Meta-level “Change-Design” Instruments for Enhanced Software Changeability during Evolution

Stan Jarzabek
Department of Computer Science
School of Computing
National University of Singapore
stan@comp.nus.edu.sg

ABSTRACT

The central theme of this paper is enhancing the visibility of changes in order to retain good structure, simplicity and changeability of an evolving software system. Other related practical problems addressed in the paper are avoiding explosion of similar component versions, dealing with the impact of multiple changes affecting software during evolution, and reuse of features across system releases, while also allowing each system release to evolve in own direction, interpedently of other releases. We point to certain difficulties to address the above problems with conventional “design for change” and Software Configuration Management techniques. We then propose to alleviate the problems with meta-level extensions to conventional techniques. The proposed approach is based on separation of change specifications (contained in meta-level structures) from architecture- and implementation-level program structures of an evolving system. In short-term, our approach eases day-to-day maintenance. In long-term, the approach provides means to unify similar evolution patterns with meta-level generics, helping us address the above mentioned evolution problems. By “similar evolution patterns” we mean both similar patterns of change specifications, as well as similarities among system components affected by changes. The approach paves the way to reuse-based evolution in which we manage multiple versions of a software system released to customers from a core of generic, reusable software representations.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools And Techniques; D.2.7 [**Software Engineering**]: Distribution, Maintenance, And Enhancement - *restructuring, reverse engineering, and reengineering*; D.2.9 [**Management**]: Software Configuration Management; D.2.10 [**Software Engineering**]: Design – *representations*; D.2.13 [**Software Engineering**]: Reusable Software - *domain engineering*

GENERAL TERMS

Design, Languages, Experimentation

Keywords

software evolution, generative programming, static meta-programming, design for change

1. INTRODUCTION

ISO 9126 quality standards mention four characteristics of maintainable software, namely analyzability, testability, stability and changeability. Still, change is hard, even if design is good and documentation is up to date. In the paper, we consider all kinds of changes affecting software during day-to-day maintenance and long-term evolution, such as fixing bugs, enhancing a program with a new user requirement, supporting a new platform, or, in extreme cases, changing software architecture [21].

Conventional techniques to deal with changes rely on modularization, module layers, abstract interfaces, information hiding [22], parameterization (e.g., generics and C++ templates) and other similar techniques. Software Configuration Management (SCM) tools [5][26] further aid developers in managing changes affecting software over time. Dealing with changes entirely within conventional program solution has well-known advantages, but also some limitations. One such limitation is inability to address changes brought in by crosscutting concerns, addressed by techniques for advanced separation of concerns such as Aspect-Oriented Programming (AOP) [19] and IBM’s hyperspaces [25].

In this paper, we analyze yet other problems that are difficult to address with conventional approaches. The central issue we focus on is enhancing the visibility of changes in order to retain good structure, simplicity and changeability of an evolving software system. Other related practical problems addressed in the paper are avoiding explosion of similar component versions, dealing with the impact of multiple changes affecting software during evolution, and reuse of features across system releases, while also allowing each system release to evolve in own directions, interpedently of other releases. Problems arise when changes are invisibly injected into an evolving system. Each new change is then re-invented afresh and implemented in its own way, despite possible similarities to past changes. Lack of consistency in how we implement changes increases conceptual complexity of an evolving software. Poor visibility of past changes promotes ad hoc implementation changes. It does not allow us to observe (and benefit from) the similarity of evolution patterns. We often redo the same kind of costly change impact analysis, multiply similar program structures (clones), increasing update anomalies, hindering traceability of information, and blowing up the overall volume of the information that must be managed during evolution.

We believe the above problems are fundamental in many evolution situations. If not addressed, they may hinder productivity during evolution, and cause code decay [9]. A common approach is to address change at the design or runtime levels, using one of many available techniques, some of which we discuss in Section 3. Then, Software Configuration Management (SCM) tools [5][26] are used to further aid in evolution. We believe there are inherent problems with handling changes at the level of concrete program components, with conventional techniques. For example, explosion of similar component versions and re-implementing the same functionality

all over again is difficult to avoid even in industrial software product line projects where special attention is paid to reuse (Deelstra et al. [8] report thousands of component versions arising in industrial projects). Similar symptoms are bound to show during evolution.

In this paper, we propose to complement conventional "design for change" and SCM techniques with generative programming [7] technique to enhance the visibility of changes. Meta-level change-design instruments separate change specifications (contained in meta-level structures) from architecture- and implementation-level program structures of an evolving system. The approach works in synergy with any conventional programming technique (including modern component platforms such as J2EE™ and .NET™), and offer features complementary to SCM. First, we apply meta-level instruments to enhance the visibility of changes during day-to-day maintenance. With better understanding of past changes, we then identify similar evolution patterns and unify them with meta-level generic design solutions, retaining changeability in long-term. This unification takes place at the level of both similar patterns of change specifications, and similarities among system components affected by changes over time. The approach leads to *reuse-based evolution* in which we exploit similarities across multiple systems released over years of evolution, and manage all such system releases from a common base of generic program representations.

The meta-level change-design mechanism described in this paper is implemented in XVCL [16][28][31]. In previous papers, we described XVCL as (1) a variability realization technique to implement product line architectures for reuse [23][29][30], and (2) a mechanism for eliminating recurring similar program structures (so-called clones) [2][14]. We reported reduction of conceptual complexity and code size of 60% or more as compared to equivalent conventional solutions. The reduced complexity and non-redundancy of meta-level solutions led to improved reusability and changeability due to smaller number of modifications required to implement enhancements, better traceability, and reduced risk of updates anomalies. However, none of the previous papers described the details of the mechanism that allowed us to achieve the visibility of changes and observe claimed improvements. This paper fills this gap. We describe the details of how changes are specified on top of conventional programs, and then continuously refined, highlighting application of XVCL as a complement to conventional support for software evolution.

We scope the evolution situations addressed in this paper in Section 2. In Section 3, we discuss difficulties to address selected evolution problems with conventional approaches. We discuss meta-level change-design approach in Sections 4 and 5. We evaluate engineering merits of the proposed approach in Section 6. A summary of XVCL experience, in both experimental studies and industrial applications, and concluding remarks end the paper.

2. EVOLUTION SITUTIONS ADDRESSED IN THIS PAPER

The evolution problems and suitability of various techniques to address those problems depend on many factors such as the size of a software system, the nature of similarities and differences among system releases, or quality requirements (e.g., reliability, time performance, or memory consumption). Below, we clarify the assumptions about the evolution situations that we are dealing with in this paper.

We consider situations in which a successful software system S evolves into a family of similar software systems released to various customers over years. In particular:

1) Each such release may implement:
    a) common features shared by all releases of S,
    b) variant features shared with some other releases, and
    c) some unique features.
2) The way the same feature is implemented may vary across system releases.
3) Each release should contain only features that its customers wish to have. This may be important for:
    a) reliability reasons,
    b) strict time performance or memory consumption requirements, for example, in real-time process control software, embedded system software, applications running on mobile devices, etc.
    c) large packages such as SAP that need to be tailored for different operating environments. Because of the huge number of features implemented in such packages, it is important to selectively inject in each custom release only those features that a customer needs.

Specific problems faced in the above evolution situations include:

Feature and component version explosion: A large number of variant features that are implemented in various combinations into released systems. The explosion of look-alike component versions is a common symptom of failing to address this problem during evolution. Some form of generic components capable of unifying groups of similar concrete components is needed to avoid such explosion.

Feature dependencies: The above problem is aggravated by feature dependencies. Feature (or change) dependencies may be of the following kinds: (1) feature f2 is functionally dependent on feature f1 (in short: f1 $^{f}\rightarrow$ f2) if we can't have f2 without f1 in any given system release, and (2) feature f2 is implementation-dependent on feature f1 (in short f1 $^{i}\rightarrow$ f2) if the presence or absence of f1 affects the way f2 is implemented. Feature dependencies cause that the implementation of the same variant feature across system releases may vary, because of the impact of other features.

Reuse of features already implemented in past releases when building new releases. Still, each release should be able to evolve independently of other releases.

Minimizing the effort of selecting/customizing baseline components for building a new release. This effort directly translates into productivity gains, therefore, it is a final test of effectiveness of an adopted approach to evolution.

Selective propagation of new features (or features enhancements) to existing releases that need those features, without affecting yet other releases that do not need them.

We believe the above is a representative, though not complete, list of problems arising in a wide enough range of evolution situations to be concerned about technical solutions that effectively address those problems.

3. RELATED WORK

Change and evolution can be addressed either at the design-time or at runtime. Design-level techniques include parameterization features of programming languages, macros, "design for change" with information hiding [22], OO, component-based design, architectural approaches and Software Configuration Management (SCM) [5][26]. Among newer approaches, there are Aspect-Oriented Programming(AOP) [19] and MDSOC [25], based on Dijkstra's separation of concerns concepts. Runtime techniques for handling change and evolution include switches controlled by parameters that enable/disable code related to variant features, design patterns, dynamic binding, dynamic meta-programming, active libraries "adaptive system" techniques and many others [7]. The choice between design-time or runtime methods to handle changes/evolution is an important engineering decision that should be evaluated based on analysis of trade-offs, taking into account the impact of the solution on the qualities that matter to us, such as program complexity (e.g., maintainability), usability, reliability or performance. While in practical situations we usually apply some mixture of design-time and runtime techniques, in this paper we focus only on design-time techniques.

Generics (in Ada, Java and C#) or C++ templates can effectively deal with type-parametric differences among classes/functions. Generics can be hardly considered a change mechanism as very few changes in evolution practice are of type-parametric nature. Still, type parameterization has a role to play in building generic solutions that in some cases can simplify evolution. Preprocessing and macros [9] work only at the implementation level, in the scope of a single file. These techniques lack parameterization mechanisms powerful enough to unify the variety of similarity patterns that arise in evolution. Failing to utilize design concepts to manage change, attempts to use macros for that purpose soon lead to overly complex code that is difficult to read and test [18].

Inheritance can explicate certain types of changes via sub-classing. It works fine if the impact of change is local, confined to a small number of classes, and, at the same time, if the impact of change is big enough to justify the need of creating new subclasses. Unfortunately, quite often this is not the case and a small change in requirements may lead to big changes due to extensive sub-classing. Even small modification of one class method requires us to derive a new class, repeating almost the same code for the modified method. There is no formal way to group all the classes modified for the same purpose or pinpoint the exact modification points. While inheritance is critical for usability reasons, we are not aware of inheritance playing a significant role in managing day-to-day changes or long-term evolution. Inheritance is not a technique of choice to build generic design solutions, either. For example, STL [24] – a hallmark of generic programming with C++ templates – has a flat class structure. Design patterns [12], playing increasingly important role in modern component platforms such as J2EE™ and .NET™, ease future changes and help in refactoring programs into more maintainable and often more generic representation. Design patterns have an important role to play in unifying similarity patterns during evolution.

In design for change with information hiding [21], we encapsulate changeable elements within program units such as functions, classes or components. When change occurs, if the impact of change can be localized to a small number of program units, we substitute the encapsulated element with new one, and as long as we do not affect interfaces – changing a program is easy. But if we could always localize and hide the impact of change in a small number of components – we would not have problems with changing programs at all. Problems start when the impact of change is not local. Many unexpected changes are of that kind as our existing design may not cater for them.

In architectural and component-based approaches, change and evolution is addressed by designing relatively stable architecture whereby changes can be implemented by replacing components, with possibly minimal impact on component interfaces. This approach is often applied to support software product lines [4]. While software architecture should be a backbone of any large software system, not all changes (or variant features) can be nicely mapped into components, and we still need a mechanism to handle variability to fight the problems of component version explosion or reuse of already implemented features [8].

Aspects (computational concerns that crosscut conventional program modules) are yet other examples of changes that cannot be localized. Aspect-Oriented Programming (AOP) [19] and MDSOC [25] are motivated by the observation that one modular decomposition cannot cater for all kinds of changes that may happen. AOP caters for changes that can be realized by weaving extra code at specified joint points in a program. However, at times, modifications may occur at arbitrary points in a program; modifications required at different program points may be similar but not the same, or completely different one from another; modifications may affect the design (component interfaces or component configuration). To address this kind of changes, we would need to bring in a new aspect for each variant. Such aspects could crosscut base code modules in arbitrary, unexpected ways, and crosscut each other. Chains of modifications could occur within certain aspects – we would need parameterize aspects, possibly with other aspects or using yet other techniques.

Even if a change is small, a typical solution is to create a new component version. This leads to explosion of component versions [8]. A common approach is to use Software Configuration Management (SCM) tools [5] and keep all the component versions in an SCM's repository. Each component version accommodates some combination of variant features. When it comes to implementing a new release of a system, we try to select either individual component versions or their configurations "best matching" the new system. Then, we customize them to fully meet requirements of the new system. This selection/customization effort determines programmers' productivity. As it is not easy to distill the overall picture of similarities and differences across system releases from the SCM repository, selecting "best matching" components and customizing them may become expensive [8]. We have conducted a detailed study of evolution supported by release history stored in CVS [17]. Despite the small size of our study, we observed some difficulties, mentioned above, which could only magnify in large-scale projects. Advanced SCM features such as a change set (or change package) allow developers to form logical groupings of files affected by change [5][20]. Molhado [27] controls changes at the system-level. Various approaches have been proposed to extract and visualize information from the SCM repository to aid in evolution [10][11][13]. These approaches address some of the evolution problems discussed in Section 2. However, we have been unable

to find a detailed enough study demonstrating which evolution challenges discussed in this paper can be mitigated using advanced SCM approaches and how.

4. CHANGE-DESIGN AT META-LEVEL

There is a difference between “change-design” concept described in this paper and “design for change” [22]. “Design for change” is a design goal that we achieve in the frame of a programming language features and design techniques such as modularization, information hiding, various forms of parameterization, table-driven design and many others. The idea of “change-design” is to create a separate meta-level plane to deal with changes, independently of any decisions that have to do with the structure of the actual program. At the meta-level plane, changes become the first class citizens, can be designed and understood in terms of program structures, but without conflicts with any required software properties such as performance, reliability, runtime architecture or component distribution. Discussion in Sections 2 and 3 motivates such separation.

An approach to change-design described below is based on meta-level program manipulation. We use a simplified form of XVCL [28][31] that implements change-design concepts. We introduce change-design instruments and supporting them XVCL Processor in a series of examples. We start by defining change-design instruments that can accomplish any modifications programmers may conceivably want to do. It is important that our mechanism can represent that without limits, in both human- and machine-readable form. Then, we proceed to more advanced mechanisms that allow us define generic solutions unifying similarity patterns in change specifications and similar components of an evolving system. Finally, we sketch the overall picture of evolution as an accumulative result and consequence of applying change-design concepts.

4.1 Specifying a single change

Let U1, …, Un be program units that must be modified to satisfy source of change C. A program unit may be any physical or logical element of a software system, such as a file, class, function/method, component (physical or logical), element of software architecture (a component layer or interface), or a subsystem. For simplicity of explanation, here we assume that units of interest are files. Each affected file Ui may be modified at distinct program points Ui.1, …, Ui.m. At each modification point Ui.j, we should be able to chose either the original code or modified code. Using cpp [18], such versioning without creating another copy of a file, is achieved by #ifdef and #define macro commands. The scope of #define is a single file. To begin with, we introduce a meta-command **<ifdef>** which is analogical to #ifdef. We also introduce meta-variables with a global scope, so that meta-variables can mark and control the chain of modifications related to a given source of change across all the affected files:

```
SPC:
<set x = whatever value>
----------------------------
Ui.j:
<ifdef x>
   modified code for change x
<else>
   original code
<end-ifdef>
```

Files instrumented with meta-commands are called meta-components. **SPC** contains top level change specifications for a given system release (or part of it). The XVCL Processor interprets the **SPC** and injects specified changes into baseline meta-components accordingly. We specify meta-components to be processed by the XVCL Processor in **<adapt>** commands.

```
SPC:
<set x = whatever value>
<adapt U1> <adapt U2> … <adapt Un>
----------------------------
Ui: … @x …
```

The XVCL Processor interprets meta-components, starting with the **SPC**. The <**adapt**> commands determine the exact processing order – whenever the Processor encounters an <**adapt**> command, the interpretation of the current meta-component is suspended and the interpretation of the meta-component designated by the <**adapt**> command begins. The Processor emits on the output any code found in meta-components “as is”, and interprets meta-commands as they are encountered during traversal. Values of meta-variables propagate to all meta-components chained by **<adapt>** commands. Unlike cpp’s #include command which includes a specified file “as is”, **<adapt>** can customize a meta-component, in different ways at each of the many points at which U is <**adapt**>ed. We return to this point later. Globally scoped meta-variables open an easy way to address simple modifications by parameterization. The Processor replaces each reference “@x” (e.g., in **Ui** above) with the value of the meta-variable. Meta-expressions [31] provide a more powerful form of parameterization.

The **<select>** meta-command is similar to **<ifdef>** but it allows us to chose one of the many options based on the value of a meta-variable:

```
<set x = x1>
<select x>
   <option x1> modified code for change x1
   <option x2> modified code for change x2
   <otherwise> original code
<end-select>
```

So far, other than a global scope of meta-variables, our mechanism is not much different from typical preprocessors. The analogy between preprocessors and our change-design mechanism stops here. Other change instruments go beyond capabilities of preprocessors such as cpp, M4 or Oberon.

Notations introduced so far have shortcomings: Firstly, it often happens that the same or similar modification must be done at multiple program points. Using **<select>** alone, we have to repeat the same change specifications in all the **<select>**s relevant to a particular change. For example, advices to be weaved at specified joint points in AOP [19] express such modification in a simple descriptive way. Secondly, using **<select>** alone, similar but not the same modifications have to be specified as if they were totally different. Thirdly, **<select>** is too coarse to show subtle modifications, and to expose all the similarities between the original and modified code.

The **<insert>** meta-command used together with **<select>** provides necessary expressiveness and flexibility in structuring change specifications, to overcome the above problems. The **<insert>** meta-command can replace sections of meta-components at the designated **<break>** points.

```
SPC:
<insert x1>
  modified code needed at break point x1
<insert x2>
  modified code needed at break point x2
<adapt U1> <adapt U2> … <adapt Un>
---
U1:
<break x1>
  original code at x1
---
U3:
<break x2>
  original code at x2
<break x1>
  original code at x1
```

As the effect of **<insert>**, the XVCL Processor replaces the original code contained in **<break**>, called **<break**>'s default, by the modified code supplied by matching **<insert>**. The **<insert>** is matched with corresponding **<break**>s by name. If there is no **<insert>** meta-command matching a specific **<break**>, then the **<break**>'s default code is in force.

The **<insert>** meta-command has two additional forms, namely, **<insert-before**> and **<insert-after**> that insert modified code before or after the matching **<break**>, respectively, preserving the **<break**>'s default.

If required modifications at different program points are similar, it is to our advantage to specify exactly what's common and what's different among them. This results in shorter, easier to understand change specifications. To illustrate this point, suppose modifications required at **<break**>s x1 and x2 (in meta-components **U1** and **U3**) are similar, but not the same. To enhance understanding of similarities and differences in change specifications, we introduce **Generic-x1-x2** (shown below) as a template for required modifications at x1 and x2. Differences among modifications needed at <**break**>s x1 and x2 are catered for by references to meta-variables y and z, and **<break**> diff. Values of meta-variables and modified code required in different contexts are specified at points where **Generic-x1-x2** is <**adapt**>ed in two different ways. Modifications are propagated to components **U1** or **U3**, as indicated by meta-variable 'z' in <**adapt**> meta-command.

```
SPC:
<insert x1>
  <adapt Generic-x1-x2>
    <insert diff>
      <set y = 1>
      <set z = U1>
        modified x1x1x1
<insert x2>
  <adapt Generic-x1-x2>
    <insert diff>
      <set y = 2>
      <set z = U3>
          modified x2x2x2
---
Generic-x1-x2:
….
<break diff> …   @y
  …. @y ….@z ….
<adapt @z>
---
U1 and U3:  the same as in the example before
```

Unique modifications can be expressed either with **<insert>**s, or with **<select**>s or as a combination of both: We **<insert>** into **<select**> options the same/similar change specifications, as well as small variations in otherwise similar change specifications.

Further simplifications of change specification are often possible by using looping supported by meta-command **<while**>. Suppose we have similar types of modifications that occur at five **<break**>s x1, x2, x3, x4 and x5. Again, we wish to specify them enhancing similarities and differences, with minimum repetitions. In the example below, meta-variables 'x' and 'i' are multi-value variables, that represent a list of values, as specified in corresponding **<set**> commands. The **<while**> loop is executed five times. The i'th iteration of the loop uses i'th value of each of the specified multi-value variables: (x1, 1), (x2, 2), etc. Each iteration of the loop specifies modifications to occur at one of the **<break**>s.

```
SPC:
<set x =  x1, x2, x3, x4, x5>
<set i = 1, 2, 3, 4, 5>
<while using-items-in = x, i >
  <insert x>
  <adapt Generic-x1-x2>
    <insert diff>
      <set y = @i>
      <set z = U@x>
        modified @x@x@xn
<end-while>
---
Generic-x1-x2, U1 and U2:  the same as in the previous example
```

This concludes the discussion of single change specifications. In the following sub-sections, we describe how to design generic components, address architectural changes and specify multiple changes, enhancing change similarity patterns. Interestingly, change-design instruments introduced so far are sufficient to deal with all those more advanced situations.

4.2 Generic components and changes at various abstraction levels

A generic component (or a pattern of components) is a template from which we can derive many specific components (or patterns of components) in variant forms. Generic components are a remedy for explosion of component versions. Generics (or templates) are examples of generic components whose instances (classes or functions) can differ in type parameters. Unlike type parameters, generic components built with our change-design instruments can capture any conceivable types of similarities and unify any types of differences among their component instances. Component's parameters are marked by meta-level commands such as references to meta-variables (meta-expressions, in general), **<select**>s and **<break**>s. A parameter may represent a program unit of any granularity – from a single statement (or part of it), to function, class, component or sub-system. Therefore, meta-level parameterization can represent arbitrary differences among groups of similar component instances.

A typical generic component instantiation mechanism is depicted below. We assume that instances x1 and x3 of **Generic-x** require unique modifications, while all the other instances require the same modifications (shown in option **otherwise**).

```
SPC:
<set x =  x1, x2, x3, x4, x5>
<while using-items-in = x >
  <select x>
    <option x1>
      <adapt Generic-x>
```

```
            change specs for instance x1
            e.g., <set y>, <insert b>
        <option x3>
          <adapt Generic-x>
                change specifications for instance x3
        <otherwise>
          <adapt Generic-x>
                change specs for the remaining instances x2, x4, x5
    <end-select>
<end-while>
-----------------------------------------------------------------
Generic-x:
    defines common design and code for components x1, x2, x3,
    x4, x5; in the body of Generic-x is parameterized by
    references to meta-variable @y, <break b>, and <adapt @x>.
```

Generic component **Generic-x** contains parameter slots that mark differences between its instances. In particular, different instances of **Generic-x** include different sub-components via **<adapt** @x> with the name of sub-component given in meta-variable 'x'. Using this simple mechanism, we can decompose a large generic component into a hierarchy of smaller units, applying usual principles of abstraction and separation of concerns. The purpose of such decomposition is to achieve simplicity of change specifications and genericity (i.e., reusability) of generic components.

Any unit of a physical program (such as file, interface definition, class, function or even any part of them) that needs to be a subject of separate manipulation for change specification purpose, may become a meta-component. The scope and scale of change exercised by <**select**>, **<insert>** and other commands may vary: The **<select>**ed code may be just a couple of statements that modify algorithmic details of certain components. But we can also **<select>** the whole **<adapt**>ed component or subsystem, or modify components' interfaces, achieving changes at architectural level. Meta-command <**while**> can be used to generate many instances of components or patterns of collaborating components at any granularity level.

By making certain **<adapt**>s conditional or placing them in a **<while**> loop for multiple adaptation, we can add more flexibility to meta-components (achieving more uniformity in change specifications), for example:

```
LU-1:
<select C>
  <option C1> <adapt @U1>
    …
  <option Cn> <adapt @Un>
</select>
<while using-items-in y>  <adapt @U>  <end-while>
```

By propagating meta-variables across units, **<insert>**ing into **<break**>s, conditionals and looping we achieve change specification goals at all levels of abstraction.

Not only names of **<adapt**>ed components, but also meta-variable names, their values and **<break**> names can be refereed to via meta-variables:

<set @x = @y>
<insert @b>
<break @x>

The mechanism is useful in forming concise generic highly parameterized and flexible design solutions. We refer to [14] and to case studies at [31] for real-world examples of generic component design with meta-level change-design instruments.

5. SPECIFYING MULTIPLE CHANGES

Instruments for single change design described in previous section are sufficient to deal with multiple overlapping changes. In particular, they are sufficient to address problems of evolution discussed in Section 2. In this section, we illustrate basic patterns of meta-level change specifications applied during evolution.

5.1 Evolution of an FRS

Our evolution example follows closely [17], to better contrast meta-level evolution with evolution with CVS described in [17]. A Facility Reservation System (FRS) helps users reserve facilities such as meeting rooms. The evolution scenario is depicted in Figure 1. Circled numbers attached to FRS releases indicate the order in which the releases were implemented. Solid arrows between releases X and Y indicate that X was "most similar" to Y. Dashed arrows between releases X and Y indicate that some features implemented in X were adapted for Y.

An initial FRS, in addition to many other features, supports a method for viewing reservations by facility (FAC, for short). **Stage 1**: Suppose one of our customers requests a new feature, to view reservations by date (DATE). Having implemented the required enhancement, we have two versions of the FRS in use, namely the original FRS and the enhanced FRS^{DATE}. **Stage 2**: After some time, yet another customer wishes to view reservations by user (USER), which results in three versions of the FRS, namely the original FRS, FRS^{DATE} and FRS^{USER}. **Stage 3** We realize that the features DATE and USER are generally useful for other customers and yet another version $FRS^{DATE,USER}$ may make perfect sense to some of them. **Stage 4**: A new customer wants an FRS that supports the concept of payment (PAY) for reservations. This includes computing and displaying reservation charges (RC), cancellation charges (CC), bill construction (BC) and Frequent Customer Discount (FCD). Name PAY refers to all such payment features. **Stage 5**: Another customer would like to make block reservations (BR), as well as support for payment. **Stage 6**: We include block reservation discount (BRD). **Stage 7**: We need an FRS with existing features USER and BR, and finally, in **Stage 8**: A customer asks us to customize the USER feature to view reservations made for only preferred range of dates (USER-PD).

FRS components are organized into the user interface, service and database tiers. Components are implemented in EJB™. The user interface components handle the initialization, display and event-handling for the various Java panels used in reservation management. Server components provide the business logic and the actual event-handling code for the various user interface widgets (e.g., buttons). The database stores data related to users, facilities and reservations.

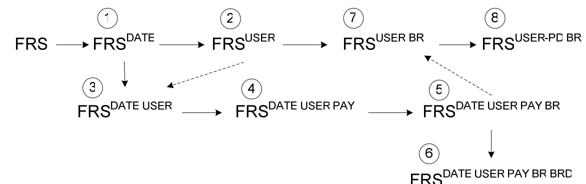

Figure 1. Stages in FRS evolution

Among user interface components, we have a component to display panels for viewing reservations, called ViewResUI. Similarly, among server components, we have a component

ViewResBL that implements business logic for viewing reservations. Components MakeResUI and MakeResBL implement user interface and business logic for making reservations.

We summarize the meaning of various FRS features for ease of reference:

- FAC: view reservations by facility
- DATE: view reservations by date
- USER: view reservations by user
- BR: block reservation – making reservations in bulk
- PAY = {RC,CC,BC,FCD} charges for reserving facilities:
  – RC: computing and displaying reservation charges
  – CC: cancellation charges
  – BC: bill construction
  – FCD: Frequent Customer Discount
- BRD: discount for block reservations
- PD: view reservations for a preferred date

Analysis reveals the following dependencies among features:

- DATE $^{\sim i}\leftrightarrow$ USER (mutually implementation-independent features)
- PAY $^{\sim f}\leftrightarrow$ BR (mutually functionally-independent features)
- PAY $^{i}\leftrightarrow$ BR
- PAY $^{i}\rightarrow$ {DATE, USER}
- PAY $^{i}\rightarrow$ FRS (meaning that most of FRS components are implementation-dependent on PAY)
- {BR, PAY} $^{f}\rightarrow$ BRD

5.2 An overview of FRS evolution

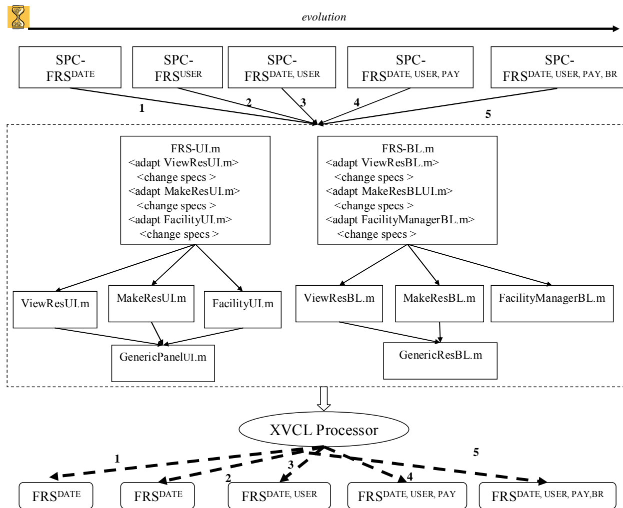

Figure 2. Meta-level support for FRS evolution

In essence, the approach works as follows: we build and refine meta-components incrementally, in response to evolutionary changes (it is analogical to the reactive approach in building product line architectures [8]). Meta-components, organized into a hierarchical structure called a *meta*-architecture (shown within dashed rectangle in Figure 2), contain an accumulative result of all the past changes implemented over time. XVCL Processor can re-construct any of the releases based on change specifications (contained in a respective SPC). Changes whose visibility is not needed anymore become integrated into the base code. This refers to changes that do not display important evolution patterns and changes related to features that have become standard rather than variant features. For past changes that still matter for evolution, we preserve the full knowledge of their impact on the base code in the meta-level change specification format. We continuously refine change specifications to enhance any similarity patterns in changes and in the system itself. This is achieved by designing generic mechanisms to handle similar changes and generic components for groups of similar components.

5.3 Specifying changes for FRS^{DATE}

Required changes include: (1) adding new option for viewing reservations by DATE to the menu in the ViewResUI, (2) adding logic for the new retrieval method and event handling code for the new retrieval method to the ViewResBL, (3) adding a new function to retrieve reservations by date from the database to the ViewResBL. The new feature DATE also affects interfaces of some components, in particular, we must add a new declaration for a data structure to retrieve reservations for a particular DATE.

As at this point we do not know yet if the feature DATE will be useful for other customers or not, we implement changes with **<insert>**s to **<break>**s:

```
SPC: construct FRS
<set affectedComponents =  UI, BL >
<while using-items-in = affectedComponents >
<adapt  "FRS-"@affectedComponents".m" >
-----------------------------------------------------------
FRS-UI.m: view reservation by date
<adapt ViewResUI.m >
  <insert  viewbyDate-Menu-UI >
      viewResMethodList.addItem ("Date");
  <insert viewbyDate-Action-UI >
     else  if  (viewResMethodList.getSelectedItem()  ==
     "Date")
   <insert viewbyDate-EventHandling-UI >
    else  if  (viewResMethodList.getSelectedItem()  ==
"Date")
<adapt>  // other components affected by change
<insert …>
-----------------------------------------------------------
ViewResUI.m:  display reservation viewing methods
class  ViewRes  extends  FRSPanels  implements
ActionListener
{ ...ViewRes(FRSClient client) // constructor
  {   ... viewResMethodList = new List ();
viewResMethodList.addItem ("Facility"); // view res. by
facility
<break viewbyDate-Menu-U I >
code to extend menu with view reservations by date here
</break>
  }
public void actionPerformed (ActionEvent ev) {
Object source = ev.getSource();
if (source == viewButton) {
if (viewResMethodList.getSelectedItem() == "Facility") {
<break viewbyDate-Action-UI >
code for new action to view reservations by date here
</break>
<break  viewbyDate-EventHandling-UI > //
code for new event handler to view reservations by date
</break>
} // end of ViewRes-UI
  original code at x2
…
<break x1>
  original code at x1
```

The change is non-intrusive in the sense that we can obtain the original component by merely removing **<break>** commands. Updating ViewRes-UI for FRS-DATE involves insertions as specified in the SPC. Other modifications of FRS components for DATE are specified in a similar way.

5.4 Specifying changes for FRS^{DATE,USER}

DATE and USER are mutually independent variant features, in both functional and implementation sense. However, they are both implemented in a similar way and we can expect yet other features of that kind in the future. Our change specifications should enhance similarity among any existing and future view reservation features, by providing a generic mechanism to deal with all of them. Below, we show the sketch of the solution.

```
FRS-UI.m : view reservation by DATE and USER
<set viewRes = DATE, USER>
<adapt ViewResUI.m>
```

```
ViewResUI.m: component to display reservation viewing methods
class ViewRes extends FRSPanels implements ActionListener
{   viewResMethodList.addItem ("Facility"); // view res. by facility
<while using-items-in viewRes >
   <select  viewers>
       <option DATE>
              viewResMethodList.addItem ("Date");
         <option USER>
              viewResMethodList.addItem ("User");
   </select>
</while>
... other Java code here
<while using-items-in viewRes>
   <select viewRes>
      <option DATE>
               else if (viewResMethodList.getSelectedItem() == "Date")
      <option = USER>
              else if (viewResMethodList.getSelectedItem() == "User")
   <end-select>
<end-while>
... other Java code here
<while using-items-in viewRes>
   <select  viewRes >
similar code to extend event handlers here
   <end-select>
... the rest of Java code here
```

Meta-variable *ViewRes* specifies reservation viewing methods needed in an FRS, for example DATE and USER. (To include only one feature, we would specify one value for *ViewRes*, namely DATE or USER.) In ViewResUI.m, we **<select>** reservation viewing methods accordingly. Changes for any other mutually implementation-independent features are specified in the similar way.

5.5 Handling feature dependencies

Payment for reservations includes features PAY = {RC, CC, BC, FCD}, explained in Section 5.1. As DATE and USER are implementation-dependent on PAY, any FRS component affected by DATE, USER and PAY features must be sensitive to these implementation dependencies. This is achieved by nested **<select>**s as shown in the example below for FRS component MakeResUI.m:

```
FRS-UI.m
<set viewRes = DATE, USER>
<set PAY = RC, CC>
<adapt MakeResUI.m>
```

```
MakeResUI.m:
<select viewRes>
   <option = DATE>
      <select PAY>
         <option = RC>
         <option = CC>
         <option = BC>
         <option = FCD>
      <end-select>
   <option = USER>
      <select PAY>
        ...
      <end-select>
   <otherwise>
<end-select>
```

Implementation of FRS^{USER,BR} illustrates an interesting case. PAY and BR (block reservation) are functionally-independent (each one may exist with or without the other) but implementation-dependent features (implementation of PAY affects BR and vice versa). Even though, in Stage 5 we implemented BR on top of PAY, we do not have much difficulty in obtaining FRS^{USER,BR} which need BR but without DATE and PAY. As long as we used the same method to handle dependencies of DATE and PAY on BR as we did above to handle dependencies of DATE on PAY, we can easily extract required features from the FRS meta-architecture and reuse them to build FRS^{USER,BR} without manual intervention.

Emergence of implementation-dependent features (or changes, in general) often triggers the need for further decomposition and refinement of existing change specifications. One may be concerned about the increasing complexity of change specifications in view of inter-dependent features. We note that the problem of combining inter-dependent features is inherently complex. The alternative to change specifications is to enumerate component versions for each combination of features of our interest, which is even more complex to work with than change specifications. As shown in above examples, the advantage of handling change specifications at the meta-level is ease of reuse of already implemented features in variant forms that may be needed in different system releases. A practical approach is to explicate only most important feature combinations, leaving addressing less important ones to the customization process.

5.6 Similarity patterns of changes

Similar modifications should be uniquely specified rather than scattered throughout change specifications. In section 4.1, we showed an example of how to capture similarity patterns in single change specifications. Using the same technique, we can capture similarities across multiple changes: We create **<break>**s at relevant **<select>** options and **<insert>** common modifications rather than repeating them at each option.

6. DISCUSSION

We evaluate the presented approach by revisiting the evolution problems discussed in Section 2.

We avoid explosion of similar component versions by designing generic, highly parameterized and adaptable components. The same mechanisms help us avoid repetitions at the change specifications level, at all levels of abstraction. Feature dependencies are dealt with by a combination of nested **<select>**s and **<insert>**s. The problem of dealing with inter-

dependent features in various configurations implemented into different system releases is inherently complex. If we work with concrete components – the problem leads to explosion of look-alike components. If we work at the level of change specifications – the problem leads to complex, crosscutting specifications. We believe that complex change specifications are the lesser of the two evils, as at least change traces are preserved in explicit form rather implicitly in many similar component versions.

At the meta-level, implementation of various features is visibly separated one from another. Traces of changes specific to the context of a given release (due to implementation-dependencies among features or different requirements for the same features in different releases) are also visible. Features can be enabled or disabled by means of meta-variables that control chains of customizations related to a given feature. Such customizations are done at the level of design and code, so the well-known problems of dealing with change at the code level only [18] can be avoided. At the same time, we can evolve each system release independently without breaking the connection with meta-architecture. Because of that, we can propagate new features to specific system releases, without affecting other system releases. Meta-variables with precisely defined scoping, related **<select>** statements, and sometimes extra parameterization with **<insert>**s are the basic means to achieve that. The same mechanism allows us to evolve the meta-architecture with or without affecting specific systems released earlier.

Change specifications enhance similarities and differences across all the systems released during evolution. Therefore, component selection and customization effort is proportional to the degree of novelty of a new system we wish to build. XVCL does not support conflict resolution and does not have any features for collaborative work (such as check in/out facilities to assist parallel development typically supported by Software Configuration Management (SCM) Tools [5][26]. SCM and XVCL complement each other, and we have found only synergy and no conflicts in using them together.

Despite the above conveniences, meta-level change instruments do not come for free. Meta-level change specifications imposed on a conventional software system embrace the family of system released during evolution. Such a generic representation poses its own challenges that must yet to be properly understood. Tool support for visualization, static and dynamic analysis of meta-programs is critical, and we are implementing an XVCL Workbench to meet this demand. Methodological guidelines for meta-level decomposition of change specifications and underlying programs are needed to minimize the scope of a meta-program that has to be analyzed for any given change. This is most essential for XVCL to fully serve project teams as effectively as today it serves small groups of skilled individuals. We included discussion of trade-offs involved in adopting meta-level techniques in papers describing specific studies [14][17][23][29][30].

7. EXPERIMENTATION

XVCL is based on frame concepts, implemented in Frame Technology™, by Netron, Inc.[1]. Frames have been applied to maintain multi-million-line COBOL-based information systems and to build reuse frameworks in companies. XVCL refines original frames into a general-purpose meta-language that blends with contemporary programming and design paradigms. We have applied XVCL in three types of projects: (1) to unify similarity patterns with generic meta-level structures for ease of maintenance in Java class library[14], STL (C++), and J2EE Web Portals [29], (2) to design generic architectures for reuse via product line approach in Java command-and–control systems [30], and ASP Web Portals [23], and (3) to manage variants in domain models [15].

In Java Buffer library [14], meta-level generic representation captured groups of similar classes in a uniform way, simplifying traceability of information, reducing the risk of update anomalies (due to non-redundancy and smaller number of modification points to implement a change), and reduced code size by 68%. This project illustrated a number of issues that have to do with scalability of the change-design mechanism described in this paper: We effectively handled implementation-dependencies among buffer features (our discussion in Section 5.5 is necessarily simplistic). We could represent design of buffer classes affected by feature combinations in generic, parameterized form. The benefit of specifying inter-dependent changes at the meta-level was twofold: (1) we could see the impact of each change on code as well as the mutual impact of changes, and (2) for any legal combination of changes that we needed, the XVCL Processor could produce a suitable custom program.

The FRS example described in this paper is based on the product line project [30]. In product lines, change specifications reflect differences across product line members in respect to generic meta-components forming a product line architecture. The essence of the approach in the product lines is, therefore, very similar to the software evolution situation. In [23][29][30], we described changes that had architectural impact - for example, some of the changes affected the component configuration and component interfaces. For such changes, we specified the whole chain of changes at the architecture and component implementation levels that led to satisfying a given source of change. In two industrial pilot projects, our industry partner ST Electronics observed estimated eight-fold saving in maintenance effort applying the change specification method in C# command and control and Web Portal applications [23].

8. CONCLUSIONS

The described approach is based on technical means (1) to enhance *understanding of past changes,* (2) to delineate *similarities and differences* among systems released during evolution, at any granularity level, and (3) to unify *similarity patterns* of evolutionary changes with generic representations.

Current form of XVCL can be called an *assembly language for change management*. Could we do better by raising the level of abstraction of change specification? XVCL contains the minimum number of low-level constructs to specify any type of changes. We consider adopting specified composition points, such as joint points in AOP [19], in addition to composition points explicitly marked with **<break>** commands. But in general, at this point, we do not know how to raise the level of abstraction of XVCL without compromising its expressive power and without disconnecting change specifications from the actual code which is undesirable for practical reasons [6]. Currently, we address the above problems with XVCL support tools. In the future, we hope to discover meta-level abstractions

that will allow us to define higher-level forms of XVCL, equally expressive but free of current pitfalls.

REFERENCES

[1] Bassett, P. *Framing software reuse - lessons from real world*, Yourdon Press, Prentice Hall, 1997

[2] Basit, H.A., Rajapakse, D.C., and Jarzabek, S. "Beyond Templates: a Study of Clones in the STL and Some General Implications," *Int. Conf. Software Engineering, ICSE'05,* St. Louis, May 2005, pp. 451-459

[3] Brooks, P.B *The Mythical Man-Month*, Addison Wesley, 1995

[4] Clements, P. and Northrop, L. *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002

[5] Conradi, R. and Westfechtel, B. "Version Models for Software Configuration Management," *ACM Computing Surveys*, 30(2), 1998, pp. 232-282

[6] Cordy, J.R. "Comprehending Reality: Practical Challenges to Software Maintenance Automation", *Proc. 11th Int. Workshop on Program Comprehension, IWPC'03*, Portland, Oregon, May 2003, pp. 196-206 (keynote)

[7] Czarnecki, K. and Eisenecker, U. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000

[8] Deelstra, S., Sinnema, M. and Bosch, J. "Experiences in Software Product Families: Problems and Issues during Product Derivation," *Proc. Software Product Lines Conference*, *SPLC3*, Boston, Aug. 2004, LNCS 3154, Springer-Verlag, pp. 165-182

[9] Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S. and Mockus, A. "Does code decay? Assessing the evidence from change management data," *IEEE Trans. Soft. Eng.*, Vol. 27, No. 1, Jan. 2001, pp. 1-12

[10] Fischer, M., Pinzger, M. and Gall, H. "Populating a Release Database from Version Control and Bug Tracking Systsems, " *Proc. Int. Conf. Soft. Maintenance, ICSM'03*, Sept. 2003, pp. 23-32

[11] Gall, H., Jazayeri, M. and Krajewski, J. "CVS Release History Data for Detecting Logical Couplings," *Proc. Int. Workshop on Principles of Software Evolution, IWPSE'03*, Sept. 2003, Helsinki, pp. 13-23

[12] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns – Elements of Reusable Object-Oriented Software*, 1995, Addison-Wesley

[13] German, D., Hindle, A. and Jordan, N. "Visualizing the evolution of software using softChange," *Proc. 16th Int. Conf. on Software Eng. and Knowledge Eng., SEKE'04,* Banff, Canada, June 2004, pp. 1-6

[14] Jarzabek, S. and Li, S. "Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique," *Proc. ESEC-FSE'03, Europ. Software Eng. Conf. and Symp. on the Foundations of Software Eng.*, ACM Press, September 2003, Helsinki, pp. 237-246

[15] Jarzabek, S. and Zhang, H. "XML-based Method and Tool for Handling Variant Requirements in Domain Models", *Proc. of 5th Int. Symp. on Requirements Engineering*, RE'01, August 2001, Toronto, Canada, pp. 166-173

[16] Jarzabek, S., Basset, P., Zhang, H. and Zhang, W. "XVCL: XML-based Variant Configuration Language," *Proc. Int. Conf. on Software Engineering, ICSE'03*, May 2003, Portland, pp. 810-811

[17] Jarzabek, S., Seow, J. and Pettersson, U. "A Case Study in Software Evolution with CVS: Some Problems and Alternatives," submitted to *ICSE'06* http://www.comp.nus.edu.sg/~stan/PAPERS/CVS evolve.pdf

[18] Karhinen, A., Ran, A. and Tallgren, T. 'Configuring designs for reuse', *Proc. Int. Conf. Software Engineering, ICSE'97*, Boston, MA., pp. 701-710

[19] Kiczales, G, Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. Aspect-Oriented Programming," *European Conf. Object-Oriented Programming*, Finland, Springer-Verlag LNCS 1241, 1997, pp. 220-242

[20] R. van der Lingen and A. van der Hoek "Experimental, Pluggable Infrastructure for Modular Configuration Management Policy Composition," *Proc. 26th Int. Conf. on Software Engineering, ICSE'04, Edinburgh, United Kingdom, May 2004,*

[21] Nedstam, J., Karlsson, E.A., and Host, M. "The Architectural Change Process," *Proc. Int. Symp. Empirical Software Engineering, ISESE'04,* Redondo Beach, California, August 2004, pp. 27-36

[22] Parnas, D., 1972. On the Criteria To Be Used in Decomposing Software into Modules, *Communications of the ACM*, Vol. 15, No. 12, December, 1972, pp.1053-1058.

[23] Pettersson, U., and Jarzabek, S. "An Industrial Application of a Reuse Technique to a Web Portal Product Line," accepted for *ESEC-FSE'05, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, September 2005, Lisbon

[24] SGI STL, http://www.sgi.com/tech/stl/ .

[25] Tarr, P., Ossher, H., Harrison, W. and Sutton, S. "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *Proc. Int. Conf. Software Engineering, ICSE'99*, Los Angeles, 1999, pp. 107-119

[26] Tichy, W. "Tools for Software Configuration Management," *Proc. Int. Workshop on Software Configuration Management*, Grassau, 1988, Teubner Verlag, pp. 1-20.

[27] Tien N, Nguyen, Ethan V. Munson, John T. Boyland and Cheng Thao "Architectural Software Configuration Management in Molhado," *Proc. Int. Conf. Software Maintenance, ICSM'04*, Chicago, Sept. 2004, pp. 296-305

[28] Wong, T.W., Jarzabek, S., Myat Swe, S., Shen, R. and Zhang, H.Y. "XML Implementation of Frame Processor," *Proc. ACM Symposium on Software Reusability*, *SSR'01,* Toronto, Canada, May 2001, pp. 164-172

[29] Yang, J. and Jarzabek, S. "Applying a Generative Technique for Enhanced Reuse on J2EE Platform," accepted for *4th Int. Conf. on Generative Programming and Component Engineering*, *GPCE'05*, Sep 29 - Oct 1, 2005, Tallinn

[30] Zhang, H. and Jarzabek, S., "An XVCL-based Approach to Software Product Line Development", *Proc. 15th Int. Conf. on Software Eng. and Knowledge Eng., SEKE'03*, San Francisco, 1 - 3 July, 2003

[31] XVCL (XML-based Variant Configuration Language) method and tool for managing software changes during evolution and reuse, http://fxvcl.sourceforge.net