abstract pointcut $pc_{i,k}$ with formal parameters consisting of one to capture target object (of type I_i) on which $\langle \_op \rangle_{i,k}$ is called plus one corresponding to each of the specified formal parameters on $\langle \_op \rangle_{i,k}$.
    - Declare advice on $pc_{i,k}$ according to the supplementary behaviour.
  - For each non-template operation $op_{i,l}$ on $PClass_i$, introduce method $m_{i,l}$ (which implements any behavioural specifications for $op_{i,l}$) on I_i.
  - For each non-template association from $PClass_i$ to some non-pattern class, introduce field on I_i.
- For each non-pattern class in *CP*, implement it directly if not already available.

Figure 5: Algorithm for mapping certain constructs in an uncomposed CP to AspectJ.

```
aspect LibraryObserver extends Observer {
  // --- Declarations ---
  declare parents:
  BookCopy implements SubjectI;

  declare parents:
  BookManager implements ObserverI;

  // --- Introductions ---
  public void BookManager.update() {
    updateStatus();
  }

  // --- Pointcuts ---
  pointcut aStateChange(SubjectI copy):
    target(copy) && args(..) &&
    (execution(* BookCopy.borrow(..)) ||
     execution(* BookCopy.returnIt(..)));
}
```

Figure 6: Aspect mapped from binding Observer CP to Library design model.

structs. The concrete aspect that results from the composition specification between the Observer CP and the base Library design is shown in Figure 6. We describe the reasoning behind the major features of this mapping below.

Concrete classes bound to pattern classes require `parents` declarations in the concrete aspect. A pattern class is represented through an interface declared on the abstract aspect. The act of binding a concrete class in the design model to that pattern class causes that concrete class to effectively become a subtype of the pattern class within the context of the CP. Therefore, a concrete class in the implementation must be effectively a subtype of the interface corresponding to the pattern class in order to be "bound" in an analogous manner. To this end, we use AspectJ's ability to add supertypes to a class. For example, Subject is represented as the interface `SubjectI`; since Subject is bound to BookCopy, `BookCopy` must implement `SubjectI`.

Concrete operations bound to template operations without supplementary behaviour require introduction of delegating methods on concrete classes. Each interface (e.g., `ObserverI`) within the abstract aspect, which corresponds to a pattern class, had declared within it a method to represent each template operation without supplementary behaviour (e.g., `update()`). Since some concrete class has been declared to implement this interface (i.e., `BookManager`), it must implement each of these methods. The binding of a concrete operation to a template operation is straightforward save for one point: their signatures can differ. So, even though the concrete class already possesses a method corresponding to that being bound to the template operation (i.e., `updateStatus()`), our concrete aspect must introduce an additional method (i.e., `update()`). This method must possess the signature expected by the aspect, and must delegate to the method already possessed by the concrete class.

Concrete operations bound to template operations with supplementary behaviour require concrete pointcuts. Finally, the abstract aspect declared abstract pointcuts that must be made concrete. Each abstract pointcut (e.g., `aStateChange(SubjectI)`) corresponds to a template operation (i.e., Subject::_aStateChange(..)) in the design model. For the concrete operations bound to this template operation, the concrete pointcut must capture the execution joinpoints of the concrete operations, while exposing the target object and arguments at each.

In the next two sections, we discuss the differences between Theme/UML and AspectJ, and consider how they both could evolve to provide for more generic aspect specification.

4. THEME/UML AS AN ASPECTJ DESIGN LANGUAGE

We have described implementation language independence as a goal for an AOSD design language. In this section, we consider Theme/UML as the design language to support AspectJ. Since Theme/UML was not designed to support AspectJ particularly, but the decomposition and composition of UML design models in gen-

[2] This assumes that the operation that delegates to this template operation is not itself invoked directly within the CP. In some such cases, the design model would not be well-formed. In the cases where it is well-formed, we have determined a clean mapping to AspectJ, but this is not discussed further in this paper.

eral (including crosscutting models), this is a reasonable initial test of its independence from implementation languages. In the following subsections, we examine categories of constructs and features of AspectJ, in turn; these categories and the quotes in each are taken from [27].

Pointcuts. A pointcut is a set of joinpoints, which are "well-defined instants in the execution of a program." Abstract pointcuts can be modelled via template parameters in composition patterns; concretisation of these abstract pointcuts can then be modelled via binding of the template parameters to concrete model elements.

Pointcut designators. A pointcut designator is a name or an expression that identifies a pointcut. Name-based pointcut designation corresponds to binding named operations to template operations in the bind[ ] attachment (e.g., see Figure 3). Theme/UML currently does not have a sophisticated expression-based semantics defined for its binding specifications; it does support simple expressions through the meta construct of the bind[ ] attachment, and limited wildcard matching. We envisage that these expressions will be extended to permit the use of the UML's Object Constraint Language (OCL) [30], which would also handle AspectJ's conditional and composite pointcut designators.

In general, the richness of the Theme/UML capabilities relating to the capture of pointcuts is constrained by the Common Behaviour semantics of the UML [21, §2.9]. This means that execution joinpoints map well to the design specification. However, the ability to distinguish between call and execution joinpoints, and to reference field-related joinpoints, exception-handler execution joinpoints, state-based joinpoints or control-flow joinpoints may require extension to the Theme/UML model. There is some evidence to suggest that the use of constraints, as provided by the OCL, will provide considerable support here. In addition, interesting developments on extensions to the UML Action model [18, 22], which deals with the specification of dynamic behaviour, should enrich the kinds of pointcuts that can be captured explicitly. We believe it to be preferable to utilise standard UML semantics for pointcut specification, than to extend Theme/UML in this regard. In addition, we plan to investigate further whether support for all the AspectJ pointcut types is actually required at the design level, and whether this can be achieved with minimal extensions to Theme/UML.

Type patterns and context exposure. In AspectJ, type patterns are "a way to pick out collections of types and use them in places where you would otherwise use only one type." The semantics for establishing correspondence of elements to be bound to templates within Theme/UML can be easily extended to include wildcard matching, and to utilise OCL to provide sophisticated matching expressions.

Advice. In AspectJ, advice is "code, similar to a method, that is executed when a joinpoint is reached." Standard UML has sophisticated support for interaction specification. This support is utilised directly by Theme/UML as the approach to the specification of advice behaviour. The **`before`**, **`after`** and **`around`** AspectJ constructs are easily mapped from the interaction diagrams. Indeed, there may be scope for AspectJ to extend its advice support based on what is available within the UML.

Static crosscutting. Theme/UML has merge integration semantics defined which are largely analogous to static crosscutting from AspectJ. Merge integration essentially joins elements, both static and behavioural, from input design models. In a sense, all the input elements are "introduced" together appropriately in a new, composed design model. At a detailed level, some differences exist in the approach to visibility and to conflict resolution, as discussed in the next paragraph.

Aspect associations. In Theme/UML, the CP construct captures the specification of crosscutting behaviour with template elements. This maps to an aspect within AspectJ. Aspects and CPs both act as scopes in which to interpret names and maintain state; both are associated with a context over which they are applied. However, Theme/UML composes a composition pattern with a base design without maintaining a dynamic correspondence; in other words, the composition pattern is not instantiable, and thus, cannot act in an identical fashion to aspect instances in any dynamic model. This should be of greatest consequence in modelling aspect associations with dynamic scope related to the control flow; these cannot be easily mimicked by a purely static scope. On the other hand, Theme/UML is not limited to a small handful of scope specifications. There is room for convergence of the two approaches here.

5. DIRECTIONS FOR INVESTIGATION

Having explored the existing support for traceability between Theme/UML and AspectJ, we now give recommendations for investigating the evolution of each language. Beyond such investigations, the study must expand to include more approaches, especially as research continues in such areas.

5.1 Evolving Theme/UML

As a priority, extensions to Theme/UML will be based on the provision of wildcards and OCL constraints for pointcut designation—or, in Theme/UML terms—to extend the kinds of templates that may be specified within a CP, and to extend the expressiveness of binding specifications on those templates. This will considerably enrich a designer's ability to specify joinpoints, type patterns, and context exposure.

The significance of the lack of a first-class aspect construct at the design level will also be considered. AspectJ's ability to associate aspects with fine-grained, dynamic scopes, such as portions of the control flow, is lacking from Theme/UML. It is possible that general, dynamic scopes could be specified by bindings that are constrained to hold only when certain interaction specifications occur; a precise, formal semantics await improvements to UML's Common Behaviour semantics. At this point, it remains unclear whether this is a significant problem or such dynamic scopes are a hammer in search of a nail.

Theme/UML as a general-purpose AOSD language. We have looked in some detail at the implementation-language independence of Theme/UML, and feel that there is considerable scope for optimism here. Space precludes us from assessing Theme/UML against our other goals, but we provide short comment. The semantics for composing design models specified using a composition relationship are defined in [3]. However, the full UML was not within the scope of the work (e.g., state models were not included). In addition, any extensions to the model will also require extensions to the composition semantics. Nonetheless, specifying such semantics is considered an integral part of any extensions. From the point of view of compatibility with other design approaches, it is also the goal of Theme/UML to utilise the full semantics of standard UML where available, and to only make extensions where required decomposition and composition capabilities are not currently available. For example, we feel that the full power of the UML should be available for interaction specifications.

5.2 Evolving AspectJ

Theme/UML has considerable support for genericity with its use of templates to parameterise crosscutting elements. There are no restrictions (other than on type) as to the elements that may be bound to the templates, except as imposed by the designer—e.g., opera-

tions of any signature could replace template operations. However, within AspectJ, genericity is available only through the provision of abstract elements within aspects; the concretisation of these elements is not as straightforward as binding template parameters, as explicit patching up of differing names and signatures must take place. These abstract elements cannot be constrained by their designers as precisely as templates can be, either.

This lack of specificity as to the types of elements that can be bound to abstract pointcuts leads to a major difference between Theme/UML and AspectJ: in a CP, a template operation can be called directly, but a pointcut can only be advised. This requires either a greater degree of coupling in aspects, as one must possess explicit names to be able to call them, or one must specify interfaces within the aspect that concrete types then implement. The latter choice leads to signature mismatching that must be manually patched up, as was the case with `ObserverI.updateStatus()`. A cleaner interface reconciliation mechanism would be useful.

The extension of Theme/UML to permit the specification of composition with highly dynamic scopes, as mentioned above, is intended to support similar abilities already present in AspectJ. However, AspectJ currently provides only a small set of such dynamic aspect associations. The full expressiveness of these extensions to Theme/UML may not be supportable by this set. Further research is required to assess the significance of these points.

6. RELATED WORK

While we have examined the mapping from a particular AOSD design language (Theme/UML) to a particular AOSD implementation language (AspectJ) in this paper, other possibilities abound in many dimensions.

6.1 Design Approaches

One of the primary contributions of Theme/UML is its capabilities relating to decomposition and modularisation of UML models. The UML [21] itself provides modularisation mechanisms such as packages and subsystems, upon which Theme/UML builds its additional composition capabilities. These are largely related to modularisation and generic composition of crosscutting design elements. Catalysis [8] also supports the decomposition of software designs along "vertical" and "horizontal" lines, providing the ability to separate both functional and technical concerns. Theme/UML provides a more generic approach, including support for both functional separation (like roles) and separation of patterns of crosscutting behaviour.

Collaboration-based design or role modelling is a compositional design approach that concentrates on decomposing designs on the basis of the roles that objects play in particular collaborations [2, 12, 14, 23]. For role modelling within OORam in particular [23], the goals are similar to those motivating separation of non-crosscutting concerns in subject-oriented design [3]. Kendall looked at role modelling and how one might map it to AspectJ [15], concluding that AspectJ did not adequately support a required level of composition for roles (e.g., merge or override).

Other approaches to providing design support for crosscutting concerns appear more tied to the AspectJ model of AOSD exclusively (e.g., [13]).

Theme/UML has taken the more independent route in extending the UML [4] to provide just those constructs required to support the decomposition (and subsequent composition specification) of design models based on requirements specifications. These requirements may be functional or crosscutting, and new design constructs are focused on how to compose the separate models, not on providing constructs to map to any particular implementation paradigm. This approach makes the model more concern centric, not implementation-paradigm centric.

6.2 Implementation Approaches

While we have focussed on only AspectJ in this paper, other compositional implementation languages and mechanisms exist. Multi-dimensional separation of concerns [26] with its associated Hyper/J language [25] arose from the subject-oriented programming paradigm [11] as has Theme/UML. Composition filters [1] are a means of intercepting and rerouting messages as they arrive at objects; they can be used to separate crosscutting concerns such as synchronisation, and have been described as an aspect-oriented technique. Adaptive programming [19] has also been described as a (special case) aspect-oriented technique. It provides a means to separate the algorithms on data from the structure of that data, allowing the structure of the data to change without requiring related changes to the algorithms. Implicit context [29] is a structuring mechanism and design philosophy concentrating on removing knowledge of the large-scale from smaller-scale components; it is related to AOSD. Others have looked to mixins [28] and mixin layers [24] as a means of realising compositional implementations of collaboration-based designs. Mixin layers are useful for product-line architectures, where features are understood from conception to be optional between different configurations of a product.

A design language would need to support most or all of these other approaches to claim to be a candidate for a standard design language for AOSD. In an earlier version of this paper (available as [7]), we examined the mapping from CPs to Hyper/J, with a reasonable level of success. We plan to continue our study of the mesh between Theme/UML and these varied implementation approaches.

6.3 Lifecycle Impact

There has been some recognition of the need for separating crosscutting concerns throughout the lifecycle. For example, Griss has proposed a development process for component-based product-lines that draws together high-level analysis- and design-composition techniques with supporting implementation-composition techniques [10]. But this process does not advise how to map the differing constructs within the combination of approaches that may be used.

The difficulties reported in re-engineering implementations to take advantage of compositional implementation techniques highlights the importance of separating crosscutting concerns across the lifecycle [20]. Being forced to manually untangle and unscatter the concerns that were identified was a difficult and error-prone process; if the systems discussed in that work had been designed with their crosscutting concerns separated in the first place, porting the implementations between the different compositional techniques studied could have been more tractable.

7. CONCLUSIONS

There is a need for a means to separate crosscutting concerns seamlessly across the lifecycle. Such an approach would help realise the benefits of software design by supporting early technical assessment of crosscutting behaviour and the evolution and non-invasive addition of such behaviour to the software artefacts across the lifecycle—e.g., designs and code. To investigate current possibilities to support this need, this paper worked with composition patterns, a part of Theme/UML, at the design level, and with AspectJ at the implementation level.

While a complete, detailed mapping from Theme/UML to AspectJ is beyond the scope of this paper, we have captured the es-

sentials and illustrated a mapping for those. As a result of mapping the Observer composition pattern to AspectJ code, we have identified areas in which both languages could evolve. For Theme/UML, extensions to the binding specifications to templates, in line with AspectJ's flexibility in pointcut specifications, would be useful. Existing, standard UML is, of course, also available to composition pattern designers within the composition pattern package. For example, as with all interaction diagrams, constraints may be defined on the execution of operations. Such constraints could potentially be used to specify useful dynamic aspect associations. For AspectJ, it would be interesting to see if extended genericity capabilities, interface reconciliation mechanisms, and more flexible aspect association specifications (as indicated by this same future work on constraints), were possible.

Just as there are many object-oriented programming languages, there is room for many aspect-oriented programming languages. However, we think that there is a place for a standard AOSD design language that is capable of supporting many of these aspect programming languages. We have identified some goals for such a design language: implementation language independence; design-level composability; and compatibility with the existing standard design language for object-oriented systems, the UML. Here, we have examined the language independence of Theme/UML through its support for AspectJ, identifying areas for improvement in both languages. The other two goals are met by Theme/UML, as discussed elsewhere [3, 4, 5, 6]. We proceed to work on extending Theme/UML to achieve and validate these goals.

8. ACKNOWLEDGMENTS

Our thanks to Martin Robillard, Gail Murphy and the anonymous reviewers for their comments on early drafts of this work.

9. REFERENCES

[1] M. Akşit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *Proc. ECOOP*, pp. 372–395, 1992. LNCS 615.

[2] K. Beck and W. Cunningham. A laboratory for teaching object-oriented thinking. In *Proc. OOPSLA*, pp. 1–6, 1989.

[3] S. Clarke. *Composition of Object-Oriented Software Design Models*. Ph.D. thesis, Dublin City University, 2001.

[4] S. Clarke. Extending standard UML with model composition semantics. *Science of Computer Programming*, 2002. To appear.

[5] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Proc. OOPSLA*, pp. 325–339, 1999.

[6] S. Clarke and R. Walker. Composition patterns: An approach to designing reusable aspects. In *Proc. Int'l Conf. on Software Engineering*, pp. 5–14, 2001.

[7] S. Clarke and R. Walker. Separating crosscutting concerns across the lifecycle: From composition patterns to AspectJ and Hyper/J. Technical Report TCD-CS-2001-15, Department of Computer Science, Trinity College, Dublin, 2001.

[8] D. D'Souza and A. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] M. Griss. Implementing product-line features by composing component aspects. In *Proc. Int'l Software Product Line Conf.*, pp. 271–288, 2000.

[11] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proc. OOPSLA*, pp. 411–428, 1993.

[12] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proc. OOPSLA/ECOOP*, pp. 169–180, 1990.

[13] W.-M. Ho, F. Pennaneac'h, and N. Plouzeau. UMLAUT: A framework for weaving UML-based aspect-oriented designs. In *Proc. Int'l Conf. on Technology of Object-Oriented Languages*, pp. 324–334, 2000.

[14] I. Holland. Specifying reusable components using Contracts. In *Proc. ECOOP*, pp. 287–308, 1992. LNCS 615.

[15] E. Kendall. Role model designs and implementations with aspect-oriented programming. In *Proc. OOPSLA*, pp. 353–369, 1999.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proc. ECOOP*, pp. 327–353, 2001. LNCS 2072.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. ECOOP*, pp. 220–242, 1997. LNCS 1241.

[18] A. Kleppe and J. Warmer. Unification of static and dynamic semantics of UML. Technical report, Klasse Objecten, 2001. `http://www.klasse.nl/english/uml/unification-report.pdf`.

[19] K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[20] G. Murphy, A. Lai, R. Walker, and M. Robillard. Separating features in source code: An exploratory study. In *Proc. Int'l Conf. on Software Engineering*, pp. 275–284, 2001.

[21] Object Management Group. *The Unified Modeling Language Specification, Version 1.3*, 1999.

[22] F. Pennaneac'h, J.-M. Jézéquel, J. Malenfant, and G. Sunyé. UML reflections. In *Metalevel Architectures and Separation of Crosscutting Concerns*, pp. 210–230, 2001. LNCS 2192.

[23] T. Reenskaug, P. Wold, and O. Lehne. *Working with Objects: The OORam Software Engineering Method*. Manning Publications Co., 1995.

[24] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. ECOOP*, pp. 550–570, 1998. LNCS 1445.

[25] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM Research, 2000.

[26] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proc. Int'l Conf. on Software Engineering*, pp. 107–119, 1999.

[27] The AspectJ Team. The AspectJ programming guide. `http://www.aspectj.org/`, 2001.

[28] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proc. OOPSLA*, pp. 359–369, 1996.

[29] R. Walker and G. Murphy. Implicit context: Easing software evolution and reuse. In *Proc. Int'l Symp. on the Foundations of Software Engineering*, pp. 69–78, 2000.

[30] J. Warmer and A. Kleppe. *The Object Constraint Language. Precise Modeling with UML*. Addison-Wesley, 1999.