From Aspectual Requirements to Proof Obligations for Aspect-Oriented Systems

Shmuel Katz[†], Awais Rashid
Computing Department, Lancaster University, Lancaster LA1 4YR, UK
katz@cs.technion.ac.il, awais@comp.lancs.ac.uk

Abstract

Aspect-oriented software development (AOSD) techniques support systematic modularization and composition of crosscutting concerns. Though AOSD techniques have been proposed to handle crosscutting concerns at various stages during the software life cycle, there is a traceability gap between the aspects at the requirements level and those at later development stages. It is not clear what proof obligations about an aspect-oriented implementation follow from the initial aspectual requirements.

This paper presents PROBE, a framework for generation of proof obligations for aspect-oriented systems from the initial aspectual requirements and associated trade-offs. The abstract proof obligations are expressed in standard linear temporal logic. Key components of the framework include an extended Ontology with parametric temporal formulas and functions, and extensive treatment of conflicts among requirements. The resultant temporal logic assertions, grouped into specifications of aspect implementations, can then be instantiated in terms of the implementation and verification tools.

1. Introduction

An unfortunate gap exists between, on the one hand, descriptions of system requirements, and, on the other hand, the actual proof obligations that need to be demonstrated about the implementations of those systems. Refinements of requirements during design and implementation are intended to bridge this gap, but are often too informal to guarantee any connection between validation tasks about the implemented system and the initial system requirements.

Moreover, the modularity achieved by grouping requirements into concerns is vital for complex systems, where a *concern* denotes a coherent set of functional or non-functional elements of a system [9] and not only non-functional requirements as in some RE approaches, e.g., [27, 28]. It is, therefore, essential that clear traceability links between concerns and their associated trade-offs are established and maintained from the requirements level through to design and implementation, hence facilitating validation of the resulting system. In this paper, we focus on such traceability and validation support for the particular class of *crosscutting* concerns tackled by aspect-oriented software development (AOSD) [1].

AOSD techniques provide systematic means for the modularization of broadly scoped, crosscutting concerns, such as security, mobility, real-time constraints and persistence. They also facilitate composition of such modules with other concerns in a system. The initial focus of AOSD has been at the programming level. Consequently, a number of aspect-oriented programming (AOP) [11, 18] techniques have been proposed. These range from language extensions [2] and enterprise platforms [16] to filter-based [3] and traversal-oriented techniques [20] through to multi-dimensional [30] and hybrid approaches [24]. The composition can be done statically, e.g., in AspectJ – an aspect language for Java [2], or dynamically as in Java Aspect Components (JAC) [23] and JBoss [16].

After initial successes at the programming level, the focus of AOSD techniques is now expanding to earlier software development stages. A number of UML extensions have been proposed to support aspect-

[†] On leave from Computer Science Department, The Technion, Haifa, Israel.

oriented modeling and design [7, 8, 14, 17]. The *early aspects* initiative [10] focuses on systematic treatment of crosscutting concerns at the requirements engineering and architecture design levels. In the specific area of aspect-oriented requirements engineering (AORE), [13] proposes a characterization of diverse requirements-level aspects of a system that each component provides to end users or other components. Cosmos [29] offers a schema to model multi-dimensional concern spaces. The AORE approach and the supporting tool ARCADE in [25, 26] build upon the work in [27, 28] to support modularization and composition of aspectual requirements through concern-specific operators. The approach also facilitates establishment and negotiation of trade-offs among requirements-level aspects before the architecture is derived.

Although [25, 26] provide some intuitive indicators of the mapping of requirements-level aspects to later software development stages, there is no systematic means to trace the refinement of aspectual requirements through to an aspect-oriented design and implementation. Nor is there any means to confirm that the trade-offs established among requirements-level aspects are preserved and respected by the implementation. Consequently, despite the homogeneity of handling crosscutting concerns at all stages from requirements through to implementation by the use of AOSD techniques, it is not possible to validate an implementation against the initial aspectual requirements and their trade-offs. The problem is further compounded by the fact that aspects at the requirements level do not necessarily map onto aspects at the implementation level. In certain cases they may map onto a conventional object or a decision (e.g., for architecture choice) [25, 26].

In this paper, we present PROBE, a framework for generation of proof obligations for aspect-oriented systems from the initial aspectual requirements and associated trade-offs. The framework takes into account the refinement and mapping of aspectual requirements and their trade-offs onto the design and implementation, thereby facilitating traceability while providing essential validation support. The proof obligations in PROBE are expressed in standard linear temporal logic [21]. These temporal logic assertions can then be used as an input to formal methods tools, e.g., model-checkers [5] or deductive proof systems, or in the specification and generation of test cases.

Section 2 in this paper introduces the AORE model proposed in [25] which is used for requirements specification in PROBE. Section 3 introduces the framework and Section 4 focuses on the key task of generation of temporal logic assertions, using an extended ontology of generic temporal formulas. Section 5 treats conflict analysis and the implications for proof obligations and traceability of conflicts among requirements. Section 6 discusses the integration of obligations derived from requirements and design as well as their instantiation for an implemented system. Section 7 summarizes the proof obligation framework. Section 8 discusses some related work while Section 9 concludes the paper.

2. Aspect-Oriented RE

Requirements specifications are traditionally given in natural language, where the restriction on the description is minimal, thus avoiding over-commitment to any particular architecture or design. The requirements engineering model proposed in [25] maintains the use of natural language for the specification but provides a semi-structured framework based on the eXtensible Markup Language (XML) to group requirements and constraints in a fashion appropriate for systematic management of crosscutting concerns and their trade-offs. The model has been instantiated for viewpoint-based requirements engineering [12]. Note that the model is generic and can be instantiated for other requirements-level separation of concerns mechanisms such as use cases [15] or goals [19]. In the viewpoint-based instantiation, a requirements document has three types of basic modules:

- *Viewpoints* encapsulating the stakeholders' requirements;
- *Aspects* encapsulating the requirements pertaining to a broadly scoped concern that cuts across several viewpoints;
- *Composition rules* employing informal yet semantically meaningful, and often concern-specific, actions and operators to specify how an aspectual requirement influences or constrains the behavior of a set of viewpoint requirements (note that such a set spans multiple viewpoints).

The instantiation of the model is supported by the Aspectual Requirements Composition and Decision Support Tool (ARCADE), which automates the validation of relationships specified by the composition rules and identification of interaction and trade-off points among the aspects.

We discuss the modules introduced above in more detail using fragments from an example from [25], which will also form the basis for demonstrating the

generated proof obligations described in later sections. The example treats the toll collection system on Portuguese highways, where drivers wishing to be charged automatically at a toll gate register with the system through an ATM and obtain a gizmo to be installed in the windshield. Each time an authorized car passes a toll gate, a green light is shown and the amount to be debited from the driver's registered account is displayed. If an unauthorized vehicle passes the toll gate, a yellow light is shown, an alarm is sounded and a photograph of the number plate is taken to be passed on to the police.

Figs. 1 and 2 show example aspects from the requirements specification for this system namely, *Availability* and *ResponseTime*. Each aspect definition is enclosed in *<Aspect> </Aspect>* tags while each requirement description is enclosed in *<Requirement> </Requirement>* tags. An *<Aspect>* tag has an attribute *name* used to specify the name of the aspect while each *<Requirement>* tag has an attribute *id* uniquely identifying the requirement within its defining scope, i.e., the aspect. These descriptions typically use natural language to identify the concern, and list conditions or key events for which the aspect must apply. Clearly, the term "react in time" in the main requirement for *ResponseTime* needs to be defined in an ontology or elsewhere. We will see that it implies that the key actions listed are restricted by bounding events and real-time limitations, and must have specified effects. Other aspects in the system are *Correctness* (dealing with the preciseness of the toll fee calculations), and *Compatibility* (requiring interoperability with external modules). *Security* (dealing with privacy of data) is another common aspect not in the original example, which easily could be added as a new concern.

Viewpoints, such as *Vehicle* (cf. Fig. 3) or *PayingToll* (not shown here), have a syntactic structure similar to that of aspects, but with definitions enclosed in *<Viewpoint> </Viewpoint>* tags. They generally define the basic required functionality of the viewpoint, and could have constraints of their own, although often these are left to the aspects.

```xml
<?xml version="1.0" ?>
<Aspect name="Availability">
    <Requirement id="1"> The system must be available for:
        <Requirement id="1.1">reacting to
        stimuli;</Requirement>
        <Requirement id="1.2">data exchange;</Requirement>
        <Requirement id="1.3">updates.</Requirement>
    </Requirement>
</Aspect>
```

Fig. 1: The *Availability* aspect

```xml
<?xml version="1.0" ?>
<Aspect name="ResponseTime">
    <Requirement id="1"> The system needs to react in-time
    in order to:
        <Requirement id="1.1">read the gizmo
        identifier;</Requirement>
        <Requirement id="1.2">turn on the light (to green or
        yellow);</Requirement>
        <Requirement id="1.3">display the amount to be
        paid;</Requirement>
        <Requirement id="1.4">photograph the plate number
        from the rear;</Requirement>
        <Requirement id="1.5">sound the
        alarm;</Requirement>
        <Requirement id="1.6">respond to gizmo activation
        and reactivation.</Requirement>
    </Requirement>
</Aspect>
```

Fig. 2: The *ResponseTime* aspect

```xml
<?xml version="1.0" ?>
<Viewpoint name="Vehicle">
    <Requirement id="1">The vehicle enters the system when
    it is within ten meters of the toll gate.</Requirement>
    <Requirement id="2">The vehicle enters the toll
    gate.</Requirement>
    <Requirement id="3">The vehicle leaves the toll
    gate.</Requirement>
    <Requirement id="4">The vehicle leaves the system when
    it is twenty meters away from the toll
    gate.</Requirement>
……
</Viewpoint>
```

Fig. 3: The *Vehicle* viewpoint

Fig. 4 shows a few of the composition rules (enclosed in *<Composition> </Composition>* tags) for the *ResponseTime* aspect. Note that each composition rule in the figure uses the *<Constraint>* tag and associated concern-specific actions and operators to specify how an aspectual requirement constrains the behavior of a set of viewpoint requirements. The *<Requirement>* tags in composition rules differ from those in aspect and viewpoint definitions in that they also include the defining scope as an attribute along with the requirement *id* (this is essential to identify a specific requirement). Furthermore, sub-requirements, if present, have to be explicitly included or excluded using the *children* attribute. The *<Outcome>* tag defines the result of constraining the viewpoint requirements with an aspectual requirement. The action value describes whether another viewpoint requirement or a set of viewpoint requirements must be *satisfied* (i.e., some requirements elsewhere in the system are dependent upon the application of the aspect to the requirement in question) or merely the constraint

specified has to be *fulfilled* (i.e., it is not linked to other requirements).

```xml
<?xml version="1.0" ?>
<Composition>
  <Requirement aspect="ResponseTime" id="1.1">
    <Constraint action="enforce" operator="between">
      <Requirement viewpoint="Vehicle" id="1" />
      <Requirement viewpoint="Vehicle" id="2" />
    </Constraint>
    <Outcome action="satisfied">
      <Requirement viewpoint="Gizmo" id="1"
      children="include" />
    </Outcome>
  </Requirement>
  <Requirement aspect="ResponseTime" id="1.2">
    <Constraint action="enforce" operator="between">
      <Requirement viewpoint="Gizmo" id="1"
      children="include" />
      <Requirement viewpoint="Vehicle" id="3" />
    </Constraint>
    <Outcome action="satisfied" operator="XOR">
      <Requirement viewpoint="PayingToll" id="1" />
      <Requirement viewpoint="PayingToll" id="2" />
    </Outcome>
  </Requirement>
  ……
</Composition>
```

Fig. 4: Composition rules for the *ResponseTime* aspect

The first composition rule connects the event associated with the requirement 1.1 of *ResponseTime* (i.e., reading the gizmo) with the bounding events seen in requirements 1 and 2 of *Vehicle* (i.e., the vehicle entering the system, and entering the toll gate, respectively). The effect connects to the *Gizmo* viewpoint (not shown) that includes subactivities of reading the gizmo. The second composition rule connects the event of turning on the light, seen in *ResponseTime* requirement 1.2 to the *Gizmo* event 1 (i.e, that the gizmo is read) and that from *Vehicle* 3 (i.e., that the vehicle leaves the tollgate), and to the effect of the events from the *PayingToll* viewpoint (also not shown) of turning on either the green or the yellow light.

The composition rules are used to compose the aspects with the viewpoints. During composition, each aspect's influences on the viewpoints are clear via the composition rules. These influences are compared against those of other aspects in order to identify potential trade-off points – where two aspects directly or indirectly influence the same set of viewpoint requirements. Provided two or more aspects contribute negatively to each other with respect to a potential trade-off point, negotiations are carried out amongst stakeholders following which fuzzy-value based priorities are assigned to aspects. This results in a contribution matrix, which is also part of the requirements specification.

We further explain the semantics of the relevant composition operators and actions in later sections. Interested readers are referred to [25] for a full description of the AORE model, the semantics of the operators and actions, and the toll gate case study.

3. Proof Obligation Framework

The input to the proof obligation framework is thus an ARCADE specification (available from the AORE tool) with viewpoint requirements, aspectual requirements, and composition rules, as well as contribution matrices that identify potential conflicts, and prioritize requirements in conflict according to a ranking function. These need to be augmented during the design phase, by either refining the requirements or replacing them with requirements that follow from the design, say in an extended UML that handles aspects, e.g., as in [7, 8, 17].

The final result we seek should have a collection of temporal logic assertions for each implementation of an aspect, instantiated in terms of the classes over which it is applied (where these often correspond to viewpoints). Moreover, as will be explained, there should be assertions that relate to conflicts among aspects which potentially could interfere with each other, and assertions that the implementation of the aspect does not invalidate any of the specification of the classes over which it is applied (derived from the viewpoint requirements).

The PROBE framework is shown in Fig. 5. Note that the XML requirements specification from ARCADE and UML design with aspect-oriented extensions as in [17] form the initial inputs to PROBE, while at later stages user-provided mappings and the implemented code are provided. The modules of PROBE include:

- ***Temporal Logic Generator***, which parses the XML representation of the aspectual requirements, viewpoint requirements and composition rules in the ARCADE specification, and uses it to generate individual temporal formulas which, in turn, are grouped into aspect proof obligations;
- ***UML Analyzer***, which determines separate proof obligations by analyzing the aspect-oriented UML design of the system;
- ***Conflict Analysis*** module, which generates proof obligations from the aspect trade-offs and conflicts from the ARCADE specification;

- ***Ontology***, which provides predefined generic temporal formulas or state functions, and records predicate names used to connect to the implementation;
- ***Integration*** module, which integrates the outputs of the previous modules to generate groupings of temporal logic assertions, in terms of predicate names from the ontology
- ***Instantiation*** module, which generates the final concrete proof obligations for the implementation.

All of the modules except the ***UML Analyzer*** are considered in greater detail in the following sections.

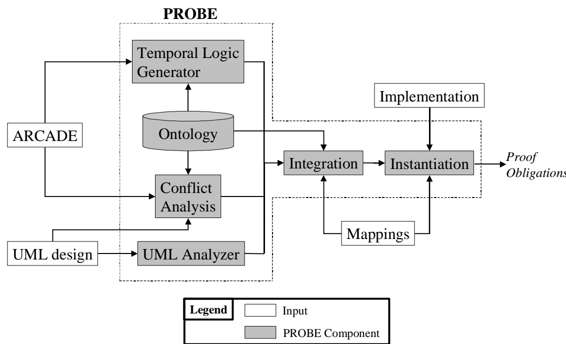

Fig. 5: The PROBE Framework

The ***UML Analyzer*** module of PROBE analyzes design components for both classes and aspects in UML, and generates temporal assertions for them. It is based on the extensions to UML seen in [17] to incorporate designs of aspects as augmentations to any of the usual UML model notations, and to identify their common sub-aspects in Conflict Diagrams. This analysis is done independently from the analysis of the ARCADE requirements descriptions, to guarantee that connections are not inferred without justification. Since this module does not directly relate to requirements descriptions, it is not further described here.

4. Generating Temporal Assertions

4.1 Individual requirements in Temporal Logic Generator

In expressing the individual requirements, linear temporal logic is used to express restrictions on possible sequences of states of the system. Here we use the unary modal operators “F”, and “G”, and the binary “U”:

- “F” represents the future, or eventually modality, asserting that the following predicate is true in some future state;
- “G” represents the “always” or “from now on” modality indicating that the following predicate should be true in every state;
- “U” represents a strong Until operator in which the predicate on the left must remain true until the one on the right becomes true and moreover the predicate on the right must become true.

An assertion without temporal modalities is understood to be true in the first state of an execution sequence. Such assertions about a system are interpreted to require that every possible sequence of states that can occur as an execution of the system is satisfied by the assertion.

The generation of temporal logic formulas from natural language descriptions of systems has been treated in [22], where a temporal logic analyzer is added to a natural language understanding system, to generate temporal logic expressions of natural language descriptions. Here, because the semi-structured ARCADE notation is used, many of the keywords can be associated with pre-determined patterns of temporal formulas. The general natural language analysis is only used for the non-structured sentence fragments. For example, the composition rule for *ResponseTime* uses a keyword “between” with two arguments, say E1 and E2, and is applied to key events, say E. This means that E must occur between E1 and E2. With the given temporal modalities, this could be expressed as the pattern

$$G(E1 \rightarrow ((F\ E2) \wedge ((\neg E2)\ U\ E)))$$

That is, throughout any execution sequence, whenever E1 occurs, then eventually E2 will occur, but E2 will not occur until E has occurred.

Use of such patterns can give better results than general natural language analysis, especially in conjunction with the extended ontology, discussed below. The ***Temporal Logic Generator*** module identifies any keywords treated by the ontology, and uses the ***Ontology*** module to extract the parametric temporal assertions associated with the keyword. It also generates predicates and functions associated with the requirements, that later are connected to states or events of the implementation. These are substituted in place of parameters in the temporal assertions, and retained in the Ontology. Thus from the *Availability* aspect, *react-to-stimuli*, *data-exchange*, and *update* predicates are defined, that will be true when the associated events occur in the implementation.

4.2 Extended Ontology

The ontology of the natural language terms that identify a concern has proof implications. A usual ontology defines terminology used in expressing requirements in natural language, to guarantee that clients, requirements engineers, and system designers have a common understanding of the terms. Here, we use an extended ontology to provide generic (parametric) temporal logic assertions. These include state-predicates that also must be instantiated whenever the terminology is used.

These assertions provide a semantics for the terminology, and especially for aspects such as *Availability*, *Correctness,* and *ResponseTime*, as well as possible later additions such as *Security*. The assertions give those terms a meaning in terms of what is to be proven about the implemented system with this aspect, as opposed to a system without it.

Just as the aspect implementation code has parameters that bind it to the system to which the aspect is applied, the proof obligations of the ontology must be related to the particular system under development by binding parameters and defining predicates in terms of the system.

The (Extended) ***Ontology*** should be prepared in advance (independently of any specific application system) to connect terminology to collections of parametric assertions. Of course, this component should be extendable, to enable inclusion of new terminology as it is needed.

We consider the *ResponseTime* aspect. "Response time" is a concern that might be defined in a usual ontology as "showing that specified activities or events occur within given periods, including a time bound." When identified as an aspect in an ARCADE requirements document, the key activities or events must be listed in the aspect requirements, and then expanded in the composition rules to a link with viewpoints, where the definitions of the relevant periods must be given. In order to associate appropriate proof obligations, this intuitive definition must be expressed using the linear temporal logic notation.

As seen earlier, the composition rules for *Response Time* have the form that "key event E must be treated between event E1 and event E2," (where each event is a parameter to be later instantiated as seen above). They also have an outcome action, represented by event E3 that is "caused" by the key event, and an implied real-time restriction between the bounding events. The actual elapsed real-time requirement can be expressed by assuming a function *time*(E) that returns a real value of the nearest time at which some event E occurs[1]. Thus an assertion *time*(E2) – *time*(E1) < N means that E2 occurs within N time units of E1. The ontology would have this real-time bound, the connection between E and E3, and the previously given formula defining the bounds using "between" as proof obligations associated with *ResponseTime*. That is, the formula would be

$$G((E1 \rightarrow ((F\ E2) \wedge ((\neg E2)\ U\ E))) \wedge (E \rightarrow E3) \wedge (\ time(E2) - time(E1) < N\ ))$$

As seen in Section 4.1, the ***Temporal Logic Generator*** would use this formula with predicate names like *read-gizmo* substituted, and add these names to the Ontology for later association with implementation events.

5. Conflicts from ARCADE Requirements

An analysis of the individual aspect requirements, as well as the contribution matrix seen in the ARCADE methodology can be used to detect potential conflicts. In the notation, for each aspect there is a collection of aspect requirements given directly or through the ontology. These often relate to activities of the system. Composition rules for the aspects then connect these activities to viewpoint requirements, thereby restricting particular requirements of the viewpoint. When overlapping sets of viewpoint requirements are constrained by two or more aspects, the aspects are potentially in conflict. Often, the aspect development process recognizes such conflicts and resolves them before implementation, by changing the requirements.

However, in some cases they remain in the final design and implemented system. According to the ARCADE methodology, this can occur when the conflicts have been "resolved" either by (1) showing they do not interfere with each other, or (2) preferring one over another when they are in real conflict. The first possibility for resolving conflicts is to show that they are actually compatible in the particular context in which they are to be applied (or at least do not conflict in a way which is "crucial to correctness"). In some cases it is sufficient to recognize that the kinds of system augmentation required by each involve distinct parts of the viewpoint, and do not interact. On the other hand, when the conflicts in requirements from different aspects are genuine, the problem can be resolved by preferring one over the other. In ARCADE, this is done

[1] This is only one of the common ways in which real-time requirements can be expressed in temporal logic, but is sufficient for purposes of demonstration.

by ranking the requirements to indicate the importance of each aspect requirement relative to each viewpoint.

5.1 Conflicts and interference-freedom

In terms of proof obligations, apparent conflicts are eliminated between aspects if the implementations of the aspects do not *interfere* with each other. An aspect *A* interferes with an aspect *B* if *B* satisfies its specification when *A* is absent, but does not satisfy its specification if *A* has also been applied to the system. Note that interference is asymmetric: *A* interfering with *B* does not mean that *B* interferes with *A*. A system is *interference-free* if none of the aspects suffer from interference. This definition is general in the sense that even aspects that change the same fields or variables might not interfere with each other.

A proof technique for interference used in verification of shared memory parallelism using threads can also be used here: an aspect *A* interferes with another *B* by invalidating the reasoning needed to justify the correctness of *B*. Consider a situation where both *A* and *B* are implemented by code-level (say AspectJ) aspects *a* and *b*, respectively, applied over classes that implement the viewpoints with overlapping requirements. The implemented versions *a* and *b* are interference free if the proof that *a* adds the requirements of *A* to the underlying requirements is still valid when *b* is also added, and vice versa. Thus for aspects like security and availability, even when they each influence the same classes, if security is achieved by encoding all information as received, and decoding it only in secure environments, while availability is achieved by internally duplicating all processing in back-up servers, the validity of each is not invalidated by the additional application of the other. On the other hand, for availability and response-time, the additional duplication and message load caused by the solution for availability very well could make the system not achieve the response-time requirements, even when it did before availability was added as an aspect.

Note that even when the system does not maintain the implementation of a requirements-level aspect as a separate code-level aspect, the above considerations hold of the system either with or without the requirements-level aspect.

5.2 Conflicts and weakening requirements

When aspects do interfere with each other, one solution is to change the level of importance of some requirements, by indicating that one requirement has priority over another. In ARCADE, priority is established by ranking the importance of each aspect requirement relative to each viewpoint requirement. When requirement *i* of A conflicts with *j* of B, and *i* is ranked higher, then *i* has priority and *j* should be weakened. Expressed more formally, when *i* has priority over *j* then we require

$$i \wedge (j \vee ((i \Rightarrow \neg j) \Rightarrow weakened(j))).$$

That is, we require *i* and moreover either *j* should hold, or if *i* and *j* cannot both be true, then a weakened *j* should hold. Here we assume that *weakened(j)* cannot contradict *i*. Of course, variants are possible where *i* is also weakened (presumably less than *j*).

A weakened version of such requirements must be defined, either in advance in the ontology, or as the need arises. Consider a response time example, where event E should occur between events E1 and E2, and E2 should occur within N time units of E1. Then in one possible weakened version, E should still occur between E1 and E2, but E2 can now occur within N+D units of E1, where D is some reasonable delay time. The definition of *weakened(ResponseTime)* then becomes

$$G((E1 \Rightarrow ((F\ E2) \wedge ((\neg E2)\ U\ E))) \wedge (E \Rightarrow E3) \wedge (time(E2) - time(E1) < N + D))$$

In another possible definition of weakening, E could occur after E2, but no more than D time units later. The associated formula is then

$$G((E1 \Rightarrow (F\ E2)) \wedge (E \Rightarrow E3) \wedge (time(E2)-time(E1) < N) \wedge (time(E) - time(E2) < D)).$$

(Note that the new real-time clause makes the Until clause extraneous.) In yet another option, E might not occur at all if the requirement cannot be otherwise achieved. In this and many other examples, we assume that as little weakening as possible is applied, so that the strongest weakening possible is used (e.g., the smallest possible value for the delay time D).

It should be noted that conflicts among requirements from different aspects are actually cross-cutting in terms of the aspects, and do not "belong" to the requirements of any one aspect. The proof obligations that arise from the conflicts need to be refined by a precise definition of weakening when it is needed. Although not the subject of this paper, such an obligation can be proven by deductive methods analyzing the code of the aspects and their specifications. Alternatively, software model checking can be applied to a version of the system with all needed aspects woven in. If all requirements are not satisfied, it is valuable for debugging to determine

whether the reason is interference among aspects, by applying them one at a time.

5.3 Conflicts between Aspects and Viewpoints

Note also that conflicts may arise between the original requirements of the viewpoints and those added by the aspects. Although the intention is to add new requirements, there are situations where the aspect requirements change those of the viewpoints. Often this can be shown to not be a problem: a requirement of availability could be satisfied through system duplication, but usually would not influence the original viewpoint requirements.

On the other hand, an aspect to treat overflow of variables might disturb some invariant relations that previously held among the variables of the system. In a more typical situation, where the viewpoint merely lists expected functionality, an implementation might have a class with methods that correspond to the needed events. Yet an aspect such as security could conflict with the basic assertion that all of the functionality is implemented, by preventing methods from being called that violate new security requirements. Such conflicts should usually be dealt with at the requirements stage, by weakening either the viewpoint or aspect requirement when they are combined, just as for two aspects.

Even when weakening has been applied, an implementation of an aspect could inadvertently violate a viewpoint requirement of the original system. Therefore, in general, a default proof obligation is added that implementations of aspects do not invalidate system properties that follow from viewpoint requirements. This is clearly so if the entire augmented system is verified for all required properties. However, a modular proof is often desirable, where the original system is verified without some aspects and we would like to show that added aspect code now satisfies both the original requirements and the new ones of the aspect without a total reproof. The implementation of the aspects must then be shown both to add the new requirements of the aspect, and to "do no harm", i.e., not to invalidate any of the original system requirements.

6. Integration and Instantiation

The ***Integration*** module must determine the mappings from the requirements level aspects to their refinements at the design level. Additional user input will often be needed for this task. Moreover, this module must merge the temporal properties from the requirements level with those from the design level for each aspect. Since here we concentrate on the proof obligations from requirements, we only mention some issues to be treated in this module. In particular, this module should analyze the assertions to determine when those from the design replace assertions from requirements level statements (and imply the requirements will be satisfied), and when the assertions from both sources must appear in an aspect specification for an implemented system.

Sometimes a design decision implies satisfaction of a requirement. For example, the *Availability* aspect could be satisfied by a design where system components are duplicated (e.g., a backup server constantly updated, that would replace the main server in case of hardware downtime). The original availability requirements are thus discharged by the design, leaving only the basic requirement that this design is actually implemented. As another example, if an aspect requiring *Security* were added to our system, it could lead to a design where information is encrypted before being transmitted from a secure class, and decrypted upon receipt in another secure class. The requirement of information not being readable outside secure classes is then replaced by a specification that the implemented system follows the encryption policy of the design.

Because of the limited expressiveness of most design notations, and particularly of UML, there are many requirements that are not fulfilled explicitly from a design, and must be passed forward to concrete proof obligations. As a simple example, a global eventuality property, for example that every request is eventually answered, or those seen in the *ResponseTime* aspect, cannot be expressed in any UML notation.

Finally, the ***Instantiation*** module must express the temporal logic assertions in the notation of whatever tool is to be used to analyze the implemented system, and must instantiate the predicates to relate to implementation events. Typical implementation events could be expressed as location predicates true whenever a label is reached, method calls, or changes in fields expressible as AspectJ join points, or even dynamic joinpoints that require richer notations than AspectJ. The final result is a collection of concrete proof obligations about the implemented aspects.

Note that only at this stage does the precise implementation language, and the tool used for verification have to be considered. In this sense it is the "back-end" of the PROBE system, and like compiler back-ends, different versions can be provided for each language and verification tool.

7. Summarizing the Proof Obligations

We can summarize the proof obligations generated from ARCADE requirements.

1. Globally, all aspect implementations are asserted to not violate any class property derived from a viewpoint requirement.
2. All aspect implementations are asserted to not interfere with (possibly weakened versions of) requirements associated with other aspects.
3. For each aspect, the implementation of that aspect is asserted to satisfy the (possibly weakened) parametric formulas from the aspect requirements, for every possible instantiation of the parameters.

Actually, in a particular system, the aspect is asserted to satisfy the collection of instances of the formulas obtained by analyzing the requirements from the Composition rules connecting the aspect to the system, and substituting the appropriate event predicates (true when the event occurs in a system state).

For example, the implementation of the *ResponseTime* aspect is asserted to satisfy an instance of the parametric formulas from the ontology given earlier, assuming no weakening is required, for each requirement in the appropriate composition rules, using the state predicates mentioned earlier. Thus, the first requirement in the composition rules in Fig. 4 corresponds to

$$G((\textit{veh-ent-sys} \rightarrow ((F\,\textit{veh-ent-toll}) \wedge ((\neg \textit{veh-ent-toll})\,U\,\textit{read-gizmo}))) \wedge (\textit{read-gizmo} \rightarrow \textit{gizmo-effects}) \wedge (\textit{time}(\textit{veh-ent-toll}) - \textit{time}(\textit{veh-ent-sys}) < N))$$

Here *veh-ent-sys* (replacing E1) is true in states where the vehicle has just entered the system by being sensed on the road, and *veh-ent-toll* (E2) is true when the actual toll gate is reached by the vehicle. The key event *read-gizmo* (E) is true when the system has just read the gizmo, and *gizmo-effects* (E3) expresses the internal actions when the gizmo has been read, as required in the Gizmo requirements.

The remaining assertions are similarly generated by substitution of the event-predicate associated with the restrictions. If the second composition rule requirement in Figure 4 were weakened due to a conflict with *Availability*, using the first possibility for weakening in Section 5.2, then substitution of predicates yields

$$G((\textit{read-gizmo} \rightarrow ((F\,\textit{veh-leave-toll}) \wedge ((\neg \textit{veh-leave-toll})\,U\,\textit{turn-on-light}))) \wedge (\textit{turn-on-light} \rightarrow (\textit{green-on} \vee \textit{yellow-on}) \wedge (\textit{time}(\textit{veh-leave-toll}) - \textit{time}(\textit{read-gizmo}) < N + D))$$

Note that E3 is replaced by a disjunction, reflecting that a green or a yellow light may be lit, depending on whether the gizmo effects include proper payment.

8. Related Work

[6] uses design elements based on partial subjective views of the design as a means to support alignment and hence traceability of requirements through to implementation. The approach requires use of the subject-oriented approach at design and implementation level. In contrast, the PROBE framework facilitates generation of proof obligations for validation of any aspect-oriented implementation. It also accounts for the fact that requirements, design and implementation modules are not necessarily in alignment as a requirements level aspect may or may not map onto an implementation level aspect. Moreover, PROBE preserves traceability of both the aspectual requirements and also their associated trade-offs.

The PROBE framework also relates to goal-oriented approaches to requirements engineering [4, 19], which facilitate traceability of system goals and trade-offs from requirements through to implementation. Goals are refined to sub-goals, alternative trade-offs are explored, concrete requirements are identified and consistency verified. In a similar fashion, PROBE facilitates traceability of broadly-scoped concerns and their trade-offs, with the resulting proof obligations providing integration with practical formal methods tools such as model-checkers for verification purposes. Since the RE approach used in PROBE can be instantiated to goal-oriented techniques [25] the framework may be used to validate an implementation against the specified goals of the stakeholders especially whether any soft goals have been satisfied within acceptable limits.

9. Conclusion

The PROBE framework connects aspect-oriented requirements to proof obligations in temporal logic that should hold for implemented aspects of the system. This both provides a precise semantics for the requirements and facilitates effective use of formal methods tools and test case generators.

The key elements of the framework are the use of an extended ontology with parametric temporal logic formulas, state predicates and functions, and the treatment of conflicts identified at the requirements stage. This includes the definition of a weakened requirement when during requirements definition its

priority is asserted to be lower than another, more crucial, requirement. PROBE facilitates tracing aspect refinements and trade-offs through to implementation in AOP. The framework thus is a significant step towards clarifying the precise connections among aspect-oriented requirements, design, and implementation, connections that previously have been largely unexplored.

Acknowledgements

This work is supported by EPSRC Grant GR/S70159/01. The work on AORE and ARCADE was carried out in collaboration with Ana Moreira and Joao Araujo at Universidade Nova de Lisboa, Portugal. The authors wish to thank Peter Sawyer at Lancaster for helpful discussions.

References

[1] AOSD, http://aosd.net, 2004

[2] AspectJ, http://www.eclipse.org/aspectj/, 2004

[3] L. Bergmans and M. Aksit, "Composing Crosscutting Concerns using Composition Filters", *CACM*, 44(10), pp. 51-57, 2001.

[4] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*: Kluwer, 2000.

[5] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*: MIT Press, 1999.

[6] S. Clarke, W. Harrison, H. Ossher, and P. L. Tarr, "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code", Proc. OOPSLA, 1999, ACM, pp. 325-339.

[7] S. Clarke and R. J. Walker, "Composition Patterns: An Approach to Designing Reusable Aspects", Proc. ICSE, 2001.

[8] S. Clarke and R. J. Walker, "Towards a Standard Design Language for AOSD", Proc. AOSD, 2002, ACM, pp. 113 - 119.

[9] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice Hall, 1976.

[10] EarlyAspects, "Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design", http://early-aspects.net, 2004.

[11] T. Elrad, R. Filman, and A. Bader (Eds.), "Theme Section on Aspect-Oriented Programming", *CACM*, 44(10), 2001.

[12] A. Finkelstein and I. Sommerville, "The Viewpoints FAQ." *BCS/IEE Software Engineering Journal*, 11(1), 1996.

[13] J. Grundy, "Aspect-Oriented Requirements Engineering for Component-based Software Systems", 4th IEEE Int'l Symp. on RE, 1999, IEEE Computer Society Press, pp. 84-91.

[14] J. Grundy, "Multi-perspective specification, design and implementation of software components using aspects", *International Journal of Software Engineering and Knowledge Engineering*, 20(6), 2000.

[15] I. Jacobson, *Object-Oriented Software Engineering - a Use Case Driven Approach*: Addison-Wesley, 1992.

[16] JBoss AOP Webpage, from http://www.jboss.org/, 2004.

[17] M. Katara and S. Katz, "Architectural Views of Aspects", Proc. AOSD, 2003, ACM, pp. 1-10.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming", Proc. ECOOP, 1997, Springer-Verlag, LNCS 1241, pp. 220-242.

[19] A. Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour", 5th Int'l. Symp. on Requirements Engineering, 2001, IEEE Computer Society Press, pp. 249-261.

[20] K. J. Lieberherr, D. Orleans, and J. Ovlinger, "Aspect-Oriented Programming with Adaptive Methods", *CACM*, 44(10), pp. 39-41, 2001.

[21] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*: Springer-Verlag, 1991.

[22] R. Nelkin and N. Francez, "Automatic Translation of Natural Language System Specifications into Temporal Logic", Proc. CAV, 1997.

[23] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin, "JAC: A Flexible Solution for Aspect-Oriented Programming in Java", Proc. Reflection Conf., 2001, Springer-Verlag, LNCS 2192, pp. 1-25.

[24] A. Rashid, "A Hybrid Approach to Separation of Concerns: The Story of SADES", Proc. Reflection Conf., 2001, Springer-Verlag, LNCS 2192, pp. 231-249.

[25] A. Rashid, A. Moreira, and J. Araujo, "Modularisation and Composition of Aspectual Requirements", Proc. AOSD, 2003, ACM, pp. 11-20.

[26] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo, "Early Aspects: A Model for Aspect-Oriented Requirements Engineering", Proc. RE, 2002, IEEE Computer Society Press, pp. 199-202.

[27] I. Sommerville and P. Sawyer, *Requirements Engineering - A Good Practice Guide*: John Wiley and Sons, 1997.

[28] I. Sommerville, P. Sawyer, and S. Viller, "Managing Process Inconsistency using Viewpoints", *IEEE Trans. on Software Engineering*, 25 (6), 1999.

[29] S. M. Sutton and I. Rouvellou, "Modeling of Software Concerns in Cosmos", Proc. AOSD, 2002, ACM, pp. 127-133.

[30] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns", Proc. ICSE, 1999, ACM, pp. 107-119.