# The Tension between Interdependencies and Flexibility in Event-Based Infrastructures: managing variability in practice

Roberto Silveira Silva Filho, David F. Redmiles
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3430 USA
{rsivlafi, redmiles}@ics.uci.edu

***Abstract***

*In the past years, new levels of flexibility in software have been required. The need to support variability in software product lines, the multitude of applications in an organization and the fast evolution of software applications has pushed the development of flexible software infrastructures. Practitioners and researches sought for new ways of achieving reuse and variability in their domains. As part of this process, middleware, and more specifically, publish/subscribe infrastructures strive to provide the services demanded by different applications. This forces have resulted in growing research in the are of flexible middleware. As proposed by David Parnas in his seminal paper (Parnas 1978), in order to support this constant demand for change, software must be flexible, i.e. designed not as a single system but as a family of systems that can grow and shrink in order to accommodate new requirements. Publish/subscribe infrastructures are not different. As supporting infrastructures, they need to evolve to support the specific needs of applications that use their services. The implementation of Parnas' flexibility in such infrastructures, however, has shown not to be a trivial task. In special, interdependencies between the many variability dimensions of a pub/sub infrastructures may pose a harsh limit in its flexibility. This paper analyses such dependencies in the context of a flexible architecture for pub/sub infrastructures. It also points out some strategies used to address some of those issues, discussing what was successful and what was not.*

## 1 Introduction

The publish/subscribe architectural style (a.k.a. implicit invocation) provides an inherent loose coupling communication mechanism between information publishers and consumers, which defines clear separation of communication from computation, and carries the potential for easy integration of autonomous heterogeneous components into complex systems that are easy to evolve and scale (Dingel, Garlan et al. 1998). It also defines a one-to-many communication mechanism with which multicast and broadcast-based applications can be implemented. For such characteristics, they have been used as the basic communication and integration infrastructure for many application domains such as software monitoring, awareness, enterprise integration, groupware, distributed user monitoring and so on.

We have been working with event-based architectures in the past years, including EDEM, Knowledge Depot, CASSIUS and other applications, and lately we have been developing a peer-to-peer collaboration tool called Impromptu. By observing the commonalities in those systems, in special, their use of events in different ways (EDEM uses advanced event processing for expectation-driven monitoring, Knowledge Depot employs event (e-mail messages) classification and categorization; CASSIUS supports event-driven awareness applications); and Impromptu (that relies on events for monitoring file access and to perform distributed interface synchronization), we looked for a single event-based infrastructure that could be used in the support of those heterogeneous applications. In other words, we sought a common infrastructure that could be customized and extended to support the needs of different applications. By surveying existing literature and systems (see survey paper), we could not find a single solution that enabled us completely to achieve our goal. Existing solutions were too inflexible (either bulky or too simple), to support such a diversity of requirements.

As a result, we have been developing YANCEES, a common event-based infrastructure that can be tailored to support those different applications. Our goal on the design of YANCEES is the implementation of a

generalized event-based integrator, router and processor that can be extended and customized to address different application scenarios, in special, software engineering and collaborative tools including awareness, software monitoring and application integration.

During YANCEES design, our requirements pointed out for a combination of the routing capability, such as those provided by existing event notification servers with, additional services demanded by different applications such as advanced event processing, event persistency, and different protocols such as peer-to-peer federation. Hence, on the at one end, we have requirements similar to pub/sub infrastructures that focus on providing fast event forwarding; and, at the other end, requirements such as those provided by awareness serves such as CASSIUS that specifically supports organizational awareness applications. As a result, our initial design combines characteristics of pub/sub infrastructures with the extra services of specialized notification servers such as CASSIUS. More specifically, we designed a flexible infrastructure that combined the event routing capabilities provided by existing notification servers, with additional services such as advanced event processing, event source advertisement, multiple notification mechanisms and many other functionality useful to the domains we sought to support.

In the design of any flexible infrastructure, it is important to define, beforehand, the main variability dimensions to be supported. For such, we based our design on an existing model proposed by Rosenblum and Wolf, and our previous experience in implementing pub/sub systems (CASSIUS). We also surveyed existing research and industry pub/sub infrastructures and notification servers such as Siena, Elvin, CORBA-NS and others (See our survey) in order to identify their main characteristics and strategies in supporting different application domains. Based on this theoretical model and on the special requirements of our applications, we proposed our own extensions to Rosemblum and Wolf's model (mainly the addition of the protocol model), to address existing issues with our scenarios.

In our implementation, we used well know existing software engineering techniques such as Object-oriented programming, design patterns, plug-ins and extensible languages. We sought techniques that have been proved in related middleware infrastructures in other domains such as HTTP servers (Apache and Tomcat), Interface tools (web browsers), and IDEs such as Eclipse, where plug-ins are used to add extra features to the server as well on the client side. As a result, decided to apply the same techniques in the development of our own extensible and customizable event-based infrastructure.

The implementation of such an infrastructure, however, showed to be non-trivial. We were not able, using the aforementioned approach, to find a total modular implementation that could address all the requirements proposed. The problem lied in the fact that pub/sub infrastructures have fundamental interdepenendencies in the design dimensions that we proposed to support as our variability dimensions. In other words, the change in one or another design dimension impacts the other design dimension and vice versa. For example, the subscription language is dependent on the event model and representation.

In this paper, we describe our experience in the design and implementation of YANCEES, describing the main problems faced; highlighting where traditional software engineering techniques succeed and where they fail. We then analyze existing attempts to implement flexible pub/sub infrastructures based on a dependency model we propose, showing how designers address the flexibility problem in their domains by fixing one or more variability dimensions and adopting specific software engineering techniques. We then conclude that the use of a dependency analysis in the way exemplified by our model would help in the identification of the main interdependencies in a domain where variability (product line architectures) are being built.

The contributions of this paper are: the analysis of variability of pub/sub infrastructure, by means of a theoretical model that presents the interdependencies between the design dimensions of pub/sub infrastrcutures, a prototypical system, YANCEES, that employs existing variability techniques to implement the infrastructure; an analysis of limitations of those approaches, an analysis of related work and their strategies to handle this problem, a research agenda and promising techniques that can help in addressing the fundamental problems identified here.

## 2 Motivation

Our previous work with awareness, software monitoring and knowledge management systems exemplify different uses of a common communication style, the publish-subscribe, also known as implicit invocation. In the first scenario, events from end user interaction with GUIs are captured and processed and compared with predefined expectations. On another scenario, different applications and gauges use events from different sources in order to leverage awareness in organizations, on a third scenario, events are used to synchronize user interfaces in a peer-to-peer setting (Impromptu), and to monitor security related accesses in each peer repository. Finally, on knowledge depot asynchronous pieces of information (e-mail) are classified according to user-defined filters (or subscriptions), and then processed in the form of summaries that are periodically sent to end users.

On those scenarios, asynchronous information, in the form of events are exchanged, filtered, processed and routed to different parties based on different interests. In other words, they all benefit from the implicit invocation communication style, also known as publish-subscribe. In spite of this commonality, each application require specific functionality in each one of those phases.

Current pub/sub infrastructures are not flexible enough to support the requirements of those applications. In fact, in a previous work (Cleidson's paper), some of the limitations in current technology are discussed, especially with respect to the user interaction and support for different applications. The notification styles, subscription capability, representation of events and other factors limit the applications that can be built. In fact, it is usually the case that the applications need to adapt to the infrastructure and not the opposite. Instead of supporting the application requirements, the infrastructure ends up limiting their possibilities.

What is needed is a way to customize the infrastructure to the application requirements. Traditional ways of developing pub/sub infrastructures (minimal or all-encompassing approaches) cannot adequately support the application needs, and usually result in the best case, the implementation from scratch of new infrastructures. For example, CASSIUS.

The supporting of each one of those scenarios require a combination of services: pub/sub interaction, routing, event processing and application-specific policies. The integration of existing systems is not usually enough to support those systems. One could argue for combining existing event processing layers with pub/sub infrastructures such as Elvin, with database systems and others. The use of those systems as black boxes not

Is pub/sub + extra services enough for my scenarios? I do not think so, but we need an event-based integrator that could respond to our application requirements. For example, client-side filtering, sequence detection capability, QoS such as speed, event representation and so on. We want to support EDEM but not reimplement EDEM in YANCEES alone. Another question is why one needs to extend the service and not add services around it.

CASSIUS case: support the extra event source browsing, persistence and pull notifications

EDEM case: Deploy agents from source to destination; collect event summaries; perform advanced event processing (ECA rules as an agent if possible).

## 3 Background - Software product line families

Software needs to evolve and adapt due to many factors. According to Parnas (Parnas 1978), software change is usually driven by the need to support: (1) Extensions motivated by social, organizational or technological evolution; (2) New and different hardware configurations; (3) Differences in its input and output data, while its function is preserved; (4) Different data structures and implementations due to differences in the available resources; (5) Differences in the size of data input and output; (6) And the need of some users of only a subset of features provided by the software. The idea of supporting software evolution is not new in software engineering. As observed by Parnas, the majority of software systems are not designed for change: instead, they are built to solve specific and well defined problems, which ends up hindering their ability to evolve due to its high costs of maintainability. On the light of this issue, Parnas finally proposes that in order to support evolution and variability, software must be designed and implemented not as a sin-

gle program, but as a family of programs that can be extended and contracted according to different application needs.

Recently, this problem has been the focus of software evolution and product lines research. A software product line aims at reducing the cost of development and evolution of software products by the implementation of a product family. A software product line defines a set of features that are common to all members of a product family, allowing developers to focus on the implementation of the specificities of each product by the reuse of this common feature set (van Gurp, Bosch et al. 2001). A feature is defined by (Bosh 2000) as a *"logical unit of behavior that is specified by a set of functional and quality requirements"*. The ultimate goal is software development cost reduction by the reuse of the existing common software core (which implements a set of required features in the domain). One of the main issues in product line design is the separation of commonality and variability through the analysis and implementation of a general core that can be extended and configured to produce the many product family members. This extension takes place through the incorporation of optional, variant or external features (van Gurp, Bosch et al. 2001) to this common core. The process of building a product line is costly and requires extensible domain analysis. In this context, different domain analysis and software design techniques have been developed in order to identify as early as possible in the software development cycle the commonalities and variable features of the product line family being built. Those techniques include the FODA (Feature-Oriented Domain Analysis) model (Kang, Cohen et al. 1990) and more recently, the FORM (Feature-Oriented Reuse Method) model (Kang 1998), both based on the concept of features. There is a n to m relation between applicaoitn requirements and features that implement them. Also, as will be discussed later, there is not always a one to one relation in a feature and the software components that implement them. In the design of a product line, a tension between generalization and specificity takes place. The more generalizable a product line is, the more valuable it is to the developers. However, this comes with higher development costs and possible feature entanglement. The decision of what to vary and what to fix will impact in the development costs of such an product line infrastructure.

Our goal in this paper is characterize the role of dependencies in the value (or cost) of a product line, helping researchers and practitioners in their decision of what aspect of a product line should be fixed and which one should be variable. In other words, we address the issue of identification of variability in the model proposed by (Svahnberg, Gurp et al. 2005). Moreover, we focus on publish/subscribe infrastructures, but the lessons learned here can be generalized to other kinds of software.

# 4 Main ideas

## 4.1 The ideal world

Ideally, we want an infrastructure where each dimension can be easily customized by the simple selection of existing components that implement each one of the application-required features. In other words, we want to be able to program our infrastructure based on all possible combinations of features that our application may need. In an ideal world, we have an infinite space of design possibilities comprised of the dimensions we want to accomplish. Each dimension hosts a set of components that implement the options of our system. The systems are then composed using each combination of components providing the required functionality set seamlessly. There is a one-to-one mapping between components and features and their entanglement is null, allowing the perfect replacement of the parts of a system, as well as its evolution.

Such ideal world is hard to be found on concrete artifacts and much harder to find in complex artifacts such as software systems.

## 4.2 The problems in designing such systems in general

In practice, however, we need to face many issues related to the parameters we want to vary, the management of the infrastructure, the evolution, the addition of new features, and so on. Some of those issues are discussed elsewhere (Svahnberg, Gurp et al. 2005).

In this paper, we want to focus on two main issues, the first related to the interdependencies between the design dimensions, their parameters we choose to vary, and the second, related to how variability is realized in existing software engineering techniques. In practice this means that designers cannot build a system or parts of it without making assumptions about its context and the interactions that the parts have with the whole. We cannot achieve the proper separation of concerns proposed by Parnas.(Parnas 1972) In other words, it is hard to decompose a system into independent and generalizable parts that together can be used to build not only a single but a family of systems. A good example is provided by Baldwin and Clark in the introductory chapter of their book (Baldwin and Clark 2000), where the authors exemplify this parameter independency phenomena in the design of a simple cup. For example, in a pub/sub system, one cannot assume that the event model being supported is constant, which impacts the whole design of the system as we will show in the next sections.

In other words, the generalization of the design dimensions is difficult since a component in one dimension needs to support a range of options instead of affixed set of input and output parameters. Normal software engineering techniques such as DFDs, for example, assume that the requirements in terms of data format are well defined as a single parameter (or maybe a variety of parameters).

A summary of steps involved in the implementation of a product line architecture together with terminology, concepts of variability and a framework of variability realization patterns is presented by (van Gurp, Bosch et al. 2001). This work, however, does not focus on the role of the interdependencies between the features in the design of a product-line architecture, which is one of the main focus of our work.

### 4.2.1 Coupling (fundamental dependencies)

Coupling is the fundamental problem in design for flexibility. All the other problems are a result of it.

The first issue are interdependencies (or coupling) between the different variability dimensions of a system. Coupling is defined by (Stevens, Myers et al. 1999) as the measure of the strength of associations established by a connection from one module (or design dimension in our case) to another. Different types of coupling exist, such as of data, control and reference as defined by (Stevens, Myers et al. 1999). For example, events and subscriptions have data dependencies.

Myer's define different kinds of coupling, each level more severe than the other, as follows:

0. Independent Coupling - No coupling between the modules.

1. Data Coupling - Two modules are data coupled if they pass data through scalar or array parameters.

2. Stamp Coupling - Two modules are stamp coupled if they pass data through a parameter that is a record. Stamp coupling is perceived as worse than data coupling because any change to the record will affect all of the modules that refer to that record, even those modules that do not refer to the fields that are changed.

3. Control Coupling - Two modules are control coupled if one passes a flag value that is used to control the internal logic of the other.

4. External Coupling - Two modules are external coupled if they communicate through an external medium such as a file.

5. Common Coupling - Two modules are common coupled if they refer to the same global data.

6. Content Coupling - Two modules are content coupled if they access and change each other's internal data state or procedural state.

Page-Jones [13] also introduced the notion of tramp coupling, where data may flow through many intermediate modules from where the data are defined to where they are used. This differs from the other coupling levels in that it measures the coupling among many modules instead of just two modules. Tramp coupling is usually a pathological form of data coupling, but can also be stamp or control coupling.

Coupling exists in every software system and cannot be avoided. In the context of variability, coupling manifest itself in the implementation of the features we want to vary. A feature is an incremental change or option in a product family. Depending on the design of a system, features can be implemented If a feature can be implemented as a single component in a well

**Feature interaction.** In the context of features, coupling manifests itself as the phenomena of feature interaction which is defined by (Griss 2000) as: *"The problem is that individual features do not typically trace directly to an individual component or cluster of components - this means, as a product is defined by selecting a group of features, a carefully coordinated and complicated mixture of parts of different components are involved."*.

**Crosscutting features**. In an ideal situation, features (such as push delivery) are implemented in a single module that can be added and removed from a system in an independent way from other modules. Some features, however, require the presence of other features in the final product. Generally speaking, if a module, that defines a feature in a product line depends on other modules, those dependencies make it impossible to implement this feature as a single component using traditional OO techniques (i.e. in a single object). This phenomena is call feature interaction (Griss 2000; van Gurp, Bosch et al. 2001).

The feature interaction problem becomes more critical when one of the modules that implement a feature is also required in the context of another feature. In this case, this shared module may provide slightly different services to satisfy different contexts (or features). Those features that share a common module are called crosscutting. When features crosscut, a change in one feature may require modifications in those common modules, thus impact other features of the system. In other words, the evolution of a feature that crosscuts other features may indirectly impact other parts of the system.

For example, pull notification requires persistency, a pull method call in the API. This feature is modeled as more than one component in YANCEES. What YANCEES does is to make those variability point first-class entities. The problem with YANCEES is that it is hard to predict all those variability points. Sometimes, one needs to change the internals of the events, for example, in a way not foreseeable in the model.

Moreover, an application requirement may be expressed as a set of features. For example, fast forwarding may require push and a fast switching core, together with a simple subscription language.

**Incompatibilities**. Finally, another issue are the incompatibilities, derived from those inter-dependencies. For example, a timing tag in the subscription requires a timing guarantee from the infrastructure. A followed-by plug-in requires partial or total event sequence guarantee.

**Fundamental and accidental coupling.** We want to introduce here the notion of fundamental coupling. The idea is that some of the dependencies faced in the design of a system can be addressed by the use of different software engineering techniques, whereas other ones are part of the problem itself, and will be present in the design of a system regardless of the software engineering approach used.

For example, in publish/subscribe infrastructures, the notion of information producers, consumers, and of subscriptions (the act of expressing interst ion the information) are essential to the problem. Those components express fundamental dependencies. For example, temporal dependencies (or coupling): consumers cannot receive events before expressing interest on the information. And, in order to express this interest, the existence of the information is usually assumed to be true. This existence may imply in knowing the kind of the content (topic), the structure of it (record), or the content (content-based filtering). Hence, those fundamental dependencies represent options that must be taken into account regardless of the scope of variability that one wants to support. In other words, the scope of variability is a funcitonn of this more fundamental dependency model.

### 4.2.2 Inconsistencies (incompatibilities, invalid configurations or exclusive options)

Because some kinds of coupling require the presence or absence of other components, as well as the proper order of execution between then, flexibility must account for inconsistencies that may occur in the system. Those inconsistencies also known as invalid configurations or exclusive variability options express the same concerns related to the fact that some features are implemented by components that cannot co-exist

with other components or that some features should be implemented by more than one component. Inconsistencies are a result of dependencies and exclusive variations in the architecture of the system.

Example of variability inconsstencies: content-based routing is incompatible with encapsulation of events, employed in many topic-based systems, which externalize only a special field in an event record.

Example of dependency variability: Pull requires event persistency, otherwise it is incomplete (in the case event persistency is modularized as a service in the case of YANCEES).

Some of those inconsistencies are not fundamental, but accidental. They are usually a function on how the infrastructure is implemented, and the architectural styles used. Depending on the implementation, push and pull notifications may not be able to co-exist, if for example, the system assumes that all

Sequence detection cannot be implemented without a total order guarantee of the events from the infrastructure, which can be implemented by many mechanisms as logical clocks or accurate timestamps in the events. If timing is not supported. This is also another kind of coupling.

Timing inconsistencies: some event processing must occur before others. For example, a filter that performs some pre-processing as event time stamping, should come before a filter that classifies those events by timestamps in a persistency repository. Another example, a match must precede an abstraction component or a sequence detector. Components must be placed in the proper order to implement the desired output. This is a consequence of time coupling.

### 4.2.3 Reuse issues

Reuse and customization of existing solutions, our experiences with Siena and Elvin for the case of Impromptu shows that a simplified routing algorithm is better than to use one of those tools since the application may require specific performance. By the use of an open implementation where one can change the internals of the pub/sub infrastructure, such as YANCEES, this can be done.

Reuse of existing solutions is hard: composition versus native implementation [Parnas and Clements]

Right component but wrong implementation due to misassumptions with respect to timing, synchronization and environment. It is also a kind of coupling mismatch.

### 4.2.4 Choosing the appropriate variability realization technique

This problem is addressed by the paper (Svahnberg, Gurp et al. 2005), which surveys the most common strategies applied by the industry in supporting product line architectures and implementing the variability required by them.

## 4.3 The problems in designing event-based middleware specifically

Middleware tension: The need to separate application-specific features (or services) from more general (or generalizable) services, a compromise. In other words, taking a feature-oriented perspective, we must keep the mandatory features and design the variability of the system to accommodate the optional, variant and external features.

Within the mandatory features, there is a tension The tensions involved in the design of Internet-scale pub/sub systems include expressiveness and scalability, common in pub/sub systems: one needs to balance routing, filtering and speed. In other words, even the mandatory features have interdependencies between them, and those interdependencies issues in trade-offs that must be accounted for in the design of a system.

### 4.3.1 Pub/sub feature graph

How those generic software engineering concerns manifest themselves in the pub/sub domain.

In the pub/sub domain, we have the following interdependencies by categories

And the following incompatibilities

### 4.3.2 Feature dependency (or interaction) graph

Here we present the dependency graph based on our feature graph. We show that the interdependencies between the features (crosscutting features), make it hard to design a generic solution to this specific problem.

## 4.4 The solutions (options and trade-offs in the design of such systems)

[HERE WE CAN ORGANIZE THE SOLUTIONS IN THE FORM OF PATTERS (Gamma, Helm et al. 1995):], and relate them to the classification of variability realization techniques proposed by (Svahnberg, Gurp et al. 2005). One of the contribution of this paper is to analyse the applicability of those techniques in the publish/subscribe domain.

Intent

Motivation

Solution

Where in the lifecycle to apply

Consequences

Examples

How existing systems address those interdependencies. List the most interesting strategies here such as adopted by modern systems (attribute/value pair representations), use of extensible languages, component technology, configuration managers, parsers, agents and programming language, reflection, AOP and other strategies listed in the survey paper. Try to categorize them into subgroups of solutions.

At the end, show how YANCEES, more specifically, applies some techniques to achieve flexibility (variability and configurability) our better saying usable flexibility or versatility.

### 4.4.1 Framework versus toolkits

Frameworks fix the control whereas the adaptation points are variable; toolkits fix the interfaces of the components and some parameters, letting the control vary. Hence, toolkits are more flexible. Frameworks are indicated when the variability of control is minimal, and their commonality is maximized. Toolkits are indicated when the control may vary (for example user interfaces), but the elements or components connected by this control have low variability, usually addressed by parameterization techniques.

### 4.4.2 Parameter fixing (variability constraint)

The idea is to limit the variability by fixing one or more dimensions, solving, for example, fundamental cycles or interdependencies. In the case of pub-sub, the common strategy is to fix and adopt a common event model. This is adopted by all infrastructures in fact.

If subscription is not variable, this is also simplified to include the minimal set of operators. For example, Siena fixes its filtering language in order to address expressiveness and scalability.

Other systems, however, use agents to implement subscription, improving its flexibility with programmability.

Example: topic and channel-based event models fix the event representation, or completely handle events as black boxes. They solve the problem of subscription-event representation interdependency.

### 4.4.3 Choosing a variability realization technique

Refer to Bosh's group taxonomy (Svahnberg, Gurp et al. 2005)

The challenge here is to choose a component model that reduces coupling

Filters as used by YANCEES are successful when their coupling are low in the scale. Some filters have been used in YANCEES successfully.

Plug-ins that use the listener pattern are forced to make few assumptions about the environment. They can be further composed by parsers as exemplified by YANCEES. Plug-ins cannot get rid of the dependency with the events if the function they perform requires opening and inspecting the plug-in content. The good news, however, is that if events are handled in a level of abstraction that consider them as black boxes, plug-ins can be very useful as exemplified by YANCEES.

### 4.4.4 Moving dependencies to configuration Managers and parsers

Configuration managers solve the problem of interdependencies. They are used in YANCEES and FACET to enforce composition rules

### 4.4.5 Adopting generalized data representations

For example, use attribute/value pairs

### 4.4.6 Using reflective architectures

Some systems use reflection OpenORB for example.

This is more a consequence. Reflective architectures are ways of deferring the binding decision to runtime. In other words, are ways of reacting to the need of the end user at runtime. Reflection will allow programmers to choose the best component to service a request. This must be matched by design patters such as Abstract Factories, Builders and others

### 4.4.7 Using programming languages as subscriptions

Some papers I surveyed use this approach. They incorporate the ability of programming langueage to handle first-order logica (if statements), as well as their built capabiity for supporting things such as mobile agents (as Java)

### 4.4.8 Using aspects

Here we cite the Jacobsen paper on feature convolution, the FACET paper. Aspects as described in the taxonomy paper (Svahnberg, Gurp et al. 2005), are ways of modifying existing code by combining (or weaving) code fragments that implement the desired variability.

### 4.4.9 How YANCEES address those problems (our approach)

We adopt a feature-oriented view of the problem, defining our variant features along an extended version of the Rosemblum and Wolf design model. With this model and the requirements of the domain, we built the following feature diagram, that shows the variant features (design dimensions), and their variability.

The main idea in the design of YANCEES is the mapping of those variability dimensions into a framework where the variation points are made explicit, they are implemented by a set of techniques using traditional OO programming. Crosscutting features are implemented by providing and combining components from different dimensions in this model. Whereas this is not a perfect solution, it is as far as you can get using traditional software engineering techniques (i.e. non AOP parsers and weavers). As we will see, it addressed many of the issues.

**Extensible languages and plug-ins** (as far as I know, YANCEES is the only one doing it). They provide a layer on top of existing content, topic or channel-based filtering, handled by other YANCEES components that allow the implementation of domain-specific subscription language. The way YANCEES is implemented allows this to be accomplished as long as the plug-ins adopt a common event representation and assumption about the underlying timing dimension. Due to those two interdependencies, this limits the re-

use of plug-ins if some of those dimensions change. However, in many cases, it is successful. The principle applied here is of a process trellis (Factor 1990).

**Input and output filters, wrappers and adapters.** The reason filters, wrappers and adapters work so well is that they allow the interception of the publish and notify methods (filters), or the existing interfaces (adapters and wrappers), working as composition filters (Bergmans and Aksit 2001), introducing points of adaptation that allows the changing of the behavior of the system in response to feature requirements.

Require late binding and parsers. This comes for free from the plug-in and extensible languages model, allowing the extension of the language event at runtime. This has a side-effect of allowing runtime change of the system by installing new plug-ins. This is implemented by the use of factories, parsers, and the use of the listener pattern to implement the plug-ins in the language.

Configuration management (partially through the use of subscription, and partially through the use of a specification language with dependency analysis). Subscription languages use XML and parsers to guarantee that the subscriptions are well formed. They have a side-effect of preventing inconsistent configurations. On the infrastructure side, the same principle prevents inconsistencies. The system performs dependency checks to prevent invalid or incompatible plug-ins from being installed in the architecture.

This shows that many of the practical problems of variability management and software evolution can be addressed by traditional software engineering techniques. In other words, the implementation of a product-line familty and event of some crosscutting concerns can be implemented this way using design patterns and architectural styles that minimize coupling. However, this is not perfect. Changes in timing and the event representation impact the majority of the system.

#### 4.4.9.1 What is not fully addressed

The use of listener pattern, filters and adapters do not fully address the coupling between the event model and the other dimensions, nor it solves the timing problem because if any of those change, some of those components need to change. However, notification policies, sequence detections and other not so event coupled components can be reused.

Open implementation or aspect-oriented architectures may help in this matter by allowing the components of the system to follow the configuration.

The event model can be fixed to attribute/value pairs only. Attribute/value pairs can be used to implement topic and record-based event models. They are more generic. But, there is a trade-off, though. The use of this more generic representation may be less efficient.

### 4.4.10 Promising technologies

Aspect-oriented programming. See the paper of Martin Griss on Implementing Product-Line Features By Composing Component Aspects, and of Jacobsen on managing the convolution problem.

The point here is to model the dependencies as apects that can be weaved on “less dependent” components handling the specifics of each implementation. Dependencies become crosscutting. In the case of pub-sub. The concerns related to the event format are crosscutting.

We propose here the use of aspects to address those specific problems that are not fully implemented in YANCEES, since we believe the OO model provides a more traditional and compatible way of implementing it.

# 5 Related work

A survey of variability techniques is presented here (Svahnberg, Gurp et al. 2005)

# 6 Conclusion

In supporting our collaborative applications, we needed to customize some aspects of pub/sub infrastructures. This customization process was not possible with current pub/sub infrastructures. We searched the literature, current implementations and research for such tools. We decided to build our won. This paper repots our experience in designing such a tool, the main issues faced, the lessons learned. We generalize those ideas in a model and a design approach that can guide the design of similar systems.

Even though there is no universal solution to this problem, designers and software engineers can employ different strategies to manage dependencies and their impact in the design of product line systems. This paper presents some of those approaches to the domain of pub/sub systems and draws some conclusions from our experience with YANCEES.

The main problem of pub-sub is the fact that the events create an implicit contract between producers and consumers of events. Content-based routing also break the encapsulation of the events. This limits the use of filters, plug-ins and extensible languages proposed by YANCEES. In practice, however, designers tend to simplify some of those dependencies and employ software engineering techniques as parsers, configuration managers, rules, and other artifacts to manage the interdependnecies.

# Acknowledgements

This research was supported by the U.S. National Science Foundation under grant numbers 0205724 and 0326105, and by the Intel Corporation

# References

Baldwin, C. Y. and K. B. Clark (2000). Design Rules, Vol. 1: The Power of Modularity. Cambridge, MA, MIT Press.

Bergmans, L. and M. Aksit (2001). "Composing Crosscutting Concerns Using Composition Filters." Communications of the ACM **44**(10): 51-58.

Bosh, J. (2000). Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach, Addison-Wesley.

Dingel, J., D. Garlan, et al. (1998). Reasoning about implicit invocation. 6th International Symposium on the Foundations of Software Engineering (FSE-6), Lake Buena Vista, FL, USA.

Factor, M. (1990). The process trellis architecture for real-time monitors. 2nd ACM SIGPLAN symposium on Principles & practice of parallel programming, Seattle, Washington, United States.

Gamma, E., R. Helm, et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company.

Griss, M. L. (2000). Implementing Product-Line Features with Component Reuse. 6th International Conference on Software Reuse, Vienna, Austria.

Kang, K. C. (1998). "FORM: A Feature-Oriented Reuse Method with Domain Specific Architectures." Annals of Software Engineering **5**: 345-355.

Kang, K. C., S. G. Cohen, et al. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study - CMU/SEI-90-TR-021. Pittsburgh, PA, Carnegie Mellon Software Engineering Institute.

Parnas, D. L. (1972). On the Criteria to Be Used in Decomposing Systems into Modules. Communications of the ACM. **15:** 1053-1058.

Parnas, D. L. (1978). Designing software for ease of extension and contraction. 3rd international conference on Software engineering, Atlanta, Georgia, USA, IEEE Press.

Stevens, W. P., G. J. Myers, et al. (1999). "Structured design." IBM Systems Journal **38**(2-3): 231 - 256.

Svahnberg, M., J. v. Gurp, et al. (2005). "A Taxonomy of Variability Realization Techniques." Software Practice and Experience **35**(8): 705-754.

van Gurp, J., J. Bosch, et al. (2001). On the notion of variability in software product lines. Working IEEE/IFIP Conference on Software Architecture - WICSA'2001, Amsterdam, IEEE.