A FRAMEWORK FOR MANAGING COMPONENTS USING NON-FUNCTIONAL PROPERTIES[*]

Jean-Michel Hufflen
LIFC (EA CNRS 4269), University of Franche-Comté, 16, route de Gray, 25030 Besançon Cedex, France

Keywords: Non-functional properties, Component-based approach, Component configuration, TACOS project, XML.

Abstract: As part of a component-based approach, we propose a framework to group specifications of component hierarchies, possibly including the specification of non-functional properties. We show how we are able to choose a particular implementation of a component—or change an implementation to another—regarding non-functional properties, and how we are able to express configurations of component-based architectures. Our approach uses programs related to XML, and non-functional properties are managed by means of a terminology originating from the metadata used by the *Dublin Core*.

1 INTRODUCTION

It is known that *functional* requirements specify what a system is supposed to do whereas *non-functional* requirements express what a system is supposed to be. Non-functional requirements include constraints and quality factors (Sommerville, 2006, Ch. 2). Examples are time-constrained response or requested services' availability. Presently, there is no consensus about classifying these properties, as reported in (Glinz, 2007). In this article, we propose a framework to handle some hierarchies of components, within a component-based approach, suitable for systems with high-safety requirements. By 'some hierarchies', we mean that *several* implementations may be associated with a component's specification. The choice among such implementations of a same specification can be guided by considering non-functional properties. We assume that preliminary steps of designing some versions about components and interactions among them have already been done. We aim to assist designers when they wish to compare different configurations of the same service. For example, they can study what is induced by the replacement of a component by another one.

At first glance, our approach may be compared with using *Makefiles*, in order to build executable files from source ones by means of the `make` command[2]. But it is well-known that such files contain information redundancy since dependences have to be put explicitly. As an example, such a specification is related to files written using the C programming language:

$$\texttt{f.o: f.c } dep_1\texttt{.h ... } dep_n\texttt{.h}$$

expresses that the `f.o` object file has to be rebuild if a file belonging to `f.c`, dep_1`.h`, ..., dep_n`.h` is newer than the present `f.o` file, or if this last file does not exist. But this information must be put even if it is already expressed within files. Since we consider source files written in C, the dependence of the `f.o` file on the dep_1`.h` file has probably been expressed within the `f.c` source file by the macro-instruction:

```
#include "dep1.h"
```

In particular, that is why some additional programs—e.g., `automake` (Vaughn et al., 2000) or (cmake, 2009)—have been developed to supply this information from source files. The same drawback related to information redundancy exists with the configuration files used by the Ant program[3]: a `target` element includes a `depends` attribute, even if this information can be deduced from source files.

Our configuration files use XML[4]-like syntax. We also use the terminology originating from (Dublin Core Metadata Initiative, 2008) for *metadata* related to non-functional properties. In addition, types qualifying the possible values for non-functional properties can be specified by the rigorous approach of XML

[*]This work is supported by the French agency for research (TACOS ANR-06-SETI-017-03).

[2]See (Oram and Talbott, 1991) for more details.

[3]See (Tilly and Burke, 2002) for more details.

[4]e**X**tensible **M**arkup **L**anguage.

Schema (W3C, 2008). We explain all these points in § 2. Then § 3 gives the different steps of the use of our framework. § 4 discusses our *modus operandi*, and § 5 sketches which ways are open. Reading this article only requires basic knowledge of XML.

2 BASIS

We consider a basic notion of component: a unit of construction implementing some services *via* interfaces. If a component is built from other subcomponents, it is said to be *composite*. Otherwise, it is *simple*. Such a simple component can be specified by the `tacos:component` element[5] (several successive `tacos:implements` elements are allowed):

```
<tacos:component id="id-0" path="path-to-id-0">
  <tacos:implements ref="interface-0"
                    role="server" name="..."/>
  <tacos:nonfunctional-properties>
    <tacos:nf-property name="nfp:complexity"
                       as="nfp:performance"
                       value="linear"/>
    <tacos:nf-property
      name="nfp:reliability" as="nfp:specific"
      check-up="http://tacos.loria.fr/..."/>
    ...
  </tacos:nonfunctional-properties>
</tacos:component>
```

The `id` attribute is unique—of type `xsd:ID`—whereas `path` is used to localise the component. Then non-functional properties are grouped. For each of them, if the `check-up` attribute is present, its value is a `URI`[6] denoting a program that applies to what is located at `path`'s value and reports about this property. For example, if the *reliability* is expressed as the mean time between failures, it can be computed by a tool and the result—the `value` attribute—is the type `xsd:decimal`. Likewise, a non-functional property related to efficiency can be reported by a tool running the component with benchmarks. If a non-functional property is supposed to be not checkable by means of a program, the `value` attribute is to be supplied by designers. Most non-functional properties are predefined, they belong to the namespace identified by the `nfp` prefix. A designer can add new properties by refining our XML schema. The `as` attribute is used to group non-functional properties into several families: it allows us to retrieve information concerning non-functional properties belonging to a same class. A composite component can be specified as follows:

```
<tacos:composite-component id="id-2"
                           path="...">
  <tacos:implements .../>
  <tacos:refers-to ref="id-0" nb="2"/>
  <tacos:refers-to ref="id-1"/>
  <tacos:nonfunctional-properties>...</...>
</tacos:composite-component>
```

There are three subcomponents of this last component: two instances of the `id-0` component (the `nb` attribute gives the number of replications), and one instance of `id-1` (`nb` defaults to `1`). Non-functional properties specified for this composite component hold for the whole of it, without reference to its subcomponents' properties. The general layout is:

```
<tacos:components ...   (Namespace definitions.)   >
  <tacos:general-metadata>
    <dc:title>Example</dc:title>
    <dc:creator>H., J.-M. (...)</dc:creator>
    ...
  </tacos:general-metadata>
  <tacos:component-specifications>
    <tacos:component id="id-0">...</...>
    <tacos:composite-component id="id-2">
      ...
    </...>
  </tacos:component-specifications>
</tacos:components>
```

The elements introducing metadata use the basic elements of the Dublin Core, prefixed by `dc`. The specification of all the components is flatten, in the sense that no component is defined inside the specification of a composite component. A reference to another component is expressed by an `tacos:refers-to` element, as shown above.

3 STEPS

Here are the five successive steps of our method.

1. This step is sketched at § 2, it just states the result of the conception of a hierarchy of components. This result is close has been shown, but without the specification of non-functional properties.
2. Elements of non-functional properties are added by designers, with accurate values associated with the `check-up` or `value` attributes.
3. We consider all the `tacos:nf-property` elements of our configuration file. If the `check-up` attribute is used, the corresponding program is called, and the result is given as a new or updated `value` attribute. So each `tacos:nf-property` element is given a `value` attribute. In addition, some metadata elements are computed, e.g., the following element is put at the specification's end of the `id-0` component:

[5] The prefix originates from the TACOS project, cf. § 4.

[6] **U**niform **R**esource **I**dentifier (Network Working Group, 2002).

```
<tacos:technical-metadata>
  <dcterms:isReferencedBy>id-1</...>
</tacos:technical-metadata>
```

The dcterms prefix is used for the elements belonging to the *Qualified Dublin Core*: such elements refine the semantics of Dublin Core's basic elements. This step is performed by applying an XSLT[7] program (W3C, 2007c). The result is given in a new configuration file, extending the original one, and so-called *complete*.

4. Several complete configuration files modelling hierarchies can be merged into one, in order to share the common parts—interfaces or components. In addition, metadata about alternatives and refinements are added, e.g., if another configuration file contains another component id-3, implementing the same interface than id-0, the technical metadata of the specification of id-0 will include:

```
<dcterms:alternative>id-3</...>
```

an analogous information being added to the specification of id-3. This step is also performed by applying an XSLT program.

5. Using a configuration file as a data base, we can ask for information about components, including non-functional properties. As a simple example, the following fragment, written in the XQuery language (W3C, 2007b), yields all the paths of the components such that the non-functional properties classified as nfp:performance are 'good', the result being an XML text.

```
<answers>{
  for $component in
      (doc("...")/tacos:components/
       tacos:component-specifications/*)
      return
      if (some $nf-property in
          $component/
          tacos:nonfunctional-properties/
          tacos:nf-property satisfies
          $nf-property/@as eq
          "nfp:performance" and
          check-good($nf-property/@value))
          then <a>{$component/@path}</a>
          else ()
  }
</answers>
```

We can also assembly components in order to build complete software, using selection criteria based on non-functional properties. If this operation succeeds, a *version name* is chosen and concerned components' metadata are updated:

```
<dcterms:isVersionOf>version-0</...>
```

this step being performed by an XSLT program.

6. We should be able to derive files usable by make—or Makefile.in files, used by the configure program (Vaughn et al., 2000)—or Ant.

4 CRITICISM

This approach has been put into action as part of the TACOS[8] project, proposing a component-based approach suitable for land transportation systems. These systems, which are both distributed and embedded, require to express functional properties, as well as non-functional ones. As a good example of our method, there are several versions and variants of the localisation component of a vehicle. These versions use the same basic components but the composite components grouping them are organised differently. There are much debate within the working group about these versions. Our approach allows us to make easier these comparisons, in particular regarding non-functional properties. More precisely, Steps 1 to 5 have been implemented using the Saxon program (Kay, 2008), providing an XQuery and XSLT processor; Step 6 is almost finished.

A close approach exists within the SCA framework (Service Component Architecture, 2007), that provides a programming model for building applications based on a service-oriented architecture. SCA also uses files written using XML-like syntax, and similar notions exist: interfaces, simple and composite components. But this approach is more restrictive: interfaces can be Java or WSDL[9] interfaces. Likewise, SCA allows a limited choice among several implementation types for a component's implementation. The most used is Java, but other languages, such as C++ or C, are also allowed. That is, we are not wholly independent of the language chosen, even if a wide variety is available. Properties can be specified, and may be deduced from programs' texts, e.g., implementations written in Java can use annotations. The main difference with our approach is that an SCA text describes only one assembly; there is no way to specify alternatives in SCA, no possible replacement of a component by another. Some elements used within SCA can be viewed as metadata, but they are defined in an *ad hoc* way. Likewise, UniFrame (Raje et al., 2001) creates a comprehensive framework that enables the discovery, interoperability and collaboration of components via

[7]e**X**tensible **S**tylesheet **L**anguage **T**ransformations.

[8]**T**rustworthy **A**ssembling of **C**omponents: fr**O**m requirements to **S**pecification. See this project home page http://tacos.loria.fr for more details.

[9]**W**eb **S**ervices **D**efinition **L**anguage (W3C, 2007a).

software generative techniques. Non-functional properties are handled as *quality-of-service parameters*, SQL requests are used to query such a frame about QoS parameters. The selection of alternative componenents is possible, too. Nevertheless, our notion of composite component is more powerful, and the relationships among components we model by means of Dublin Core elements are more refined. In addition, as far as we know, UniFrame does not provide tools to perform the merge of several hierarchies sharing common components. On the contrary, we do not provide the automatic generation of glues and wrappers, as UniFrame does, but as part of the TACOS project, that can be done by means of Fractal, a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications (Bruneton et al., 2004).

5 FURTHER WORK

Many tools are used within TACOS project, e.g., Fractal. It does not consider non-functional properties, but handles XML configuration files. We succeeded in getting such XML files, transforming them, and adding specification of non-functional properties. So we can reuse the conception done by a Fractal user. An advantage of using elements originating from Dublin Core: we plan to use some tools related to the Semantic Web. This idea is promising: as part of studying services, this connection with the Semantic Web has already been proposed in (Gerede et al., 2008). Such metadata are also used within the *Web Services Semantics* (W3C, 2005; WSMO, 2006).

6 CONCLUSIONS

We have wanted to show that we follow a rigorous approach, with the advantages and drawbacks of a general one, without any hypothesis about languages and paradigms used. Such an approach requires the development of many additional tools, in particular, to deal with non-functional properties. But the application fields of such a framework is potentially high. Of course, we need more case studies to experiment our approach, but we are optimist because of the first results we got from Fractal configuration files.

REFERENCES

Bruneton, É., Coupaye, Th., and Stefani, J.-B. (2004). *The Fractal Component Model*. http://fractal.objectweb.org/specification/index.html.

cmake (2009). *CMake*. http://www.cmake.org/.

Dublin Core Metadata Initiative (2008). *Dublin Core Metadata Initiative*. http://dublincore.org.

Gerede, C. E., Ibarra, O. H., Ravikumar, B., and Su, J. (2008). Minimum-cost delegation in service composition. *Theoretical Computer Science*, 409(3):417–431.

Glinz, M. (2007). On non-functional requirements. In *Proc. RE 07*, New-Delhi, India.

Kay, M. H. (2008). *Saxon. The XSLT and XQuery Processor*. http://saxon.sourceforge.net.

Network Working Group (2002). *Uniform Resource Identifiers (URIs), URNs, and Uniform Resource Names (URNs): Clarifications and Recommendations*. http://www.ietf.org/rfc/rfc3305.txt. Edited by M. Mealling and R. Denenberg.

Oram, A. and Talbott, S. (1991). *Managing Projects with make*. O'Reilly & Associates, Inc., 2 edition.

Raje, R., Bryant, B., Auguston, M., Olson, A., and Burt, C. (2001). A unified approach for integration of distributed heterogeneous software components. In *Proc. of the 2001 Monterey Workshop Engineering Automation for Software Intensive System Integration*, pages 109–119.

Service Component Architecture (2007). *Assembly Model Speficiation*. http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf?version=1.

Sommerville, I. (2006). *Software Engineering*. Addison-Wesley, 8 edition.

Tilly, J. and Burke, E. M. (2002). *Ant: the Definitive Guide*. O'Reilly & Associates, Inc.

Vaughn, G. V., Ellison, B., Tromey, T., and Taylor, I. L. (2000). GNU *Autoconf, Automake, and Libtool*. Sams.

W3C (2005). *HyperText Markup Language Home Page*. http://www.w3.org/MarkUp/.

W3C (2007a). *Web Services Description Working Group*. http://www.w3.org/2002/ws/desc/.

W3C (2007b). *XQuery 1.0: an XML Query Language*. http://www.w3.org/TR/xquery. W3C Recommendation. Edited by Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie and Jérôme Siméon.

W3C (2007c). *XSL Transformations (XSLT). Version 2.0*. http://www.w3.org/TR/2007/WD-xslt20-20070123. W3C Recommendation. Edited by Michael H. Kay.

W3C (2008). *XML Schema*. http://www.w3.org/XML/Schema.

WSMO (2006). *Web Service Modelling Ontology*. http://www.wsmo.org/TR/d2/v1.3/. Edited by Dumitru Roman, Holger Lausen, and Uwe Keller.