PROBE: From Requirements and Design 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*

Computing Department, Lancaster University, Technical Report Number: COMP-002-2004

Abstract

Aspect-oriented software development (AOSD) techniques support the systematic modularization and composition of crosscutting concerns, the so-called "aspects". Though AOSD techniques have been proposed to handle crosscutting concerns at various stages during the software life cycle, there are gaps between the aspects at the requirements level, those at the design level, and those implemented at later development stages. It is not clear what proof obligations about an aspect-oriented implementation follow from the initial aspectual requirements, or from design elements given in UML. This validation problem is further compounded by the lack of traceability of aspectual requirements and their associated trade-offs through to subsequent design and implementation-level refinements.

This paper presents PROBE, a framework for generating proof obligations for aspect-oriented systems from the initial aspectual requirements and associated trade-offs, as well as from designs of aspects. 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 proof obligations generated from designs of aspects are integrated with those from the requirements stage. The resultant temporal logic assertions, grouped into specifications of aspect implementations, can then be instantiated in terms of the implementation and intended verification tool. The result provides input to formal methods tools such as model-checkers, or can be used in the specification and generation of test cases.

1. Introduction

An unfortunate gap exists between, on the one hand, descriptions of system requirements and designs, 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., [31, 32]. 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

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

the particular class of crosscutting concerns, tackled by the emerging aspect-oriented software development (AOSD) [1] techniques.

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, 22] techniques have been proposed. These range from language extensions [2] and enterprise platforms [19] to filter-based [3] and traversal-oriented techniques [24] through to multi-dimensional [34] and hybrid approaches [28]. The composition can be done statically, e.g., in AspectJ – an aspect language for Java [2], or dynamically as in Java Aspect Components (JAC) [27] and JBoss [19].

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-oriented modeling and design [7, 8, 15, 20]. 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), [14] proposes a characterization of diverse requirements-level aspects of a system that each component provides to end users or other components. Cosmos [33] offers a schema to model multi-dimensional concern spaces. The supporting tool ARCADE, seen in [29, 30], is built upon the work in [31, 32] 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 [29, 30] 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) [29, 30].

In this paper, we present PROBE, a framework for generating proof obligations of aspect-oriented systems from the initial aspectual requirements and associated trade-offs, as well as from elements of the design. 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 [25]. 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 [29], which is used for requirements specification in PROBE, and the UML extensions used to express aspects and their interactions with the rest of the system at the design level. Section 3 introduces the framework itself and explains its components. Then Section 4 focuses on the key task of generating 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 explains the proof generation process for the designs of aspects, while Section 7 treats the integration of design-level obligations with obligations from the requirements stage, demonstrates some proof obligations generated for one of the aspects, and explains the final instantiation of the proof obligations for an implemented

system. Section 8 discusses some related work, Section 9 outlines an implementation plan for PROBE, and Section 10 summarizes the approach.

2. Aspect-Oriented Requirements Engineering and Design

2.1 Requirement descriptions in ARCADE

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 at too early a stage. The requirements engineering model proposed in [29] 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 [18] or goals [23]. In the viewpoint-based instantiation, a requirements document can have 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. Once the trade-off points are identified, provided two or more aspects contribute negatively to each other with respect to a specific 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 discuss the modules introduced above in more detail using an example, which will also form the basis for demonstrating the generated proof obligations described in later sections. We have chosen to use fragments from an example in [29], i.e., the toll collection system on Portuguese highways. In this system, 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 onto 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. Note that the defining scope for aspect names is the specification itself. 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. However, their definitions are enclosed in *<Viewpoint>* *</Viewpoint>* tags and the defining scope of requirement *id*s is the particular viewpoint. 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 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 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 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* (so that the fulfillment of the new requirement is dependent on already existing ones) or merely the constraint specified has to be *fulfilled* (and is not linked to other requirements).

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.

```
<?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

We further explain the semantics of the relevant composition operators and actions while discussing the proof obligation framework in later sections. Interested readers are referred to [29] for a full description of the AORE model, the semantics of the operators and actions, and the toll gate case study.

2.2 Design descriptions of aspects in UML

Design descriptions of aspects could be presented in any of several versions of an extended UML that handles aspects, e.g., as in [7, 8, 20]. PROBE is based on the particular extensions to UML for aspects seen in [20].

The description of an aspect uses all standard UML diagrams and views, with each element labeled either as a new addition with a *<provided>* tag, or an assumption about an existing part of the system, with a *<required>* tag. The *<provided>* tag can usually be omitted, since it is the default assumption. Some provided elements can be tagged as *<hidden>* in order to denote that they are not part of the interface of the aspect. The fundamental unit of design is called a *concern*, and is composed of a partially ordered collection of aspect designs. Such a collection is the design component that corresponds to an ARCADE aspect treating a system concern.

As part of the design of a particular system, each concern design must be woven into the remainder of the design. The elements with a *<required>* tag are then viewed as formal parameters, to be bound to system elements in other UML descriptions, of either an underlying system or other aspects. The *<required>* design artifacts can be viewed both as describing the "join points" to which the aspect is to be applied, as well as the wider context in which it is used.

In Fig. 5, a typical design aspect describing an alarm is shown, with its class diagram and statechart.

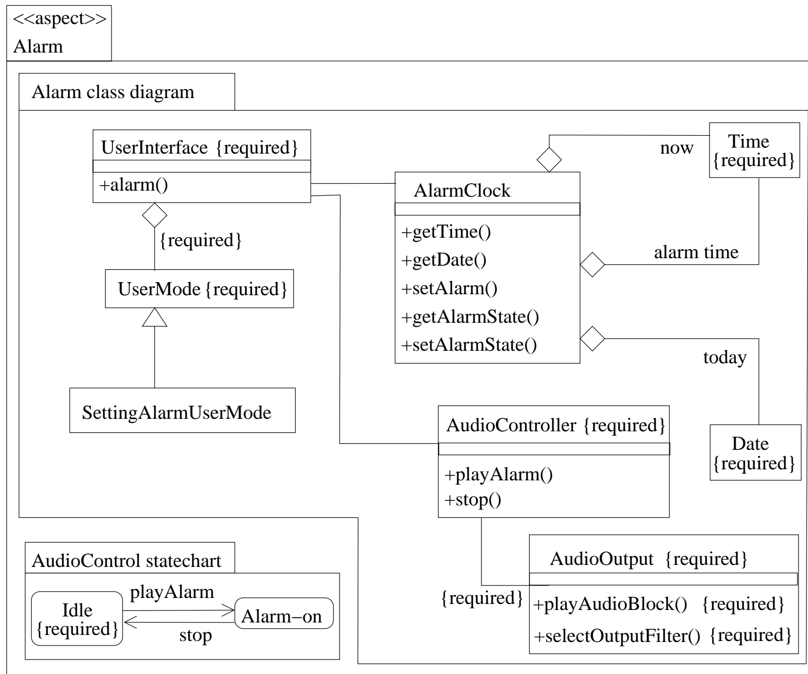

Fig. 5: An aspect in extended UML

It requires, among others, *AudioController* and *AudioOutput* classes and adds methods *playAlarm* and *stop* to *AudioController*, as well as adding an *AlarmClock* class, and extending a statechart by adding two transitions and a new state to a state *Idle* that is required to be present. This aspect could be part of the

implementation of the *ResponseTime* concern of the toll road system, to enable implementing the real-time requirements.

In addition there is a concern architecture diagram that shows the interactions of the aspects and their overlapping groupings into concerns, and in particular which subaspects are used in common by aspects either in conflict or cooperating, and how aspects depend on other aspects. When a *<required>* element of an aspect B is bound to an element of an aspect A, then B *depends* on A, and there is an arrow from the icon of B to that of A in the concern architecture. Intuitively, A should be woven first to the system, and only then B can be applied.

In Fig. 6 a typical concern diagram is shown with three concerns that overlap in their implementation aspects.

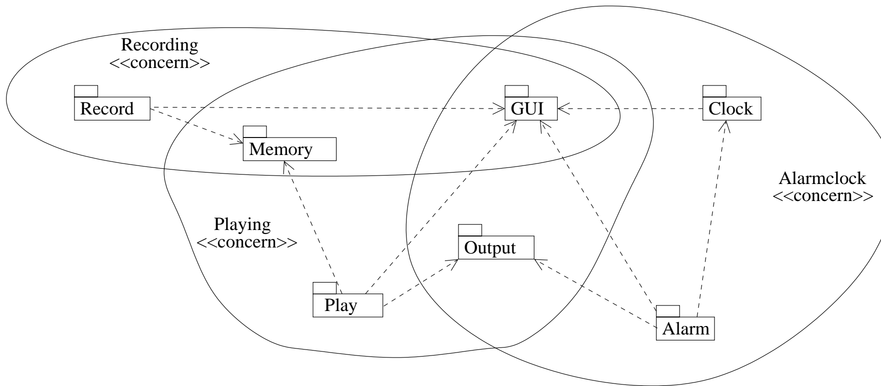

Fig. 6. A concern diagram

These concerns deal with an alarmclock, recording an announcement, and playing the recorded message using the alarm, and might be part of the implementation of the toll road user interface and response time concerns. Note that the alarm aspect depends on a lower-level clock aspect (that could be a wrapper), as well as on GUI and on Output aspects.

3. Proof Obligation Framework

The input to the proof obligation framework is thus an ARCADE specification in XML (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. From the design phase, an XML representation of the extended UML that handles aspects is also input to PROBE. As seen, this includes aspect descriptions, their groupings into concerns, and concern diagrams that describe dependency and overlap among concerns.

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 rest of the specification of the classes over which it is applied (derived from the viewpoint requirements).

The PROBE framework is shown in Fig. 7. Note that the XML requirements specification from ARCADE and UML design with aspect-oriented extensions as in [20] are the initial input to the modules

in the framework. At later stages, the implementation of the aspects is provided, as well as additional user input in the form of mappings from terms in the requirements to elements in the design, and then to locations and other elements of the implementation. The modules of PROBE (to be described in greater detail in the following sections) 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 additional proof obligations by analyzing the aspect-oriented UML design of the system;
- ***Conflict Analysis*** module, which treats the aspectual trade-offs and conflicts from the ARCADE specification and the design to determine proof obligations;
- ***Ontology***, which provides predefined generic temporal formulas or state functions for the ***Temporal Logic Generator*** and ***Conflict Analysis*** modules, as well as function and predicate names used in refinement;
- ***Integration*** module, which integrates the outputs of the ***Temporal Logic Generator***, ***Conflict Analysis*** and ***UML Analyzer***, using terms from the ***Ontology*** and ***Mapping*** information to generate groupings of temporal logic proof obligations, still in terms of predicates from the requirements and design.
- ***Instantiation*** module, which uses the result of the ***Integration*** module, along with the ***Implementation*** itself, and additional ***Mapping*** information, to generate concrete proof obligations in terms of the implemented system, in whatever notation is expected by the particular model checking or testing tool to be used in the verification.

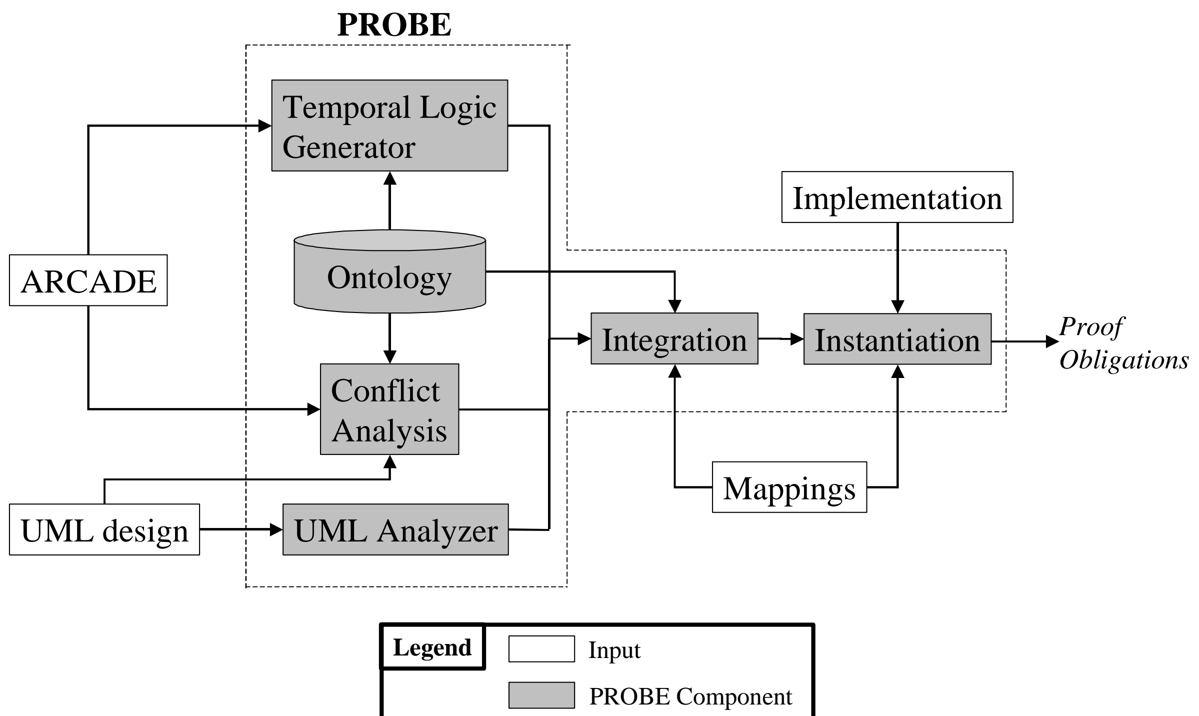

Fig. 7: The PROBE Framework

4. Requirements to temporal logic 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 "eventuality" 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" modality 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 [26], where a temporal logic analyzer is added to a natural language understanding system, to generate temporal logic expressions of natural language descriptions. Because the semi-structured ARCADE notation is used here, many of the keywords can be associated with pre-determined patterns of temporal formulas. 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. Names of state or event predicates generated from the ARCADE requirements are substituted in place of parameters in the parametric assertions in order to connect the pattern to the terms in the particular system being specified. This process will be demonstrated in later sections.

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 establish 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 *ResponseTime* 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 realtime restriction between the bounding events. The actual elapsed realtime 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 realtime 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

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

When the abstract formula is used in an aspect that lists key events or conditions, the ontology is extended to include predicate and function names associated with those events or conditions. These later will be connected to method calls, state functions, or events of the implementation when proof obligations are generated. 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. The predicates associated with *ResponseTime* are similarly derived from the text in Fig. 2, and include *read-gizmo*, *turn-on-light*, *display-amount*, etc. As explained previously, these terms will be substituted for the parameters in the abstract formula (E, E1, E2, and E3, in the example above), but also will be retained in the ***Ontology*** module, so that later they can be associated with design and, ultimately, implementation elements such as locations, events, or conditions.

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

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

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 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 they may not interfere with each other. For example, 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 there might be a real interference: 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

$$G((E1 \rightarrow (F\ E2)) \wedge (E \rightarrow E3) \wedge (time(E2)\text{-}time(E1) < N) \wedge (time(E) - time(E2) < D)).$$

(Note that the new realtime 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).

Similarly, weakened versions of the other aspects, such as *Availability*, or *Security*, can be defined. Although these aspects may seem to be absolute requirements, they are always somewhat restricted, and making these restrictions explicit can provide more realistic proof obligations.

It should be noted that conflicts among requirements from different aspects are actually crosscutting 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 totally new proof. 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. UML Analysis

6.1 UML *Analyzer* module

The ***UML Analyzer*** module of PROBE analyzes design components for both classes and aspects in UML, and generates temporal assertions for them. As already explained, it is based on the extensions to UML seen in [20] 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 initially done independently from the analysis of the ARCADE requirements descriptions, to guarantee that connections are not inferred without justification.

The ***UML Analyzer*** module is given an XML representation of the design aspects and of the concern diagrams, in addition to standard UML components. The underlying proof obligation for the implemented system is clearly that this given design is actually implemented. Each of the UML diagram types is a potential source for specific proof obligations in temporal logic for the implementation. Below we briefly consider the main viewtypes of the extended UML, namely use-cases and sequence diagrams to express intermodule interaction and communication, class diagrams and statechart diagrams for structural relations and internal behavior of methods within classes, and concern diagrams to show dependencies and overlap among aspects and concerns.

6.2 Use-case scenarios and sequence diagrams

Any descriptions using use cases or sequence diagrams can be turned into assertions about whether the implementation *conforms* to those scenarios. This means that every system computation should be equivalent to a sequence of scenarios, one after another. The notion of equivalence used is that equivalent computations may differ by the order in which independent operations were executed. The Technion’s CNV system [13] can be used to directly check conformance.

Although the specific syntax of CNV becomes relevant only in the ***Instantiation*** module, this type of specification differs from the others in that temporal logic expressions are not convenient as an intermediate notation for assertions about conformance. Several other researchers have recognized the attractiveness of adding regular expressions to linear temporal logic, and such a possibility should be used for this type of proof obligation. The role of the PROBE ***UML Analyzer*** module thus is to generate temporal expressions extended with a regular expression language, when presented with UML use-case scenarios or sequences of operations extracted from sequence diagrams in aspect descriptions.

The idea is that one way to describe the aspect is to give typical sequences of interactions that the aspect changes in the system, and that any implementation should have such sequences, interspersed with sequences irrelevant to the aspect. An assertion such as *conforms(abc, efg)* would mean that the system should conform to sequences of scenarios with *abc* and *efg*, interspersed with other activities. For the toll example, one scenario might be *<veh-ent-toll, charge-gizmo, veh-leave-toll>*. Techniques for such an analysis can also be found in the book by Harel and Marelly [16] on using “play-in” from scenarios and sequence charts to describe the effects new inputs should have.

6.3 Class and statechart diagrams

The augmented class diagrams show how to add subclasses or methods to "formal" parametric classes, that are denoted as <*provided*> . These can then be bound to classes of an underlying system, or to classes of other aspects, to show desired class augmentations. The proof obligations that follow are largely static type consistency, indicated by the resultant augmented class inheritance hierarchy. Since the formal methods tools to which we connect our proof obligations emphasize behavioral properties, these structural relations are less emphasized. It is also unusual to violate such design decisions, since the class hierarchy is by then fixed for the implementation.

Statecharts show behavioral descriptions of abstract states and the transitions among them, with conditions for applicability, and specified side-effects. The proof obligations that follow can be extremely detailed, or be left more abstract. Thus, a single state-transition-state triple <s1,t,s2> could lead to the proof obligation:

$G(in(s1) \wedge enabled(t) \rightarrow E(F(in(s2) \wedge s2 = t(s1))))$.

This means simply that whenever we are in state s1 and the transition t is enabled, there is a continuation where we are in state s2 expressing the change in the state done by executing t. This particular form actually uses branching time temporal logic, rather than the linear form we have seen so far. The collection of all such assertions is just another way of representing the state diagram. Such a detailed collection of local assertions is unlikely to add much to the analysis of the implemented system, and generally more global properties are of greater interest (that hold for all possible continuations, and thus are expressible in linear temporal logic). However, there can be key states that correspond to high level requirements (e.g., when a toll has just been paid) that will be of interest for proof obligations. The ***Integration*** module, discussed in Section 7, will need to identify such key transitions and states.

6.4 Concern diagrams

Just as a conflict expressed in ARCADE affects proof obligations, so does the overlap among concerns that is shown by the concern diagrams. These are an additional source for discovering potential interference among aspect implementations, even when the requirements do not seem to conflict. They can show situations where the design uses one aspect as a common denominator that is then extended in different ways (by applying additional aspects over it) to treat different concerns. Such identifications are particularly helpful when the aspects in a system change as the system evolves. A collection of aspects treating a concern correctly in isolation should be rechecked when used in a design where joint aspects are used in treating multiple concerns. Thus overlapping concerns are flagged, and proof obligations that show their non-interference are generated.

7. Integration and Instantiation

7.1 The Integration module

The ***Integration*** module must determine the mappings from the requirements level aspects to their refinements at the design level, and then integrate the combined results to obtain the final (but still abstract) proof obligations. Additional user input will often be needed for this task. For each aspect, the temporal properties from the requirements level are merged with those from the design level. In particular, this module will 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.

Thus, when a requirement is refined during design, the new requirements must be accepted as a concrete fulfillment of the abstract requirement, and then replace the requirements level description as the source for proof obligations. As another example, if an aspect requiring *Security* were added to our system, it could lead to a system 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 indeed follows the encryption policy of the design.

The ***Integration*** module includes a user interface that incorporates input on the mappings between the requirements and design levels. This will be driven by the (so-far undefined) names of predicates and functions that were generated from the natural language ARCADE requirements when the temporal logic assertions were generated (often using the ***Ontology*** module). These terms will either be associated with design elements, or left undefined for later integration with the implementation. Thus *yellow-on* could either correspond to a test of a system variable, or to the completion of a method call whose purpose is to send a signal lighting a yellow light.

As noted, sometimes the entire requirement is replaced with assertions about the design. More commonly, some predicates are defined in terms of the design, but otherwise the original requirement itself is used to generate proof obligations about the implementation. 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, an invariant property of a system may not be expressible except by using the Object Constraint Language OCL, which is not presently part of the UML standard. Similarly, 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, and yet should hold for the implemented system.

7.2. Summarizing the proof obligations

We can summarize the proof obligations generated by the ***Integration*** module from ARCADE requirements and extended UML designs for aspects.

1. Globally, all aspect implementations are asserted to not violate any class property derived from a viewpoint requirement, or from the structural associations seen in the class diagrams for the aspect.
2. All aspect implementations are asserted to not interfere with (possibly weakened versions of) requirements associated with other aspects, from the ARCADE contribution matrix and identified conflicts, and from the UML concern diagram.
3. Each aspect is asserted to conform to the collection of regular expressions that describe sequences of acceptable scenarios or sequences of messages from sequence diagrams of the aspect design, interleaved with the computations of the underlying system.
4. 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, augmented by (or replaced by) eventuality assertions derived from transitions in the statechart diagrams involving key states or events.

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

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

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 4.2, then substitution of predicates yields

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

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. The Integration module also has associated the event predicates to design elements such as states, method calls, or transitions.

7.3 Instantiation

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. As one example, to use the Bandera toolset [17] for model checking Java programs, the programs must be annotated using particular keywords, and the temporal logic assertions must be translated to their particular BSL notation (which has expressive power equivalent to standard linear temporal logic).

As already noted, conformance of a system with scenarios expressed as regular expressions can be checked by inputting a particular syntax to the CNV system. If this backend is used, the ***Instantiation*** module obviously would have to express the regular expressions in the CNV notation. CNV then transforms the scenarios, the original implementation, and information on independence of operations into an extended system with a history window, and a series of model checking tasks to be done on the extended system.

If at all possible, the instantiation of the predicates and events should be expressed in a *verification aspect*, as suggested in [21]. The idea is to describe needed functions, location predicates, and conditions about the implementation (e.g., as needed for annotating an implementation in anticipation of activating Bandera) in a generic aspect that is then woven into the implementation as needed to annotate it in

preparation for verification. This approach is clearly preferable to annotating the implementation code by directly inserting the needed labels, predicates, etc., since hand annotation suffers from the scattering and tangling problems that aspects are intended to prevent. Moreover, such a separation allows changing the implementation and/or the properties in the proof obligations independently.

8. Related Work

Design elements based on partial subjective views of the design are used in [6] as a means to support alignment and hence traceability of requirements through to implementation. That work requires use of the subject-oriented approach at the design and implementation levels. In contrast, the PROBE framework facilitates generation of proof obligations for validation of any aspect-oriented implementation. It also allows for the fact that requirements, design, and implementation modules are not necessarily in alignment: a requirements level aspect may or may not map onto an implementation level aspect. It could map onto a regular object or an engineering decision. Furthermore, PROBE preserves traceability of not only the aspectual requirements but also their associated trade-offs.

The PROBE framework also bears a relationship with goal-oriented approaches to requirements engineering [4, 23], which facilitate traceability of system goals and trade-offs from requirements through to implementation. This in turn supports validation. 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. Furthermore, the RE approach used in PROBE can be instantiated to goal-oriented techniques [29]. In such an instantiation, aspects, due to their support for capturing systemic concerns, would provide means to capture goals. In that case, the proof obligation framework can be used to validate an implementation against the specified goals of the stakeholders.

9. Implementation Plan

The design of the PROBE framework seen in this document has clarified many relationships among the requirements, designs, and implementations of aspects. Since both the ARCADE requirements and the extended UML design are given in XML notation, the input can easily be parsed and analyzed for the modules in PROBE. In fact, both kinds of input already have parsers that can be used for this purpose.

In the first stage of implementation, the ***Temporal Logic Generator***, and ***Ontology*** modules will be implemented, with the ***Conflict Analysis*** following closely. These modules can be used in PROBE with a stub ***UML Analyzer*** and a trivial ***Integration*** module that passes the requirements-level proof obligations and predicate names directly to the ***Instantiation*** module. This will allow showing proof of concept, and directly connect the requirements to the implementation proof obligations. The ***Instantiation*** module would only have to allow defining the predicate and function names from the ***Ontology*** in terms of the implemented aspects, since there would be no obligations from the design. The ***UML Analyzer*** and complete ***Integration*** modules will follow, to provide the full PROBE functionality.

10. Conclusions

The PROBE framework is based on connecting aspect-oriented requirements and design 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 design notations and facilitates effective use of formal methods tools and test case generators that are becoming increasingly practical for software validation.

One key element of the framework is the use of an extended ontology with parametric temporal logic formulas, state predicates and functions. The predicate and function names generated during the

requirements analysis phase are used to drive the acquisition of the needed mappings, both to design elements, and then directly to the implementation. The treatment of conflicts identified at the requirements stage is also unique to the framework. This includes the option of defining weakened versions of a requirement during requirements definition, when its priority is asserted to be lower than another, more crucial, requirement.

The PROBE proof obligation framework facilitates precise tracing of aspect refinements and trade-offs from requirements, to design, through to implementation, all in terms of aspects and concerns. This is a significant contribution, since until now the precise connections among aspect-oriented requirements, design, and implementation have been largely unexplored.

Acknowledgements

This work is supported by UK Engineering and Physical Sciences Research Council (EPSRC) Grant GR/S70159/01. The work on aspect-oriented requirements engineering 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 Computing Department, Lancaster University for helpful discussions.

References

[1] AOSD, "Aspect-Oriented Software Development", http://aosd.net, 2004.

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

[3] L. Bergmans and M. Aksit, "Composing Crosscutting Concerns using Composition Filters", *Communications of the ACM*, Vol. 44, No. 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", ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 1999, ACM, pp. 325-339.

[7] S. Clarke and R. J. Walker, "Composition Patterns: An Approach to Designing Reusable Aspects", International Conference on Software Engineering (ICSE), 2001.

[8] S. Clarke and R. J. Walker, "Towards a Standard Design Language for AOSD", 1st International Confernece on Aspect-Oriented Software Development, 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", *Communications of ACM*, Vol. 44, No. 10, 2001.

[12] A. Finkelstein and I. Sommerville, "The Viewpoints FAQ." *BCS/IEE Software Engineering Journal*, Vol. 11, No. 1, 1996.

[13] M. Glusman and S. Katz, "Model Checking Conformance with Scenario-based Specifications", International Conference on Computer-Aided Verification (CAV), 2003, Springer-Verlag, Lecture Notes in Computer Science, 2725, pp. 328-340.

[14] J. Grundy, "Aspect-Oriented Requirements Engineering for Component-based Software Systems", 4th IEEE International Sympsium on RE, 1999, IEEE Computer Society Press, pp. 84-91.

[15] J. Grundy, "Multi-perspective specification, design and implementation of software components using aspects", *International Journal of Software Engineering and Knowledge Engineering*, Vol. 20, No. 6, 2000.

[16] D. Harel and R. Marelly, *Come, Let's Play: Scenario-based Programming Using LSC's and the Play-Engine*: Springer-Verlag, 2003.

[17] J. Hatcliff and M. Dwyer, "Using the Bandera Toolset to Model-check Properties of Concurrent Java Software", CONCUR, 2001, Springer-Verlag, Lecture Notes in Computer Science, 2154, pp. 39-58.

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

[19] JBoss, "JBoss Aspect Oriented Programming Webpage", http://www.jboss.org/index.html?module=html&op=userdisplay&id=developers/projects/jboss/aop, 2004.

[20] M. Katara and S. Katz, "Architectural Views of Aspects", 2nd International Conference on Aspect-Oriented Software Development, 2003, ACM, pp. 1-10.

[21] S. Katz and M. Sihman, "Aspect Validation using Model Checking", International Symposium on Verification in Honour of Zohar Mana, 2003, Springer-Verlag, Lecture Notes in Computer Science, 2772, pp. 389-411.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming", European Conference on Object-Oriented Programming (ECOOP), 1997, Springer-Verlag, Lecture Notes in Computer Science, 1241, pp. 220-242.

[23] A. Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour", 5th International Symposium on Requirements Engineering, 2001, IEEE Computer Society Press, pp. 249-261.

[24] K. J. Lieberherr, D. Orleans, and J. Ovlinger, "Aspect-Oriented Programming with Adaptive Methods", *Communications of ACM*, Vol. 44, No. 10, pp. 39-41, 2001.

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

[26] R. Nelkin and N. Francez, "Automatic Translation of Natural Language System Specifications into Temporal Logic", International Conference on Computer-Aided Verification (CAV), 1997.

[27] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin, "JAC: A Flexible Solution for Aspect-Oriented Programming in Java", 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), 2001, Springer-Verlag, Lecture Notes in Computer Science, 2192, pp. 1-25.

[28] A. Rashid, "A Hybrid Approach to Separation of Concerns: The Story of SADES", 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), 2001, Springer-Verlag, Lecture Notes in Computer Science, 2192, pp. 231-249.

[29] A. Rashid, A. Moreira, and J. Araujo, "Modularisation and Composition of Aspectual Requirements", 2nd International Conference on Aspect-Oriented Software Development, 2003, ACM, pp. 11-20.

[30] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo, "Early Aspects: A Model for Aspect-Oriented Requirements Engineering", IEEE Joint International Conference on Requirements Engineering, 2002, IEEE Computer Society Press, pp. 199-202.

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

[32] I. Sommerville, P. Sawyer, and S. Viller, "Managing Process Inconsistency using Viewpoints", *IEEE Transactions on Software Engineering*, Vol. 25, No. 6, 1999.

[33] S. M. Sutton and I. Rouvellou, "Modeling of Software Concerns in Cosmos", International Conference on Aspect-Oriented Software Development, 2002, ACM, pp. 127-133.

[34] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns", International Conference on Software Engineering (ICSE), 1999, ACM, pp. 107-119.