Tool support for semi-automatic modularization of existing code bases

[Extended Abstract]

Robert Mischke, Doreen Seider, Andreas Schreiber
German Aerospace Center (DLR), Simulation and Software Technology
Cologne, Germany
Robert.Mischke@dlr.de, Doreen.Seider@dlr.de, Andreas.Schreiber@dlr.de

ABSTRACT
Many component-based systems and frameworks require the integration of external codes, for example, providing numerical functionalities. These numerical codes can be either sequential or parallelized, written in languages such as C, Fortran, Python, or Java. Frameworks provide support for workflow management, data management, using distributed computing resources, or a graphical user interface. Today, modern systems are based on Eclipse and OSGi or similar technologies.

For many frameworks, tight integration of pre-existing or third-party code requires manual source code changes to add the specific component interfaces to such code. As this is error-prone and time-consuming, especially when large code bases must be integrated, tool support for these steps becomes useful, or even necessary.

Tool support for automatic integration of existing code (in different languages) comprises several sub-problems such as code analysis, code transformation, generation of wrapper code, generation of proper user interfaces, and others. In this paper, we focus on the aspect of modularization of existing Java OSGi workflow systems and present a new Eclipse-based tool which provides end-user support for the migration of previously unmodularized software into modules or components.

Categories and Subject Descriptors
C.2.4 [Distributed Systems]: Distributed applications;
D.1.2 [Programming Techniques]: Automatic Programming

1. INTRODUCTION
In many scientific and engineering domains, multidisciplinary coupled simulation is commonly used, for example, to design complex technical systems (ships, spacecrafts, automobiles etc.). Usually for each discipline, certain numerical codes exist which are often integrated into component-based frameworks. Typical frameworks provide support for workflow management, data management, using distributed computing resources, or a graphical user interface. The numerical codes are written in languages such as C, C++, Fortran, Python, or Java. Often, they are parallelized and require HPC resources to run.

For many frameworks, tight integration of pre-existing or third-party code requires manual source code changes to add the specific component interfaces to such code. The component interface is defined by the component model of the framework (e.g., specified by CCA [3], OSGi, CORBA [4], or Web Services). A specific topic is the development of suitable wrapper code for external applications where the source code is not available or for code written in a different language than the framework. For example, for Java-based component frameworks codes in C or Fortran could be integrated by suitable JNI wrapper code. As the integration of numerical codes can be error-prone and time-consuming, especially when large code bases must be integrated, tool support for these steps becomes useful, or even necessary.

Tool support for automatic integration of existing code in different languages comprises several sub-problems such as code analysis, code transformation, generation of wrapper code, generation of proper user interfaces, and others. In this paper, we focus on the aspect of modularization of existing Java code as a first step towards the long-term goal of generic, automatic, cross-language integration of code into OSGi-based systems. As part of this effort, a new Eclipse-based tool is developed. This tool provides end-user support for the migration of previously unmodularized software into modules with explicitly defined boundaries, exports, imports, and embedded module-private libraries. While there are separate tools for static dependency analysis, visualization, or rule checking, none of them are combined to provide useful OSGi modularization support inside the Eclipse IDE yet. Future versions of the tool are planned to support other programming languages, frameworks, and programming models.
Data
Host
plug-ins
added and managed as be easily extended by application specific services which are RCE allows the transparent access to resources. RCE can
tures. The seamless integration of Grid technologies into is platform independent and can be used on most architec-
ton OSGi (Open Services Gateway initiative), therefore RCE
adaptable to a variety of application domains. It is based
and services. RCE is application independent and easily
management. A workflow in RCE consists of plug-ins (i.e.,
resources. Some have a high demand for computational power, others need a huge storage capacity. RCE is realized as a distributed system (Figure 1). It can run each plug-in of a workflow on the re-
source which meets its needs best. The only requirement is
an installed RCE instance on each resource.

For workflows with complex, computationally intensive sim-
ulations, RCE provides a Grid interface that enables plug-
ins to source computations out to the Grid. This inter-
face is realized by integrating the Grid Application Toolkit
(GAT) [1] into RCE. GAT provides a uniform API to access
heterogeneous compute and data resources. By making use of adaptors it is able to support different Grid middleware (e.g., Globus or UNICORE) and resource systems by having
one uniform and stable API. New technologies will be sup-
ported by adding an adaptor without the need to modify the
API. This Grid interface of RCE allows workflows to be executed on the Grid. More precisely, their components
(i.e., plug-ins) are enabled to run computations in the Grid. This concept allows the combination of plug-ins running in the Grid or on HPC resources with plug-ins being executed in RCE.

3. ECLIPSE TOOL SUPPORT
On the lowest abstraction level, the developed Eclipse tool provides a condensed model of dependencies between the Java types located in the analyzed projects and their sub-
projects. Using the abstract source tree (AST) parser of the
Eclipse Java Development Tools for character-level analysis, dependency patterns are detected and categorized to help the creation of higher-level models.

Binary types and packages are analyzed as dependency tar-
gets as well, together with references to the deployment containers (JAR files) from which they emerged. This in-
formation is important for several reasons. First, the OSGi
specification forces the developer to either define an explicit import on required binary packages, or to completely embed the required JAR file inside the defined module. Having access to the deployment information of the referenced
types allows the user to decide whether the automatic cre-
ation of import statements, or a complete inclusion of the
affected JAR file is more appropriate.

A second reason why dependencies to binary types are im-
portant is the possible use of the developed tool in platform
or library migration. By omitting the old platform from the
list of valid dependencies, the user is presented with a list
of forbidden references, which provides him with an auto-
matic guideline for the necessary source code adaptations.
This supports a “soft” migration, keeping the source code in
a consistent and compiling state as long as possible. This
option has a similarity to the Eclipse “Access Rules” feature,
but has the advantage of operating on a virtual module def-
inition. This allows to define the migration rules in a more
flexible way. For example, the new rule set can define both
source code as well as library elements as invalid, and is also
not restricted to Eclipse project boundaries.

3.1 Visualization and User Interface
The most detailed view of the dependencies between the
various elements is the modularization tree view (Figure 2).
By various grouping options, the user can easily assess the
source, destination and type of the current dependencies,
with the default view showing only those that violate the
access rules derived from the current modularization. Based
on this information, the user can choose to manually elim-
nate certain dependencies, or to semi-automatically adapt
the module configuration to the current dependencies. An
analysis of dependency cycles on the type and package level
is also provided.

While the tree view is useful in untangling low-level interde-
pendencies, it does not scale well for large sets of intercon-
ected packages. To address this situation, a dependency
structure matrix [5, 7] is also provided (Figure 3). In its
most basic form, its columns and rows are the same, with
the cells showing the number of dependencies from the col-
umn to the row element. The cell background indicates the
type of access permission between these elements under the
current modularization design. By selecting a matrix cell,
the underlying dependencies are shown in a smaller tree view below to assist the user in understanding the nature of the (possibly conflicting) access constellation. To improve on this basic design, several adaptations are being evaluated, including filtering and partitioning options, as well as allowing non-quadratic matrices and mixing of package and module elements along the axes.

3.2 Migration Workflow

As choosing a top-level architecture for modular decomposition of an existing software is an inherently semantic process, no attempt is made to automate the selection of root modules. Instead, an incremental approach is proposed, gradually transforming a set of unassigned source packages into a network of modules.

Initially, the user is presented a list of all source packages within the project scope. From these, the user defines an arbitrary number of modules that outline the basic modularization intent. Additionally, a set of source and binary packages can be defined that describes the environment the final modules will be deployed to. Each package from the project scope can be assigned at most to one module or to this “provided” set. The unassigned packages are kept in automatic “unassigned source” and “unassigned binary” sets.

The new Eclipse tool support continually displays the incoming and outgoing references between the defined modules and these special package sets. After the initial modules have been defined, three workflow stages are repeated until a valid modularization state is reached. These three stages resolve package cycles, invalid outgoing dependencies, and invalid incoming dependencies.

In the first stage, package cycles affecting any of the defined modules are examined. If a cycle affects only one module, one option is to add the whole cyclic chain to the affected module, which, by definition, makes the cyclic dependencies valid under the OSGi access rules. Of course, this approach is only desirable if the connected packages have a high internal cohesion, in which case such a tight coupling is acceptable. If the user decides that this is inappropriate, he can use the tool support to identify the connecting dependencies and decouple them manually or by using existing refactorings. If a cyclic dependency affects more than one defined module, the user is forced to resolve this situation on the code level, as such a cycle makes an automatic assignment impossible.

After all cyclic dependencies with packages outside the modules have been resolved, the second stage aims at resolving all outbound dependencies that violate the defined access rules. The basic forms of invalid external references are dependencies on unassigned source or binary packages, references to platform-provided packages without corresponding import declarations, and references to other modules which are not backed by any (package or module) import declaration. Without going into the detailed options for each case, tool support is provided to either create missing imports, to declare referenced packages as “provided”, to define new target modules, or to merge the referenced packages into the referring module.

After completing the second stage, all modules are valid in the sense that they have no outgoing dependencies preventing their deployment as modules. However, there may, however, be unassigned source packages referencing these modules in violation of the defined access rules. The third stage of the proposed workflow covers the resolution of these invalid references. Using the tree or the matrix views, the incoming references can be examined. Based on this inspection, the user can decide between adding the external package to the affected module, creating a new module, adding the package to another module, or removing the dependency by using existing refactorings. In the cases where additional modules are affected, appropriate export and import statements can be added semi-automatically to the target and source modules.

As adding previously unassigned packages to modules in stage three can possibly introduce new cycles or new outgoing references, a new iteration from stage one is performed until no stage effects a change anymore, resulting in an exhaustive and completely valid modularization setup.

In an optional fourth stage, the exported packages of all defined modules can be examined for unnecessary exports, and possible reductions in type coupling and visibility. The amount of tool support for this stage is not determined yet.

4. FUTURE WORK

While the developed tool provides support for the migration of existing Java code to an OSGi environment, it only represents a first step towards the long-term goal of automatic integration. A useful integration support would be a tool that only leaves the essential, behaviour-defining decisions to the user, while completely automating the laborious steps of refactoring, untangling, interface wrapping, and target platform adaptation. While this goal is out of reach of current software technology, it can serve as a guideline to determine pragmatic steps towards better integration support.

There are three principal topics for medium-term work. The first is the definition of an abstract model of cross-language code integration, and the creation of tool support for the specific properties of the supported target languages. C and Fortran will be supported first, but the underlying model
should be generic enough to cover all relevant platforms. One part of this model would be a cross-language definition of all possible dependency constellations. As every target language has a different set of existing tools and possible refactorings, an abstract description of their respective properties should also be aimed at to allow a generic evaluation of design options.

The second topic will be the adaption of the current modularization support and workflow to the properties of the target languages. As the goal for all underlying programming languages is the creation of OSGi-compliant modules, the modularization model itself is unlikely to change. However, it must be evaluated how the specifics of each language can be mapped to OSGi, which was designed primarily as a Java framework. For example, code written in C might depend on precompiled, platform-specific libraries, which must be provided at runtime within the OSGi environment. Such requirements should be abstracted, and generic solutions should be collected in form of a pattern library.

The third research topics will be the integration of existing wrapper generation tools [2, 6] into the migration workflow. Together, these elements should help the integration of existing code into OSGi environments. Future versions of the tool are planned to support other programming languages, frameworks, and programming models.

5. REFERENCES


