On Refining XML Artifacts

Felipe I. Anfurrutia, Oscar Díaz, and Salvador Trujillo

University of the Basque Country - San Sebastián (Spain)
{felipe.anfurrutia,oscar.diaz,struji}@ehu.es

Abstract. Step-wise refinement is a powerful paradigm for developing a complex program from a simple program by adding features incrementally where each feature is an increment in program functionality. Existing works focus on object-oriented representations such as Java or C++ artifacts. For this paradigm to be brought to the Web, refinement should be realised for XML representations. This paper elaborates on the notion of XML refinement by addressing what and how XML can be refined. These ideas are realised in the XAK language. A Struts application serves to illustrate the approach.

1 Introduction

So far, most Web applications are conceived in a one-to-one basis. A recent study indicates a cloning rate (i.e. code repetition throughout the application) of 17-63% within Web applications of the same organisation [5]. This cloning evidences the existence of a common, although implicit, theme throughout the applications, and confirms an intuition felt in most organisations: code similarities among applications. These similarities are being handled in various ways such as IFDEFs, configuration files, installation scripts or cloned software copies *à la "copy-paste-modify"*. However, these solutions do not scale and can hinder maintenance as the number of variations increases.

One technique to handle similarities is step-wise refinement [1]. Step-wise refinement is a powerful paradigm for developing a complex program from a simple program by incrementally adding details. This approach attempts to depart from current "clone&own" practise by leveraging reuse of the common parts, and separating variable and changing parts as *program deltas*. The final product is obtained through composition: the common parts are composed with the program deltas that realise the variations for the product at hand.

Existing works focus on object-oriented representations such as Java or C++ artifacts [1]. However, recent studies revealed that the cloning rate of web-specific artifacts (e.g. mainly XML files) was considerably higher than general artifacts (e.g. *Java*, *C++*, etc) [5]. Indeed, XML artifacts play a preponderant role in current software practises, specially in the Web setting. This omnipresence of XML vindicates the existence of modularisation techniques specially tuned for XML. Attempts have been made to bring object-orientation (OO) and componentisation to the HTML realm [3], [4], [6]. But in the same way that OO falls short to face the increasing complexity of conventional software (crosscut handling is a case in point), so does it happen for XML artifacts.

This work addresses the use of refinements as a modularisation technique for XML artifacts. But, what is meant to refine an XML artifact? Does it mean that we can arbitrarily insert or delete a node anywhere in an XML tree? To this end, a language for defining refinements in XML documents is introduced: XAK (pronounced "sack"). The language is accompanied by a *validator* and a *composer*. The validator checks the correctness of XML refinements at compile time, whereas the composer builds incrementally a document by composing a base document with a refinement document. XAK composer is currently available as part of the AHEAD Tool Suite [7].

In this way, an XML artifact can also be conceived incrementally. This accounts for the following advantages: enhanced reusability of XML artifacts since commonalities and variabilities can be defined separately; flexibility in the selection of variable content and their composition; and decoupling validation of XML core artifacts from XML refinements. First, a brief on the notion of refinement is provided.

2 On the Notion of Refinement

A refinement can be thought of as a function that takes an artifact as an input, and returns another similar artifact but leveraged to support a given feature (denoted by *Feature1•Base*). In other words, a base artifact (e.g. a Java class) can now be incrementally extended (i.e. refined in our parlance) by adding a new module (e.g. a method) that extends the functionality of the base with a new feature (i.e. *Feature1*). At first sight, this resembles regular inheritance, but there is a difference: *there are not two classes but a single class that is being incrementally extended to account for a new feature.* Furthermore, the class being extended is not fixed at compile time (like in regular inheritance) but decided at composition time. In this way, a refinement behaves like a mixin inheritance, i.e. a class whose super class is parametrised [2]. Since the super class is not fixed until composition, distinct refinements on different (and unpredictable) order may be composed to yield a class. When the artifact is source code, a class refinement can introduce new data members, methods and constructors to a target class, as well as extend or override existing methods and constructors of that class. But, what is meant to refine an XML artifact? The next section introduces a sample case.

3 A Motivating Example Using Struts

Let *CurrencyConverter* be a web application that facilitates information about converting distinct currencies[1]. The application exhibits a J2EE architecture where Apache Struts is used. Struts follows the MVC pattern where the Controller separates the control-flow from the Model and the View. Space limitation makes us focus on the controller. However, similar remarks can be made for the artifacts realising the Model and the View.

The control-flow of a sample base application is realised through the *struts-config* document depicted in figure 1(b). The description so far accounts for the *base* controller. The term "base" refers to the stable core which is free for any variations. The issue arises when this base functionality needs to be leveraged with additional capabilities due to either perfective maintenance or versioning. For instance, consider two additional

[1] For a working example see *www.oanda.com/convert/classic*

```
<xs:schema>
  <xs:element name="struts-config"
               xak:modularizable="yes"> …
  <xs:element name="form-beans"
               xak:modularizable="yes"> …
  <xs:element name="form-bean"
               xak:modularizable="yes"> …
  </xs:element>
  <xs:element name="global-forwards"> …
  <xs:element name="action-mappings"
               xak:modularizable="yes"> …
  <xs:element name="action"
               xak:modularizable="yes"> …
  <xs:element name="plug-in"> ...
</xs:schema>
```

(a)

```
<struts-config   xak:artifact="struts-config.xml"
                 xak:feature="base" >
 <form-beans xak:module="mForms">
   <form-bean  xak:module="mConverterForm" …>
       <form-property name="amount" initial="1"…/>
       <form-property name="sourceCurrency" …/>
       <form-property name="targetCurrency" …/>
   </form-bean></form-beans>
 <action-mappings xak:module="mActions" >
   <action path="/converter"  xak:module="mButtons" …>
     <forward name="button.convertNow" path="/convertNow.do"/>
     <forward name="button.cheatsheet" path="/cheatsheet.do"/>
   </action>
   <action path="/convertNow" > … </action>
   <action path="/cheatsheet" > … </action>
 </action-mappings>
</struts-config>
```

(b)

Fig. 1. (a) Schema-based and (b) instance-based modularisation

features: (1) the *DateRate* feature, which allows end users to introduce a date in order to make the currency conversion with the rates at the given date. This implies to refine the *base* controller with new form properties, and some additions to the control-flow; (2) the *Customisation* feature, which permits end users to personalise the application by providing default values for both the *sourceCurrency* and the *targetCurrency* properties. This simple feature impacts all the model, the view and the controller.

Despite their simplicity, we are unaware of any mechanism that permits to incorporate these features *incrementally*. That is, start with a simple product (i.e. the *base*) and compose *deltas* to progressively add features to the base. Notice, you can add *ifdef* tags to the base code that a pre-compiler can leave or remove depending on the features to be finally exhibited by the application. But, this is more a kind of configuration or parametrisation mechanism that requires the designer to foresee all possible extensions where superfluous extensions are removed at configuration time. By contrast, refinements work the other way around: start with a simple product and apply deltas (i.e. program refinements) to progressively elaborate the desired product. Refinements are defined separately from the *base* in both time and space. In time, because the refinement can be added at any time. And in space, since the refinement is handled separately from the *base* artifact. Therefore, product synthesis rests in the ability to implement and compose refinements.

4 The Unit of Refinement

We aim at synthesising XML documents incrementally through refinements. But, what is the granularity of this refinement? A first approach could be to consider *any* XML node as the unit of refinement. However, this implies handling XML documents as mere data structures where any element node can be subject to refinement. This is too fine-grained granularity that defers the principle of modularity whereby high level abstractions (i.e. the modules) encapsulate their low level realisation (i.e. the instructions). Indeed, the

Open-Closed Principle (OCP) states that modules should be both open (for extension and adaptation) and closed (to protect the content against certain modifications).

To this end, a distinction is made between elements playing the role of modules (and hence, being subject to refinement) and elements that describe the realisation of these modules (and hence, protected against punctual updates). Thus, an XML module is defined as an element of a document that carries out a specific function and is liable to be re-used by/combined with other modules.

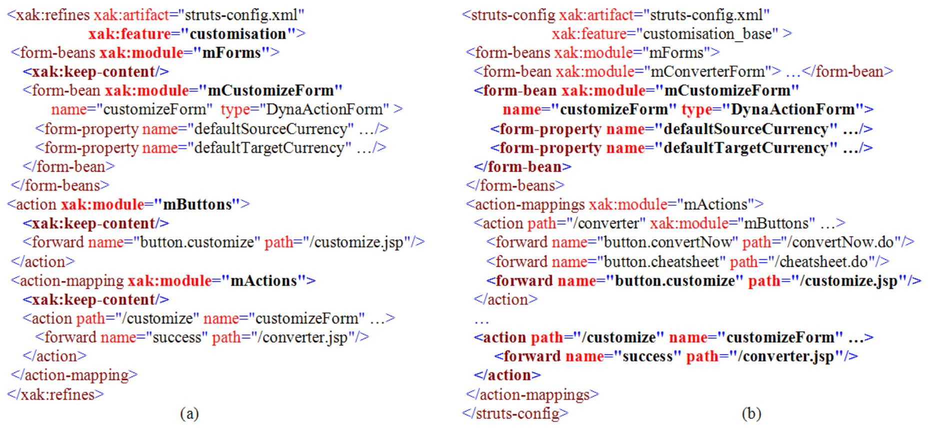
Besides realising abstractions, modules should be univocally identified. Xpath relies on element location. If the position of the element changes, so does it the Xpath expression output. Therefore, location-based Xpath expressions can not be used for element identification when the position of this element is liable to be changed, as it is the case for refinements. The order of refinements (e.g. *Feature1•Feature2•Base* vs. *Feature2•Feature1•Base)* can make the location of a given element to change. Thus, XML modules must have and preserve an ID property which permits to address this module unambiguously.

Similar to code artifacts, the identification of the main abstractions (modules) depends on the domain at hand. In an XML setting, this domain is partially defined by vocabularies. W3C XML Schema is one of the most popular schema languages. A schema states the element, attribute and atomic type names, in addition to structural constraints that instances of this schema must obey.

Next, we need a way to indicate the elements playing the role of modules. For our sample case, we want to stay that only element types <*struts-config*>, <*form-beans*>, <*form-bean*>, <*action-mappings*> and <*action*> can be modules (liable to be refined). The rest of the element types can not be refined (e.g. <*controller*> served only for implementation). This is the purpose of the *"xak:modularizable"* attribute. Figure 1(a) shows the Struts schema now annotated with this attribute. The attribute indicates whether an element type is eligible to be a module or not. For a given document instance, this does not force every occurrence of a modularisable element type to be refined, but prevents non-modularisable element types from being refined.

However, stating modularity at the schema level can be too general. Frequently, the notion of module depends on the document at hand. Hence, "schema-based" modularisation is complemented with "instance-based" modularisation (using the *xak:module* attribute). This approach permits to further restrict what can be refined among the modularisable elements. For our sample case, only *"/converter*" is a refinable <*action*>*;* whereas */convertNow* and */cheatsheet* can not be refined. Figure 1(b) illustrates this situation for our sample case. This moves the decision of what can be refined to the instance level.

"Schema-based" and "instance-based" approaches to module definition offers a good balance between the controlled approach that offers the schema, and the freedom that programmers' creativity requires. This is akin to the openness and subsidiary way of working that characterise the XML world (e.g. schema management in XML Schema). *Schema designers* can use a schema approach to define the "refinable" element types, *the schema users* can work at the instance level by indicating the "refinable" elements, and finally, *the instance users* compose the features to synthesise the final application, refining some element contents, should it be required.



```
<xak:refines xak:artifact="struts-config.xml"
            xak:feature="customisation">
 <form-beans xak:module="mForms">
   <xak:keep-content/>
   <form-bean xak:module="mCustomizeForm"
        name="customizeForm"  type="DynaActionForm" >
     <form-property name="defaultSourceCurrency" .../>
     <form-property name="defaultTargetCurrency" .../>
   </form-bean>
 </form-beans>
<action xak:module="mButtons">
   <xak:keep-content/>
   <forward name="button.customize" path="/customize.jsp"/>
 </action>
 <action-mapping xak:module="mActions">
   <xak:keep-content/>
   <action path="/customize" name="customizeForm" ...>
     <forward name="success" path="/converter.jsp"/>
   </action>
 </action-mapping>
</xak:refines>
```

(a)

```
<struts-config xak:artifact="struts-config.xml"
             xak:feature="customisation_base" >
 <form-beans xak:module="mForms">
  <form-bean xak:module="mConverterForm"> ...</form-bean>
  <form-bean xak:module="mCustomizeForm"
      name="customizeForm" type="DynaActionForm">
    <form-property name="defaultSourceCurrency" .../>
    <form-property name="defaultTargetCurrency" .../>
  </form-bean>
 </form-beans>
 <action-mappings xak:module="mActions">
  <action path="/converter" xak:module="mButtons" ...>
    <forward name="button.convertNow" path="/convertNow.do"/>
    <forward name="button.cheatsheet" path="/cheatsheet.do"/>
    <forward name="button.customize" path="/customize.jsp"/>
  </action>
  ...
  <action path="/customize" name="customizeForm" ...>
     <forward name="success" path="/converter.jsp"/>
  </action>
 </action-mappings>
</struts-config>
```

(b)

Fig. 2. (a) A XAK refinement for the *Customisation* feature and (b) the resulting document from the composition *customization●base*

5 The Ways of Refinement

Product synthesis starts with a base product and apply deltas (i.e. refinements) to progressively incorporate new features to this product. Thus, there are two kinds of artifacts: base documents (i.e. values) and refinement documents (i.e. functions).

Base document. Any traditional XML document can be a *base* document. The only difference is that now a distinction is made between XML elements, liable to be refined (i.e. modules), and those that can not be refined (i.e. the implementation). To this end, the XAK namespace provides three attributes (see figure 1b), namely: **@xak:artifact**, which specifies the name of the document that is being incrementally defined; **@xak:feature**, which indicates the name of the feature being supported[2]; and **@xak:module**, which identifies those elements that play the role of modules. Notice that the designer is not forced to turn into modules all elements of a modularisable type.

Refinement documents. A refinement is an increment in program's functionality. This is specified through the following XAK elements: ***<xak:refines>*** and *<xak:keep-content>*. The former is the root element of the refinement document. Its content describes a set of module refinements (i.e. elements annotated with the *xak:module* attribute) over a given base document (i.e. the *xak:artifact* attribute). Moreover, the ***<xak:keep-content>*** attribute indicates the place where the content of the refined module will be placed once it is synthesised.

As an example, consider our sample case. The *customisation* feature enhances the *base* by providing default values for both the *sourceCurrency* and the *targetCurrency* properties. Adding this feature impacts on all the aspects: the model, the view and the controller. Thus, three refinement documents are needed, all with *@xak:feature = "customisation"*. Let's focus on the controller, i.e. *"struts-config.xml"*. This artifact is

[2] For base documents, this attribute keeps the value *"base"*.

gradually defined as features are being composed. The *base* is shown in figure 1b where *mForms*, *mActions* and *mButtons* are set as modules. *Customisation* implies: (1) adding the *customizeForm* form-bean into the *mForms* module; (2) extending the *mButton* dispatcher action to show the *customise.jsp* page of the feature, and (3) defining a new action. At synthesis time, *customisation* • *base* will deliver the enhanced *strust-config.xml* file shown in figure 2b.

A refinement realises just an increment, i.e. a delta. Hence, it is most unlikely that the refinement obeys the schema of the type of document being refined. For instance, our previous refinement (see figure 2a) is not a valid *struts-config* document since it holds and <*action*> element outside an <*action-mappings*> element. Moreover, the elements and attributes of the XAK namespaces are intermingled with the elements of the given schema vocabulary.

Therefore, the validity of a refinement can not be checked directly against neither the schema of the document being refined (e.g. *struts-config.xsd*) nor the XAK schema. However, the element names, types and some structural constraints still hold. For instance, the elements and attributes used in the refinement should be permitted by the content model of the module being refined. This implies to define which are the laws of refinement and develop a tool for checking it.

6 Conclusions

Step-wise refinements permits to conceive artifacts incrementally, hence, distinguishing between stable, base artifacts and refinement artifacts that realise the variations. This work addresses refinement of XML artifacts. The peculiarities brought by markup languages as opposed to object-oriented ones have been exposed where the notion of refinement offers an alternative way to modularise source code for languages where no other modularisation technique is available.

References

1. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Transactions on Software Engineering 30(6), 355–371 (2004)
2. Bracha, G., Cook, W.: Mixin-based inheritance. SIGPLAN 25(10), 303–311 (1990)
3. Gellersen, H.-W., Wicke, R., Gaedke, M.: WebComposition: an object-oriented support system for the Web engineering lifecycle. Computer Networks and ISDN Systems 29(8-13), 1429–1437 (1997)
4. Klapsing, R., Neumann, G., Conen, W.: Semantics in Web Engineering: Applying the Resource Description Framework. IEEE Multimedia 8(2), 62–68 (2001)
5. Rajapakse, D.C., Jarzabek, S.: An investigation of cloning in web applications. In: Lowe, D.G., Gaedke, M. (eds.) ICWE 2005. LNCS, vol. 3579, Springer, Heidelberg (2005)
6. Schranz, M.W., Weidl, J., Goschka, K.M., Zechgmeister, S.: Engineering complex World Wide Web services with JESSICA and UML. In: Proc. of the 33rd Annual Hawaii Int. Conf. on System Sciences (HICSS'00), Maui, HI, USA (2000)
7. Trujillo, S., Batory, D., Díaz, O.: Feature Refactoring a Multi-Representation Program into a Product Line. In: Proc. of the 5th Int. Conf. on Generative Programming and Component Engineering (GPCE'06) (2006)