Managing Feature Interaction by Documenting and Enforcing Dependencies in Software Product Lines

Roberto Silveira Silva Filho and David F. Redmiles

Bren School of Information and Computer Sciences
Department of Informatics
5029 Donald Bren Hall
Irvine, CA 92697-3440
{rsilvafi, redmiles}@ics.uci.edu

Abstract. Software product line engineering provides a systematic approach for the reuse of software assets in the production of similar software systems. For such it employs different variability modeling and realization approaches in the development of common assets that are extended and configured with different features. The result is usually generalized and complex implementations that may hide important dependencies and design decisions. Therefore, whenever software engineers need to extend the software product line assets, there may be dependencies in the code that, if not made explicit and adequately managed, can lead to feature interference. Feature interference happens when a combined set of features that extend a shared piece of code fail to behave as expected. Our experience in the development of YANCEES, a highly extensible and configurable publish/subscribe infrastructure product line, shows that the main sources of feature interference in this domain are the inadequate documentation and management of software dependencies. In this paper, we discuss those issues in detail, presenting the strategies adopted to manage them. Our approach employs a contextual plug-in framework that, through the explicit annotation and management of dependencies in the software product line assets, better supports software engineers in their extension and configuration.

Keywords: Feature interaction, software product lines, product line documentation, contextual component frameworks, software dependencies, and publish/subscribe infrastructures.

1. Introduction

The need for faster software development cycles that meet the constantly evolving requirements of a problem domain has driven industrial and academic research in the area of Software Product Lines (SPL for short). The goal of SPL engineering is *"to capitalize on commonality and manage variability in order to reduce the time, effort, cost and complexity of creating and maintaining a product line of similar software systems"* [17]. In SPLs, reuse of commonality allows the reduction of the costs of producing similar software systems, while variability permits the customization of software assets to fit different requirements of the problem domain [6].

SPLs are usually designed using the concept of features and variation points [15]. Variation points represent the locations in the software that enable choices, while features represent user-observable units of variability associated to one or more of those points. In many industrial settings, commonality is implemented in the form of large pieces of software such as object-oriented frameworks, whereas features implement new behavior by direct extension and source code configuration [3].

In such approaches, features can interact in a positive way, by the combined extension of the common code in different variation points. They can also interact in a negative way, by defining behaviors that are incompatible with other features installed in the same infrastructure. In the latter cases, the interaction is also called interference. A feature interference occurs when the addition of a new feature affects or modifies the operation of the system in an unpredicted way [4]. In fact, many non-trivial feature interferences in software are a result of conflicting assumptions about service operations and system capabilities that are not explicitly documented or exposed to the programmers of those features [28]. SPLs are no exception. The dimensions of extensibility in SPLs are not always orthogonal, and their dependencies are not always explicit. As a consequence, whenever software engineers need to extend a SPL with new features, there may be dependencies within and among variation points and features that, if not documented and managed, can lead to feature interference.

Our experience in the development of YANCEES [22], a highly extensible and configurable publish/subscribe SPL, makes evident some of those issues. In particular issues associated with the lack of management and documentation of fundamental, configuration-specific, incidental dependencies and emerging system properties. Those dependencies are further explained as follows.

- **Fundamental (or problem domain) dependencies** encompass the logical relationships that are common to all software product line members. For example, in the publish/subscribe domain, the process of: publication of events, followed by their routing based on subscription expressions, and the subsequent notification to subscribers define a common behavior shared by all publish/subscribe infrastructures. The fundamental dependencies that involve this common behavior restrict variability in the problem domain and create configuration rules that must be obeyed in the extension and configuration of a SPL. Moreover, they restrict some configuration-specific dependencies.
- **Configuration-specific dependencies**. These include the compatibility relations between features that extend or refine the common SPL behavior in the implementation of the different SPL members. Those dependencies are expressed in the form of inter-feature relations such as “compatible”, “incompatible”, “optional”, “exclusive”, “alternative”, and others. For example, ‘content-based’ filtering, ‘tuple-based’ message format and ‘push’ notifications define a compatible combination of features present in many content-based publish/subscribe infrastructures, whereas ‘content-based’ filtering and events represented as ‘objects’ are usually incompatible.
- **Incidental (or technological) dependencies** are consequence of the variability realization approaches employed in the construction of the SPL. Examples of such approaches include: design patterns, parameterized classes, aspects, mixings and others [26]. The benefits provided by each one of these approaches come with extra costs: the increase of the overall software complexity and the need to

comply with their configuration and extension rules. For example, the use of software patterns such as Strategy, require the proper implementation of interfaces and the selection criteria. Moreover, these approaches usually introduce indirections in the code that, when applied in combination, may hinder its legibility and extension ([7] pp. 295).

- **Dependencies on emerging system properties** represent assumptions about system-wide guarantees for example: security, guaranteed event delivery, total order of events, and other properties that depend on different configuration parameters of the infrastructure. These system-wide properties may vary due to complex dependencies between the system components and parameters. In YANCEES, for example, the total order of events is a function of the distribution of the system. In peer-to-peer settings, for example, the total order of events is not preserved, whereas in centralized settings it is assured by the infrastructure.

The inherent variability in software product lines, together with the need to cope with fundamental, configuration-specific, incidental and system properties dependencies not only creates a configuration management problem, but also hinders the reuse and the proper extension of software product lines. It makes possible for changes in different parts of software to break implicit system assumptions, leading to feature interference. In fact, our experience in the design, implementation and use of YANCEES shows that software engineers lack appropriate knowledge of those dependencies and assumptions, what we call variability context: *the information necessary to understand, extend and customize the software product line.* They also lack automated support in the form of tools and mechanisms that enforce those relations in the SPL, providing runtime and configuration-time guarantees.

This paper describes in detail those issues in the design and development of YANCEES, a publish/subscribe infrastructure SPL and discusses the strategies used in supporting software engineers in extending and configuring this infrastructure. In particular, we argue for the use of dependency models in both design and implementation, with the elucidation and enforcement of those dependencies in the product line code. Our approach represents dependencies in the code artifacts, allowing their automatic enforcement at both load time and run time, at the same time that support software engineers in extending and configuring the product line, by supporting their understanding of the hidden dependencies and configuration rules of software.

The contributions of this paper are in different fronts. From a feature interaction perspective, we provide a case study that shows how the lack of documentation and enforcement of fundamental, configuration-specific, incidental dependencies and emerging system properties can interfere in the feature reuse and the extension of SPLs. From a feature interaction research perspective, we show how the explicit documentation of those dependencies in the SPL, combined with the use of contextual component frameworks and configuration managers can help in the detection and prevention of feature interference.

This paper is organized as follows: section 2 presents the technological background of our approach. Section 3 discusses our experience in the design, implementation and extension of YANCEES. Section 4 discusses our approach in managing those issues. Section 5 discusses some related work and we conclude in section 6.

2. Background

The work presented in this paper relies on concepts from the areas of SPL variability modeling, and software component frameworks. We introduce these concepts here.

2.1 Variability modeling

Variability modeling approaches provide a notation for representing choices and constraints (dependencies and rules) involving units of variability (features, variants, components) in SPLs. First generation modeling languages such as FODA [16], represent variability in terms of features and their compatibilities (alternative, multiple, optional and mandatory) and incompatibilities (exclusive or excludes) around predefined variation points. Researchers soon realized the importance of representing other kinds of dependencies in these models, proposing different extensions. For example, Ferber et al. [11] introduces the notion of "intentional", "environmental", and "usage" dependencies; whereas Lee and Kang [19] proposes the representation of runtime feature interactions such as "activation" and "modification" dependencies. Those models, however, suffer from a fundamental problem: the lack of representation of dependencies as first-class entities, and their traceability to implementation concerns.

The inadequate management and representations of dependencies in SPLs [8] motivated the development of second generation variability modeling approaches[24]. These approaches represent dependencies as first-class entities, and support the variability management by the use of constraint checkers. Together, they provide software engineers with an overview of variability in the system, supporting their navigation through the space of valid product configurations, and deriving individual product members that meet a valid set of quality and feature attributes. An example of variability model and environment is COVAMOF [25], which also represents overall system quality attributes and tacit knowledge in the documentation of more complicated relations between features.

While very useful in the representation of system variability, commonality, and the interaction between features, these models fail to: (1) support source code-level maintenance and evolution, and (2) support the runtime configuration management of features [18]. In this paper, we propose an approach that, by the integration of design models into the code, allow software engineers to better understand the underlying assumptions in the code implementation; whereas allows the infrastructure to automatically enforce those relations, preventing feature interference.

2.2 Contextual software component frameworks

Component models define the basic encapsulation, communication and composition rules that support the development of component-based software. Contextual Component Frameworks (CCF) [27] implement these models and support the automatic creation and composition of objects based on user-defined properties (or con-

text). A CCF uses the inversion of control (IoC) and injection of dependencies principles [12] to transparently provide user-requested services and properties to the components in the system. Dependency Injection is a form of IoC that removes explicit dependence on container APIs, separating those concerns from the component implementation. Property-based contextual composition allows software engineers to select environmental characteristics and crosscutting concerns required by the component. This is achieved with the use of properties, usually expressed in the code or associated manifest configuration files. Common properties include: transactional communication, persistency, security and other crosscutting concerns. Examples of well-known component frameworks include CORBA Component Model, COM/ActiveX and Enterprise JavaBeans.

YANCEES uses this approach to separate configuration management concerns from feature implementations and to support software engineers in the extension and configuration of software product lines. It explicitly represents variation points and inter-feature dependencies in the software source code, with the specific goal of supporting variability and preventing feature interaction caused by the lack of representation and enforcement of dependencies.

3. Case study: YANCEES, a publish/subscribe product line

This section describes our experience in the design and implementation of YANCEES, a highly configurable and extensible publish/subscribe SPL, and discusses the main variability management issues faced.

Publish/subscribe infrastructures implement a distributed version of the Observer design pattern [10], as shown in the top level of Fig 1. In its initial stage the pattern is very simple, it provides an interface (*IPubSub*) which allow polishers (*IPublisher*) to send events to the infrastructure; whereas subscribers express interest on those events through the use of subscriptions, using the *subscribe(Subscription exp)* command. A subscription is a logical expression in the content or order of the events. When a subscription is satisfied, a notification with the message matching this expression is sent to subscribers (*ISubscriber*) through the call of the *notify(Event: evt)* command in their interfaces. This pattern is used in the implementation of different publish/subscribe infrastructures in different domains. For a survey of existing pub/sub systems please refer to [23].

The majority of publish/subscribe research and commercial infrastructures fall short of mechanisms that allow their customization and configuration to comply with the evolving requirements demanded by event-driven applications [23]. Motivated by this fact, we developed a flexible publish/subscribe infrastructure called YANCEES (Yet ANother Configurable Extensible Event Service) [22], that allows the different aspects of the publish/subscribe pattern to be extended and customized. In the coming sections, we briefly present the main elements of YANCEES design and implementation.

3.1 YANCEES design and implementation

Different principles and strategies were applied in the design and implementation of YANCEES. These are: the use of a micro kernel architecture, supporting variability on different publish/subscribe dimensions; the application of different variability mechanisms such as: abstract classes and interfaces, extensible languages, dynamic and static plug-ins, generic events; and the wide use of static and dynamic configuration managers. These principles and strategies are further discussed as follows.

Table 1 Publish/subscribe infrastructures variability dimensions and examples.

MODEL	DESCRIPTION	EXAMPLE
Event model	Specifies how events are represented	Tuple-based; Object-based; Record-based, others.
Publication model	Permits the interception and filtering of events as soon as they are published, supporting the implementation of different features and global infrastructure policies.	Elimination of repeated events, persistency, publication to peers (through protocol plug-ins).
Subscription model	Allow end-users to express their interest on sub-sets of events and the way they are combined and processed.	Filtering: content-based, topic-based, channel-based; Advanced event correlation capabilities
Notification model	Specifies how subscribers are notified when subscriptions match published events.	Push; pull; both, others
Protocol model	Deals with other necessary infrastructure interactions other than publish/subscribe. They are subdivided in interaction protocols (that mediate end-user interaction), and infrastructure protocols (that mediate the communication between infrastructure components)	**Interaction protocols**: Mobility; Security; Authentication; Advanced notification policies. **Infrastructure protocols**: federation, replication, Peer-to-peer integration.

Variability Dimensions. Around a common generalized publish/subscribe micro kernel, different variability dimensions were implemented in YANCEES, as listed at Table 1. The YANCEES variation points were selected according to the main publish/subscribe design concerns described by Rosenblum and Wolf [21] model, extended to include the notion of protocols, a design concern that captures the different kinds of infrastructure distribution strategies and other sorts of user interactions outside of the publication and subscription of events.

Extensible languages and plug-is. Publish/subscribe infrastructures have special requirements of dynamism driven by its interactive characteristic. Subscriptions are dynamic in essence; they are posted and removed at runtime by their users, and are expressed in terms of commands in a subscription language. As a consequence, variability in this domain requires the simultaneous evolution of language and infrastructure capabilities. Those requirements lead us in the choice of extensible languages and plug-ins [2] as the main variability realization approach for the subscription and noti-

fication models. In particular, YANCEES is implemented as a composition framework ([27] chapter 21.1) where component instances (in our case plug-ins) are created and combined at runtime in response to composition operators (subscription, notification commands in the user's posted subscriptions) with the help of parses (or Mediators). The extensible language is implemented in XML, having its grammar defined using W3C XMLSchema standard.

Static plug-ins. Non-interactive characteristics are implemented by static plug-ins and filters, installed at load-time (i.e., when the infrastructure is bootstrapped). The publication model, for example, is implemented as a Chain of Responsibility design pattern (see [13]), where filters (as static plug-ins), are composed into event processing queues that intercept the publication of events, implementing global system policies. Features that are shared by different variation points are implemented as static plug-ins a.k.a. services.

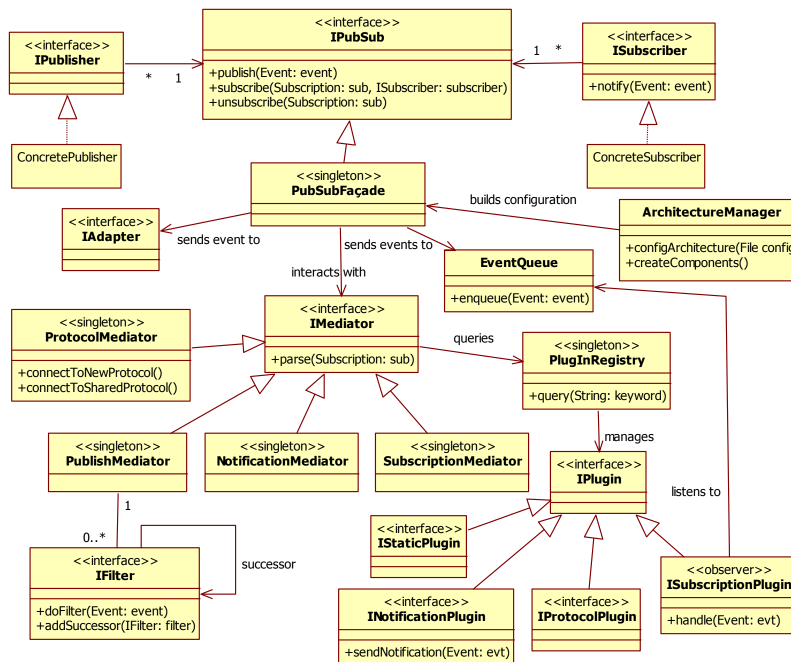

Fig 1 Overview of YANCEES core architecture, with its main components and interfaces

Generic Events. The variability in the event model is supported by the use of object wrappers that can hold different content formats (attribute/value pairs, objects, XML files or plain text) under a generic interface (*IEvent in* Fig 1).

Configuration managers and dynamic parsers. The final design decision is the implementation of variability management in the system itself by the use of configuration managers that install static plug-ins and filters, and mediators, that allocate plug-ins at runtime.

Applying those strategies the original publish/subscribe design pattern was extended as presented in Fig 1, which shows the main YANCEES core components and

interfaces. Due to space limitations, other classes such as exceptions and auxiliary objects are not represented in Fig 1. The *PublishMediator* handles the publication of events, allowing the extension of its dimension through the use of filters (implementing *IFilter* interface). The *NotificationMediator* utilized notification plug-ins to implement different notification policies; whereas the *SubscriptionMediator* handles the interpretation of different subscription language expressions allocating appropriate subscription plug-ins. The dynamic allocation and discovery of plug-ins at runtime is supported by the use of a *PluginRegistry* component. After passing through the publication filters, the events are placed on the internal *EventQueue* and/or sent to adapters (implementing the *IAdapter* interface) that allow the integration with existing pub/sub systems. The *ArchitectureManager* installs static and dynamic plug-ins in the infrastructure based on a configuration file describing the features and their implementation files.

The YANCEES core, composed of all the mediators, queue, registry and interfaces presented in Fig 1 is about 6000 LOC of Java code. The plug-ins and extensions used in different projects comprise another 3500 LOC.

3.2 Extending and configuring YANCEES

This section presents an example on how YANCEES can be extended and configured to support different application domains. In particular, we show how it was extended to support Impromptu [9], a peer-to-peer (P2P in short) file sharing tool. Impromptu provides an interface and a repository that allow users to share files in an ad-hoc peer-to-peer way. In Impromptu, events are used to monitor the activity of a local file repository from each peer, to inform the arrival or departure of new peers in the network, and to synchronize the shared visualization of user's interfaces from every peer. The peer discovery protocol is implemented using the IETF multicast DNS protocol. Every Impromptu peer executes a local YANCEES instance which is connected to other YANCEES instances in every peer in the network, thus forming a virtual P2P event bus. In this configuration, YANCEES provides both local and global event-based communication. Locally, it decouples the file repository from the GUI. Globally, it allows the monitoring of events from the file repositories in other peers, keeping their visualizations synchronized.

In the support of Impromptu, YANCEES was extended and configured with plug-ins, filters and a tuple-based event format as illustrated in Fig 2. In this example, events are represented as attribute/value pairs of variable length and number. This is achieved by extending the *GenericEvent* interface. The subscription language is also extended to support two kinds of filtering: content-based, allowing the filtering of events based on the content of all their fields; and topic-based filtering, allowing the fast switching of events based on a single field. It also supports event sequence detection that operates over each one of those filters. The subscription language extension requires two steps: (1) the implementation of the *ISubscriptionPlugin* interface, and (2) the extension of the XMLSchema of the subscription language for every new command. The notification policy is push, extended in the same way as the subscription plug-ins, i.e. implementing the *INotificationPlugin,* and extending the notification language. The protocol model supports the mDNS peer discovery, detecting the arriv-

al and departure of YANCEES instances in the local network. It also supports the publication of events between YANCEES peers, creating a virtual bus, with the help of the *PeerPublisher* plug-in that publishes to and receives events from other peers. Events from other peers are placed directly in to the event queue, skipping the publication model filtering. Finally, the publication model is extended with two filters: one that removes repeated events as they are published, thus saving network bandwidth; and another filter that forwards the events to the protocol plug-in. Publication filters must implement the *IFilter* interface. These extensions are put together with the help of the *ArchitectureManager* that assembles a valid infrastructure based on a configuration file that defines the feature names, their implementation file, and the variability dimension they extend.

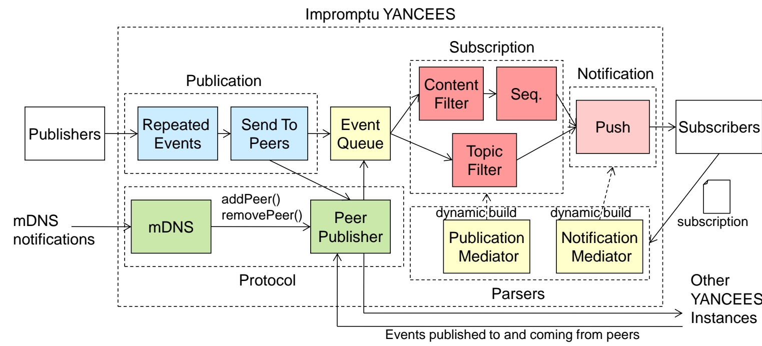

Fig 2 YANCEES configuration with Impromptu required functionality

The generality and variability approaches employed in the design and implementation of YANCEES, whereas provide the required flexibility, resulted in different issues that lead to feature interference. Those issues are further discussed in the next sections.

3.3 Feature interference in the YANCEES variability model

Fundamental (or problem domain) dependencies. Through the lack of appropriate documentation, software engineers may assume that certain variable characteristics of the infrastructure are constant. For example, plug-ins and filters may be implemented with specific timing and event formats in mind. A change in the event representation, for example, from variable attribute/value pairs to fixed records, can completely invalidate the subscription language *ContentFilter*, *SequenceDetector* or even the input filters and protocol plug-ins in the system in Fig 2.

Configuration-specific dependencies. Some features in YANCEES have their functionality implemented through the integration of different components spanning more than one variation point. In the example of Fig 2, the *SendToPeers* will only work properly if the *PeerPublisher* and *mDNS* plug-ins are both installed in the protocol variation point. This characteristic creates a dependency between these features. Moreover, changes in any of those components due to natural software evolution may invalidate the implementation of the whole feature.

Incidental (or technological) dependencies. Each variability realization approach introduces specific configuration rules which, if not accounted for, can lead to interference. In the example of Fig 2, the accidental inversion of the order of *SendToPeers* and *RepeatedEventsFilter* pug-ins would result in the erroneous publication of repeated events to all peers, interfering with the overall system performance.

Dependencies on emerging system properties. Implicit assumptions on existing system attributes also permeate the implementation of features in our model. The order of events, for example, is a function of the distribution and of the protocol algorithms used. In a centralized setting, event order is usually guaranteed, whereas in distributed settings as in this P2P model, events can arrive earlier or later than others (coming from the *PeerPublisher* plug-in for example), invalidating the *SequenceDetector* subscription command ('Seq.' in Fig 2). Those assumptions may also directly impact the behavior the *RepeatedEventsFilter.*

Generality issues. One of the main strategies of reuse in YANCEES is the implementation of a generalized common core. The use of generic interfaces throughout the system, permits specific extensions to be developed, while the common pub/sub process is preserved and reused. This approach, however, has a disadvantage of hiding implicit dependencies and assumptions. For example, the filter interface only prescribes a *doFilter(IEvent: evt)* method that must be implemented by all the filter components. It does not prescribe any timing or control dependencies between other filters, installed together in the publication model, nor explicitly represent environmental assumptions such as the impact a filter may have in other parts of the system if events are removed, modified or added by this component. The same is true to the subscription and notification models where plug-ins implement generic interfaces dependent on *IEvent* generic event representation. As a result, syntactically sound expressions can be incompatible with the current system configuration as event order, format or timing.

4. Managing feature interaction in YANCEES

In order to address and prevent the different kinds of feature interaction discussed in the last section, software engineers need a way to better understand and enforce the fundamental, configuration-specific and incidental dependencies in the SPL without jeopardizing its flexibility. In YANCEES these goals are achieved by the documentation and enforcement of design and implementation level dependencies in the code. This information is exposed to the software engineers in context, i.e. in the variation points of the system, in a way that is both human and machine readable, supporting engineers in understanding these dependencies, and the infrastructure itself enforcing these dependencies at both load time and runtime.

4.1 Modeling dependencies

The first step in the management of feature interaction is the proper modeling of dependencies. One of the most important kinds of dependencies in SPL are the fundamental dependencies. They usually become implicit in the common SPL assets, and

impact the other dependency types previously discussed. For being common to all product line members, these dependencies can be analyzed during the design of the system, being further refined as the infrastructure gets implemented.

We represent the fundamental dependencies in our model in Fig 3, using a notation similar to Ferber and Haag's approach [11]. Note that, in the diagram of Fig 3, we also introduce new dimensions (written in italic) to represent emerging properties of the SPL. In YANCEES, these properties are the *timing*, *routing* and *resource* concerns that change as a consequence of parameters selected in different variation points. Besides the representation of the problem domain dependencies between variation points, dependencies also exist between features within the same variation point and across variation points (as exemplified in Fig 2). For the lack of space, we do not provide a diagram for these dependencies in this paper.

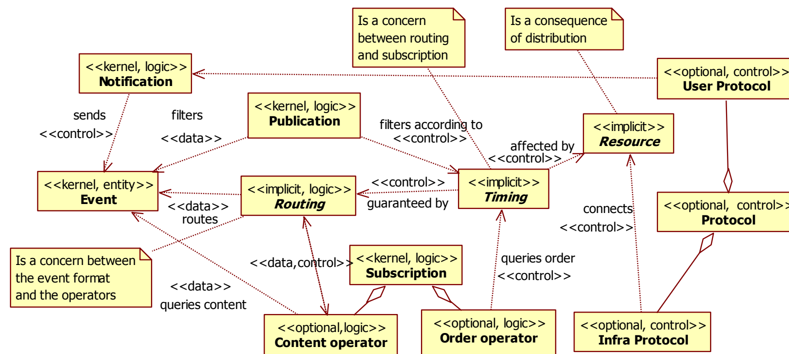

Fig 3. A dependency model of publish/subscribe main variation points and concerns

In the diagram of Fig 3, the event model and its representation directly impacts the subscription and routing models. Timing is another crucial concern in the model. A change in the way YANCEES routers are federated may affect the timing guarantees of the system (guaranteed delivery or total order of events), which will impact the subscription language semantics. A change in the resource model may also affect the timing model. For example, in a hierarchical distributed system, the total order of events may not be feasible. Finally, the notification model is orthogonal to the other features. Since it manages only events, it can vary independent form the other features.

4.2 Representing and managing dependencies in the product line assets

Once the dependencies are identified, they must be formally incorporated in the implementation of the infrastructure. In YANCEES, this is achieved by the use of source code annotations in both the common code and in the feature implementations. In particular, we use the Java annotations API (available since JDK1.5), that permits the creation of custom properties that are associated to classes, methods and fields.

Fig 4 illustrates, in general terms, the main strategies of our approach. The dependencies between the many variation points (emerging properties and fundamental

dependencies) of the system are represented by annotations in the variation points of the code (arrows between variation points and properties in the picture).

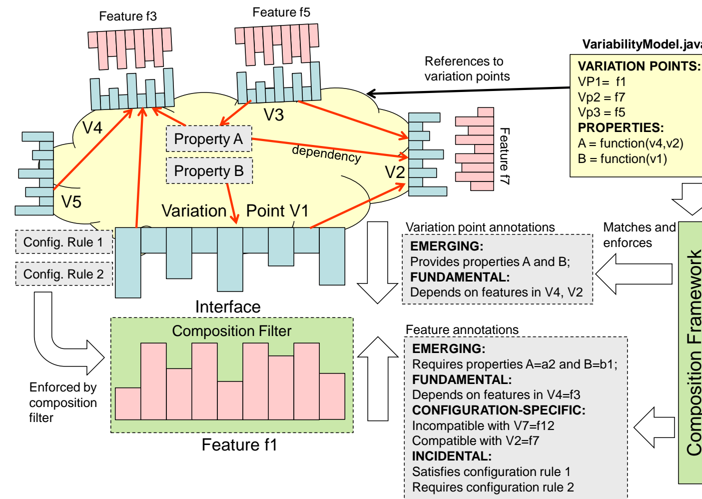

Fig 4 Summary of the approach: managing dependencies with context annotations

The variation point V1, for example, is extended with Feature f1 that has specific emerging, fundamental, configuration-specific and incidental dependencies as described in the figure. Those values are matched with the provided properties of the system. The composition framework, based on the annotations in the code, guarantees that the feature's requirements are met. In other words, all required and provided dependencies are satisfied.

Table 2 Summary of the contextual annotations used in YANCEES

DEPEND.	ANNOTATIONS	DESCRIPTION
Fundamental	`@DependsOnVP` `@DependsOnProperty`	Expresses a general dependency existing between variation points and between properties.
Configuration-specific	`@RequireFeature` `@CompatibleWithFeature` `@CompatiblwWithProperty`	Express a dependency on a specific feature on a variation point. Expresses compatibility with existing features and emerging properties
Traceability	`@ImplementsFeature` `@ImplementsVariationPoint`	Marks classes that implement variation points and features in the code.
Incidental	`@ProvidedGuarantees` `@RequiredGuarantees`	Specifies the provided and required guarantees of the extension

In our implementation, the *VariabilityModel* class (top right of Fig 4), provides a single point of access to the dependency meta-model and the emerging system properties. The emerging properties are encoded in the variability model as rules based on the features installed in each variation point. The main variation points in the infrastructure have their implementation classes referenced in this model, allowing the

navigation through their dependencies by following their annotations. The dependencies between the variation points are encoded in their respective classes using the annotations described in Table 2.

Table 3 sample annotations for the *AbstractFilter* variation point and *SendToPeers* input filter.

```
//--- Indicates fundamental dependencies on other variation points ---
@DependsOnVP(VariabilityModel.VariationPoints.EVENT)

// --- Indicates what variation point this class implements ---
@ImplementsVariationPoint(VariabilityModel.VariationPoints.PUBLICATION)

public abstract class AbstractFilter implements FilterInterface {
   //--- Abstract implementation goes here ---
}
```

```
// --- Local configuration concerns ---
@ProvidedGuarantees(modifyEventContent=false, modifyEventOrder=false,
        modifyEventType=false)
@RequiredGuarantees(intactEventContent=false, intactEventOrder=false,
        intactEventType=false)

// --- Compatibility with features and emerging properties ---
@CompatibleWithFeature(
        variationPontType = VariabilityModel.VariationPoints.EVENT,
        featureClass= edu.uci.isr.yancees.YanceesEvent.class,
        featureName="Event.AttributeValueEvent")

@CompatibleWithProperties(
        resource = VariabilityModel.Resource.ANY,
        routing = VariabilityModel.Routing.ANY,
        timing = VariabilityModel.Timing.ANY)

// --- Feature unique ID ---
@ImplementsFeature(name = "Publication.PublishToPeers", version="1.0")

public class SentToPeersInputFilter extends AbstractFilter {
   // --- plug-in implementation --- }
```

An example of the use of code annotations is presented in Table 3. In this example, two classes are presented: the *AbstractFilter* class that implements the publication variation point and the *SendToPeersInputFilter* which implements "*Publication.PublishToPeers*" feature in the publication model as discussed in section 3.2. These classes are annotated with different tags (highlighted in grey), expressing the local and global dependencies and configuration concerns of this feature. In particular, it expresses the filter intent of preserving the existing order, content and type of the events. It also expresses the guarantees this component requires from the publication variation point. This allows those extensions to require, in this example, that no other component in the chain of responsibility this filter participates with will be able to modify the attributes and content of the events. Annotations also describe the component compatibility with existing concerns and variation point's extensions.

The enforcement of the properties specified in the component annotations is guaranteed, at load time, by the YANCEES architecture manager, which checks for coherent sets of components using the dependency annotations and the information in the architecture configuration file. At runtime, the YANCEES Composition framework,

with the help of the subscription and notification mediators, check for compatibility dependencies and enforce required and provided guarantees. For such, the framework uses composition filters [1] to wrap plug-ins and data elements (events), controlling their access according to the properties provided and required by the filters.

This approach has been used to annotate features and variation points in YANCEES, reducing the feature interference issues discussed in this paper, and helping software engineers in the implementation more robust extensions. One of the advantages of our approach is the ability software engineers have to narrow or broaden the compatibility of a component based on more restrictive or broad compatibility declarations and, in doing so, control the level of enforcement a provided by the infrastructure.

5. Related work

In the field of publish/subscribe infrastructures, different approaches are used to provide flexibility to software [23]. The management of feature interaction in this domain has been, to the best of our knowledge, ad-hoc and not well described. In systems such as FACET [14], for example, the configuration management of features does not directly supports software engineers in extending and in managing feature interaction induced by dependencies.

In software product lines, variability management approaches, as described in the background section, and surveyed by [24], strive to enforce configuration rules and dependencies. Unlike those approaches, we integrate both the dependency model and the runtime guarantees in the system itself. For such, we employ a contextual framework that is part of the product line, bundling in the source code, the information necessary for its extension and customization, together with runtime and load time tools that enforce those constraints.

The use of annotations to elucidate design concerns in the code has been studied as a way to separate and integrate design concerns [5]. Our model applies a similar approach to software product line concerns, with explicit runtime support for the enforcement of those dependencies.

Finally, in the feature interaction community, Metzger et al. [20] proposes an approach for systematically and semi-automatically deriving variant dependencies. Our work complements this approach by providing a practical way of incorporating those dependencies in the management of feature interaction.

6. Conclusions and future work

Dependencies restrict the variability of a system and variability makes managing dependencies difficult. When improperly documented and managed, dependencies lead to feature interference. As a consequence, the benefits of variably require extra configuration and management measures.

The gains in software reuse and variability obtained by the use of software product lines usually come with the increase in the software complexity. This complexity is a

function of the dependencies between the many variation points and the implementation of variability realization approaches. Moreover, the use of software frameworks and other approaches that require direct access to source code, usually suffers from the lack of documentation of these dependencies, and have no automated support for the users in managing those issues. As a consequence those issues represent an important source of programming and design errors in software product line engineering, which can lead to feature interference.

In this paper, we show how those issues may lead to feature interference in publish/subscribe SPLs, and discuss the strategies used to manage feature interaction in YANCEES. In particular, our approach is based on a contextual component framework that uses source code annotations expressing dependencies and configuration rules to support software engineers in extending and configuring SPLs, preventing feature interaction. This approach allows for both static and runtime configuration of components, coping with the dynamism requirements of the publish/subscribe domain.

Currently, the modeling of dependencies in our approach come from the SPL engineers expertise and the manual analysis of dependencies in the code. In the future, we plan on automating the generation of those dependencies by the static analysis of the SPL source code using approaches such as those proposed at [20]. Future work also includes the broadening the scope of our approach, applying it to other flexible software implementations, for example Apache Tomcat.

Acknowledgments. This research was supported by the U.S. National Science Foundation under grant numbers 0534775, 0205724 and 0326105, an IBM Eclipse Technology Exchange Grant, and by the Intel Corporation.

References

1. Bergmans, L. and Aksit, M. Composing Crosscutting Concerns Using Composition Filters. *Communications of the ACM*, *44* (10). 51-58.
2. Birsan, D. On Plug-ins and Extensible Architectures *ACM Queue*, 2005, 40-46.
3. Bosch, J., Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study. in *TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, (1999), Kluwer, B.V, 321 - 340.
4. Bowen, T.F., Dworack, F.S., Chow, C.H., Griffeth, N., Herman, G.E. and Lin, Y.-J., The feature interaction problem in telecommunications systems. in *Software Engineering for Telecommunication Switching Systems*, (1989), 59 - 62.
5. Bryant, A., Catton, A., Volder, K.D. and Murphy, G.C., Explicit Programming. in *1st AOSD*, (Enschede, The Netherlands, 2002).
6. Coplien, J., Hoffman, D. and Weiss, D. Commonality and Variability in Software Engineering *IEEE Software*, 1998, 37-45.
7. Czarnecki, K. and Eisenecker, U.W. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.
8. Deelstra, S., Sinnema, M., Nijhuis, J. and Bosch, J. Experiences in Software Product Families: Problems and Issues during Product Derivation, SPLC'04. *Springer Verlag LNCS*, *3154*. 165-182.

9. DePaula, R., Ding, X., Dourish, P., Nies, K., Pillet, B., Redmiles, D., Ren, J., Rode, J. and Silva Filho, R.S. In the Eye of the Beholder: A Visualization-based Approach to Information System Security. *IJCHS - Special Issue on HCI Research in Privacy and Security*, *63* (1-2). 5-24.
10. Dingel, J., Garlan, D., Jha, S. and Notkin, D., Reasoning about implicit invocation. in *6th International Symposium on the Foundations of Software Engineering (FSE-6)*, (Lake Buena Vista, FL, USA, 1998).
11. Ferber, S., Haag, J. and Savolainen, J. Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. *LNCS. Second International Conference on Software Product Lines, SPLC'02*, *2379*. 235-256.
12. Fowler, M. Inversion of Control Containers and the Dependency Injection Pattern, http://www.martinfowler.com/articles/injection.html, 2004.
13. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.
14. Hunleth, F. and Cytron, R.K., Footprint and feature management using aspect-oriented programming techniques. in *Joint conference on Languages, compilers and tools for embedded systems*, (Berlin, Germany, 2002), ACM Press, 38 - 45.
15. Jacobson, I., Griss, M. and Jonsson, P. *Software Reuse. Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.
16. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E. and Peterson, A.S. Feature-Oriented Domain Analysis (FODA) Feasibility Study - CMU/SEI-90-TR-021, Carnegie Mellon Software Engineering Institute, Pittsburgh, PA, 1990.
17. Krueger, C. Software Product Line Concepts: www.softwareproductlines.com/ introduction/concepts.html, The Software Product Lines site, 2006.
18. Krueger, C.W., Software product line reuse in practice. in *3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, (Richardson, TX, USA, 2000), 117-118.
19. Lee, K. and Kang, K.C. Feature Dependency Analysis for Product Line Component Design. *Lecture Notes in Computer Science - 8th International Conference on Software Reuse, ICSR'04*, *3107*. 69-85.
20. Metzger, A., Bühne, S., Lauenroth, K. and Pohl, K., Considering Feature Interactions in Product Lines: Towards the Automatic Derivation of Dependencies between Product Variants. in *Feature Interactions in Telecommunications and Software Systems VIII*, (Leicester, UK, 2005), 198-216.
21. Rosenblum, D.S. and Wolf, A.L., A Design Framework for Internet-Scale Event Observation and Notification. in *6th ESEC/FSE*, (Zurich, 1997), Springer-Verlag, 344-360.
22. Silva Filho, R.S. and Redmiles, D., Striving for Versatility in Publish/Subscribe Infrastructures. in *5th International Workshop on Software Engineering and Middleware (SEM'2005)*, (Lisbon, Portugal., 2005), ACM Press, 17 - 24.
23. Silva Filho, R.S. and Redmiles, D.F. A Survey on Versatility for Publish/Subscribe Infrastructures. Technical Report UCI-ISR-05-8, ISR, Irvine, CA, 2005, 1-77.
24. Sinnema, M. and Deelstra, S. Classifying variability modeling techniques. *Information and Software Technology*, *49* (7). 717-739.
25. Sinnema, M., Deelstra, S., Nijhuis, J. and Bosch, J. COVAMOF: A Framework for Modeling Variability in Software Product Families. *LNCS*, *3154/2004*. 197-213.
26. Svahnberg, M., Gurp, J.v. and Bosch, J. A Taxonomy of Variability Realization Techniques. *Software Practice and Experience*, *35* (8). 705-754.
27. Szyperski, C. *Component Software: Beyond Object-Oriented Programming, 2nd edition*. ACM Press, 2002.
28. Zibman, I., Woolf, C., O'Reilly, P., Strickland, L., Willis, D. and Visser, J. An architectural approach to minimizing feature interactions in telecommunications. *IEEE/ACM Transactions on Networking*, *4* (4). 582-596.