

Chapter 8 Step-by-step introduction to XVCL

Summary of this chapter:

In this Chapter, we introduce XVCL mechanisms to realize mixed-strategy approach. We explain the rationale for XVCL in the context of maintenance changes and evolution first. We introduce step-by-step XVCL commands in a series of examples, explaining their role in handling various types of changes and evolution situations. In Section 8.9, we summarize XVCL commands in a more formal and systematic way.

A notation described in this Chapter is a simplified form of XVCL, which we use in the remaining part of the book. We use simplified, XML-free XVCL syntax to focus reader's attention on essentials. Appendix C defines XVCL in its XML form, and with all the details, many of which we omit from the presentation in this Chapter.

We recap the main points about building mixed-strategy solutions with XVCL first. We start the detailed description of XVCL by explaining how XVCL allows us to achieve flexible composition to address large granularity changes. Then, we show how XVCL commands allow us to specify small granularity changes. At this point, we already have a complete set of change instruments to specify any modifications programmers may conceivably want to do. Finally, we comment on how XVCL is used to unify similarity patterns in both base program components and change specifications. The former allows us to avoid explosion of similar components. The latter is essential in keeping change specifications possibly simple. As we introduce XVCL conventions, we also comment on their role in addressing challenges of reuse-based evolution.

We would like the reader to note that the same small set of XVCL commands is used for all types of changes, from architecture (at subsystem or component level) down to statement-level modifications.

8.1 Salient features of XVCL

XVCL acts as a complement to conventional programming techniques: Developers still use one of the programming languages to define the behavioral core of their program solutions (e.g., user interfaces, business logic or databases). However, when there are engineering benefits to capture similarity patterns in generic form, but conventional techniques do not allow us to do that, rather than using ad hoc solutions, we escape to XVCL mechanisms to deal with the problem.

Mixed-strategy partitioning is parallel to partitioning of a program along runtime software architecture boundaries which includes subsystems, architectural descriptions, components, interfaces, classes and implementation of all these elements. The runtime architecture is build and expressed using conventional techniques and programming languages, to meet program's behavioral and quality requirements. The boundaries of mixed-strategy partitioning are solely dictated by the concerns of generic design, and are not restricted by the rules of a programming language or the semantics of a programming problem being solved.

The overall generic design solution is decomposed into a hierarchy of parameterized x-frames (denoted by capital letters in Figure 1). X-frames may represent program elements of arbitrary kind, structure or complexity, such as functions, classes, or architectural elements (interfaces, components or subsystems). X-frames can be parameterized in fairly unrestrictive ways. Parameters range from simple values (such as strings), to types and to other x-frames. Parameters mark variation points in x-frames.

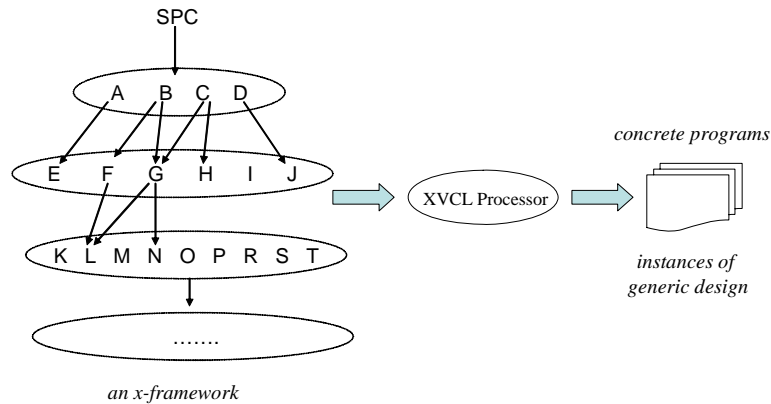


Figure 1. An overview of mixed-strategy approach

Each x-frame in the hierarchy (e.g., G in Figure 1) defines a generic design solution in terms of lower-level x-frames (L and N), and also contributes to generic design solutions at the higher level (B and C), as their building block.

The top-most x-frame, called SPC, SPeCifies global controls and parameters that allow XVCL Processor to synthesize a custom program from a generic x-framework. XVCL Processor interprets the SPC, traverses x-frame hierarchy accordingly, adapts visited x-frames and emits the custom program. By varying specifications contained in the SPC, we can instantiate the same generic design in different ways, obtaining different custom programs.

In the evolution context, an x-framework contains information about systems released during evolution – commonalities and differences among system releases, changes that led to specific releases and other design information that matters during evolution. Each system release is defined by an SPC, and XVCL Processor can produce a given release from the x-framework by interpreting its SPC.

Mixed-strategy representation unifies differences among software clones - similar program structures - with unique generic structures. Variations among similar program structures are specified as deltas from the generic structure and automatically propagated to the respective clones, instances of a generic structure. Such unification of similarity patterns occurs within and across systems released during evolution.

From the XVCL window, a designer has a precise picture of program similarities (a generic structure) and differences among all the instances of a generic structure in a program. Any future changes are also done via generic structures. XVCL provides mechanisms to exercise a full control over the cases when certain instances of a generic structure are to be treated differently from others. Unavoidably redundant code may be emitted as a result, but that code

is no longer the canonical specification of the solution. Non-redundant generic structures together with their instantiating deltas now play that role.

XVCL defines a number of mechanisms to achieve genericity and ease of adaptation: XVCL variables and expressions provide a basic parameterization mechanism to inject genericity into x-frames. Typically, names of entities (e.g., x-frames, architectural elements (such as interfaces and components), source files, classes, methods, data types, keywords, operators or even short algorithmic fragments are represented as XVCL expressions that can be then instantiated by XVCL Processor, according to the context. A `<set>` command assigns a value to a XVCL variable. During processing of x-frames, values of XVCL variables propagate from the x-frame where the value of a XVCL variable is set, down to lower-level x-frames. While each x-frame usually can set default values for its XVCL variables, values assigned to XVCL variables in higher-level x-frames take precedence over the locally assigned default values. Thanks to this scoping mechanism, x-frames become generic and adaptable, with potential for reuse in many contexts. Other commands that help us design generic and adaptable x-frames include `<select>`, `<insert>` into `<break>` and `<while>`. We use `<select>` command to direct processing into one of the many pre-defined branches (called options), based on the value of a XVCL variable. With `<insert>` command, we can modify x-frames at designated variation points - `<break>`s - in arbitrary ways. A `<while>` command allows us to iterate over certain sections of a x-frame, with each iteration generating custom output.

Similarity patterns occur at all the levels of abstraction, from software architecture, to detailed class/component design and to code. Opportunities for program simplification or reuse by capturing similarity patterns in generic form exist at all those levels. To exploit those opportunities, the above XVCL mechanisms are uniformly applied at all levels of abstraction that are involved in formulation of a required generic design solution.

8.2 Flexible composition of x-frames

We introduce XVCL change mechanisms feature-by-feature, from the perspective of various program modifications developers typically do. We hope such presentation helps reader appreciate the rationale behind various XVCL mechanisms, and relate them to usual tasks developers do during maintenance.

For ease of reference, Figure 2 shows main XVCL commands.

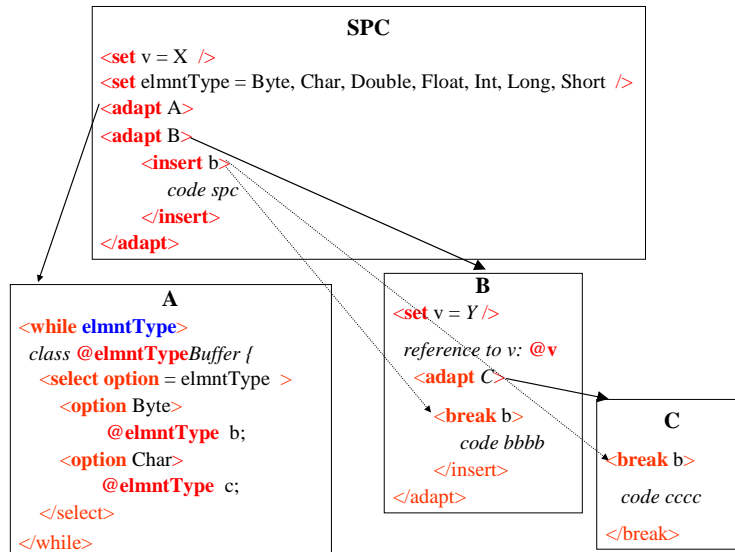


Figure 2. Main XVCL commands

X-frame *composition with adaptation* may be the most fundamental and intuitive way to picture XVCL. We start with the *composition* part.

A popular approach to handling changes is by replacing software components and/or re-configuring components. Different component versions are usually stored in a repository of a Software Configuration Management tool [3] such as CVS. Of course, a new component that replaces an old one must fit into the context of the program. Components after replacement or re-configuration must form syntactically and semantically viable program. #include command of a popular cpp [5] provides a simple form of file composition. PCL [7] offers more sophisticated rules for configuring files. Modern component platforms such as J2EE™ or .NET™ allow us to plug-in plug-out components conforming to interfaces providing the context definition. By interchanging components we can implement certain types of changes.

X-frames are units of mixed-strategy decomposition. The main criterion for decomposition is to preserve the history of an evolving software in a possibly clear and easy to work with representation.

Specific releases of an evolving software (or any custom programs in general) are built by composing x-frames. In XVCL, we compose x-frames by means of **<adapt>** commands. An x-frame may include any number of children x-frames. *Flexible composition* means that composition rules are not completely fixed. As the same x-frame is often customized in different ways depending on the reuse context, the exact composition rules may vary. For example, x-frame A may need x-frame B as its subcomponent in one context, but may not need B in some other context. Variations in composition rules may occur during the same run of the Processor over an x-framework, or in separate runs, for different SPCs. Composition is the core of the XVCL customization mechanism that allows the same x-frames to be adapted in different ways, for reuse in many different contexts. Other fine-granularity XVCL customization mechanisms (such as parameterization, selection or insertions at designated

break points) are specified on top of x-frame composition rules (along adaptation paths) and in the x-frame body.

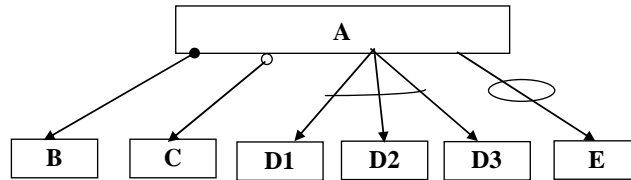


Figure 3. X-frame composition rules

Suppose x-frames A, B, C, D1, D2, D3 and E have the composition structure depicted in Figure 3.

- (1) *mandatory composition*: x-frame B is a mandatory part of A, meaning that in any context when we need x-frame A, B is also used,
- (2) *optional composition*: x-frame C is an optional part of A, meaning that we may or may not need C to build A, depending on the context,
- (3) *alternative composition*: x-frames D1, D2 and D3 are alternative parts of A, meaning that, depending on the context, we need exactly one of them to build A,
- (4) *multiple composition*: x-frame E is used many times in the process of building A.

In the context of modeling evolving software, the above rules often have the following interpretation:

- (1) we need B to build A in any system release, that is, independently of changes affecting A,
- (2) in some system release we need C to build A, but in other system releases we do not need C to build A; whether or not we need C to build A depends on specific changes affecting A,
- (3) different system releases may need D1, D2 or D3 to build A; specific changes affecting A determine which of the three x-frames should be used to build A,
- (4) an x-frame E that is used many times to build A usually is a kind-of a template from which we generate many similar components needed in system releases. Depending on specific changes affecting A, we may need different number of instances of E to build A.

Figure 3 shows one level of x-frame decomposition. In any real evolution situation, we normally decompose x-frames hierarchically, at many levels, as we could see in examples shown in Chapter 7. Such multi-level decomposition is necessary to achieve genericity (reusability) of x-frames across system releases, understanding of similarities and differences among system releases, and a good grasp of changes affecting various system releases.

8.3 Defining compositions with XVCL

Controls in XVCL are exercised by means of XVCL variables, so we briefly introduce variables first. An XVCL variable may be assigned values in any x-frame by <set> command, for example: <set x = 1 />. Variable values are interpreted as character strings.

A variable that is assigned a list of values in the respective `<set>` command is called a multi-value variable, e.g., `<set z = 1, 2, 3 />`. Multi-value variables control loops that facilitate generation of many similar components, e.g., in the multiple composition rule (4) discussed above.

When x-frames are interpreted by XVCL Processor, values of variables propagate from higher level x-frames down to adapted x-frames. A value assigned to a variable in higher-level x-frame takes precedence over any possible values assigned to the same variable in adapted x-frames. This variable value overriding rule has a role to play in achieving reusability of x-frames, so that the same x-frame may be flexibly adapted to needs of different system releases.

8.3.1 Basic x-frame compositions

`<adapt>` commands mark points where x-frame compositions occur during XVCL processing. `<adapt A>` command is analogical to cpp's `#include A` [4]. Unlike `#include` which includes a specified file "as is", `<adapt>` can customize a specified x-frame in different ways, at each of the many points at which x-frame A is needed.

We achieve mandatory composition of x-frame A and B by putting command `<adapt B>` at the required composition point in A.

It is important to keep in mind that XVCL Processor never modifies x-frames. Instead, the Processor emits the result of x-frame composition and other processing to the output. We can imagine that the Processor always works with a copy of an x-frame. Therefore, any reference of the same x-frame in subsequent processing, always uses an original x-frame.

Optional composition is achieved by putting `<adapt>` command inside the conditional `<ifdef>`. For example, the following `<ifdef>` command in x-frame A will conditionally compose B:

```
A:
<ifdef x>
  <adapt B />
</ifdef>
```

The decision whether or not to adapt B is made based on variable x: if variable x has been defined, then the body of `<ifdef x>` (that is, `<adapt B />`) is processed and composition occurs; otherwise, composition does not occur.

(Using cpp, 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.)

Alternative composition is achieved by placing `<adapt>` commands under suitable `<option>`s of a `<select>` command. For example, the following `<select>` command composes one of the x-frames D1, D2 or D3 to build x-frame A, depending on the value of variable x:

```

A:
<select option = x>
  <option d1> <adapt D1 />
  <option d2> <adapt D2 />
  <option d3> <adapt D3 />
  <otherwise> error
</select>

```

Multiple compositions are achieved by putting an **<adapt>** command inside the **<while>** loop. For example, the following **<while>** loop in x-frame A **<adapt>**s x-frame E three times:

```

A:
<set z = 1, 2, 3 />
<while z >
  <adapt E />
</while>

```

Multi-value variable 'z' controls the loop. The i'th iteration of the loop uses i'th value of variable 'z'. By changing the definition of variable 'z', we can change the number of loop iterations, and the number of times x-frame E is adapted in building A.

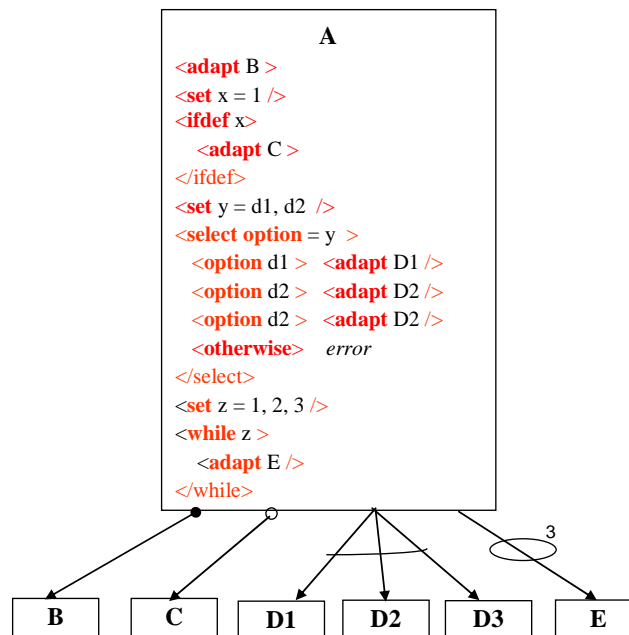


Figure 4. X-frame composition with XVCL commands

8.3.2 More flexible x-frame composition with parameters and insertions

Parameterization can considerably enhance flexibility of x-frame compositions. This is done by representing the name of an adapted x-frame by a reference to an XVCL variable or expression, and by inserting **<adapt>** commands at designated composition points in x-frames. With these extensions, our composition mechanism becomes powerful enough to define generic and flexible x-frame composition structures such as are needed in modeling and representing complex software under long-term evolution.

We can represent the name of an adapted x-frame by a reference to an XVCL variable or expression rather than by a constant given by a character string. A reference to an XVCL variable *x*, written as *@x*, may be placed anywhere in the x-frame body. When encountered during processing, the Processor emits the current value of the variable on the output. Example below explains the basic mechanism. Line numbers are for ease of reference and do not play any formal role in x-frame definition.

B:

1. `<set x = F />`
2. `@x` // the Processor emits F
3. `<adapt @x />` // adapts x-frame F
4. `<set x = FF />`
5. `<adapt @x />` // adapts x-frame FF
6. `<adapt yetOther@x-NEW />`

In line 3, the name of adapted x-frame is given by reference to variable *x*: *@x*. The Processor fetches the current value of variable *x* and `<adapt>`s x-frame F. In line 5, the Processor adapts x-frame FF.

In line 6, reference *@x* is surrounded by text shown in italics (in this case, it is Java code). The Processor concatenates value of ‘*x*’ with surrounding text “as is”, and treats the result as the name of an x-frame to be adapted. Therefore, in line 6, the name of an adapted x-frame is *yetOtherFF-NEW*.

Figure 5 shows yet another situation, where x-frame B is adapted in the context of three different x-frames, namely A1, A2 and A3. When reused in the context of A1, x-frame B `<adapt>`s x-frame F, in the context of A2 – x-frame B `<adapt>`s x-frame G, and in the context of A3 – x-frame B `<adapt>`s x-frame H. We say that value H is a default value of variable *x*, as it takes effect whenever *x* is not overridden by some `<set>` command in one of the higher-level x-frames. This example explains the reason why the value assigned to a variable in higher-level x-frame overrides any possible values assigned to the same variable in adapted x-frames.

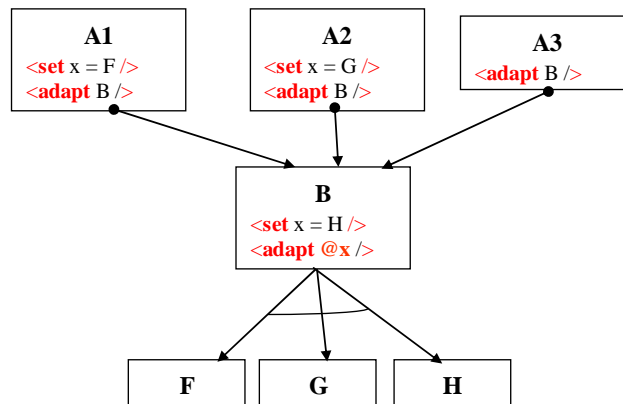


Figure 5. Context-sensitive composition (1)

An example of flexible composition of Figure 5 also illustrates one of the mechanisms to address evolution challenges discussed in Chapter 1: X-frames F and G might provide

implementation of some feature specific to system releases A1 and A2, respectively, while the default x-frame H might provide implementation of that feature for all the other releases (via x-frame A3).

Sometimes the impact of a variation (or change) on the base code is more diverse, and cannot be catered for by varying the names of adapted x-frames only, as this was the case in our last example. Figure 6 and Figure 7 show solutions that cater for such situations. “Customizations” under each of the `<adapt>` commands apply selectively only to x-frames visited on a given adaptation path.

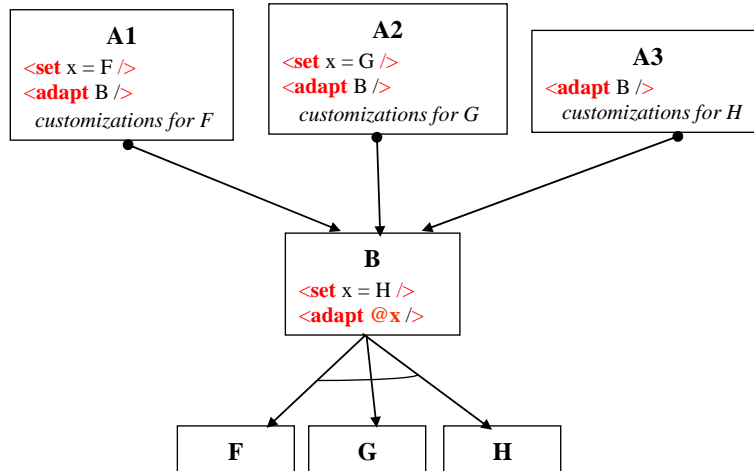


Figure 6. Context-sensitive composition (2)

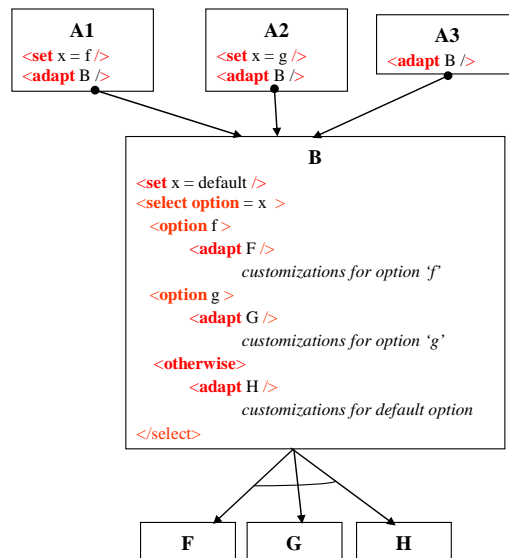


Figure 7. Context-sensitive composition (3)

We can also conduct multiple composition of x-frames by defining their names in a multi-value variable, and by placing parameterized `<adapt>` commands in a `<while>` loop, for example:

B:

```
<set x = F, G, H, I />
```

```
<while x >
```

```
  <adapt @x />
```

```
</while />
```

Consecutive iterations of **<while>** loop **<adapt>** x-frames F, G, H and I, respectively.

Insertion of **<adapt>** commands at designated variation points, called **<break>**s, gives us yet another way to vary composition rules.

We explain the basic rules for **<insert>** into **<break>** XVCL construct first. Command **<insert>** is matched with corresponding **<break>**s by name. As the effect of **<insert>**, the Processor replaces the original code contained in matching **<break>**, called **<break>**'s default, by the modified code supplied by matching **<insert>**. If there is no **<insert>** XVCL command matching a specific **<break>**, then the **<break>**'s default code is in force.

The net result of composition of x-frames in Figure 8 is the same as the situation depicted in Figure 5.

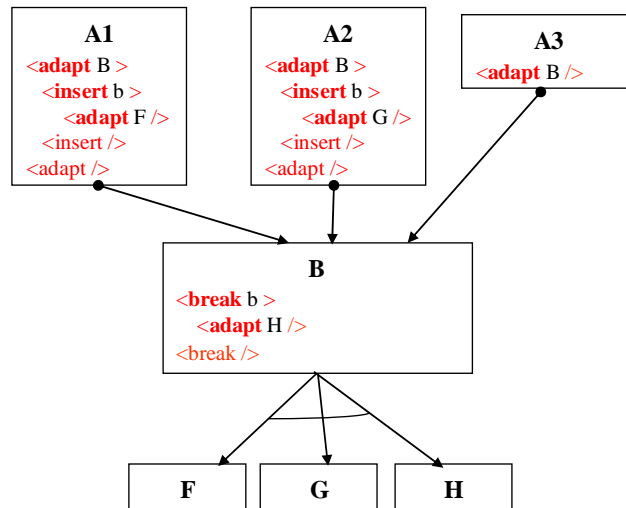


Figure 8. Context-sensitive composition with **<insert>** into **<break>**

An example of Figure 9 models more diverse impact of change on x-frame B (any x-frames adapted by B), in context of x-frames A1, A2, A3 and A4. This impact ranges from a single parameter x (in A1), to parameter x plus adaptation of x-frame G (in A2), to selection of an adapted x-frame based on the value of parameter x (in A3), to multiple composition of specified x-frames (in A4).

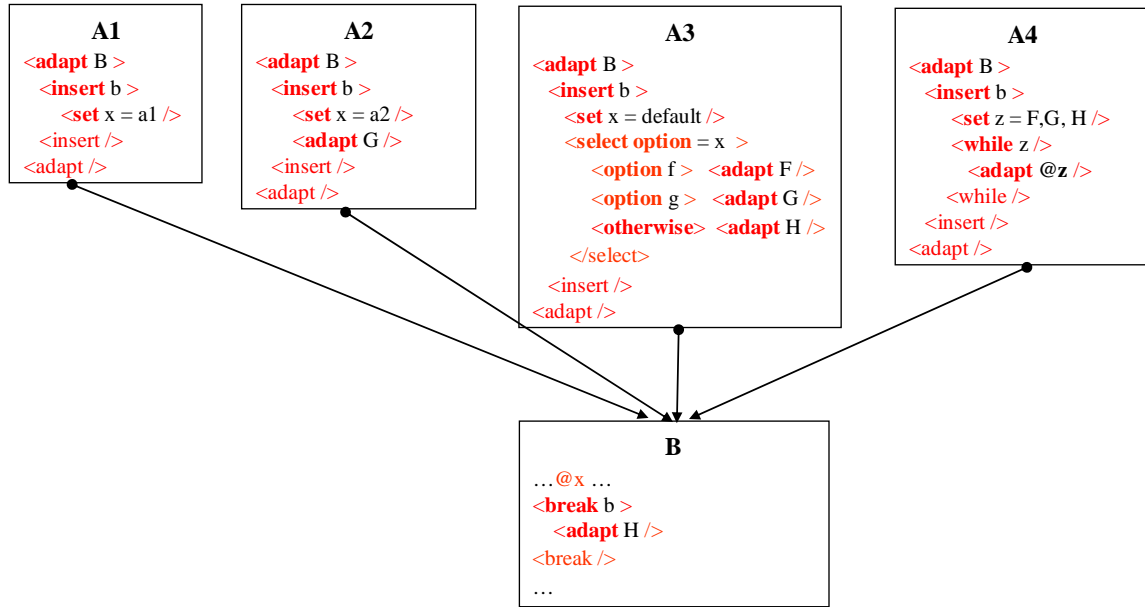


Figure 9. Yet another example of flexible composition

Methods illustrated in the above examples can be combined to achieve the desired properties of flexible composition structure. The main criteria to judge the quality of a solution is reusability across system releases, ease of extending, simplicity and achieving “small change in requirements – small impact on an x-framework” effect.

Most often, x-frames represent large granularity units of change (e.g., change at subsystem, component, or class levels). Flexible composition of x-frames described in this section addresses changes that match granularity of x-frames. By varying changes specifications, the Processor can compose the same composition structure (an x-framework) in different ways, emitting different custom code that we need. Such mechanism is sufficient to deal with large granularity changes, but is not sufficient for small granularity changes.

8.4 Specifying small granularity changes

The same XVCL commands that we used to define flexible composition are also used to specify small granularity changes.

8.4.1 Parameterization

XVCL variables and expressions simply and conveniently address certain types of small changes. In the example below, we vary code emitted from x-frame B by means of a variable that takes different values in two different adaptations of x-frame X.

A:

1. <set x = modified code for first B >
2. <adapt B />
3. <set x = modified code for second B />
4. <adapt B />

For each reference @x in x-frame B adapted in line 2 (and in any x-frame adapted directly or indirectly from there), the Processor uses value “modified code for **first B**”. For each reference @x in x-frame B adapted in line 4 (and in any x-frame adapted directly or indirectly from there), the Processor uses value “modified code for **second B**”.

In Figure 10, x-frame B is parameterized by variables x, y, and z. B also sets default values for those variables. When adapting x-frame B, x-frames A1 and A2 override defaults, while x-frame A3 accepts defaults (unless some other x-frame that adapted A3 set values to x, y or z).

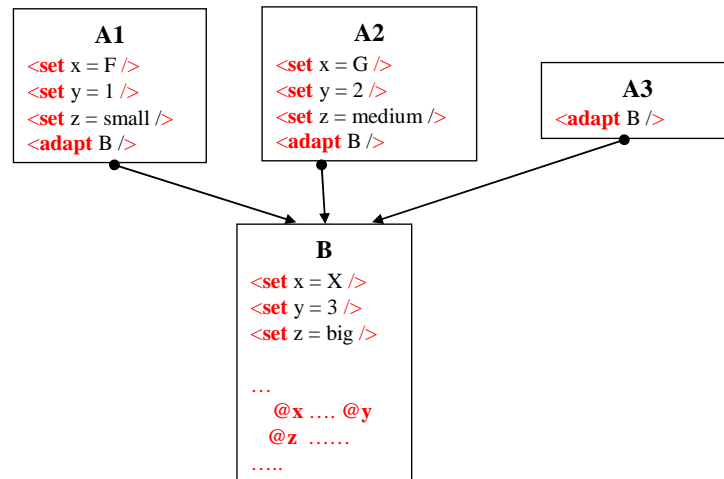


Figure 10. Context-sensitive parameterization

We discuss more about expressions in Section 8.9.

8.4.2 Selection

Command **<select>** is often used to select among the changes relevant to different system releases:

```
<set release = F3 />
<select option = release >
  <option F1> modification for release F1
  <option F2> modification for release F2
  <option F2> modification for release F3
  <otherwise> code for all other releases
</select>
```

<select> commands are used to deal with variant features arising during evolution that have become an integral part of the system evolution history. This usually means features that are shared by a number of system released systems, or can be potentially useful to yet other systems to be released in the future. On contrary, ad hoc enhancements of a single release should be handled in a non-intrusive way, usually by **<insert>**s into **<break>**s, so that they do not pollute or complicate the overall evolution history contained in an x-framework.

8.4.3 Insertions at break points

Small granularity changes can be also specified as `<insert>`s into `<break>`s. The rules here are the same as in defining flexible compositions. An example below shows the basic case:

```
A:
<adapt B>
  <insert b>
    change at b for A
  </insert>
</adapt>
```

and the following breakpoint in x-frame B:

```
B:
...
<break b>
  default code at b
</break>
```

`<insert b>` overrides the “default code at b” contained in `<break b>` by the new code “change at b for A”. Notice that this change only occurs in B as adapted in the context of A. In other contexts in which B is needed, other changes or no changes at all could be specified. In case of adapting B without any changes, the “default code at b” is in force.

`<insert>` commands are always attached to a specific `<adapt>`. In the example above, `<insert b>` applies to x-frame B and to all the x-frames reached in the chain of adaptations that originates at `<adapt B>`.

Here is further illustration:

```
A:
<set x = B1 />
<adapt B>
  <insert b>
    bbbbb
  </insert>
</adapt>

<set x = B2 />
<adapt B>
  <insert b>
    BBBBB
  </insert>
</adapt>

B:
...
<break b>
  default code at b in B
</break>
<adapt C />

C:
<break b>
  default code at b in C
</break>
```

The first `<adapt B>` in x-frame A, uses B1 as value of variable x, and replaces `<break b>` in x-frames B and C with 'bbbb'. The second `<adapt B>` in x-frame A, will use B2 as value of variable x, and replace `<break b>` in x-frames B and C with 'BBBB'.

The scoping and overriding rules governing `<insert>`s and matching `<break>`s are analogical to `<set>` commands assigning values to variables and variable reference points: The first executed `<insert b>` wins – it overrides all the other `<insert b>` commands found in the Processor execution sequence on the way to the matching `<break b>`. Therefore, any `<break>` can be affected by only one `<insert>`. The motivation for this scoping rule is the same as in the case of XVCL variables, namely, to keep lower-level x-frames generic and reusable in as many context as it is practically needed in the evolution situation.

The `<insert>` command has two additional forms, namely, `<insert-before>` and `<insert-after>` that insert change before or after the designated `<break>`s, respectively.

A more detailed discussion of `<insert>`s into `<break>` will follow in Section 8.9.

8.5 Changes at various abstraction levels

As we could see in the last two sections, the scale of change – its scope and granularity – exercised by `<select>`, `<insert>` and other XVCL commands may vary. In one case, we can `<select>` or `<insert>` the whole component or subsystem, or modify components' interfaces, achieving changes at architectural level. Even change of the value of a single XVCL variable can determine a new context that triggers a new chain of customizations, and determines which x-frames should be composed and adapted. In other cases, the `<select>`ed or `<insert>`ed code may be just a couple of statements that modify algorithmic details of a certain component, and we may need an XVCL variable may represent just names of classes, variables, operators or keywords, in a generic form. Similarly, `<while>` command can generate components, subsystems or the application, or can be used to incorporate small changes in certain components.

Any program unit (such as file, interface definition, class, function or even any part of them) that needs to be a separately represented for change specification purpose to adequately model evolution history, may become a unit of change specifications - an x-frame.

We build higher-level logical units of change specifications as grouping of lower-level units by means of `<adapt>` commands. By propagating variables across units, `<insert>`ing into `<break>`s, conditionals and looping we achieve change specification goals at all levels of abstraction.

8.6 Defining generic structures and generators

Genericity plays important role in reuse-based evolution, and makes maintenance changes easier. It is important to achieve genericity in program components that we need adapt to specific need of various system releases. It is also important to unify similar program structures wit generic forms within each release. Finally, change specifications should be also normalized for non-redundancy (i.e., be generic).

XVCL mechanisms of parameterization, selection, insertions contribute to building generic and flexible program representations to meet the above needs. `<while>` loop helps us to

design generators to produce custom instances of generic structures based on the specifications of their required properties.

Generic structures are a remedy for explosion of component versions. Generics (or templates) are examples of conventional generic components whose instances (classes or functions) can differ in type parameters. Unlike type parameters, generic structures built with XVCL can capture any conceivable types of similarities and unify any types of differences among their instances.

A typical generic component and its instantiation mechanism is depicted below. X-frame Generic-x defines a common structure for a group of similar components and their variation points, that is points at which a generic structure can be modified to produce a required custom instance. Variation points are marked with XVCL commands such as variable references, `<select>`, `<adapt>` or `<break>`s. X-frame A iterates over Generic-x. Each iteration produces a custom instance of Generic-x. X-frame A specifies how to modify variation points in each iteration. Various `<option>`s of `<select>` command define unique modifications required at variation points for a given instance. In case of our example, instances x1 and x3 require unique modifications, while all the other instances require the same modifications (shown in option **otherwise**).

A:

```
<set x = x1, x2, x3, x4, x5>
<while x >
  <select x>
    <option x1>
      <adapt Generic-x>
        modification instructions for instance x1
    <option x3>
      <adapt Generic-x>
        modification instructions for instance x3
    <otherwise>
      <adapt Generic-x>
        modification instructions for the remaining instances x2, x4, x5
  </select>
</while>
```

Generic-x:

common design and code for components x1 – x5, parameterized by: references to variable @y, `<break b>`, and `<adapt @x>`.

The mechanism is useful in forming concise generic highly parameterized and flexible design solutions.

Example below, shows slightly different design of a generator for class methods written in Java.

A:

```
<set x = foo />
<set z = 1, 2, 3 >
<set type = int, short, char />
<set arg = a, b, c />
<insert b>
  custom code
```

```

<insert />
<set size = 5 />
<adapt Generic-foo />

Generic-foo:
<while z, type, arg >
@type @x@z(@type @arg) {
// implementation of methods foo() can be customized at variation points by:
// references to variables:
    @x @z @type
// <break>s that can be overridden by custom code from higher-level frames:
    <break b>
        default code
    <break />
// <select> commands at which we can select among a number of options defining custom code
    <select option = size />
        <option 3 > custom code for size 3
        <option 5 > custom code for size 5
        <otherwise > code for any other size
    <select />
// <adapt> commands:
<ifdef extra>
    <adapt Extra_Code />
</ifdef />
</while>

```

X-frame Generic-foo defines a common structure for methods foo() and variation points. X-frame A sets parameters to instantiate variation points. The <while> loop is controlled by three multi-value variables, namely z, type and arg. The t'th iteration of the loop uses the i'th value of each of the control variables, namely (1,int,a), (2,short, b) and (3, char, c). In case of a reference to a control variable in the body of a loop, the Processor emits a variable value as of a current iteration. Each iteration of the loop synthesizes a different header of Java class method, possibly needed in different system releases.

While in a very simple form, the above examples illustrates the main idea of how we build XVCL generic structures and generators. A generic structure is a template from which we can derive its variant forms.

We discuss more examples of generators in later Chapters.

8.7 Capturing change traces and similarity patterns in evolutionary changes

The first step towards understanding evolutionary changes is to keep a record of the whole chain of modifications related to a given source of change. The second step is to unify similar components within and across released systems with generic components (discussed in the last section). The third step is to unify similarity patterns in change specifications arising during evolution.

The “source of change” may be just any reason for changing software such as fixing a bug, enhancing a single or a number of selected system releases with new feature, or modification of certain feature. In XVCL, variables have a global scope and their values propagate from

the variable definition point (in `<set>` command) to adapted x-frames. XVCL variables are a simple yet effective means to chain together detailed modifications across the x-framework related to a given source of change. Variation points are marked with XVCL commands such as `<adapt>`, `<ifdef>`, `<select>`, `<break>`, `<while>` and others.

Using `<select>` command alone may 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. Secondly, sometimes modifications may be similar but not the same. Using `<select>` alone, we have to specify such modifications as if they were totally different. If required modifications at different program points are similar, we should specify exactly what's common and what's different among them. This can result in shorter, easier to understand change specifications. Thirdly, `<select>` is too coarse to show subtle modifications, and to expose all the similarities between the original and modified code.

The `<insert>` XVCL command used together with `<select>` provides necessary expressiveness and flexibility in structuring change specifications, to overcome the above problems.

Similar modifications should be uniquely specified rather than scattered throughout change specifications. Having observed recurring pattern of change specifications, we should try to unify possible differences in change specifications so that similar modifications can be uniquely specified rather than scattered throughout change specifications. One way to achieve such unification is to `<insert>` common modifications into `<option>`s of `<select>` command rather than repeating them at each `<option>`. The following example illustrates the general structure of such a solution:

A:

```
<set x = x1, x2, x3, x4, x5>
<adapt X>
  <insert sameX>
    the same modification in scope of X
  </insert>
  <insert similar1>
    similar modification in the scope of X
  </insert>
  <insert similar2>
    yet other similar modification in the scope of X
  </insert>
</adapt />
```

X:

```
<while x >
  <select option = x >
    <option = x1> default
    <option = x2> <break sameX> ... </break>
    <option = x3> ... <break similar1> ... <break similar2> </break>
    <otherwise> unique modification required in x4, x5
  </select>
</while>
```

Global modifications are under option x1, the same modifications in the scope of component X are under option x2, and similar modifications in the scope of component X are under `<otherwise>`.

Unrestricted parameterization is the essence of the above mechanism for unifying similar patterns of changes. Genericity is needed in both change specifications and generic component design. The same mechanism is used for both. In real programs, we find arbitrary similarities and differences. To deal with them, we need build highly parameterized structures, with parameters ranging from simple values (such as integer or string) to sub-component hierarchies. The `<insert>`ed code or `<select>` option may contain just simple code to cater for the former, and `<adapt @x>` or yet other nested `<select>` XVCL commands to cater for the latter case. Indirect referencing to `<adapt>`ed components and `<break>`s further enhances genericity.

8.8 Handling implementation-dependencies among features

When one modification affects other modifications, things get necessarily more complicated. Typically, such situation is handled by nested `<select>` (or `<ifdef>`) XVCL commands. Below, we sketch the situation of two features, F1 and F2, with two variants of F1 (F1.1 and F1.2) and four variants for feature F2 (F2.1 – F2.4):

SPC:

```
<set F1 = F1.1>
<set F2 = F2.1, F2.3>
```

MakeRes-UI:

```
<select F1>
  <option = F1.1>
    <select F2>
      <option = F2.1>
      <option = F2.2>
      <option = F2.3>
      <option = F2.4 >
    </select>
  <option = F1.2>
    <select F2>
      ...
    </select>
  <otherwise>
</select>
```

XVCL cannot eliminate the inherent complexity of the problem. The question remains if we should specify all the combinations of inter-dependent variant features implemented in systems released during evolution or only some of them. The first solution will allow us to easily reuse existing variant features “as is”. But as change specifications get more complex, it will inevitably become more difficult to customize features for specific systems and evolve the whole evolution architecture. It is necessary to carefully evaluate such trade-offs before making these important engineering decisions.

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. A practical approach is to explicate only most important feature combinations, leaving addressing less important ones to the customization process.

8.9 Summary of XVCL rules

We summarize basic XVCL rules for ease of reference. At times, we also formalize the meaning of XVCL commands that in the previous sections were introduced in very informal way. The reader can find a complete definition of XVCL in its XML syntax in the Appendix X.

8.9.1 Basic XVCL processing rules

The Processor traverses an x-framework in depth-first order, as dictated by `<adapt>`s embedded in x-frames. Processing starts at the top-most x-frame called SPC. The Processor interprets XVCL commands embedded in visited x-frames, and emits a custom program into one or more files. XVCL commands in x-frames are processed in the sequence they appear in the x-frame. Whenever the Processor encounters an `<adapt>` command, e.g., `<adapt B>`, processing of the current x-frame, say A, is suspended, and the Processor starts processing x-frame B. Once processing of x-frame B has been completed, the Processor resumes processing of x-frame A. The processing is completed when the Processor reaches the end of the SPC.

Recursive adaptations are not allowed.

Customization commands are (optionally) specified for each `<adapt>` command. `<adapt B>` command instructs the Processor to:

1. process x-frame B and all x-frames adapted from B
2. perform customizations of all the adapted x-frames as specified in the body of `<adapt B>`
3. interpret any XVCL commands embedded in visited x-frames
4. emit the result to the specified output file.

For a given SPC, the processing flow is a trace that goes through the visited x-frames, starting with SPC. Suppose p1 and p2 are two, not necessarily distinct, points in the same or different x-frames. We say that:

p1 **precedes** p2 (and p2 **follows** p1) if p1 appears before p2 in the processing flow. We use notation: $p1 \rightarrow p2$ to indicate processing flow from p1 to p2.

Given an x-framework and SPC, for any two x-frames A and B, we say that:

- A is an **ancestor** of B, if A `<adapt>`s (directly or indirectly) B in the processing flow defined by the SPC; we call B a **descendant** of A
- A is an **immediate ancestor** (or **parent**) of B, if A directly `<adapt>`s B in the processing flow defined by this SPC; we call B an **immediate descendant** (or **child**) of A. Note that for A to be a parent of B, A must contain an `<adapt>` command whose name attribute yields the name of x-frame B.

X-frames are read-only. The Processor creates and modifies a copy of the adapted x-frame and never changes the original x-frame.

8.9.2 XVCL variable and expressions

Generic names increase flexibility and adaptability of programs and play an important role in building generic, reusable programs. XVCL variables and expressions provide powerful means for creating generic names and controlling the x-framework customization process.

It should be noted that most of the elements in XVCL commands can be defined by XVCL expressions. This includes variable names and values in `<set>` commands, x-frame names in `<adapt>` commands, break names in `<insert>` and `<break>` commands, control variables in `<select>` and `<while>` commands, values in `<option>`s, etc. Only the name of an x-frame must be a defined by a constant.

An XVCL variable may be assigned values in any x-frame by `<set>` command, for example: `<set x = 1>`. Variables are type-less. Variable values are interpreted as character strings.

A variable that is assigned a list of values in the respective `<set>` command is called a multi-value variable, e.g., `<set x = F, G, H >`. The main use of multi-value variables is in controlling loops. A reference to multi-value variable in the loop yields the value of that variable in a given iteration

B:

```
<set x = F, G, H />
<while x >
  @x
  <adapt @x />
<while />
```

In the first iteration of the above loop, the Processor outputs character 'F' and `<adapt>`s x-frame F. In the second iteration, the Processor outputs 'G' and `<adapt>`s x-frame G. In the first iteration of the above loop, the Processor outputs character 'F' and `<adapt>`s x-frame F. In the third iteration, the Processor outputs 'H' and `<adapt>`s x-frame H.

XVCL expressions are formed by variable references as follows:

A direct reference to variable C is written as `@C`. Each extra symbol '@' in the front of a variable name indicates a level of indirection. So:

`@C` means value-of (C)

`@@C` means value-of (value-of (C))

`@@@C` means value-of (value-of (value-of (C))), and so on.

XVCL processor replaces references to variables by variables' respective values. Here, we should mention, that XVCL processor stores all the variables defined so far in the Symbol Table along with their current values, as assigned to variables in `<set>` and `<set-multi>` commands.

The Processor reports an error for a reference to a variable that does not exist in the Symbol Table.

Table 1 gives an example of Symbol Table we use in the examples.

Table 1. The Symbol Table with variables

<i>Name</i>	<i>Value</i>
C	U
U	BU
BU	V
AV	W

For example, the value of @C is U, the value of @@C is BU, and the value of @@@C is V.

So-called Name Expressions are written as follows: ?@A@B@C?. the Processor interprets this Name Expression as: value-of (“A” | value-of (“B” | value-of (C))), where symbol ‘|’ means text concatenation.

For example, referring to the Table 1, the evaluation of Name Expression, ?@A@B@C? is done as follows:

1. get the value of variable C
 - the intermediate result is U
2. concatenate B and U and get the value of variable BU
 - the intermediate result is V
3. concatenate A and V and get the value of variable AV
 - the final result is W.

After each evaluation step, the intermediate value computed is concatenated with the character string on the left to form a new variable name that is looked up in the Symbol Table. Evaluation of a Name Expression continues until the whole Name Expression is evaluated.

8.9.3 Variable scoping and propagation rules

Variable scoping rules are the same for both single-value and multi-value variables.

Within an x-frame, each subsequent assignment of value to a variable overrides any previous <set> that assigns value to that variable. Variable values propagate to adapted x-frames, as shown in Figure 11.

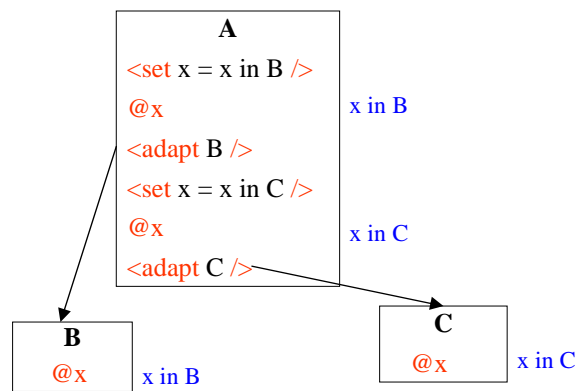


Figure 11. Variable value propagation

During processing, values of variables propagate through the whole adaptation chain of x-frames visited by the processor. The `<set>` command in the ancestor x-frame overrides `<set>` commands in all the subsequently processed descendent x-frames. That is, once an x-frame `<set>`s the value of variable `x`, any further attempts to re-define that variable `x` in descendent x-frames will be ignored. Example below clarifies the variable propagation rule.

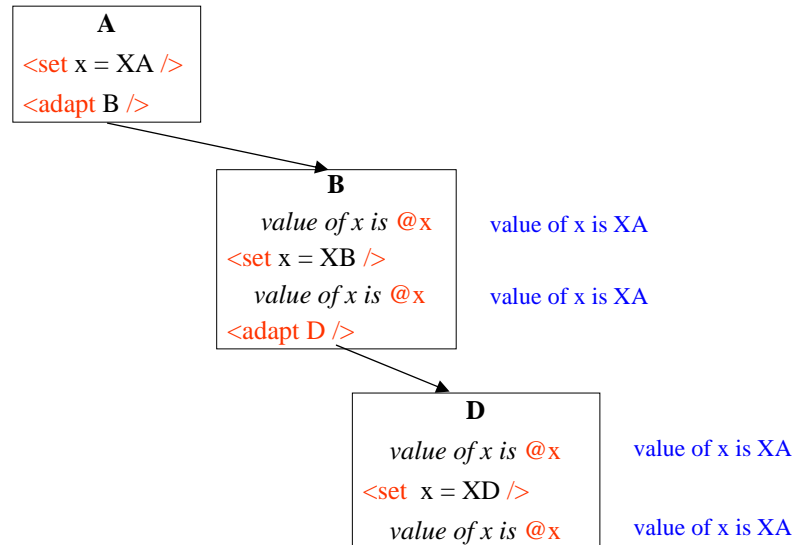


Figure 12. Variable value propagation along adaptation chains

It is a good practice to make an x-frame `<set>` default values to variables it refers to. In case no ancestor x-frame `<set>`s value to a given variable, the default value is in force.

A reference to undefined variable the Processor treats as an error that terminates processing of an x-framework.

8.9.4 `<select>` command

`<select>` command is used to select zero or more of the pre-defined branches (`<option>`s) for further processing. `<select>` contains zero or more `<option>`s followed by an optional `<otherwise>`. For example:

```

<select option = x />
  <option a > option-body
  <option b > option-body
  <option c > option-body
  <otherwise> option-body
</select>
  
```

Options are selected based on the value of control variable. If control variable is undefined (that is it does not exist in the Symbol Table), the Processor issues an error message and terminates processing.

An option-body may contain code written in one of the base languages and any XVCL commands.

The XVCL processor checks **<option>**s in sequential order and selects for processing **<option>**s as follows: the value of the control variable of **<select>** is compared against the *values* specified at each **<option>**. Each **<option>** whose value matches the value of the control variable is selected for processing in the order of their appearance in **<select>** command. The Processor processes each of the selected option clauses immediately upon selection. If none of **<option>**s is selected, then **<otherwise>** is processed, if present.

A value specified in **<option>** may be an expression. We refer to details to the Appendix X.

8.9.5 **<while>** command

<while> command iterates over its body. **Multi-value** variables listed in **<while>** are loop's control variables. All control variables must have the same number of values. Each loop iteration uses the *i*th value of each of its control variables, therefore, the number of iterations is equal to the number of values in each of the multi-value control variables.

The while-body may contain code and XVCL commands. Values of the loop's multi-value control variables can be referenced in the while-body. If used inside **<while>** command, these multi-value variables behave like a single-value variables and can be referenced just like a single-value variable.

A **<while>** loop in the example below generates three Java class methods foo, with some variants in the header and an empty body.

A:

```
<set z = 1, 2, 3 >
<set type = int, short, char />
<set arg = a, b, c />
<adapt Generic-foo />
```

Generic-foo:

```
<while z, type, arg >
@type @x@z(@type @arg) { }
</while>
```

The loop is controlled by three multi-value variables, namely z, type and arg. The *t*'th iteration of the loop uses the *i*'th value of each of the control variables, namely (1,int,a), (2,short, b) and (3, char, c). In case of a reference to a control variable in the body of a loop, the Processor emits a variable value as of a current iteration. Each iteration of the loop synthesizes a different header of Java class method, possibly needed in different system releases.

8.9.6 **<ifdef>** and **<ifndef>** commands

These tow commands have the following syntax:

```

<ifdef x >
  if-body
</ifdef>
<ifndef x >
  if-body
</ifndef>

```

Variable specified in **<ifdef>** or **<ifndef>** is a control variable. If a control variable is defined (i.e., it exists in the Symbol Table), the Processor processes the if-body of **<ifdef>** command. Otherwise, the if-body is ignored. The **<ifndef>** command acts in the opposite way – its if-body is processed only if a control variable is undefined.

The if-body may contain code and any XVCL commands.

8.9.7 <insert> into <break> command

<break> marks a variation point in an x-frame. **<insert>** replaces the contents of matching **<break>**s (e.g., *default x in b* in Figure 13) with new contents defined in **<insert>** (e.g., *xA* in Figure 13). If there is no **<insert>** matching a given **<break>**, the default contents contained in the **<break>** is processed. **<insert>** matches **<break>** by name.

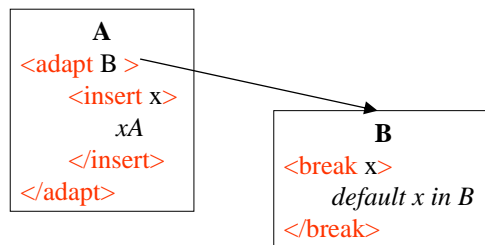


Figure 13. A basic rule for **<insert>** into **<break>**

<insert> commands can only be specified in the body of an **<adapt>** command. In Figure 13, **<insert x>** is specified for **<adapt B>**. **<insert>** propagates to all the subsequently adapted x-frames and may reach matching **<break>**s there (Figure 14). Notice that **<insert x>** does not affect **<break x>** in x-frame C.

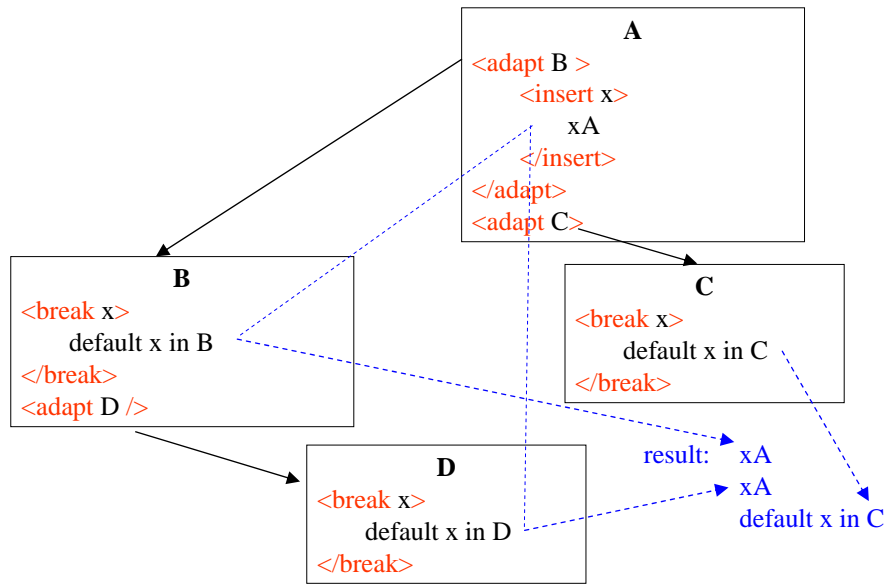


Figure 14. **<insert>** into multiple **<break>**s

Only **<insert>** command executed first in the processing flow matches (affects) a **<break>**.

This first executed **<insert>** overrides **<insert>**s commands that may appear in all the subsequently adapted x-frames, as illustrated in Figure 15.

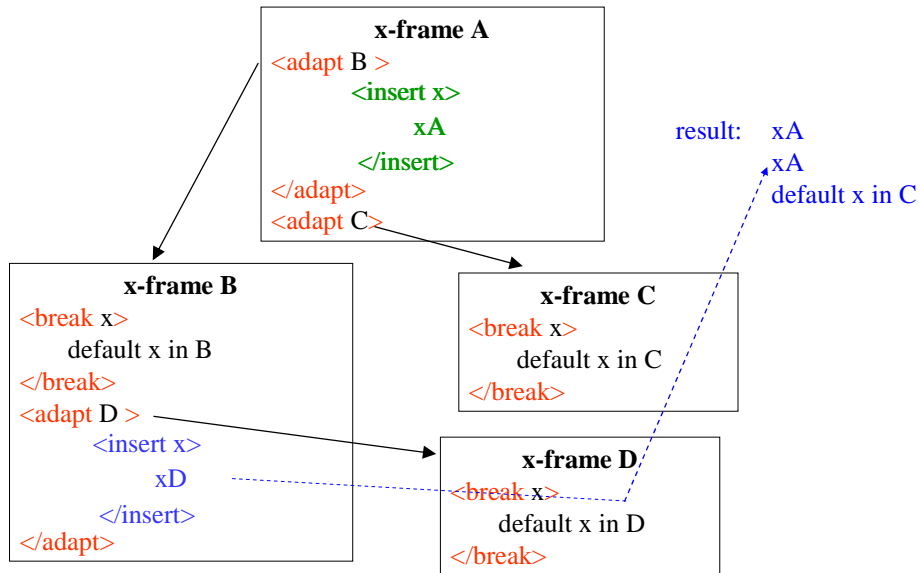


Figure 15. Upper-level **<insert>** overrides lower-level **<insert>**s

This **<insert>** overriding rule, similar to variable value overriding rule, has the purpose of making x-frames adaptable and reusable in multiple contexts (Figure 16).

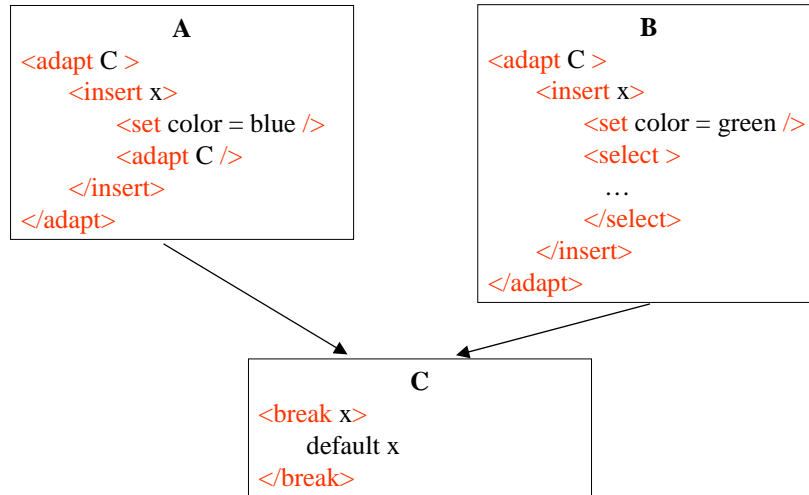
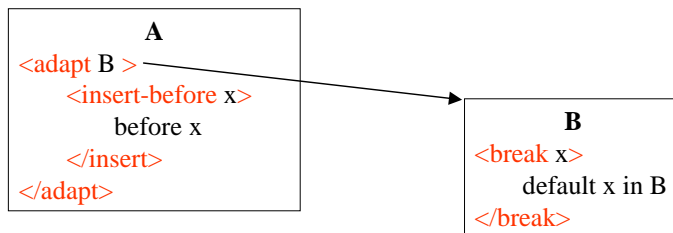


Figure 16. X-frame C adapted with `<insert>`s in two contexts

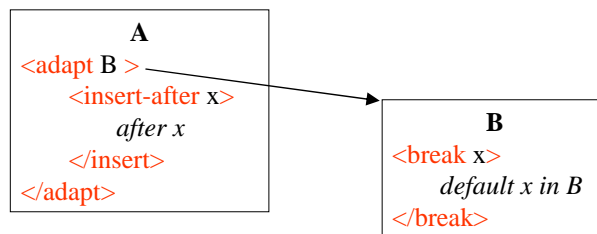
Commands `<insert-before>` and `<insert-after>` are similar to `<insert>`. `<insert-before>` inserts its Insert-Content before the matching `<break>`s commands.

The `<insert-before>` inserts its contents before the matching `<break>`s. The `<insert-after>` inserts its contents after the matching `<break>`s. Notice that the `<insert-before>` and `<insert-after>` do not replace the `<break>`'s default contents.



result : before x
default x in B

Figure 17. `<insert-before>`



result : default x in B
after x

Figure 18. <insert-after>

Propagation and overriding rules for <insert-before> and <insert-after> are the same as for <insert>: first <insert-before> overrides any subsequent ones, and first <insert-after> overrides any subsequent ones.

Each of the <insert>, <insert-before> and <insert-after> commands may match (affect) the same <break>, yielding an accumulative result, as shown in Figure 19 and Figure 20.

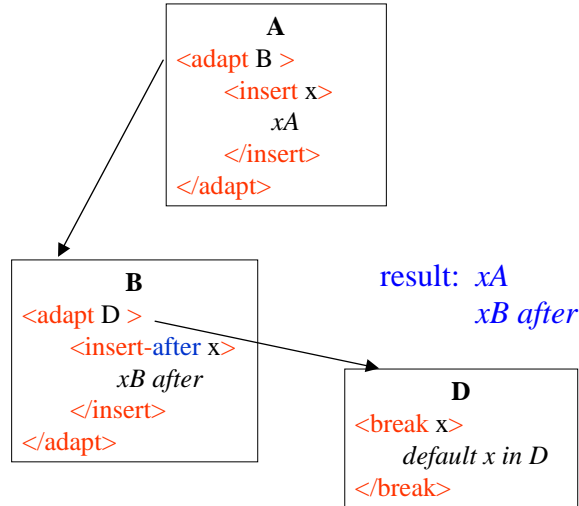


Figure 19. Accumulative result of <insert> and <insert-after>

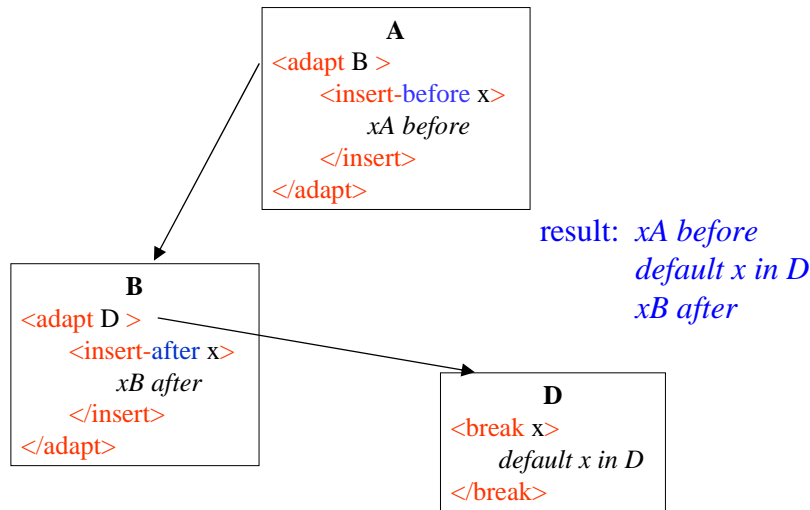


Figure 20. Accumulative result of <insert-before> and <insert-after>

More rules governing <insert> into <break> are described in Appendix X.

8.9.8 Attributes **outdir** and **outfile**

Attributes **outdir** and **outfile** specify the directory name and the file name, respectively, to which the XVCL Processor should emit the processing output. These attributes may be

attached to x-frame or to a specific <adapt> command. Values of **outdir** and **outfile** may be given by an expression.

Figure 21 illustrates how attributes **outdir** and **outfile** are used to direct the output to different files and directories.

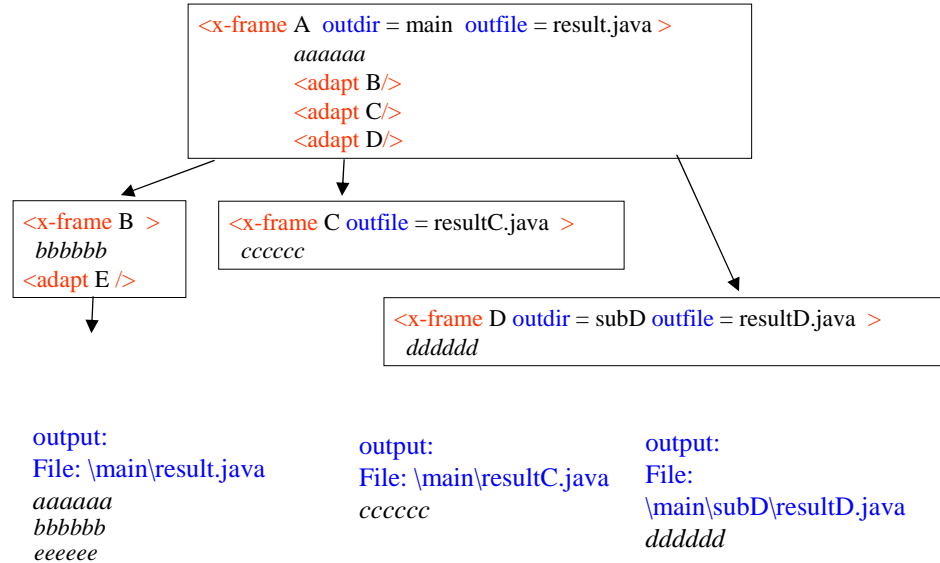


Figure 21. Attributes **outdir** and **outfile**

Here is more detail explanation. We discuss attributes **outdir** and **outfile** for x-frame only. The case of attributes **outdir** and **outfile** of <adapt> command is similar, and we refer the reader to Appendix X for explanation.

outdir = *dir-name*

The value of attribute **outdir**, say *dir-name*, must be an XVCL expression (in simple case – it is just a constant). If the specified directory *dir-name* in attribute **outdir** does not exist, the Processor creates one. The *dir-name* can be either an absolute path like “c:\mydir\test” or a partial path, including the null path, for example the name “test” or “xvcl\test”.

If *dir-name* is a partial path, the XVCL processor appends it to the output directory path of the current x-frame’s parent. When the Processor traverses back up the x-framework, it removes the directory paths that are appended at each level. This way, the directory structure may grow as the processor traverses down and shrink accordingly when the Processor traverses back up the x-framework, placing outputs from various x-frames in different directories. For example, suppose the parent’s output directory is “c:\mydir\”. Defining “test” in **outdir** attribute of the current x-frame will make the Processor emit the output from the current x-frame into the file that will be placed in “c:\mydir\test” directory. This feature is useful, for example, when the Processor needs emit Java code from different x-frames into different directories.

If *dir-name* is an absolute path, the Processor emits output of the current x-frame into this directory. The current output directory is set to be this path and descendent x-frames can

append their own partial paths to it. In this way, subsequently processed x-frames can create new output directories, as desired if these do not exist.

If the **outdir** attribute is not defined in the current <x-frame>, then the parent's output directory path is used.

If no ancestor x-frame, including SPC, has defined an output path in the **outdir** attribute, the Processor emits output from the current x-frame (as well as from all its ancestors) to a directory where SPC is stored.

If the *dir-name* of SPC's **outdir** attribute is a partial path, for example "test", then the Processor appends "test" to the directory path where SPC is stored and emit output from the SPC to that directory.

When both the <**adapt**> command in the parent x-frame and the adapted x-frame define the output directory path in their **outdir** attributes, the <**adapt**> command's output directory path is used.

For example, if x-frame X <adapt>s Y as follows:

```
<adapt Y outdir = c:\ancestor\ outfile = b.java />
```

and x-frame Y is defined as:

```
<x-frame Y outdir = c:\descendent\ />
```

then the output from x-frame Y will be emitted to the directory "c:\ancestor\b.java".

Value *dir-name* is often defined by an XVCL expression that involves variables. In such a case, an ancestor x-frame can override *dir-names* in the descendant x-frames via variables that are <set> in ancestor x-frames and referenced in *dir-name* expressions in descendent x-frames.

outfile = file-name

This attribute specifies a file to which the Processor emits the output from the current x-frame. The *file-name* must be an expression that yields a legal file name or absolute directory path including the file name. Partial (relative) paths are not allowed in the **outfile** attribute.

An extension (if any), is treated as an integral part of the file name. While the processor does not check file name extensions or make any use of them, it is a good practice to use extension .xvcl for files containing x-frames.

When the Processor emits output to file *file-name* for the first time in processing flow, and file *file-name* exists, the existing file is deleted before the output is emitted. If any subsequently processed x-frame emits output to the same file during the same run of the processor, the output is appended to the contents of that file.

A Processor option -A allows the user to avoid deleting an existing file. If the Processor is invoked with option -A, the Processor appends emitted output to file *file-name* even if the file *file-name* already existed before the Processor was invoked.

If **outfile** is omitted, the processor checks the **outfile** attribute of the parent x-frame that adapts the current one as follows: If **outfile** attribute is defined in the parent's command that adapts the current x-frame, that file name will be used. Otherwise, the parent's x-frame output

file is used, if defined. If an output file name is not defined in any of the ancestor x-frames including SPC, the SPC's name is as the *file-name*.

When both the parent's x-frame **<adapt>** command and the current x-frame define the output file name in their **outfile** attributes, the parent's **<adapt>** command's output file name is used. This is consistent with XVCL's variable scoping rules.

For example, if x-frame X **<adapt>**s Y as follows:

```
<adapt Y outfile = a.java />
```

and x-frame Y is defined as:

```
<x-frame Y outfile = b.java />
```

then the output from x-frame Y is emitted to the file "a.java".

8.9.9 Other commands and features

<remove x /> command un-defines a variable x by removing it from the Symbol Table. Removing an undefined variable produces a warning message. The variable is removed if and only if the **<remove>** command appears in the x-frame that originally defines that variable (i.e., the x-frame that entered that variable into the Symbol Table). Otherwise, the command is ignored and a warning message is produced.

<message> command displays a message on the screen. This command does not affect the output emitted by the XVCL Processor. **<message>** command is used for debugging purpose, e.g., to check variable values in order to trap error situations.

There are a number of yet other XVCL features such as attribute **samelevel** of **<adapt>** command, and defer-evaluation in **<set>** command, and arithmetic expressions. These features are not often used and we refer the reader to the Appendix X for details.

8.9.10 XVCL Processor's options

Processor options and environmental variables defined in a configuration file allow the user to tune the processing as follows:

-A option and variable optionA

Option **-A** allows the user to avoid deleting an existing file when the processor attempts to emit output to this file for the first time. When the Processor is invoked with option **-A**, for example:

```
java -jar xvcl.jar -A SPC
```

the Processor appends emitted output to file *file-name* even if the file *file-name* already existed before the Processor was invoked.

-T option and variable optionT

When the Processor is invoked with **-T** option, for example:

```
java -jar xvcl.jar -T SPC
```

the Processor includes the comment line with the name of an x-frame as the first line in the output emitted from that x-frame.

The user should specify comment symbols in 'begin-comment' and 'end-comment' variables. The default for begin-comment is //. The default for end-comment is nil (nothing).

-B option and variable optionB

When the Processor is invoked with -B option, extra white spaces (blank, space, etc.) normally emitted by the Processor are removed.

8.10 Conclusions of this Chapter

In this Chapter, we introduced step-by-step description of XVCL mechanisms to realize mixed-strategy approach. We used simplified, XML-free syntax to focus attention on essentials. Full specifications of XVCL are in Appendix C. In initial sections, we introduced XVCL commands informally, in the context of various maintenance and evolution tasks. In the last section, we summarized XVCL commands in a more formal and systematic way.

The simplicity and uniformity of XVCL change instruments signifies a certain level of maturity of the technology. It is not surprising if we recall that XVCL is based on principles of Frame Technology that emerged from industrial maintenance battlefield, and have been refined in projects dealing with millions lines of code.

We consider a current form of XVCL an assembly language of change specifications. XVCL's explicit and direct articulation is the source of its expressive power but also the source of its weakness, as x-frames may become overly verbose. Currently, we address this problem with tools that reveal simplified, abstract views of x-frames. In the future, we hope to discover mixed-strategy 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] Clements, P. and Northrop, L. *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002
- [3] Conradi, R. and Westfechtel, B. "Version Models for Software Configuration Management," *ACM Computing Surveys*, 30(2), 1998, pp. 232-282
- [4] 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 Transactions on Software Engineering.*, Vol. 27, No. 1, Jan. 2001, pp. 1-12
- [5] Ernst, M., Badros, G., and Notkin, D. "An Empirical Analysis of C Preprocessor Use," *IEEE Transactions on Software Engineering*, vol. 28, No. 12, December 2002, pp. 1146-1170
- [6] Karhinen, A., Ran, A. and Tallgren, T. 'Configuring designs for reuse', *Proc. Int. Conf. Software Engineering, ICSE'97*, Boston, MA., pp. 701-710
- [7] Sommerville, I. and Dean, G. 'PCL: A language for modeling evolving system architectures', *IEE Software Engineering Journal*, pp. 111-121 (1996)