Run-Time Variability through Component Dynamic Loading

Leonardo Murta[1], Aline Vasconcelos[1,3], Ana Paula Blois[1, 2],
Marco Lopes[1], Carlos Junior[1], Marco Mangan[1, 2], Cláudia Werner[1]
1.COPPE/UFRJ – System Engineering and Computer Science Program
P.O. Box 68511 – ZIP 21945-970 – Rio de Janeiro – RJ – Brazil
2. Pontifícia Universidade Católica do Rio Grande do Sul- PUCRS
Av. Ipiranga, 6681 – Prédio 30 – Bloco 4 – ZIP 90619-900 - Porto Alegre – RS
3.CEFET Campos (Centro Federal de Educação Tecnológia de Campos)
R. Dr. Siqueira, 273 – Pq. Dom Bosco – ZIP 28030-130 - Campos dos Goytacazes- RJ
{murta, aline, anablois, mlopes, carlosjr, mangan, werner}@cos.ufrj.br

Abstract

This paper presents a tool for dynamic loading of components into run-time environments. This tool was implemented in the context of the Odyssey environment. Using this tool, it is possible to select, at run-time, the desired set of functionality for the Odyssey environment. Depending on the selected configuration, new components are downloaded and dynamically plugged into the environment. Moreover, the dependencies among components are analyzed to keep the consistency of the whole environment.

1. Introduction

Along the last six years, COPPE's software reuse group has been working on the Odyssey project [10]. Its aim is to construct a reuse infrastructure based on domain models, product lines and component based development. This reuse infrastructure, named Odyssey environment [12], entails different tools to support reuse activities. Several of them were presented in previous editions of SBES tools session since 1998.
However, the Odyssey environment became a huge infrastructure, with drawbacks related to size, performance, and usage complexity. Many tools are not necessary in some particular usage scenarios, and it can represent an overhead to the whole environment in these scenarios. For example, LockED is a tool for concurrent development of domain models. This tool can be suppressed when only one person is working on a given domain model. Moreover, some tools are functionally equivalent, and only one of them is necessary to allow the environment execution, which is the case of alternative persistence mechanisms.
To solve this problem, some reengineering had to be applied to the environment in order to reduce its complexity and to allow its customization to specific needs. First, each environment functionality was analyzed and labeled as mandatory or optional. Mandatory functionalities were placed in a kernel module and optional functionalities were placed in plug-in tools. After the reengineering, the kernel and correspondent modules were named Odyssey-Light. Moreover, a generic interface named *Tool* was created to encapsulate plug-in tools into software components managed by the kernel module. Finally, a dynamic loading mechanism was implemented in the Odyssey-Light kernel to allow run-time variability management, through the selection, download, and installation of plug-in tools.

Considering the described scenario, the goal of this paper is to present the dynamic loading mechanism adopted in the Odyssey environment along with the specifications that must be followed by any tool that should be plugged in the infrastructure. The paper is organized in five sections. Following this Introduction, Section 2 presents some related works on the topic of system variability and Section 3 discusses the proposed approach for run-time variability through components dynamic loading. Section 4 shows a usage example of dynamic loading of components into Odyssey-Light. Finally, Section 5 presents some contributions, limitations, and future work.

2. System Variability

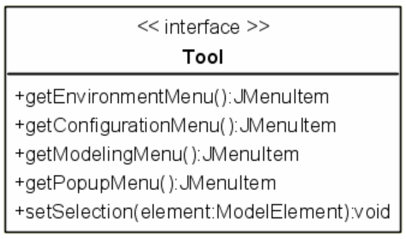
System variability is the ability of software to change its behavior during its life cycle [11]. Due to cost-benefit relation, systems need to be released by means of different flavors, with different sets of functionalities (e.g. desktop, standard, professional, enterprise) [7]. Furthermore, independently of the situation, it is important to allow the system configuration to fit the particular needs of customers.
System variability can occur at different phases of the software life cycle. Deployment is the most known phase of system variability occurrence. Tools such as InstallShield [1] and InstallAnywhere [13] allow the selection of the desired functionalities during the deployment phase. This approach is less flexible than the ones based on run-time variability, since it is not possible to install new functionalities during the software execution. Moreover InstallShield and InstallAnywhere can only support the customization through options already offered by the software manufacturer, not allowing third-party tools to be integrated in the software.
At the system specification phase, different techniques allow the representation of system variability, such as the usage of optional and variable elements in product lines to support product selections [4] and the selection of features inside domain models within a reuse process [8]. The advantage of these approaches is the possibility to manage and trace variabilities since the analysis phase. However, they don't support system evolution without system modifications as the approach proposed in this paper. Finally, at coding and building phases, compiler directives are commonly used to decide what functionalities are needed.
Besides these specific approaches for particular phases of the software life cycle, [5] suggests the usage of software architectures to guide the variability selection at anytime in the life cycle, including run-time variability. This approach makes possible to define variability at design time and apply this variability at design-time, invocation-time and run-time. In our case, it is important to allow the Odyssey's development team to describe variability at development time and let software engineers (Odyssey users) to select the functionalities at run-time.

3. Run-Time Variability through Components Dynamic Loading

An extensive study about plug-in approaches in the context of development environments was performed. Some integrated development environments, such as Eclipse [3], JBuilder [2] and NetBeans [9], were analyzed. This analysis provided a few evidences of the main requirements regarding building a dynamic loading infrastructure for the Odyssey environment, which are: (1) common interface for all tools, (2) packaging specification, (3) dependency description, and (4) dynamic loading mechanism.

3.1. *Tools* Interface

As mentioned before, all plug-in tools were purged from the Odyssey environment to allow the creation of its Light version. However, every interaction point between these plug-in tools and the environment were cataloged during the purge process. These interaction points were used to build a generic interface named *Tool*.
The *Tool* interface, shown in Figure 1, is accessed by the Odyssey-Light environment every time it needs to consult or notify plug-in tools. Each plug-in tool informs a list of menus to be placed in the kernel user-interface by realizing the Tool interface. The methods *getEnvironmentMenu()*, *getConfigurationMenu()*, and *getModelingMenu()* ask the plug-in tool for menus to be shown, respectively, in the tools and preference sections of the main window and in the modeling window of the Odyssey environment. The method *getPopupMenu()* must declare popup menus to be shown over selected model elements.



<< interface >>
Tool
+getEnvironmentMenu():JMenuItem
+getConfigurationMenu():JMenuItem
+getModelingMenu():JMenuItem
+getPopupMenu():JMenuItem
+setSelection(element:ModelElement):void

Figure 1: *Tool* interface.

Finally, the method *setSelection()* informs the tools about the current selected model element in the modeling environment. The tool developer may consult this and other model elements navigating through the Odyssey API. This functionality is relevant for tools that need to present contextual information.

3.2. Packaging Specification

In our approach, a tool becomes a component when it implements the *Tool* interface and is packaged in a Java Archive (JAR) file. Attributes in the JAR manifest file contain meta-data relevant to the mechanism, such as the class name that implement the *Tool* interface. Figure 2 shows the manifest defined for Ares, a reverse engineering tool.

```
Manifest-Version: 1.0
Tool-Class: br.ufrj.cos.lens.odyssey.tools.reverseEngineering.ares.AresFacade
Created-By: 1.4.2_01-b06 (Sun Microsystems Inc.)
```

Figure 2: Manifest of the Ares tool inside the tool JAR file.

3.3. Dependency Description

Usually, components require and provide services from/to other components. A tool, packaged as a component, depends on other tools and has a set of attributes, such as name, type, description and repository location. These characteristics and dependencies are prescribed in the document type definition (DTD) shown in figure 3.
The available types of components are kernel, plug-in, and library. Kernel components must be present inside the Odyssey environment (e.g. the Odyssey diagram editor). Plug-in components may be selectively activated, providing variability to the environment (e.g. the UML Criticism Assistant tool). Finally, library components are required by another component (e.g. a specific JDBC library is required by the

relational persistence tool). Figure 4 shows an excerpt from Odyssey-Light component descriptor (http://sety.cos.ufrj.br/releases/1.0.0.xml).

```
<!ELEMENT components (component)* >
<!ELEMENT component (dependency)* >
<!ELEMENT component EMPTY >
<!ATTLIST component
        type (kernel | plugin | library) #REQUIRED
        name ID #REQUIRED
        description CDATA #IMPLIED
        location CDATA #REQUIRED
>
<!ATTLIST dependency
        name IDREF #REQUIRED
>
```

Figure 3: DTD for component description.

```
<component type="plugin" name="odysseyxmi-0.0.5.jar" description="Odyssey XMI"
location="http://sety.cos.ufrj.br/releases/components">
  <dependency name="jmi.jar" />
  <dependency name="jmi-uml-1.4.jar" />
  <dependency name="mof.jar" />
  <dependency name="jmiutils.jar" />
  <dependency name="mdrapi.jar" />
  <dependency name="nbmdr.jar" />
  <dependency name="openide-util.jar" />
</component>
```

Figure 4: An excerpt from the unstable release of the components descriptor.

This excerpt defines a tool component for importing and exporting UML models using XMI files. This tool, named Odyssey-XMI, has dependencies to other seven library components. All these components must be downloaded and installed in the Odyssey environment prior to the installation of the XMI tool itself.

3.4. Dynamic Loading Mechanism

The dynamic loading mechanism is responsible for the communication between the Odyssey environment instance and the component directory, on the Odyssey download server. Moreover, it uses algorithms that are similar to those defined by [6] to allow dependencies detection and it implements an independent Java class loader to access Java classes by means of a dynamically defined classpath.

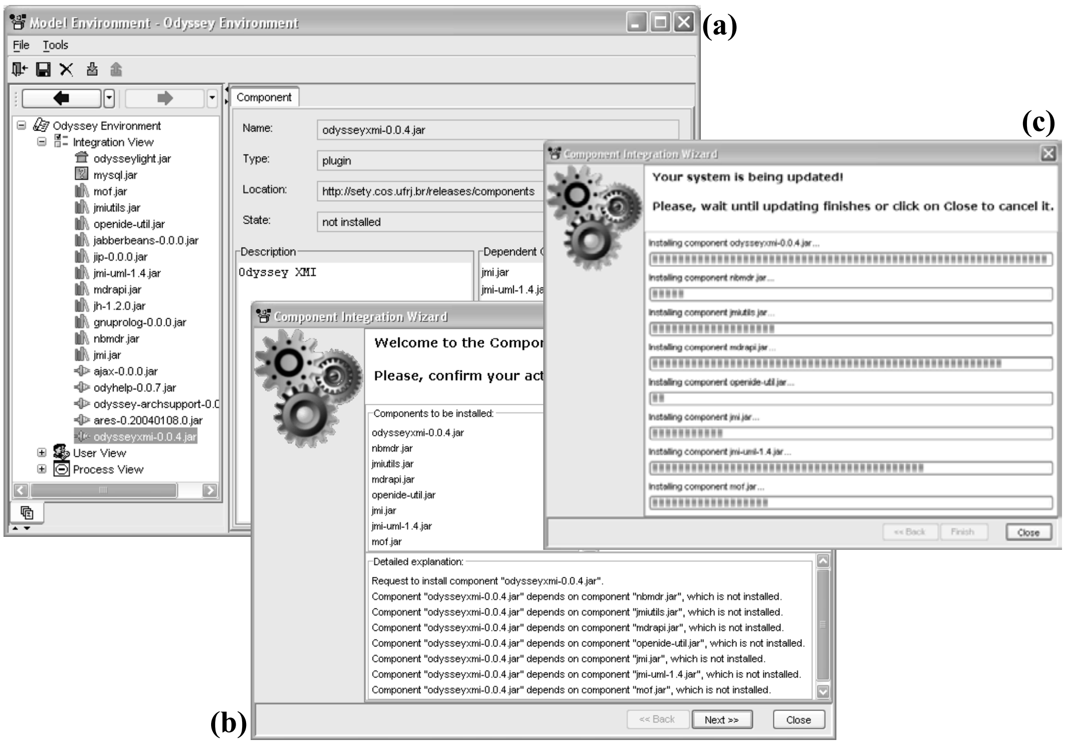
The communication between the Odyssey environment instance and the component directory, including the download of component descriptors and implementations, occurs by means of the HTTP protocol. The download of multiple components occurs in parallel, decreasing download time. All the installation process is presented to the end-user, including the component download progress.

After the selection of new components, but before the download, some dependency analysis is performed. This analysis search for other components that must be installed in order to satisfy restrictions declared in the component descriptor. In addition, all installed components are subjects of a reference counting algorithm, which detects dangling library-type components and mark them for removal.

After the download phase, a new Java class loader is created to access the downloaded classes. This new class loader is set as the context class loader of all existing execution threads. Finally, the Java reflection API is used to access the class declared in the manifest file of the component. This class is cast to the *Tool* interface and put in a collection that contains all installed tools. This collection is used by the Odyssey environment to communicate with the installed tools.

4. Usage Example

Dynamic loading resources are accessed in the integration view of the Odyssey environment. This view allows users to set the URL of the component descriptor and to set the local directory for components. It is also possible to refresh the list of available components and to install or remove a specific component, as shown in Figure 5.a.


(a)
(b)
(c)
Model Environment - Odyssey Environment
File
Tools
Component
Component Integration Wizard
<< Back
Next >>
Finish
Close

Odyssey Environment
Integration View
odysseylight.jar
mysql.jar
mof.jar
jmiutils.jar
openide-util.jar
jabberbeans-0.0.0.jar
jip-0.0.0.jar
jmi-uml-1.4.jar
mdrapi.jar
jh-1.2.0.jar
gnuprolog-0.0.0.jar
nbmdr.jar
jmi.jar
ajax-0.0.0.jar
odyhelp-0.0.7.jar
odyssey-archsupport-0.0
ares-0.20040108.0.jar
odysseyxmi-0.0.4.jar
User View
Process View

Name: odysseyxmi-0.0.4.jar
Type: plugin
Location: http://sety.cos.ufrj.br/releases/components
State: not installed

Description
Odyssey XMI

Dependent C
jmi.jar
jmi-uml-1.4.ja

Welcome to the Compor

Please, confirm your act

Components to be installed:
odysseyxmi-0.0.4.jar
nbmdr.jar
jmiutils.jar
mdrapi.jar
openide-util.jar
jmi.jar
jmi-uml-1.4.jar
mof.jar

Detailed explanation:
Request to install component "odysseyxmi-0.0.4.jar".
Component "odysseyxmi-0.0.4.jar" depends on component "nbmdr.jar", which is not installed.
Component "odysseyxmi-0.0.4.jar" depends on component "jmiutils.jar", which is not installed.
Component "odysseyxmi-0.0.4.jar" depends on component "mdrapi.jar", which is not installed.
Component "odysseyxmi-0.0.4.jar" depends on component "openide-util.jar", which is not installed.
Component "odysseyxmi-0.0.4.jar" depends on component "jmi.jar", which is not installed.
Component "odysseyxmi-0.0.4.jar" depends on component "jmi-uml-1.4.jar", which is not installed.
Component "odysseyxmi-0.0.4.jar" depends on component "mof.jar", which is not installed.

Your system is being updated!

Please, wait until updating finishes or click on Close to cancel it.

Installing component odysseyxmi-0.0.4.jar...
Installing component nbmdr.jar...
Installing component jmiutils.jar...
Installing component mdrapi.jar...
Installing component openide-util.jar...
Installing component jmi.jar...
Installing component jmi-uml-1.4.jar...
Installing component mof.jar...

Figure 5: Component dynamic loading in the Odyssey environment.

After the install or remove options are selected, the integration wizard reports component dependencies, as shown in Figure 5.b. Finally, the wizard downloads the components, as shown in Figure 5.c, and dynamically loads them into the environment, creating new menus and performing additional integration issues.

5. Conclusions

This paper presented a tool for component dynamic loading inside a run-time environment. This tool was designed and implemented to provide a run-time variability feature to the Odyssey environment. In this context, the Odyssey functionality set was clustered and mapped into components by means of a Java interface (*Tool interface)* and a component descriptor.

The main contribution of this approach is the automation and decentralization of software distribution and customization issues. The dynamic loading of components benefits both Odyssey end-users and developers. The end-user benefits from the ability to customize the environment, reducing the interface cluttering by unused menu options and increasing run-time and load-time performances. The developer benefits from the ability to develop new tools using a well-know and non-intrusive programming interface. The description of the mechanism may be useful to other developers facing similar problems in software maintenance and evolution.

The Odyssey-Light environment kernel is available through InstallAnywhere [10]. Additional tool components and libraries can be downloaded and plugged through the dynamic loading mechanism. Some new features, such as the description of conflicts among components, will become available in the next releases of this tool. Also, some of the original tools are still being wrapped into a component and will be available soon. Finally, to integrate the tools in other environments it is necessary to remove the dependencies from the plug-in tools to the Odyssey-Light and to build Adapters that should intermediate the communication between the tools and the selected environment, encapsulating these dependencies. In addition, for each new environment a particular packaging specification should be provided.

Acknowledgements
The authors would like to thank CAPES and CNPq, for their financial support, and all participants of Odyssey Project, who direct or indirectly supported this work.

References

1. Baker, B. The Official InstallShield for Windows Installer Developer's Guide. John Wiley & Sons, 2001.
2. Borland. Java Development Solution for Java Applications: Borland JBuilder X. In: http://www.borland.com/jbuilder/, Accessed in 03/02/2004.
3. Eclipse Foundation. Main Page. In: http://www.eclipse.org, Accessed in 03/02/2004.
4. Garg, A., Critchlow, M., Chen, P., et al. An Environment for Managing Evolving Product Line Architectures. In: International Conference on Software Maintenance, Amsterdam, Netherlands, September 2003, p. 358-367.
5. Hoek, A. Design-Time Product Line Architectures for Any-Time Variability Science of Computer Programming, special issue on Software Variability Management, 2004.
6. Larsson, M. Applying Configuration Management Techniques to Component-Based Systems. Licentiate Thesis, Department of Information Technology, Uppsala University, Sweden, 2000.
7. Leon, A. A Guide to Software Configuration Management. Norwood, MA, Artech House Publishers, 2000.
8. Miler, N., Werner, C. M. L., Braga, R. M. M. O uso de Modelos de Features na Engenharia de Aplicações. In: IDEAS´00, Cancun, México, April 2000, p. 85-96.
9. NetBeans Community. In: http://www.netbeans.org, Accessed in 03/02/2004.
10. Odyssey. Projeto Odyssey. In: http://www.cos.ufrj.br/~odyssey, Accessed in 02/03/2004.
11. Svahnberg, M., Gurp, J., Bosch, J. A Taxonomy of Variability Realization Techniques. Blekinge Institute of Technology, Sweden, 2002.
12. Werner, C. M. L., Mangan, M. A. S., Murta, L. G. P., et al. OdysseyShare: an Environment for Collaborative Component-Based Development. In: IEEE Conference on Information Reuse and Integration (IRI), Las Vegas, Nevada, October 2003, p. 61-68.
13. Zero G Team. InstallAnywhere Tutorial and Reference Guide. Addison-Wesley Pub Co, 2004.