Serdica J. Computing **5** (2011), 79–100

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

MODERNIZING LEGACY PHYSICS APPLICATIONS FOR REUSE IN WEB AND SOA*

Anna Malinova

Abstract. This paper concerns the application of recent information technologies for creating a software system for numerical simulations in the domain of plasma physics and in particular metal vapor lasers. The presented work is connected with performing modernization of legacy physics software for reuse on the web and inside a Service-Oriented Architecture environment. Applied and described is the creation of Java front-ends of legacy C++ and FORTRAN codes. Then the transformation of some of the scientific components into web services, as well as the creation of a web interface to the legacy application, is presented. The use of the BPEL language for managing scientific workflows is also considered.

ACM Computing Classification System (1998): D.2.12, D.2.13, D.2.11, H.3.5, I.6.8.

Key words: reengineering, wrapping, legacy software, JNI, SOA, BPEL, scientific workflows, web services.

*This article presents the principal results of the doctoral thesis "Software system for computer simulation of metal vapor lasers" by Anna Malinova (Paisii Hilendarski University of Plovdiv), successfully defended before the Specialized Academic Council for Informatics and Mathematical Modelling on 2 November 2009. Additional research results and basic aspects of future development are also presented.

This work was partially supported by the NSF of the Bulgarian Ministry of Education and Science, project VU-MI-205/2006, and the RS-2009-M13 project of the Scientific Fund of the Paisii Hilendarski University of Plovdiv, Bulgaria.

1. Introduction. The intensive development of computer technology, informatics, and information technologies gave a new meaning to the concepts of scientific experiment and modern physics laboratory. An important direction of physics research is solving the problems of plasma physics, which is in the basis of studying the processes of a broad spectrum of technical devices, lasers and metal vapor lasers in particular.

As a result of the growing interest in simulating metal vapor lasers, new results for the improvement of the characteristics of the laser have been obtained in recent years and many mathematical and statistical methods and corresponding numerical algorithms have also been developed. For instance, some of our group's results are: In [13], [14] and [15] a new analytical formula for determining the gas temperature profile of the gas discharge in CuBr, ultraviolet CuBr and He-$SrBr_2$ lasers was obtained. These results are used to produce corresponding codes for computer simulation of various temperature and cooling processes under varying laser operating conditions. In [12] a numerical model for simulation of the electric field intensity in copper bromide lasers was developed. The model is applied in the created BPEL process for simulation of the electric field potential and intensity [24], as discussed further in this paper. Most of these codes solve particular concrete tasks of the entire process of computer simulation of metal vapor lasers and have become a basis for searching solutions for integration and reuse of legacy physics applications.

After a survey of a large number of existing software products in the domain of plasma physics and laser simulations, several conclusions have been pointed out [26]:

- A significant amount of resources has been invested toward the development of high-performance scientific simulation software, including numerical libraries, visualization, software frameworks, and physics packages;
- Applications are based on complicated mathematics and physics results and have been repeatedly tested and refined over the years;
- Developing such systems requires interdisciplinary teams;
- General-purpose physics software systems can serve only for a part of the simulation process in contrast to specialized applications;
- Many of the applications developed and used for numerical simulations are command-line or desktop applications written in FORTRAN, C or C++;

- Most of the applications are not designed for interoperability and reuse. It is difficult to share these software packages among applications due to differences in implementation language, programming style, or calling interfaces.

These applications are often considered *legacy* code, because they are developed with technologies that precede contemporary approaches and best practices. For reasons of reliability and efficiency, it is highly desirable to be able to reuse such large and complicated software packages without having to devote much time to reengineering them.

It is obvious that as simulations become increasingly complex and interdisciplinary no single person or even single group can develop scientific software in isolation. Development teams rarely have enough scientific expertise in all required domains to successfully create a complex application from scratch. In addition, contemporary scientific applications are expected to be distributed, multiplatform, scalable, and to meet such requirements as reusability, flexibility, and interoperability.

Awareness about these arguments in favour of reusing and modernization led us to the idea of creating Java front-ends of the native legacy software. *Native* software denotes code that is "implemented in platform-dependent code, typically written in another programming language such as C, C++, FORTRAN, or assembly language", as stated in the Java language specification [16]. In order to address the integration requirements involved in building connected applications we decided to apply a web services–based approach and to convert some of the created Java modules into web services. Next these web services may be chained into a BPEL process for simulation [24].

This paper presents the principal results of the application of the abovementioned information technologies for creating software for numerical simulations and in particular simulation of metal vapor lasers [23]. This work is connected with performing modernization through wrapping of native legacy physics software and with building a Service-Oriented Architecture (SOA) that allows, through integration of modules developed by separate teams, for automation of the design process to be achieved, as well as obtaining and evaluation of basic laser characteristics, investigation and control over the laser generation, based on well-known mathematical, statistical and optimization models.

Section 2 presents the process of wrapping native physics codes in order for them to be used in a Java-based environment for conducting numerical simulations of metal vapor lasers. The creation of the wrapping code includes application of the basic principles of mixed language programming, the use of methods through which a Java application communicates with legacy code that is part of the same process—Java Native Interface, and wrapping an application as a whole by launching it in a process that is external to the virtual machine. The building of the wrapper code is surveyed from a point of view related to the object-oriented design and design patterns [29]. Described are: the wrapping of a solver for calculating the distribution of the potential and intensity of an electric field [26], [25]; the wrapping of an entire legacy application—a finite element mesh generator [25]; the wrapping of the basic functionality of Plasimo—a software system for simulation of low-temperature plasma [28], [8].

Section 3 presents the application of Service-Oriented Architecture for building a software system for simulation of metal vapor lasers. The use of the Business Process Execution Language (BPEL) for building scientific workflows is presented, as well as an analysis of the BPEL specification in the context of the scientific workflow's requirements [27]. Also described are: the building of a BPEL process for simulation of the distribution of the electric field potential and intensity [24]; the application in the Plasimo software system—building the WebPlasimo prototype [8].

Section 4 presents the basic aspects of future development. Section 5 provides concluding remarks.

2. Integration through wrapping of legacy physics software.

In [30] an overview of the general approaches to legacy software modernization is provided. These are considered in the context of the basic activities related to system evolution, which are maintenance, modernization and replacement. Discussed are black-box and white-box modernization strategies, wrapping and reengineering, respectively. It is demonstrated that although white-box and black-box approaches suggest wrapping as an alternative strategy to reengineering and redevelopment, quite often wrapping is introduced as one of the techniques to carry out the reengineering, or it is defined as a “black-box reengineering task”. This assumes a broader understanding of the reengineering process that depends on the level of abstraction at which wrapping has been performed. For instance, the wrapping techniques and practical experience presented further in the recent paper show that wrapping is most often not entirely a “black-box” approach and requires some level of reverse engineering for better understanding of the wrapped legacy interfaces, class hierarchy or objects interrelations. In this process a need for re-documentation and design recovery may appear. In addition, in our work, after the completion of the wrapping process, a subsequent process of forward engineering has been performed over the wrappers in order to extend the functionality of the legacy system, add safety or new features in wrappers by the use of the new technologies that became available as a result of the overall reengineering process. Further, in [30] primary legacy modernization techniques are outlined, such as automated migration, re-hosting, package implementation, reengineering/re-architecturing, SOA integration. Special attention is paid to the modernization towards an SOA environment and its realization through reengineering or wrapping.

This section presents the process of wrapping legacy scientific codes in the domain of plasma physics and simulation of metal vapor lasers. Wrapping was chosen as a preferred modernization strategy because this is the only approach that does not entail performing code changes to the legacy system and because this technique allows for reusing on one hand, and adoption of modern technologies on the other.

The aim of this work was to create Java front-ends of existing modules, written in FORTRAN, C and C++, and thus enable them to be further reintegrated into a web application or a Service-Oriented Architecture, as will be shown in Section 3. Some of these native components were our legacy codes, while others were created by disparate teams.

2.1. Methodology. Two approaches allowing for reusing without modification have been applied and evaluated in our work. These methods are not mutually exclusive and the main reason for choosing one or the other is the level of interaction with the legacy system. The first method uses the Java Native Interface (JNI) which enables integration with native legacy code by providing a programming interface to the native environment that a Java virtual machine is running on. In this case the interaction between the wrapper and the target system involves class instantiation, method invocation, and control over the created objects' lifecycle. The second method wraps an application as a whole and is connected with invoking the precompiled executable by forking off a separate process, external to the Java runtime environment. This method has many limitations compared to JNI and is appropriate when the interaction requirements are relatively simple.

2.1.1. Creating Java wrappers of legacy modules through JNI. As a part of the Java virtual machine implementation, the JNI is a two-way interface that allows Java applications to invoke native code and vice versa [17]. For instance, the Java code can invoke native methods written in programming languages such as C and C++; also, native methods can create, update and inspect Java objects and call their methods. Java applications call native methods in the same way they call methods implemented in the Java programming language. Behind the scenes, however, native methods are implemented in another language and reside in native libraries. We use JNI to write the "glue" code between the two parts. Basic aspects of using JNI are: how a Java program calls native methods; how a native method can access Java class members; how the corresponding types are mapped; how exceptions are handled. These situations involve understanding of the use of opaque references, reflection support, callback functions, and such operating system specifics as locating and loading native libraries and linking native methods with native libraries that implement them.

When **wrapping C/C++ code** the following procedure is applied [20], [26]: 1) Define the Java wrapper with the declarations of native methods. This Java class usually loads the native library and invokes the native methods; 2) Generate the header file to be included into the intermediate C/C++ code; 3) Create the native C/C++ implementation containing the JNI "glue" code; 4) Compile the intermediate C/C++ code and generate dynamic (.dll) or shared object (.so) library.

When **wrapping FORTRAN code** additional C/C++ code is needed around the legacy FORTRAN module. Integrating C and FORTRAN codes has been practised since the advent of the first C compilers. FORTRAN–C interactions are two-way and do not require an additional interface like the JNI. A C/C++ function calls a FORTRAN function or subroutine like any other C function taking into account the mixed-language programming issues.

There are several key issues that should be of concern when applying **mixed-language programming** ([4], [32], [20]). Usually there are important differences in the way languages implement the following: calling conventions; naming conventions; passing by value or by reference; handling data types in multiple languages. These issues are important to both interactions—Java-C/C++ and FORTRAN-C/C++. For instance, calling conventions vary for different native languages. In addition, different implementations of the same language may follow different calling conventions, but generally we may say that C++ follows the `cdecl` calling convention, while FORTRAN follows the `stdcall` calling convention. In [20] it is discussed that the JNI requires the native methods to be written in a specified standard calling convention in a given host environment. Thus the JNI follows the C calling convention under UNIX and the `stdcall` convention under Win32.

Since the JNI wrapping adds an additional layer around the native code, the impact of that layer on the overall system **performance** should be considered. Writing critical portions of a Java application in native code is usually intended to improve performance. In [38] it is argued that after the appearance of the JIT compilers this is no longer a good solution. It is shown that the overhead of crossing the Java/C boundary may be severe enough to compensate the performance gains that could be obtained by moving code to C. Similar conclusions are reached in [19]: for native methods with very small amounts of computing, the additional invocation overhead can exceed the performance benefits. In our work concerning simulations, the transitions Java–native code were reduced to the possible minimum number, i.e., native invocations are "coarse-grained" and connected with native code that represents a basic step of the simulation process and involves intensive calculations. This way, by reducing the number of JNI calls and by leaving the cpu-intensive operations into the compiled native libraries, the efficiency of the real simulation practically does not depend on the wrapper layer.

2.1.2. Wrapping an entire application. This method wraps the native application at operating system level by invoking it as an external to the Java runtime environment process. Hence, interaction is limited to sending arguments to the executable via the standard input and reading results via the standard output. The wrapping process generally includes the use of two Java classes—`java.lang.Runtime` and `java.lang.Process`. Every Java application has a single instance of the class `Runtime` that allows the application to interface with the environment in which the application is running. A reference to the current runtime can be obtained from the `getRuntime` method. We use that reference to run external programs by invoking the `Runtime.exec` method [25]. This invocation creates a `Process` object—actually an object of the `Process` subclass is created, since the `Process` class is an abstract class and a specific subclass of it exists for each operating system.

This is the most straightforward way for a Java application to interact with standalone applications written in other languages. The Java API provides methods for performing input from a process, performing output to the process, waiting for the process to complete, checking the exit status of the process, and destroying the process. This approach, of course, sacrifices the low level integration for ease of use. Actually, ease of use may be regarded as a disadvantage of this method—in practice there are hidden problems. For instance, in the JDK Javadoc documentation it is stated that because some native platforms only provide limited buffer size for standard input and output streams, failure to promptly write the input stream or read the output stream of the subprocess may cause the subprocess to block and even deadlock. Hence buffering of the above operations is necessary. Other disadvantages of this method are that a predetermined knowledge of the name and location of the wrapped application is required and that the `Runtime.exec` method does not accept every shell command as could be mistakenly inferred from the documentation.

2.2. Application for simulations of metal vapour lasers.

2.2.1. Design of the wrapper code. In [29] different design approaches to the process of Java wrapping of native legacy scientific codes in the domain plasma physics and simulation of metal vapor lasers are presented. There are two integration options when encapsulating a native legacy application, as discussed in [5]. There could be either **one wrapper** for the entire application or several, one per a needed functionality. These wrappers can be used to form a **library of wrapper classes**, corresponding to different native classes or base services. Both approaches allow replacement of legacy services when the new application no longer needs a native legacy application to provide a service. In this case, if a new implementation of this functionality is provided, one method invocation would simply be replaced with another inside the wrapper class [29].

In the context of creating Java wrappers of native legacy applications, some well-known object-oriented techniques can be discussed, such as design patterns. The design patterns considered in [29] are the GoF's **Adapter** and **Proxy** [11]. Both are not directly applicable since the client and the adapted code are written in different languages, but can help with further understanding of the wrapping process and better structuring of the wrapper code. In addition, since it is often a question of adapting non-object-oriented legacy software, the **Wrapper Facade** design pattern [36] is also considered.

The most straightforward way to create wrapper classes through JNI is the **one-to-one mapping** [17], [20]. This approach requires us to write one stub function for each legacy native function we want to wrap. Hence, each Java method declared as native maps to a single native stub function, which in turn maps to a single legacy native method definition. The stub serves two purposes: adaptation of the native function's argument calling convention to what is expected by the Java virtual machine; conversion between the Java programming language types and native types. The object version of the Adapter design pattern could be related to the process of JNI wrapping of native legacy codes—the client sends a request to the Java wrapper class; then the JNI stub functions, implementing the Java methods declared as native, make the corresponding invocations of the underlying native functions. As a result, the legacy native interface is adapted to a Java interface.

One-to-one mapping addresses the problem of wrapping native functions. However, if an instance of a C++ class is created in a JNI stub function, another problem arises: how can C++ classes be used by a Java program and keep objects around while the program is running. One way to handle this situation is to define a Java class called "**peer class**" that corresponds to the C++ class [17], [20]. Peer classes directly correspond to native data structures. Each instance of the peer class corresponds to a C++ object, tracking the state of that object. The Proxy design pattern can be considered when creating Java peer classes that wrap native structures. In [11] the Proxy is defined as a surrogate or placeholder for another object in order to control the access to it. Thus the Proxy pattern makes the client of an object communicate with a representative of this object rather then the object itself. Such a representative can serve many purposes determined by its pre- and post-processing of requests. Java peer classes that wrap native data structures apply both Adapter and Proxy design patterns. On one hand the existing C++ interface is adapted to a Java interface and on the other—the peer class serves as a proxy to the native C++ class it represents, taking care of creating and deleting the instances of this class, and providing interface that is identical to or a subset of the wrapped one.

Integration with **non-object-oriented code**, written in such languages as C and FORTRAN, is discussed in [31]. The object-oriented re-architecturing technique presented there implies using object-oriented architecture (wrapper) around internal elements that are not object-oriented. The Wrapper Façade design pattern, presented in [36], encapsulates the functions and data provided by the non-object-oriented legacy native API within more concise, portable and maintainable object-oriented class interfaces. As far as the Java wrapping is concerned, the Wrapper Façade pattern corresponds to creating a C++ class that invokes the non-object-oriented code. That class is then wrapped through JNI. This additional C++ class (classes) provides a higher-level object-oriented interface that is easier to maintain and reuse. The alternative to the Wrapper Facade is to create a Java class that directly accesses the non-object code through the JNI stubs. This is the approach we have used in our work because of the small amount of the wrapped non-object-oriented code. This implies that all methods declared as native in it are also static. However, the Wrapper Façade can simplify the wrapping if there is a large amount of native functions to wrap.

2.2.2. Wrapping a legacy electric field intensity solver. This section describes the creation of a Java wrapper of a legacy FORTRAN solver we use to calculate the distribution of the electric field potential and intensity as part of the process of simulation of metal vapor lasers [26], [24]. Electric field intensity is one of the most important discharge parameters—its value has a direct influence on the current flow, space distribution of electric power, temperature profile of the gas discharge, electron energy, and ionization processes of gas molecules.

The solver is used under Windows and has a relatively simple interface—it receives arrays of input data and returns tables of numbers, written in external text files. The implementation of the wrapper code consists of: creating a C/C++ wrapper of the FORTRAN code; creating the JNI "glue" code; creating a Java wrapper class. In our work, however, the additional C++ wrapper and the JNI code were merged. Here an additional step to the procedure of wrapping C/C++ code is compiling both the FORTRAN and the C++ code into a dynamic link library which can be loaded and linked into the Java Virtual Machine.

As for mixed language issues, when wrapping FORTRAN code, it can be pointed out that the C/C++ code hides the various symbol-naming conventions used by FORTRAN compiler vendors. For instance, FORTAN symbols are sometimes uppercase, sometimes lowercase (or mixed as in C) and there may be one or two trailing underscores appended. In addition the C/C++ wrapper hides the name mangling of FORTRAN module symbols. In order for C++ compliers to recognize code generated by the FORTRAN compiler, name mangling must be switched off for these routines [7].

2.2.3. Wrapping a finite element mesh generator. This section describes the wrapping of an entire application through starting it as external to the Java virtual machine process, as discussed in Section 3.1.2. Our aim was to provide a web service interface to an existing application [25]. This way a web service can wrap an entire application, enclosing it and invoking it without having to modify the application. We have used the QMG 2.0 code [35] as an example. It is written in C++, but we didn't want to wrap it directly using the JNI. Instead, we invoked the precompiled QMG executable by forking off a separate process.

The QMG package generates finite element meshes in two and three dimensions. The mesh generator takes as input a "brep", which is a boundary representation of a two- and three-dimensional geometric object, and produces as output triangulation of that "brep". The user of QMG invokes the functions through a console interface. QMG does not have its own console interface—instead it relies on the scripting capabilities of other software packages (Matlab and Tcl/Tk). The QMG uses the scripting language as a front end for all the geometric modeling routines and the mesh generator itself.

We have created a Matlab script that uses the QMG programming interface to describe a geometry representing a cross section of a He-Cd laser. The laser is manufactured from two tubes: an internal Al_2O_3 tube, inserted in an external quartz tube and equipped by two outer longitudinal electrodes. Then we used the `exec` method of the `Runtime` class to run the Matlab application as a separate process. This returns a `Process` object to control the process and obtain information from it. The Java class containing the above code was easily converted into an Apache Axis web service. In [25] a sample Java server page that acts as a client to invoke the Web service is presented. The web service produces as output a triangulation of the boundary representation that was provided by the user.

Because of the limitations of this method compared to JNI we haven't used it any further. It can be concluded that this method is useful for simpler integration solutions.

2.2.4. Wrapping the basic functionality of Plasimo—a framework for modelling low-temperature plasma sources. This section presents the process of creating Java front-ends for a part of the basic functionality of the Plasimo simulation software [34], [9]. The Plasimo code is a multi-physics code for simulating a variety of plasma sources with various degrees of equilibrium, electromagnetic field configurations, flow regimes and geometries [8]. Plasimo is a framework written in C++ and the application of different wrapping techniques was investigated. The Java Native Interface was used to produce a class library that wraps a set of Plasimo's functions and classes. The aim of this wrapping was to give the legacy code access to new web technologies and best practices. For details we refer to [28], [8] and [29].

The techniques "one-to-one mapping" and creating Java peer classes were applied. In addition, such issues like exception handling and reflection support provided by the JNI are also discussed in [28]. We have created a number of Java peer classes that correspond to basic Plasimo classes. The native methods are called within the peer classes and are the link between the peer classes and the C++ classes. Furthermore, we have created an abstract Java peer class that all peer classes extend. This class provides some common functionality and contains a 64-bit field that refers to the corresponding C++ instance of a PLASIMO class. Subclasses of the abstract peer class assign specific meaning to that field. If we are on a platform with 32-bit pointers, we can simply store this pointer in an `int`; if we are on a platform that uses 64-bit pointers, we store it in a `long`.

Figure 1 presents the dependency relationships between different Plasimo components after the Java front-end was created. The application JPlasimo is a test application that invokes the native methods of the created Java wrapper classes. The implementation of the native methods is provided by a set of libraries consisting of JNI stub functions, which in turn invoke the Plasimo functions inside the Plasimo compiled libraries.

3. Application of the service-oriented architecture for scientific software. The increasing use of distributed applications in different scientific and academic organizations has replaced traditional desktop applications. The contemporary virtual physics laboratory has to meet more and more the requirements of contemporary business applications, i.e., to be a reliable and scalable application used by multiple users.

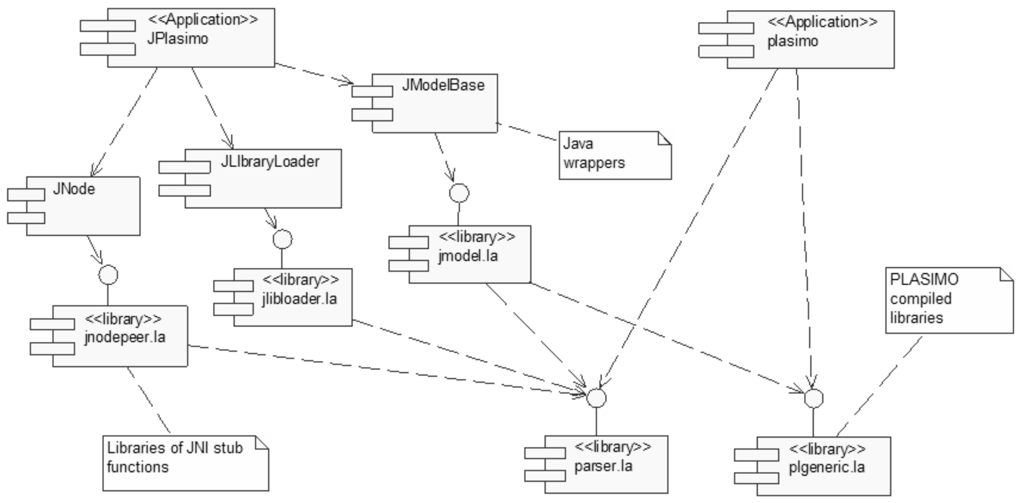


Fig. 1. UML component diagram showing the dependency relations between different Plasimo components after the Java wrapping

In order to address the integration requirements involved in building connected applications we have applied a web services-based approach with the aim of adopting Service-Oriented Architecture. According to [33] the term SOA refers to a style of building reliable distributed systems that deliver functionality as services, with the additional emphasis on loose coupling between interacting services. In [33] a service is defined as a software component that can be accessed via a network to provide functionality to a service requester. Loose coupling implies that the interacting software components minimize their inbuilt knowledge of each other: they discover the information they need at the time they need it [37]. SOA and web services are two different things, but web services are the preferred standards-based way to implement SOA [22].

The Business Process Execution Language (BPEL) [7] is an XML-based language used for integration of a number of web services into more complex composite services. Thus BPEL enables the top-down realization of SOA through composition, orchestration, and coordination of web services. The BPEL composite services are called business processes and are managed by a workflow engine.

3.1. Using BPEL for managing scientific processes. The spectrum of what might be called scientific workflow is wide and includes scientific discovery workflows, workflows that automate manual procedures or reengineer custom tools, and data and compute-intensive workflows. In this section we provide a number of common requirements of scientific workflows: service composition and reuse, scalability, detached execution, reliability and fault tolerance, user interaction, monitoring, "smart" re-runs, data provenance, etc. These are also discussed in [1], [18], [21].

In general, the BPEL vocabulary is tailored more to the requirements of business processes, which often have different requirements compared to scientific workflows. For example, in [21] it is shown that business workflow approaches focus on control-flow patterns and events, whereas dataflow is often a secondary issue. Scientific workflow systems, on the other hand, tend to have execution models that are much more dataflow-oriented.

In [27] we provide an analysis of the BPEL specification in the context of the requirements above listed. We do this also in the context of the implementation technology we have adopted—the Oracle BPEL Process Manager that is a part of the Oracle SOA Suite.

• **Service composition and reuse:** Web services can be combined in two ways: orchestration and choreography. BPEL supports two different ways of describing business processes that support orchestration and choreography: Executable processes—they follow the orchestration paradigm and can be executed by an orchestration engine; Abstract business protocols—they allow specification of the public message exchange between parties only. They do not include the internal details of process flows and are not executable. They follow the choreography paradigm.

• **Scalability:** Some scientific workflows involve large volumes of data and/or require high-end computational resources, e.g., running many parallel jobs on a cluster computer. Concurrency is provided in BPEL with the ¡flow¿ activity. BPEL also provides features to support handling of multiple requests—by creating multiple instances of the process, one for each interaction, and by declaring a correlation set in order to identify a particular instance of among a set of instances of that process.

• **Detached execution:** Long-running scientific workflows require an execution node that allows the workflow control engine to run in the background on a remote server, without necessarily staying connected to a user's client application that has started and is controlling the workflow execution. In a BPEL process a web service can be invoked as a synchronous or asynchronous operation. Asynchronous web services do not block the BPEL process and are useful for environments in which a service can take a long time to process a client request.

• **Reliability and fault tolerance:** A scientific workflow might incorporate a service that "fails" often, changes its interface, or just becomes unacceptably slow. Thus the workflow definition should support the definition of failure-handling mechanisms. BPEL provides a flexible structure for dealing with failures. Fault and compensation handlers are used to reverse the effects of partially completed interactions. The execution of these handlers is tied in with the concept of scopes.

• **User interaction:** Many scientific workflows require user decisions and interactions at various steps. BPEL 1.1 and 2.0 do not include human interactions and are limited to service orchestration. The Oracle BPEL Process Manager that we have used provides a manual task web service to integrate people and manual tasks into BPEL processes [6].

• **Monitoring:** Scientific workflows are potentially long-running activities and it is of importance for scientists to be able to observe and monitor the ongoing execution of a workflow. The Oracle BPEL Manager provides sensors to monitor BPEL activities, variables, and faults during runtime [6]. The following types of sensors can be defined, either through the BPEL Designer or manually by providing sensor configuration files: activity sensors; variable sensors; fault sensors.

While the above list of requirements for scientific workflow systems is not complete, it captures many of the core characteristics and we conclude that BPEL is fully applicable for managing scientific services. Other requirements may include "smart" re-computations, data provenance, and the use of an intuitive GUI to allow the user to compose a workflow visually from smaller components in order to animate workflow execution, to inspect intermediate results, etc., but these are not related to the BPEL specification itself. Rather, they are BPEL Designer and engine dependant.

3.2. Application of BPEL for building scientific processes for simulation of metal vapour lasers. In this section an example workflow for simulating the distribution of the electric field potential and intensity is presented. In [24] a detailed problem formulation, graphical representation of the created BPEL process, and description of the participating web services are provided.

The web services we have developed and orchestrated are: *GetInput-Service*, *ParametersApproval*, *ElectrodeService*, and *PoissonService*. The BPEL process waits for an incoming message from the client, which starts the execution of the simulation process. At the beginning of the process execution, the flow takes a *SimulationOrder* XML document as input. In a BPEL process everything is XML, including the messages that are passed into and returned from the BPEL process, the messages that are exchanged with external services, and any local variables used by the flow itself. The input XML file is the body of the generated SOAP request that initiates the simulation.

The flow gets the specified laser type as a string from the input XML file and then invokes the synchronous *GetInputService* service to request the relevant parameter set for this particular laser device. With a parameter set we denote a group of related parameters (physics constants, variables, etc.) with additional metadata describing various aspects of the parameters, such as help information, valid ranges (e.g., min, max, etc.), default values, whether the parameters are required or optional, etc. Thus we can manage different parameter sets relevant to different laser types, as well as have different parameter set instances of a particular laser type. Each of those XML files is an instance of an XML schema file we have created to describe the parameter types.

Once the required parameter set is obtained, the simulation process starts a human task, the service *ParametersApproval*, for a customer representative to manually approve the parameter values. This enables the user to change some of the predefined values of the physical constants, geometrical parameters, etc.

When the parameters have their approved values, the processing becomes automated. First the *ElectrodeService* is invoked synchronously. This is our legacy FORTRAN component wrapped as a web service. This component generates the appropriate mesh for discretization and classifies the points in the different sub-regions: outer electrodes, laser tube, etc. basic input parameter is the voltage applied to the electrodes, which is then used to determine the boundary conditions. Then the *PoissonService* is invoked. This is also one of our FORTRAN legacy codes, wrapped as web service, which is used to calculate the electric field potential and intensity distributions by solving a two-dimensional quasi-stationary Poisson equation. The service generates an output file with results.

3.3. Building the WebPlasimo prototype. The WebPlasimo prototype provides new interfaces to the Plasimo framework for modelling low-temperature plasma sources [8]. The main tasks we have set while building the WebPlasimo prototype were: 1) to create a web interface to the Plasimo framework; 2) to expose certain Plasimo functionalities as web services for use by other scientific teams. Both tasks involve creating Java wrappers of basic Plasimo functionality as was shown in Section 3.2.4. The basic components of WebPlasimo are: the server-side part of the application developed through the Apache Struts 2 framework [3]; a web-based client that is a Rich Internet Application developed using the Dojo framework [10]; web services developed through the Apache Axis 2 framework [2]; Java wrappers of Plasimo modules created through the JNI interface.

The following tools have been developed as parts of the WebPlasimo prototype:

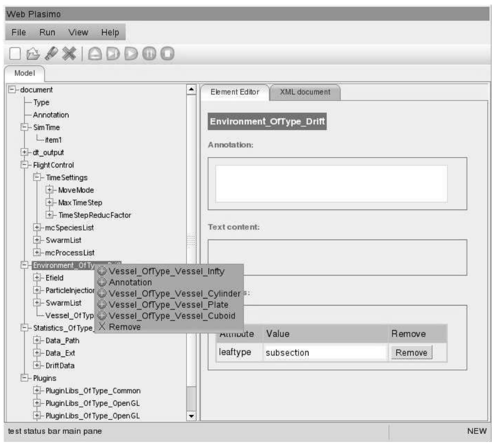
- **Generator of XML files:** generates an initial XML file based on a specified XML Schema file describing the necessary configuration data. This tool's purpose is to alleviate the creation of the simulation input file which would contain the mandatory parameters, the default and fixed values, and other related initial configuration data.

- **Web-based XML editor:** its purpose is to provide a user-friendly interface for editing XML files. For this purpose a number of utility functions for processing XML files have been created. A related tool for generating a GUI representation of an XML element being edited is developed as well. Views of the web-based XML editor are shown in Figure 2 and Figure 3.

- **Controller of the Plasimo simulation model being executed:** provides an interface for controlling the executed simulation model, e.g., install the model, start the model, perform one step of the simulation, pause, and stop the execution. The controller uses the corresponding JNI wrapped Plasimo classes.

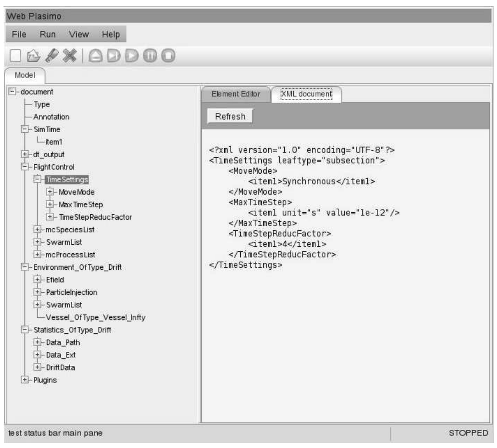
- **Tool for a web visualisation of the results of the simulation:** the Plasimo visualizing capabilities have been extended with C++ classes alleviating the web browser visualization of the textual and graphical representation of some Plasimo data types. These C++ classes have also been wrapped through JNI in order to be used by the WebPlasimo application as shown in Figure 4 and Figure 5.

Different **aspects of the web services applications** to the Plasimo framework and other physics software have been discussed in [8], such as: providing web services based access points to Plasimo as a whole, as well as to its individual building blocks; providing a web services-based post-processing on data servers in order for them to serve derived data rather than just “raw” data sets.


Web Plasimo
File Run View Help
Model
Element Editor
XML document
Environment_OfType_Drift
Annotation:
Text content:
Vessel_OfType_Vessel_Infty
Annotation
Vessel_OfType_Vessel_Cylinder
Vessel_OfType_Vessel_Plate
Vessel_OfType_Vessel_Cuboid
Remove
test status bar main pane
NEW

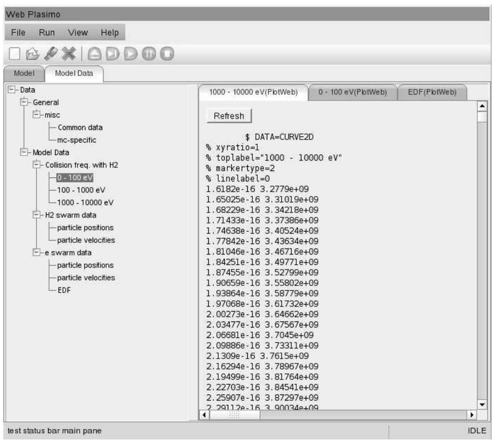
Attribute	Value	Remove
leaftype	subsection	Remove

Fig. 2. WebPlasimo—view of the XML editor



```
<?xml version="1.0" encoding="UTF-8"?>
<TimeSettings leaftype="subsection">
    <MoveMode>
        <item1>Synchronous</item1>
    </MoveMode>
    <MaxTimeStep>
        <item1 unit="s" value="1e-12"/>
    </MaxTimeStep>
    <TimeStepReducFactor>
        <item1>4</item1>
    </TimeStepReducFactor>
</TimeSettings>
```

Fig. 3. WebPlasimo—visualizing XML content



```
        $ DATA=CURVE2D
% xyratio=1
% toplabel="1000 - 10000 eV"
% markertype=2
% linelabel=0
1.6182e-16 3.2779e+09
1.65025e-16 3.31019e+09
1.68229e-16 3.34218e+09
1.71433e-16 3.37386e+09
1.74638e-16 3.40524e+09
1.77842e-16 3.43634e+09
1.81046e-16 3.46716e+09
1.84251e-16 3.49771e+09
1.87455e-16 3.52799e+09
1.90659e-16 3.55802e+09
1.93864e-16 3.58779e+09
1.97068e-16 3.61732e+09
2.00273e-16 3.64662e+09
2.03477e-16 3.67567e+09
2.06681e-16 3.7045e+09
2.09886e-16 3.73311e+09
2.1309e-16 3.7615e+09
2.16294e-16 3.78967e+09
2.19499e-16 3.81764e+09
2.22703e-16 3.84541e+09
2.25907e-16 3.87297e+09
2.29112e-16 3.90034e+09
```

Fig. 4. The prototype after one step of the simulation—textual representation

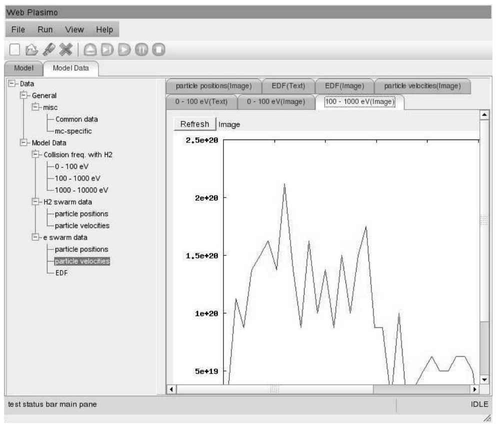


Fig. 5. The prototype after one step of the simulation—graphical representation

The web services we have created with testing purposes provide: operations for converting Plasimo-specific configuration files into XML format, and *vice versa*; an operation for obtaining the final results of the simulation—the results are in textual representation and step-by-step execution and pausing are not provided since the created web services do not manage state. In perspective, the next step is to develop stateful asynchronous web services in order to expose as a web service a fully functional simulation involving obtaining intermediate or partial results and step-by-step execution.

4. Future work. The future work in the domain of reuse and modernization of legacy physics codes and building linked applications in an SOA environment will focus on the following aspects:

- The ongoing research on the methods to process and share physics data, e.g., physics data servers to do some processing in order to provide derived quantities, not just "raw" data sets, as discussed in [8]; providing web services-based access to this additional functionality of the data servers.
- The ongoing development of the created XML schemas describing the parameter types.
- Development of stateful asynchronous web services using the session management provided by the SOAP frameworks used.
- Development of web services that provide access points to legacy physics functionality that can be used outside the context of simulation of metal vapor lasers and plasma physics.
- Transforming the created tool *Web-based XML editor* into a web services test console.

Although the above list is by no means complete, it shows our belief that future research on the adoption of SOA in scientific applications is important and needed because of the increasing requirements that a modern physics laboratory has to meet.

5. Conclusions. A significant amount of high-performance simulation software in the domain of plasma physics, created during the last two decades, is written in native languages, such as C, C++, and FORTRAN. Reasons of reliability and efficiency make it highly desirable to be able to reuse these large and complicated software packages. In the process of modernization towards web and SOA such a modernization technique as wrapping native codes continues to be a question of present interest. The approaches presented in this paper, wrapping an entire application at operating system level and wrapping a particular legacy functionality through the Java Native Interface, showed that this technique is not entirely a "black-box" modernization effort and includes a number of reengineering tasks.

The main conclusion from the presented work is that the choice of SOA and web services provides a unique opportunity to handle a complex simulation process involving multiple applications developed by disparate teams. In addition, it is shown that web and SOA enabling, which involves JNI wrapping, is feasible and does not affect the legacy application's performance, nor does it alter its function.

REFERENCES

[1] Akram A., D. Meredith, R. Allan. Application of Business Process Execution Language to scientific workflows. *Int. Trans. on Systems Science and Applications*, **1** (2006), No 3, 289–302.

[2] Apache Axis 2. `http://ws.apache.org/axis2/`.

[3] Apache Struts 2 framework. `http://struts.apache.org/2.x/`.

[4] Arnholm A. Mixed language programming using C++ and Fortran 77, version 1.1., 1997. `http://arnholm.org/software/index.htm`

[5] Asman P. Legacy Wrapping. `http://www.hillside.net/plop/plop2k/proceedings/Asman/Asman.pdf`

[6] Bradshaw D., M. Kennedy. Oracle® BPEL Process Manager Developer's Guide10g. `http://download-east.oracle.com/docs/cd/B31017_01/integrate.1013/b28981/toc.htm`

[7] Business Process Execution Language. `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`.

[8] Van Dijk J., A. Malinova, V. Yordanov, J. van der Mullen. New Interfaces for the Plasimo Framework. In: AIP Conf. Proc., 6th Int. Conf. on Atomic and Molecular Data and Their Applications, Beijing, China, 27–31 Oct. 2008, Vol. **1125**, 2009, 176–187.

[9] Van Dijk J. Modelling of Plasma Light Sources: an object oriented approach. PhD thesis, Eindhoven University of Technology, The Netherlands, 2001.

[10] Dojo Toolkit. `http://dojotoolkit.org/`.

[11] Gamma E., R. Helm, R. Johnson, J. Visslides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[12] Gocheva-Ilieva S., I. Iliev. Mathematical modeling of the electric field in copper bromide laser. In: Proceedings of the Int. Conf. of Numerical Analysis and Applied Mathematics, Sept. 16–20, 2007, Corfu, Greece, Vol. **CP936**, 527–530.

[13] Iliev I., S. Gocheva-Ilieva, N. Sabotinov. Analytic study of the temperature profile in a copper bromide laser. *Quantum Electron*, **38** (2008), No 4, 338–342.

[14] Iliev I., S. Gocheva-Ilieva, K. Temelkov, N. Vuchkov, N. Sabotinov. Modeling of the radial heat flow and cooling processes in a deep ultraviolet Cu+ Ne-CuBr laser. Mathematical Problems in Engineering, Hindawi Publ. Corp., Vol. **2009**, ID 582732.

[15] Iliev I., S. Gocheva-Ilieva, K. Temelkov, N. Vuchkov, N. Sabotinov. Analytical model of temperature profile for a He-SrBr2 laser. *Journal of Optoelectronics and Advanced Materials (JOAM)*, **11** (2009), No 7, 1025–1032.

[16] Java Language Specification,
`http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`.

[17] Java Native Interface Specification,
`http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html`.

[18] Joncheere N., W. Vanderperren, R. Straeten. Requirements for a Workflow System for Grid Service Composition. In: Proceedings of the 2nd Int. Workshop on Grid and Peer-to-Peer Based Workflows (GPWW 2006), Vienna, Austria, September 2006, Lecture Notes in Computer Science, Vol. **4103**, Springer-Verlag, 2006, 365–374.

[19] Kurziniec D., V. Sunderam. Efficient cooperation between Java and Native Codes-JNI Performance Benchmark. In: Proceedings of the 2001 Int.

Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001), 2001.

[20] Liang S. The Java Native Interface: Programmer's Guide and Specification, Addison-Wesley, 1999.

[21] Ludascher B., I. Altintas, C. Berrkley, D. Higgins, E. Jaeger, M. Jones, E. Lee, J. Tao Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice & Experience*, **18** (2006), 1039–1065.

[22] Mahmoud Q. Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI). 2005. `http://java.sun.com/developer/technivcalArticles/WebServices/soa/index.html`

[23] Malinova A. Software system for computer simulation of metal vapor lasers, Abstract of PhD thesis, Plovdiv University Press, 2009 (in Bulgarian).

[24] Malinova A., S. Gocheva-Ilieva. Application of the Business Process Execution Language for building scientific processes for simulation of metal vapor lasers. In: Proceedings of the 3rd Balkan Conf. in Informatics, Sofia, Bulgaria, 27–29 Sept., 2007, Volume **2**, 75–86.

[25] Malinova A., S. Gocheva-Ilieva, I. Iliev. Web services—based simulation of metal vapor lasers. In: Proceedings of the IX Int. Conf. on Laser & Laser Inf. Techn. & V Int. Symp. on Laser Techn. & Lasers ILLA/LTL'2006, Smolyan, Bulgaria, October 4–7, 2006, 315–321.

[26] Malinova A., S. Gocheva-Ilieva, I. Iliev. Wrapping legacy codes for Numerical simulation applications, In: Proceedings of the III International Bulgarian-Turkish Conf. Computer science, Istanbul, Turkey, October 12–15, 2006, Part II, 202–207.

[27] Malinova A., S. Gocheva-Ilieva. Using the Business Process Execution Language for managing scientific processes. *International Journal "Information Technologies and Knowledge"*, **2** (2008), 257–261.

[28] Malinova A., V. Yordanov, J. van Dijk. Leveraging existing plasma simulation codes. International Book Series "Information Science & Computing", No 5, Suppl. to the Int. J. "Information Technologies & Knowledge", **2** (2008), 136–142.

[29] Malinova . Design Approaches to Wrapping Native Legacy Codes, Scientific Works, Plovdiv University, Vol. **36**, Book 3, 2009, 89–100.

[30] Malinova A. Approaches and Techniques for Legacy Software Modernization. Scientific Works, Plovdiv University, Vol. **37**, Book 3, 2010-Mathematics, 77–85.

[31] Meyer B. Object Oriented Software Construction. 2nd ed., Prentice Hall, 1988.

[32] Mixed-Language Programming. `http://msdn.microsoft.com/library`.

[33] OGSA Glossary Terms v 1.5. `http://www.ogf.org/documents/GFD.81.pdf`.

[34] Plasimo simulation software. `http://plasimo.phys.tue.nl`

[35] QMG 2.0 Mesh generation software.
`http://www.cs.cornell.edu/home/vavasis/qmg2.0/qmg2_0_home.html`.

[36] Schmidt D., M. Stall, H. Rohnert, F. Buschmann. Pattern-Oriented Software Architecture—Patterns for concurrent and networked objects, Volume **2**, Willey, 2000.

[37] Srinivasan L., J. Treadwell. An Overview of Service-oriented Architecture, Web Services and Grid Computing, Nov 2005.
`http://h71028.www7.hp.com/ERC/downloads/`
`SOA-Grid-HP-WhitePaper.pdf`.

[38] Wilson S., J. Kesselman. Java Platform Performance: Strategies and Tactics, Prentice Hall, 2001. `http://java.sun.com/docs/books/`
`performance/1st_edition/html/JPTitle.fm.html`.

Anna Malinova
Department of Computer Technologies
Faculty of Mathematics and Informatics
University of Plovdiv
236 Bulgaria Blvd
4003 Plovdiv, Bulgaria
e-mail: `malinova@uni-plovdiv.bg`

Received Junuary 10, 2011
Final Accepted February 7, 2011