Aspectual Support for Specifying Requirements in Software Product Lines

Harvey Siy, Prasanna Aryal, Victor Winter, Mansour Zand
University of Nebraska at Omaha
Department of Computer Science
{hsiy,paryal,vwinter,mzand}@mail.unomaha.edu

Abstract

We present an aspect-oriented requirements specification system for software product lines. We encapsulate non-functional concerns as a set of advices for transforming parameterized requirements to product-specific requirements.

We apply our system to the Health Watcher case study to demonstrate our approach. We sort out system requirements, exception handling requirements (alternate flows) and non-functional requirements and represent them as aspects in our framework. We have implemented a prototype transformation tool which takes these aspects along with the basic functional requirements as input and produces a requirements document with all applicable aspects woven in.

1. Introduction

We introduce a system for specifying functional and non-functional requirements for software product lines with the support of simple aspect-oriented mechanisms. A software product line [1] is a family of applications that share common features to meet the needs of a particular application domain. The set of products supported by a product line is determined by the specified product line variability. Like product-specific requirements, product line requirements are also subject to similar problems resulting from crosscutting concerns, especially due to nonfunctional requirements. Hence, the use of aspect-oriented mechanisms gives similar benefits in localizing some crosscutting concerns. Furthermore, we illustrate how aspect-oriented specification, coupled with parameterized requirements, can be used to express nonfunctional requirements that are dependent on the product configuration.

In the rest of this section, we discuss the rationale for considering the Health Watcher requirements as set of product line requirements and our general approach to specifying crosscutting concerns for product lines. In Section 2, we present the concerns and how we identified them. We also present the requirements specification language we will use in the rest of the paper. In Section 3, we present the crosscutting concerns and why we classified them as such. In Section 4, we present our aspect composition mechanism. While we do not support trade-off point identification here, we discuss in Section 5 how trade-offs may be analyzed in the context of configuring a specific product. In Section 6, we discuss the development status of our tools for supporting the generation of a requirements with all relevant aspects woven in. And we conclude with a summary and discussion of limitations in Section 7.

1.1. The Health Watcher Case Study as a Product Line

We apply our system to restructure the Health Watcher requirements. It is shown in the original paper [8] that the system may take several possible configurations. Thus, this requirements document can be treated as requirements for a product line, albeit at a high level. We view the Health Watcher requirements document as being composed of two distinct sets of requirement: (a) a common set of system functionality that form the core deliverable, and (b) a variable set of configurable parameters and nonfunctional requirements. Therefore, we characterize the Health Watcher system as a product line whose members vary by system configuration yet retain the same core functionalities as expressed in its use cases.

1.2. Overview

Managing varying requirements to develop different product line members has been a challenge in Software Product Line (SPL) systems. At the requirements level, the challenge transforms itself into coming up with a requirements specification for the system such that requirements are modularized and variabilities are localized and composable.

While not explicitly addressed in many product line engineering methods, it is advantageous to create a product

line requirements document [2]. A product line requirements specification is a more detailed refinement to the commonality analysis document[1] and provides the basis for implementing the libraries and components that are part of the product line core assets. Specific requirements specification documents can be generated for actual product line members for purposes of validation and verification.

Our work focuses on developing semantics for a language to specify non-functional requirements of each product line member as localized aspects that can be separately woven into the functional requirements to form a complete product member specification. This language represents functional requirements as viewpoints and non-functional requirements as concerns. It also supports parameterized requirements. Based on the way parameterized requirements are instantiated, some nonfunctional concerns may or may not be woven into the requirements document.

The approach of separating functional requirements into viewpoints and nonfunctional requirements into concerns is similar to Arcade [5]. We differ from Arcade with our support for parameterized requirements. The composition rules were also simplified and more closely approximate the weaving rules in our previous work on invertible weaving [10].

2. Identified Concerns

We identified viewpoints using the viewpoint classification from VORD [3].

There are two sets of direct viewpoints: *interactor* viewpoints and *subsystem* viewpoints. Interactor viewpoints are the direct users of the system, citizen and employee.

To derive the subsystem viewpoints, we assume the following product line architecture which is consistent with the architecture in the original paper [8]: a client for user interaction, an application subsystem, a messaging subsystem (which can be internal or network), and a persistent data subsystem (which can be file-based or one of several database technologies). Of these, the user interface interaction objects (such as servlets), the platform (hardware and software), the message passing subsystem and persistent data subsystem can be considered external to the Health Watcher software and thus considered subsystem viewpoints.

We also identified *engineering* concerns which are considered indirect viewpoints. The engineering concerns have to do with the nonfunctional attributes of the product and include: usability, availability, performance, security, standards. These are well-known crosscutting concerns and are discussed in Section 3.

[1] The basic commonality analysis document is a list of the product line commonalities and their variabilities.

2.1. A Requirements Specification Language

We use a set of DSLs to specify the product line requirements. The DSL notation chosen was loosely based on Nakatani's work on InfoWiz jargons [4], with some extensions to support aspects and wildcard matching.

The basic syntax is:

$$;term(attr_1, attr_2, \ldots, attr_n)[memo]$$

In this expression, $term$ roughly corresponds to an XML element, $attr_i$ corresponds to an XML attribute, and the $memo$ can have natural language text or other terms embedded. The semicolon marks the beginning of a term, the parentheses enclose the attribute list and the square brackets enclose the memo.

We use the keyword ;viewpoint to represent a viewpoint. A viewpoint carries one attribute which is its name. The memo of a viewpoint consists of one or more requirements.

We use the keyword ;req to represent a requirement. A requirement carries one or more attributes, which we will refer to as *tags*. These tags will be used later in aspect composition (see Section 4). The memo of a requirement consists of natural language sentences and zero or more subrequirements.

2.2. Health Watcher Viewpoints

With this notation, the use cases are represented in a straightforward way. For example, this fragment shows part of FR01:

```
;viewpoint(Citizen)
[
  ;req(UseCaseQueryInformation)
  [
    ;req(choosequery)
    [
      The citizen selects type of query.
      ;req(retrieveinfodblist)
      [
      In the case of query on specialties grouped by health units,
        the system retrieves the list of health units.
        ;req(retrieveinfodbdetails)
        [
        The system retrieves the details of each health unit
          such as its description and specialties.
        ]
        ;req(displayinfodblist)
        [
         The list of health units is presented to the user on
          their local display.
        ]
      ]
```

```
        ;req(retrieveinfodblist)
        [
        In the case of query on health units grouped by specialties,
           the system retrieves the list of specialties.
           ;req(retrieveinfodbdetails)
           [ ... ]
     ...
  ]
  ;req(UseCaseSpecifyComplaint)
  ...
```

As the example above indicates, requirements can be nested in other requirements to arbitrary depths. Note that requirements tags (e.g., retrieveinfodblist) need not be unique but are used to denote some distinctive characteristic of the requirement (see Section 4 for its usage).

2.3. Parameterized Requirements

We use parameterized requirements as one way of representing variability in product requirements. We apply an illustration in our message passing subsystem:

```
;viewpoint(SubMessagePassing)
[
  ;req(client2appcomm)
  [
    The message passing subsystem must pass messages
     between the client and the application subsystem.
     ;req(linktype, ;par[client2applink])
     [
        This communication is through a
        ;par[client2applink] link.
     ]
  ]
  ;req(app2dbcomm)
  [
    The message passing subsystem must pass messages
     between the application subsystem and the
     persistent data subsystem.
     ;req(linktype, ;par[app2dblink])
     [
        This communication is through a
        ;par[app2dblink] link.
     ]
  ]
]
```

The ;par[] term is a parameter. This enables the requirement to be used in different product configurations. In this example, the link type can either be internal or network. The instantiation mechanism is through an advice and will be covered in Section 4.

Here are the parameters of variation we identified for the subsystem viewpoints:[2]

1. User interface objects (assuming servlets) - browser, webserver, servlet container
2. Platform - hardware platform, software platform (the values for both of these parameters were fixed in the requirement)
3. Message passing - client to application link, application to database link
4. Persistent data - arrays, relational database, object-oriented database

3. Identified Crosscutting Concerns

We identified two sets of crosscutting concerns, the nonfunctional requirements and exceptions.

3.1. Nonfunctional requirements

The engineering viewpoints (usability, availability, performance, security, standards) are considered to be crosscutting because they apply to requirements across multiple use cases.

In contrast, we do not consider the remaining nonfunctional concerns to be crosscutting at the requirement level. These are: hardware and software, distribution, user interface, and storage medium. These concerns correspond to our configuration subsystem viewpoints, platform subsystem, message passing subsystem, user interface objects, and persistent data storage subsystem, respectively. While these concerns are crosscutting at the design and implementation level, their specification can be localized at the requirement level. Later, we show how their parameterized requirements can affect the other engineering concerns.

Table 1 shows which viewpoints are affected by each of the crosscutting nonfunctional concerns. When a crosscutting concern affects a viewpoint, it is implied that additional requirements will be added to that viewpoint in order to satisfy the concern.

3.2. Exceptions

We also classified as crosscutting any requirement that can be applied to more than one other requirement. These included exceptions and subsystem-related requirements.

All exception requirements are classified into one of 5 exception concerns:

1. Communication problem - all exceptions on cross-component communication problems.
2. Retrieval problem - all exceptions associated with retrieving information from the database.

[2] It is also possible to parameterize requirements in the interactor viewpoints. For instance, the type of object being queried and updated can be parameterized, but we decided not to do so as we wanted to follow the original functional requirements as much as possible.

	Customer	Employee	UI objects	Platform	Message Passing	Persistent Data
Usability	√	√				
Availability			√	√	√	√
Performance - Capacity - Response	 √	 √	 √ √	 √ √	 √ √	 √ √
Security	√	√			√	
Standards			√		√	

Table 1. Concerns and the viewpoints they crosscut.

3. Invalid input - all exceptions on invalid user input (e.g., wrong passwords).
4. Inconsistent data - exceptions associated with inability to assure consistent data.
5. Writing problem - exceptions associated with problems storing data.

The actual specification of these crosscutting concerns is closely tied to the way their composition is specified and are discussed in Section 4.

4. Composition

Our composition process borrows much from the aspect-oriented programming metaphor. A crosscutting concern is represented as an aspect. An aspect can have a set of advices whose contents modify functional requirements. In our join point model, an advice can be prepended or appended to a requirement. An advice can also add new requirements. Each advice has a pointcut expression which specifies whether the advice is to be prepended, appended or added as a new requirement.

4.1. Language Extensions to Support Aspects

We extend the Infowiz notation with aspect constructs. We use a keyword aspect to distinguish crosscutting concerns from viewpoints. Within an aspect, the keyword advice begins a new advice.

The pointcut expression refers to a set of viewpoints or requirements on which to apply the advice. Viewpoints and requirements are referred to by their *tags*. Requirements can be qualified by their viewpoints and other upper level requirements:

```
;viewpoint(...);req(...);req(...) ...
```

A set of requirements can be referred to in a pointcut expression through wildcards on tags, i.e., *, retrieve*, etc.

4.2. Designators: before and after

In addition, we use pointcut *designators* to specify where and how an advice is to be weaved to the requirements. As in AspectJ, we have before and after designators which prepend or append an advice to the given requirements. As an example, the exception aspect for inconsistent data is specified as follows:

```
aspect CheckConsistency
[
  advice[;viewpoint(*);req(access*)]: before
  [
    The system ensures the information is
    consistent.
  ]
  advice[;viewpoint(*);req(access*)]: after
  [
    If inconsistent data cannot be assured,
    abandon the retrieval attempt and
    raise an error.
  ]
]
```

In this example, the aspect weaver looks for requirement tags that match the particular wildcard pattern, prepends the before and appends the after advice to the matching requirements. To illustrate, given the requirement:

```
;req(access,searchbyID)
[
  The unique identifier is used by the system to
  search for the selected health unit.
]
```

The weaver transforms the requirement into:

```
;req(access,searchbyID)
[
  The system ensures the information is
  consistent.
  The unique identifier is used by the system to
  search for the selected health unit.
```

```
    If inconsistent data cannot be assured,
    abandon the retrieval attempt and
    raise an error.
]
```

4.3. Weaving in new requirements

We define the introduce designator to add new requirements to a viewpoint or new subrequirements to a requirement. We illustrate this with the security concern:

```
aspect Security
[
  advice[;viewpoint(Citizen);req(UseCase*)] : introduce
  [
    ;req(accesscontrol)
    [
      Only a citizen may execute these
      functionalities.
    ]
  ]
  advice[;viewpoint(Employee);req(UseCase*)] : introduce
  [
    ;req(accesscontrol)
    [
      Only a registered employee may
      execute these functionalities.
    ]
  ]
]
```

Each of these add an access control requirement to each use case defined in the Citizen and Employee viewpoints.

4.4. Instantiation

We define an instantiate designator in order to instantiate parameterized requirements into concrete requirements. For example, the following concern binds one parameter of the message passing requirements defined in Section 2.3.

```
aspect client2appnetworklink
[
  advice[;viewpoint(SubMessagePassing);req(*)] : instantiate
  [
    ;replace(client2applink)
    [
      network
    ]
  ]
]
```

The requirements under the viewpoint SubMessagePassing are treated as a requirements template and all instances of ;par[client2applink] are replaced by network.

One benefit of this parameter instantiate mechanism is that it affords us with some flexibility in specifying requirements that are conditioned on certain configurations. For example, we illustrate how we would represent the secure protocol requirement as part of the Security concern:

```
aspect Security
[
  advice[;viewpoint(SubMessagePassing);req(*)
      ;req(linktype, network)] : introduce
  [
    ;req(secureprotocol)
    [
      The system should use a security protocol
      when sending data over the network.
    ]
  ]
  ...
]
```

This advice is woven into the MessagePassing viewpoint only if the link type (a parameter) has been instantiated to network.

5. Trade-off Point Identification and Resolution

Trade-off point identification is not supported for this case study. From the product line standpoint, trade-offs are analyzed as part of the process of instantiating parameters. For instance, the choice of a network link rather than an internal link impacts availability as network links can go out of service. In order to automate this type of reasoning, we rely on mathematical models of the nonfunctional concerns. For instance, a reliability model of system availability based on component availability (e.g., probability of availability of a network link, etc.) can be used to calculate the system availability given a particular set of configuration choices, and thus whether the system availability requirement can be reached.

However our system needs quantifiable goals and quantifiable information on component-level and subsystem-level nonfunctional attributes. An example can be found in our ongoing work on specifying performance requirements for a telecomm product line [7].

6. Tool Support

We have developed a prototype weaver based on the HATS transformation system [9]. We have implemented the syntax and semantics for specifying requirements as viewpoints and aspects. We currently support advices with before, after and introduce designators. Similarly we support logical expressions for our pointcuts. Wildcard support for pointcut expression is being implemented.

This weaver is part of a larger system for automating much of the selection of aspects to be woven. Figure 1

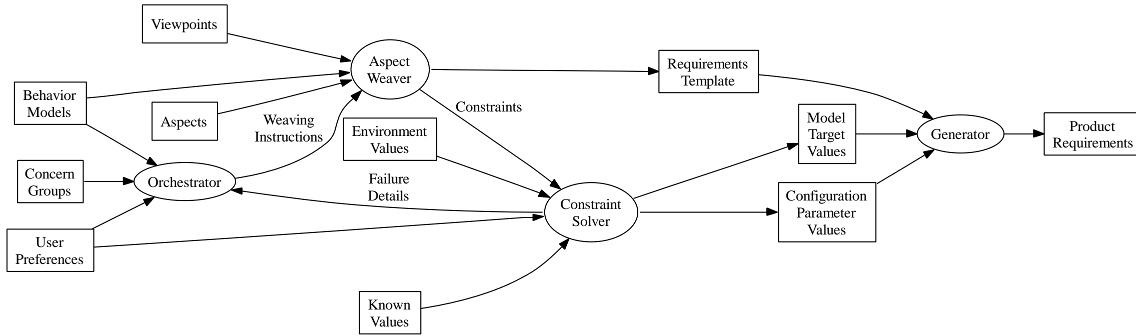

Figure 1. Dataflow diagram for aspect-oriented product line requirements specification system.

is our proposed architecture for a system that supports the composition of aspect-oriented product line requirements. This system is designed to provide automated support for generating product-specific requirements that satisfy nonfunctional goals, which in turn vary based on user preference and environmental conditions. The driver for selecting which set of weavings to perform is an orchestrator that selects a group of configuration concerns, such as the client2appnetworklink concern from Section 4.4, and generates a set of weaving instructions that enable the weaver to first instantiate a product configuration and then weave in the concerns. A constraint solver checks if the resulting configuration satisfies the user preferences. Mathematical models of the nonfunctional concerns provide the basis for the constraints.

7. Conclusions

We have introduced a system for defining aspect-oriented product line requirements and have used it to restructure the original Health Watcher requirements. The full specification can be viewed in [6]. We have made explicit those parts of the requirements which can be treated as parameterizable. This enabled us to define specific nonfunctional requirements that are conditioned on certain parameter values, thereby giving more flexibility to the requirement specification process.

As the readers may note, our composition mechanism is strongly dependent on the use of requirement tags. Aspects must know which tags to weave to. We are in the process of devising a systematic naming scheme that can be applied consistently to all requirement tags.

References

[1] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, 2001.

[2] S. R. Faulk. Product-line requirements specification (PRS): An approach and case study. In *Proc. of the 5th IEEE International Symposium on Requirements Engineering (RE'01)*, pages 48–55, Toronto, Canada, Aug. 2001.

[3] G. Kotonya and I. Sommerville. Requirements engineering with viewpoints. Technical Report CSEG/10/1995, Lancaster University, 1995.

[4] L. H. Nakatani, M. A. Ardis, R. G. Olsen, and P. M. Pontrelli. Jargons for domain engineering. In *Proc. of the 2nd Conference on Domain-Specific Languages (DSL'99)*, pages 15–24, Oct. 1999.

[5] A. Rashid, A. Moreira, and J. Araújo. Modularisation and composition of aspectual requirements. In *Proc. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 11–20, Boston, MA, March 2003.

[6] H. Siy, P. Aryal, V. Winter, and M. Zand. Aspect-oriented product line requirements specification for the Health Watcher Case Study. http://cs.unomaha.edu/∼hsiy/aspl/health.html.

[7] H. Siy, P. Aryal, V. Winter, M. Zand, and P. Zhang. Specifying performance requirements for product lines. Technical report, University of Nebraska at Omaha, 2007. http://cs.unomaha.edu/∼hsiy/aspl/vqreqs.pdf.

[8] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proc. of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02)*, Seattle, Washington, Nov. 2002.

[9] V. Winter and J. Beranek. Program transformation using HATS 1.84. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generational and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCS*. Springer-Verlag, 2006.

[10] V. Winter, H. Siy, M. Zand, and P. Aryal. Aspect traceability through invertible weaving. In *Early Aspects Workshop at AOSD'06*, Bonn, Germany, March 2006.