Proceedings of GT2007
ASME Turbo Expo 2007: Power for Land, Sea and Air
May 14-17, 2007, Montreal, Canada

GT2007-27086

ADVANCED CAPABILITIES FOR GAS TURBINE ENGINE PERFORMANCE SIMULATIONS

A. Alexiou
Research Associate
Laboratory of Thermal Turbomachines
National Technical University of Athens
PO BOX 64069, Athens 15710, Greece
Email: a.alexiou@ltt.ntua.gr

E.H. Baalbergen
System Architect
Collaborative Engineering Systems Department
National Aerospace Laboratory, NLR
PO BOX 90502, 1006BM Amsterdam, The Netherlands
Email: baalber@nlr.nl

O. Kogenhop
Gas Turbine Performance Engineer
Gas Turbines & Structural Integrity Dept.
National Aerospace Laboratory, NLR
PO BOX 153, 8300AD Emmeloord,
The Netherlands
Email: okogenho@nlr.nl

K. Mathioudakis
Professor
Laboratory of Thermal Turbomachines
National Technical University of Athens
PO BOX 64069, Athens 15710,
Greece
Email: kmathiou@central.ntua.gr

P. Arendsen
Structural Design Specialist
Collaborative Engineering Department
National Aerospace Laboratory, NLR
PO BOX 153, 8300AD Emmeloord,
The Netherlands
Email: arends@nlr.nl

ABSTRACT

This paper describes the integration of advanced methods such as component zooming and distributed computing, in an object-oriented simulation environment dedicated to gas turbine engine performance modelling.

A 1-D compressor stage stacking method is used to demonstrate three approaches for integrating numerical zooming in an engine model. In the first approach a 1-D compressor model produces a compressor map that is then used in the engine model in place of the default one. In the second approach the results of the 1-D analysis are passed to the 0-D component through appropriate 'zooming' scalars. In the final approach the 1-D compressor component directly replaces the 0-D one in the engine model.

Distributed computing is realized using Web Services technology. The implementation steps for a distributed scenario are presented. The standalone compressor stage stacking method, in the form of a shared library, is placed in a remote site and can be accessed over the internet through a Web Service Operation (server side). An engine simulation is set up containing a 1-D compressor component which acts as the client for the Web Service operation.

Future development of the tool's advanced capabilities is finally discussed.

Keywords: simulation, zooming, distributed computing, web services

INTRODUCTION

The tough competition between gas turbine manufacturers dictates a drastic reduction in both development time and cost for all new engine programs. This results in multi-partner, co-operative projects where each partner is responsible for an engine's subsystem. Furthermore, the industry increasingly relies on the use of computer simulation technology to reduce the number of hardware tests required. Traditionally, individual partners have their own simulation methodology as well as simulation tools (commercial or in-house) which make it difficult to integrate different simulation modules in a single engine model and/or compare simulation results. Some of these different simulation tools are presented in [1]. Consequently, each partner builds and maintains its own engine model resulting in effort duplication and hence significant waste of resources and increasing the scope for error in the data transfer due to the lack of common modelling and simulation standards.

From these observations, it is clear that a common simulation environment providing shared standards and methodologies will greatly improve technical communication capabilities between partners as well as the many different disciplines involved in gas turbine engine research and development programs. For these reasons, within the integrated project VIVACE (**V**alue **I**mprovement through a **V**irtual **A**eronautical **C**ollaborative **E**nterprise) [2], a consortium of European universities, research institutes and corporate companies is developing PROOSIS[1] (**PR**opulsion **O**bject **O**riented **SI**mulation **S**oftware), a flexible and extensible object-oriented simulation environment. US engine manufacturers have already developed such a tool (NPSS, [3, 4]) but this is generally not available to the European industry. The main aim of PROOSIS is to perform all kinds of engine simulations as well as generic system simulation (e.g. control, thermal, hydraulic, etc.). It features an advanced graphical user interface allowing for modular model building using either the standard or any custom library of engine components. It is capable of both steady and transient simulations as well as customer deck generation. Different calculation types (design, off-design, test analysis, optimisation, etc) can be performed. The tool is also required to provide advanced capabilities such as multi-disciplinary, multi-fidelity and distributed simulations. These capabilities are not yet commonly available in performance departments of European gas turbine manufacturers.

The integration of such capabilities in PROOSIS, namely component 'zooming' and distributed simulations, is the subject of this paper. Different zooming implementations are described using a 1-D compressor stage stacking analysis code as the higher fidelity representation of compressor performance (compared to a conventional map). Distributed simulations are demonstrated by executing remotely the same code. Hence, for understanding the framework in which the advanced capabilities are implemented, a general overview of PROOSIS and a brief description of the 1-D code are firstly presented.

PROOSIS OVERVIEW

PROOSIS is a standalone, multi-platform, object-oriented simulation environment. It shares the philosophy of the commercial simulation tool described in [5, 6]. It uses a high-level object-oriented language (EL), for modelling engine systems. EL offers all the benefits of this type of programming: encapsulation, inheritance, aggregation, abstraction, polymorphism, etc. The most important concept in EL is the Component (equivalent to a class in C++); it contains a mathematical description of the corresponding real-world engine component. Components communicate with each other through their ports. Ports define the set of variables to be interchanged between connected components. Components and ports are stored in a library.

PROOSIS comes with a Standard Library of engine components and ports. Their modelling is based primarily on the work of Walsh and Fletcher [7] and respects the international standards [8-11] with regards to nomenclature, interface, object oriented environment and standard performance methodology. However, the use of the standard library is not compulsory and the user can build custom components and/or libraries. Components from different libraries can be combined in constructing a model as long as connected components share the same communication interface (e.g. ports). Figure 1 shows the tool's graphical user interface for all the different phases of model building and running.

[1] In Greek, PROOSIS (ΠΡΟΩΣΙΣ) means propulsion.

A model, be it a single component, a sub-assembly or a complete engine, can be constructed graphically by 'drag-and-drop' icons from one or more library palettes to the schematic window.

The model's mathematical description (called a Partition) is set with the help of wizards. Built-in mathematical algorithms process the equations symbolically, resolve high index problems, solve algebraic loops, suggest boundary conditions and finally sort the equations for efficient calculation. The simulation tool allows for non-causal modelling; the order and form of equations does not matter.

Different simulation cases (Experiments) can be performed for a Partition. Within the Experiment window (Fig. 1) and using the object-oriented language EL, one can initialise variables, set the values of boundary condition variables and component data, run single and multiple steady state simulations, integrate the model over time (transient operation) and generate reports (write results to file or screen). With the help of internal (EL) or external (C, C++, FORTRAN) functions it is possible to create very complicated simulations (e.g. multi-point design, optimisation, test analysis, etc.). Experiments can run either in batch mode or graphically.

Currently, many of the partners involved in the development of PROOSIS have successfully 'translated' engine models from their own simulation tools to PROOSIS in order to suggest improvements at both kernel and interface levels, report any weaknesses or bugs and get used to this new simulation philosophy. The official release of PROOSIS is scheduled for 2008.

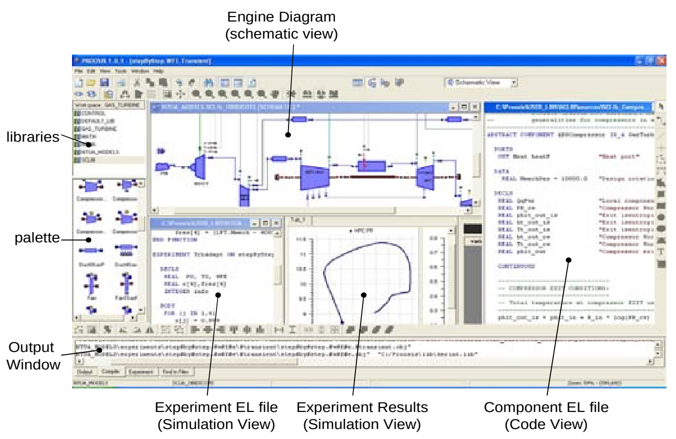


Figure 1: PROOSIS interface

Compressor Stage Stacking

The overall performance of a multi-stage axial compressor depends on the performance of its constituent stages. The performance of a single stage can be represented by its non-dimensional characteristic curves Φ-Ψ, Φ-η where Φ is flow coefficient, Ψ is pressure coefficient and η is efficiency. At any given operating point, knowledge of these curves along with flow area, mean radius and stage exit angle allows the calculation of individual stage exit properties (static and total pressure and temperature, velocity, etc). By 'stacking' the stages together the overall compressor performance is evaluated.

The particular model used here is a modified version of the one described in [12]. The code is written in FORTRAN and compiled as a static library (LIB). There is a main subroutine that accepts as input the stage geometry and characteristics, the compressor inlet conditions and information regarding bleeds. The output consists of compressor overall pressure ratio, mass flow and isentropic efficiency, for a user specified number of points along the speed line (between stall and choke mass flow values), corresponding to the specified compressor rotational speed. Then in PROOSIS this subroutine is declared as an external FORTRAN function with the corresponding arguments. An alternative approach is to create a C++ wrapper for the FORTRAN subroutine and then declare it in PROOSIS as a public method of an external class with the same arguments. Although this adds an extra step, the final outcome is in-line with object-oriented modelling; one can declare an object of this class and use its methods. The latter approach is used for demonstrating distributed simulations where a slightly different version of the stage stacking code (in the form of a shared DLL) is used in which, for the same inputs, the output is simply the compressor overall pressure ratio and isentropic efficiency, for the specified compressor inlet mass flow and rotational speed. The former approach is used for studying different zooming methods as described in the following.

COMPONENT ZOOMING

In the context of whole engine performance simulation, component zooming or variable complexity or multi-fidelity analysis refers to the execution of one (or more) higher order analysis code and the integration of its results back into the 0-D engine cycle. In this way, simulation accuracy increases as it is based on more detailed, physics based component characteristics compared to traditional component maps. Additionally, component design teams can rapidly evaluate the effects of their designs on the whole engine performance as well as the other components. The clear and multiple benefits of component zooming have been demonstrated in a number of recent publications [13-17], where different approaches to zooming have been implemented. One approach, referred to as 'de-coupled' zooming in [13], is to execute the higher fidelity code for a number of different operating conditions in order to produce a map (or 'mini-map' as in [14]) that can be subsequently used directly in the 0-D model. Another approach uses an iterative process between the high-fidelity component representation and the 0-D engine cycle model until an engine operating point is established [15]. This 'semi-coupled' approach uses scaling factors in the 0-D component to communicate the high-fidelity results back to the engine simulation [16]. Finally, [17] presents a 'fully-coupled' approach where an engine model can be constructed from mixed-fidelity components. Depending on user's needs, available resources and modelling philosophy, any of the three methods may be the more appropriate to implement, for a given simulation case. Hence the implementation of the three approaches in PROOSIS is described next.

The Engine Model

For demonstrating the different zooming implementations, a model of the single shaft version of an industrial gas turbine engine [18] has been created in PROOSIS using standard library components. Figure 2 shows the engine's schematic diagram.

For the 15-stage axial compressor, a smoothed BETA version of the map presented in [18] is used (available in GasTurb10 Map collection [19]). The turbine's off-design performance is acquired by scaling the default PROOSIS turbine map. The model is validated against a proven simulation model created using an in-house simulation tool [20].

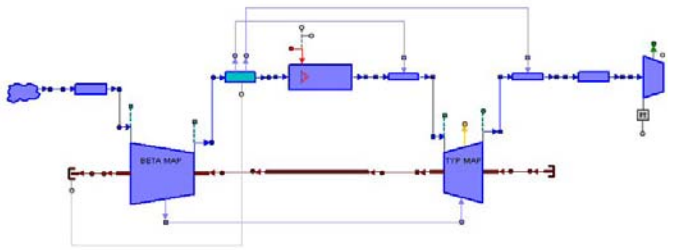


Figure 2: Engine schematic in PROOSIS

The 'de-coupled' Approach

A 1-D compressor component is developed in EL that follows the architecture of the standard library in PROOSIS. The compressor hierarchy adopted is shown in Fig. 3. Hence, abstract 1-D compressor Component has inherited ports to communicate with other components and can be used like any other component in a library to be arranged in a schematic (engine) diagram. Within the component there is a call to the external FORTRAN stage-stacking function. The function arguments consist of the compressor operating conditions, received through the component's inlet ports, and the stages' geometry and characteristics, specified by the user. For the engine model used herein, the stage geometry and characteristics derived in [21] are used. They employed the adaptive stage-stacking technique introduced by [12] to reproduce the map given in [18].

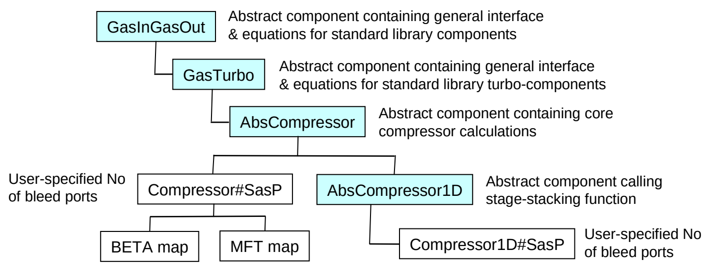

GasInGasOut
Abstract component containing general interface & equations for standard library components
GasTurbo
Abstract component containing general interface & equations for standard library turbo-components
AbsCompressor
Abstract component containing core compressor calculations
User-specified No of bleed ports
Compressor#SasP
AbsCompressor1D
Abstract component calling stage-stacking function
BETA map
MFT map
Compressor1D#SasP
User-specified No of bleed ports

Figure 3: Compressor 'Inheritance Tree'

For creating a 1-D map, an instance of the 1-D compressor is firstly created. This is simply achieved by placing the icon of the 1-D compressor in a schematic window and compiling it. A Partition (a PROOSIS equivalent to a configuration) is then specified with boundary conditions the component's inlet port variables (total pressure & temperature, water-to-air ratio, rotational speed and either mass flow or pressure ratio). Finally, in an Experiment (a PROOSIS equivalent to a simulation case), a multi-point steady state calculation is defined where the rotational speed is varied, at ISO conditions. For each rotational speed, the stage-stacking function provides the compressor overall pressure ratio, mass flow and isentropic efficiency for a user specified number of points between the stall and choke mass flow rate values. The results are written to a file in the form of a BETA map so that it can directly replace the 0-D one in the engine model. Thus the only change required in the original engine model is the name of the new compressor map file. The two maps are shown in Fig. 4 as pressure ratio and isentropic efficiency versus mass flow rate for relative corrected speed values of 0.9, 0.95 and 1.0 (for lower than 0.9 values the geometry of the first 5 stages changes to ensure adequate surge margin).

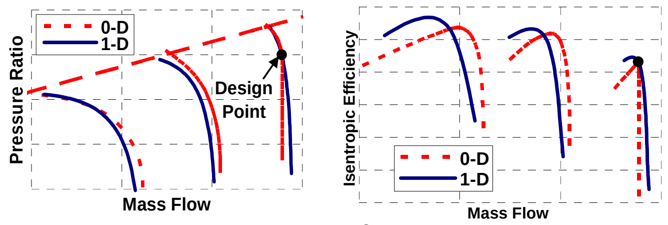


Figure 4: Comparison of 0-D and 1-D Maps

Figure 5 shows the percentage difference in heat rate at different loads at design speed when the 1-D map replaces the 0-D one in the engine model.

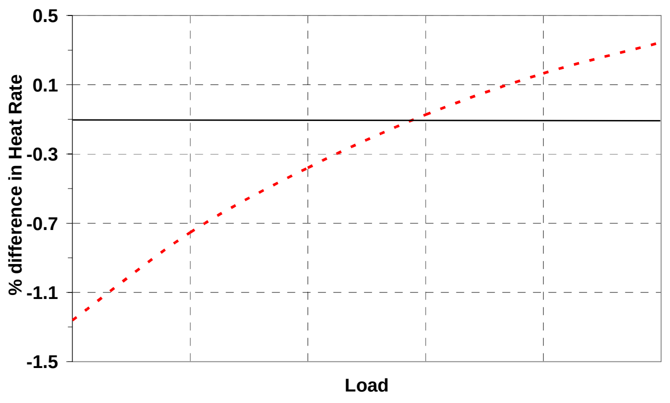


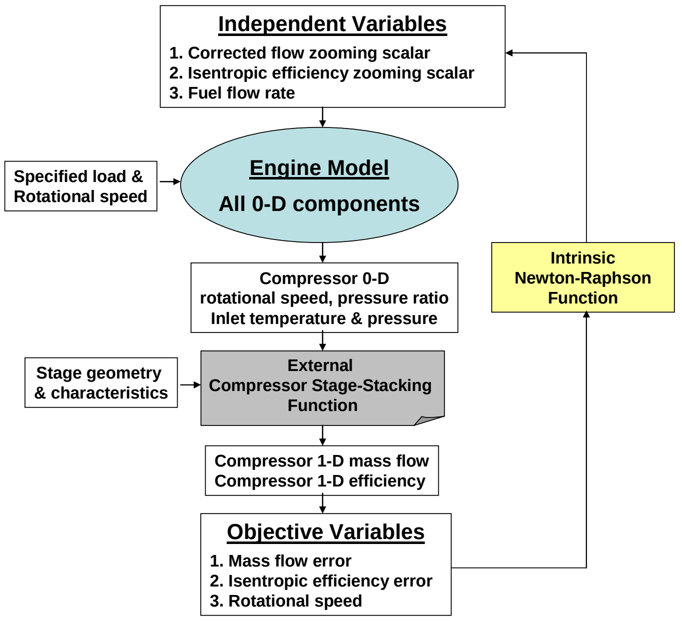
Figure 5: Effect of Compressor Zooming on Heat rate vs Load Characteristic

In the de-coupled approach, apart from changing the compressor map file (a component attribute), the original model is used with the same schematic diagram, partition and experiment. From within a single experiment, the user can run consecutive simulations specifying a different compressor (or any other component) map each time and comparing the results. In fact, the final user may not even be aware that a higher fidelity analysis has been performed beforehand.

Finally, it should be noted that a dedicated component just calling the stage-stacking function and accepting as input the stage geometry and characteristics could also have been used instead of the 1-D compressor, for creating the map. Even a void component can be used with all the information entered at experiment level as explained in the next section. The 1-D compressor is developed for use in the fully-coupled zooming approach and so it makes sense to use it with this method too, as it is already available.

The 'semi-coupled' Approach

In this approach the original 0-D model is also used as is (same schematic and partition), but the zooming is performed at experiment level. The 1-D compressor component is not employed this time. Instead the stage stacking function is used directly in the experiment in a process shown schematically in Fig. 6. Zooming scalars on corrected mass flow rate and isentropic efficiency are incorporated in the 0-D compressor component. For a specified load and rotational speed (single-shaft industrial engine for electricity generation), the 0-D model converges to the required fuel flow rate value. The compressor inlet total pressure and temperature, pressure ratio and rotational speed are passed to the stage stacking function that evaluates the 1-D mass flow and isentropic efficiency for these conditions and based on the specified stage geometry and characteristics. Error terms are formed by comparing these values with the corresponding 0-D ones. The solver adjusts the component zooming scalars until these error terms are within a user specified tolerance. Figure 7 shows the variation of the scalars with external load for a relative corrected speed of 0.95.


Independent Variables
1. Corrected flow zooming scalar
2. Isentropic efficiency zooming scalar
3. Fuel flow rate
Specified load & Rotational speed
Engine Model
All 0-D components
Compressor 0-D rotational speed, pressure ratio Inlet temperature & pressure
Intrinsic Newton-Raphson Function
Stage geometry & characteristics
External Compressor Stage-Stacking Function
Compressor 1-D mass flow Compressor 1-D efficiency
Objective Variables
1. Mass flow error
2. Isentropic efficiency error
3. Rotational speed
Figure 6: Zooming Scheme Implemented in Experiment

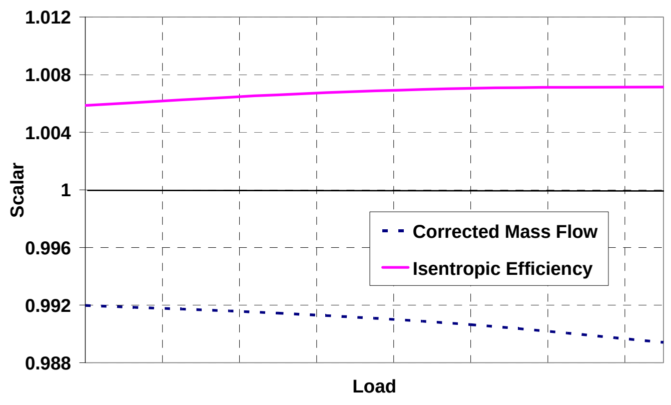


Figure 7: Variation of Zooming Scalars with Load

Using this approach, a multi-point experiment can also be created for converting a complete 0-D map to a higher fidelity one through the calculated scalars. Hence obtaining in one step (creating and using a 1-D map) the same result as in the two-step de-coupled approach (first create 1-D map and then use it).

The 'fully-coupled' Approach

In the PROOSIS palette of the standard library components there are icons for both 0-D compressor and its 1-D counterpart. The fully-coupled approach to zooming is realised by simply deleting from the engine schematic the 0-D compressor icon and inserting the 1-D one described earlier. After connecting its ports (same way as in Fig. 2) the schematic is compiled. It is now possible to create a Partition. In the all 0-D component version of the model there is a single boundary condition (the fuel flow rate), three algebraic variables (the compressor map auxiliary coordinate BETA, the turbine map axial coordinate and the engine inlet mass flow rate) and one dynamic variable (the shaft rotational speed). When the 1-D compressor is used there is no BETA variable and the compressor pressure ratio is 'flagged' as an independent parameter. The solver then finds the value of pressure ratio for which the mass flow calculated from the stage-stacking function is equal to the compressor inlet flow. The rest of the partition is identical to the 0-D version. Using the fully-coupled approach, a simulation is performed for the design case, where there is no difference between the 0-D and 1-D compressor characteristics, but with a 1.5% inter-stage bleed for turbine cooling, taken from the 10th compressor stage. The same simulation case was also run for the 0-D compressor model (1.5% bleed flow, 10/15 work fraction). Table 1 shows the percentage difference in compressor performance and fuel flow rate between the two cases.

Table 1: The Effect of Zooming

PARAMETER	% DIFFERENCE
Fuel Flow Rate	0.289
Compressor Inlet Flow	0.111
Compressor Delivery Temperature	0.438
Compressor Pressure Ratio	0.211
Compressor Polytropic Efficiency	-0.238
Compressor Power	0.583

This approach is more intuitive and direct as it does not require the generation of a compressor map like in the de-coupled approach or the creation of a complex experiment like in the semi-coupled approach. On the other hand it is heavier on computer resources as the high fidelity code is directly 'embedded' in the cycle calculations. For PROOSIS and for the stage-stacking code used in the calculations, this translates to an increase in simulation time by an order of magnitude, albeit a few seconds rather than fractions of a second as it is for example the case in the de-coupled approach (Intel Celeron 1.6 GHz, 512MB RAM, Windows XP based PC).

DISTRIBUTED SIMULATIONS

Distributed simulation refers to technologies "that enable a simulation program to execute on a computing system containing multiple processors, such as personal computers, interconnected by a communication network" [22, 23]. Distributed simulation technology facilitates integrated simulation on multiple sites, enabling engine subsystem simulations to interact with each other in a controlled way. Not only does it enable possibly geographically dispersed engineers to efficiently collaborate in modelling, it also obsoletes the need to reinstall subsystem models at a single site, and hence suppresses costs for extra hardware, software, verification, maintenance. Subsystem models may be developed, verified, maintained, and deployed for use in the integrated simulation at an engineer's own site. There is no need to recreate the subsystem's context, including resources such as databases and sometimes expensive software tools accessed from the model, at a different site. An engineer may modify or replace a submodel, as long as its interface to the integrated simulation remains unaffected. Distributed simulation techniques also support the protection of model ownership. It enables a party to allow other parties to use a model, usually in a restricted form, in a particular setting and with a predefined subset of model parameters, without granting direct access to the underlying data and software implementing the model. In addition, distributed simulation may lead to reduction of simulation time, through distribution of the computational load over several computers, e.g., using Grid technology [24]. Distributed simulation also supports scale up of the model; a model may grow with respect to size and complexity irrespective of the capability of the underlying computing infrastructure. Moreover, distributed simulation gives rise to reuse of submodels in different integrated simulations.

Implementing Distributed Simulations

Distributed applications are supported through middleware (software that connects applications), usually in an operating-system independent way. Today, different technologies are available to accomplish a distributed application, depending on the required situation and operational context. Widely known technologies are:

- **CORBA**. The Common Object Request Broker Architecture (CORBA) created and controlled by the Object Management Group, a consortium aiming at setting standards for distributed object-oriented systems. CORBA provides applications with platform and location transparency for sharing objects across a network of computers. The objects are well-defined in terms of attributes and methods, using an interface definition language (IDL). CORBA defines application programming interfaces, communication protocols, and object/service information models to support the interoperation of heterogeneous applications written in various languages on various platforms. Through the years, several commercial as well as free implementations became available, called object request brokers (ORBs). For example, JacORB is a freely available ORB for Java. CORBA defined the Internet Inter-Orb Protocol (IIOP) to enable the several ORBs to interoperate, hence allowing different applications realised using different ORBs to interoperate. IIOP is an implementation of the General Inter-ORB Protocol (GIOP) for the TCP/IP protocol which is the basis for Internet. CORBA has often been used in aerospace industry to accomplish distributed applications, e.g.

NASA's Numerical Propulsion System Simulation (NPSS) [25, 26]. However, since achieving a proper level of security is difficult in CORBA, CORBA seems mainly used for distributed applications within local company networks. Despite its use in several industrial distributed applications, CORBA is loosing its popularity. Important reasons are its complexity, its slow and weak response to the rapidly growing Web developments and demands, its high run-time costs (for commercial ORBs), the difficulty in achieving the appropriate level of security, and its lack of a proper versioning enabling commercial software based on CORBA to ensure backwards compatibility [27]. Also, Microsoft never supported CORBA.

- **DCOM**. The Distributed Component Object Model (DCOM) is Microsoft's technology enabling distributing software components across a computer network to communicate with each other. It was built on Microsoft's Component Object Model (COM) introduced in 1993, enabling programming language-independent interprocess communication and dynamic object creation. DCOM was a major competitor of CORBA. Since DCOM is very powerful and is considered to provide "too" much functionality, it raises security problems. Over the years, hackers discovered their way around in DCOM, and exploited and abused the plethora of possibilities to gain illegal access to systems. Like CORBA, DCOM was not able to catch up with the Web developments, thereby failing to provide a secure distributed environment over Internet firewalls and containing unknown and insecure systems. DCOM has been deprecated in favour of Microsoft's .NET framework.
- **Java RMI**. The Java Remote Method Invocation (RMI) is a Java application programming interface for invoking an object's methods. Java RMI is Java specific. Nowadays, Java RMI is also considered to be an intermediate solution, being obscured by Web Services. The Object Management Group established the standard RMI-IIOP (RMI over IIOP), to simplify the development of CORBA applications while preserving the popular RMI style of programming.
- **XML** and **SOAP**. Developments and further standardisation by the World-Wide Web Consortium (W3C, [28]) resulted in the definition of the Extensible Markup Language (XML) for describing different kinds of data. XML facilitates the exchange and sharing of data across different, heterogeneous, usually Internet-connected systems. In addition, the Simple Object Access Protocol (SOAP) – originating from Microsoft as an object access protocol – was adopted and further maintained by the W3C as protocol for exchanging XML-based messages, using the HyperText Transfer Protocol (HTTP), over a computer network. XML and SOAP are known to be much slower than binary protocols used in, e.g., CORBA and RMI. However, their possibilities to construct secure distributed and web-based applications in wide-area set ups possibly involving Internet firewalls, and the present support for deploying the two standards, make XML and SOAP far more popular.
- **Web Services**. Today, World Wide Web technology is used more often for communication between applications and, consequently, for accomplishing distributed computing and simulations. Based on lessons from the past, and developments and demands with respect to the Web, the W3C established Web Services. This technology provides software applications with programmatic interfaces, enabling interoperability between different applications running on different platforms and in different frameworks. Web Services facilitate the combination of software and services from different companies to form integrated services. Web Services use open standards and protocols, such as SOAP and XML. In the Web Service approach, the interface (Application Programming Interface, API) of a service, including details of its bindings to specific protocols, is well-defined in terms of a Web Service Definition Language (WSDL) description, in XML-format. The WSDL description enables clients to interact with the service. Web Services also define service "broker" technology. Service providers may register services with a service broker. Service requesters may use the service broker to find a particular service and retrieve the service's WSDL description in order to subsequently invoke the operations implemented by the service. Web Services are commonly used to implement a distributed system in the style of a Service Oriented Architecture (SOA) [29]. This software architecture defines the use of individual, loosely-coupled services on a computer network to support the implementation of an integrated software system comprising the services. The computational resources from a network, including the applications, are available as independent services that are accessible without knowledge of the services' actual implementation and underlying platforms. Support for the application of Web Services is emerging. For example, the Java development environment NetBeans enables software developers to have a first Web Service implementation for their application operational quickly. Also, Microsoft's .NET supports Web Services. Although Web Services seem promising, critics state that Web Services still are too complex for the software developers, and that the performance is poor compared to RMI, CORBA, and DCOM, resulting from the use of the text-based XML. The complexity will certainly be reduced in the next years, with the advance of standards and tools based on Web Services technology. The performance issue is addressed in the on-going developments in the areas of XML and W3C's Message Transmission Optimization Mechanism (MTOM). Web Services technology – supported by initiatives to further reduce its alleged complexity – is expected to become a standard for enabling organizations to share data and services with customers and business partners. For example, eBay, Google, and Amazon provide a Web Services based interface (or actually a client library to facilitate the Web Services access) to their services.

In order to establish distributed simulations to run across company boundaries while achieving a proper level of security, and to use standard and platform independent technology, the choice was made to use Web Services technology for carrying out distributed simulations in Proosis. Web Services technology is also successfully used in VIVACE for development of a workflow management framework across company borders, over the Internet. It is foreseen that entire workflows will be included in distributed simulations as well.

Prototype Development

A prototype was developed to demonstrate the feasibility of calling the compressor stage stacking function remotely, using Web Services technology. The actual stacking function is developed by, and is proprietary code of, NTUA. The function is written in Fortran and available to NTUA users as a shared library, or Dynamically Linked Library (DLL) on Windows. The function is made available for use by simulations in Proosis by integrating the DLL into Proosis as a so-called customer library (also called component or class), which is the mechanism to extend PROOSIS with customer-written (C++) software in engine simulations. The library is, and may be, only locally available at NTUA. However, although the DLL may not be installed elsewhere, the function may be used in simulations running outside NTUA and accessed over the Internet. A mechanism to accomplish this remote usage is provided through the notion of PROOSIS Web Component, enabling libraries to be shared among PROOSIS simulations without the need to distribute the code of the libraries. A prototype PROOSIS Web Component is created for the stacking function. It is implemented as a PROOSIS customer library that can act as a client for a Web Service operation. For the server side, the Web Service operation is developed that gives access to the actual stacking function. It uses a plug-in mechanism which makes it easy to replace the shared library with another shared library (with the same external interface) containing a different implementation. The shared library can even be replaced while the Web Service is active. A future target is to develop a specific, but reusable external PROOSIS library that can easily be extended to remotely call any function via Web Services technology.

Figure 8 gives a schematic view of the test environment that is used to test the created prototype. The PROOSIS simulation is run mainly at NLR. However, the shared library containing the stacking function is replaced by the PROOSIS Web Component (PROOSIS plug-in *WebComponent.lib*), which accesses the actual stacking function via a Web Service operation. The server at NTUA provides the stacking function via a Web Service operation that redirects the call from the PROOSIS Web Component to the original shared library.

To accomplish the set up as depicted in Fig. 8, we must take into account that PROOSIS only supports customer libraries developed in C++ and FORTRAN. However, the Web Services software is written in Java, since nowadays Web Services implementations (server engines, tools, code generators) are mainly supported for the Java platform. Support for C++ (e.g., Apache Axis) is yet limited. But the language incompatibility is not a problem since Java provides support for calling C++ functions from Java, and vice versa, through the Java Native Interface (JNI). Also, the stacking function is implemented as a FORTRAN function, which again is not a problem since it can easily be used from C++ code. Potentially this overhead may cause calculation times to be larger when comparing these timings with local instantiations of the stage stacking function.

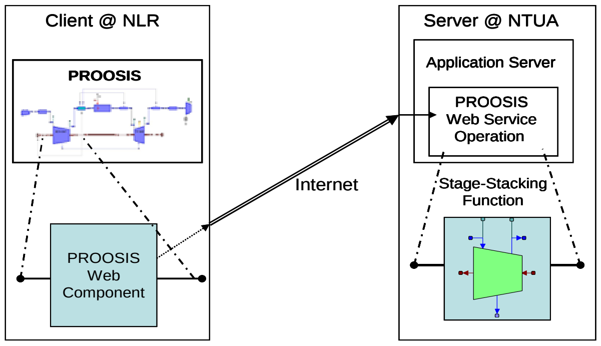


Figure 8: Compressor Zooming via Remote Web Service Invocation between NLR and NTUA

.

Application of Web Services technology, thereby taking the programming language constraints into account, resulted in an implementation of the prototype existing of 5 layers:

1. PROOSIS custom library containing code that interacts with Java through JNI (C++). This layer serves as plug-in for the remote use of the stacking function in the PROOSIS simulation.
2. Web Service client implementation (Java). This layer comprises the Java modules that enable layer 1 to call the stacking function remotely as a Web Service operation.
3. PROOSIS Web Service Component implementation that interacts with C++ through JNI (Java). This layer provides the Web Service operation for calling the stacking function. It receives a request for the stacking function from the Internet, and passes the call to layer 4.
4. Intermediate code that calls the FORTRAN implementation (C++). This layer acts as the glue between the Java code on the one hand, and FORTRAN code compiled into a DLL on the other hand.
5. Actual implementation of stacking function (StgStk) in FORTRAN. This function has 20 parameters in total, including arrays and 3 output parameters. For the prototype, its interface is used as the reference interface through all layers; no generic useable interface is used.

A schematic overview of the layered structure is displayed in Fig. 9, including the programming language in which each layer is implemented. The numbers next to the layer represent the 5 layers above. Note that the JNI layer is not numbered as a separate layer, since JNI is not part of the implementation; it is an API that is by default provided with the standard Java distribution (J2SE).

The tools and development environments used for the implementation of the prototype are listed in the Appendix.

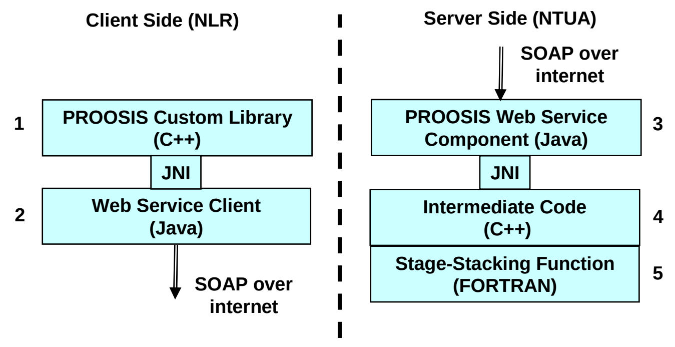
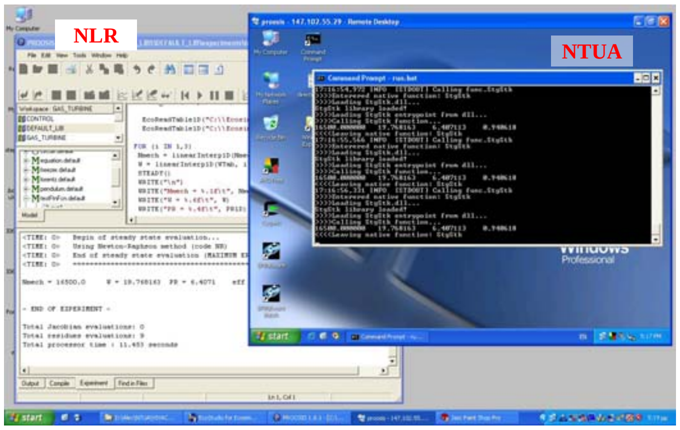

Client Side (NLR)
Server Side (NTUA)
SOAP over internet
1
PROOSIS Custom Library (C++)
JNI
2
Web Service Client (Java)
SOAP over internet
PROOSIS Web Service Component (Java)
3
JNI
Intermediate Code (C++)
4
Stage-Stacking Function (FORTRAN)
5

Figure 9: Layered structure of prototype (5 layers)

The prototype was used successfully for distributed engine simulations, with the PROOSIS engine simulation running at NLR premises (the Netherlands) and the stacking function being installed and used at NTUA (Greece). A 'live' public demonstration of a distributed simulation using the prototype was also performed during VIVACE Forum-2 [2]. Figure 10 shows the PROOSIS simulation on the left (client) and the execution of the stage stacking function at the remote location on the right (the server). A typical calculation using the de-coupled zooming approach for obtaining a set of operating points (69 points, representing the upper part of a compressor map) takes approximately 40s using a high speed internet connection compared to 4s when it is ran in a similar but local experiment (without the Web Services/Java layer in Fig. 9). This prototype has demonstrated the feasibility of using Web Services technology in distributed simulations using PROOSIS, enabling companies to collaborate in the engineering activities.



```
EcoReadTable1D("C:\\Ecosi
EcoReadTable1D("C:\\Ecosi

FOR (i IN 1,3)
 Nmech = linearInterp1D(Nmec
 W = linearInterp1D(WTab, i
 STEADY()
 WRITE("\n")
 WRITE("Nmech = %.1f\t", Nm
 WRITE("W = %.6f\t", W)
 WRITE("PR = %.4f\t", PR1I)
```

```
<TIME: 0>   Begin of steady state evaluation...
<TIME: 0>   Using Newton-Raphson method (code NR)
<TIME: 0>   End of steady state evaluation (MAXIMUM E
<TIME: 0>   *******************************************

Nmech = 16500.0     W = 19.768163  PR = 6.4071    eff

- END OF EXPERIMENT -

Total Jacobian evaluations: 0
Total residues evaluations: 9
Total processor time : 11.453 seconds
```

```
17:16:54,972 INFO  [STDOUT] Calling func.StgStk
>>>>Entered native function: StgStk
>>>>Loading StgStk.dll...
StgStk library loaded!
>>>>Loading StgStk entrypoint from dll...
>>>>Calling StgStk function...
16500.000000   19.768163    6.407113    0.940618
<<<<Leaving native function: StgStk
17:16:55,566 INFO  [STDOUT] Calling func.StgStk
>>>>Entered native function: StgStk
>>>>Loading StgStk.dll...
StgStk library loaded!
>>>>Loading StgStk entrypoint from dll...
>>>>Calling StgStk function...
16500.000000   19.768163    6.407113    0.940618
<<<<Leaving native function: StgStk
17:16:56,331 INFO  [STDOUT] Calling func.StgStk
>>>>Entered native function: StgStk
>>>>Loading StgStk.dll...
StgStk library loaded!
>>>>Loading StgStk entrypoint from dll...
>>>>Calling StgStk function...
16500.000000   19.768163    6.407113    0.940618
<<<<Leaving native function: StgStk
```

Figure 10: Distributed Simulation in PROOSIS

Future Developments

The prototype is still under development and some recommendations for future work are presented in this section. For each recommendation, the advantages and disadvantages will be given if applicable.

Design of a more generic (re-usable) interface

In the current prototype, one particular function (i.e., the stacking function) was put available as a Web Service operation. Currently, the Web Service contains one component, with one operation named after the function. This Web Service operation expects the same input arguments as the original function. If this Web Service component needs to provide more operations, the existing framework has to be extended for each operation to be added. This means that a function/method/operation mapping has to be created in all 5 layers described. Hence, the software developer has to know the details about each layer and needs to have the complete software environment to rebuild the framework when a new function is added.

It is worthwhile to design an interface that can be reused by multiple function implementations. This means that the Web Service component does not contain specific operations anymore, but a single generic operation. The arguments of this operation then need to be very generic, so that it can easily be reused. The implementation near the front-end (layer 4) and back-end (layer 2) then needs to be modified to adapt the generic interface to the specific interface (of, e.g., the stacking function).

The advantage of this method is that not all layers need to be modified for each new operation that is added. The software developer only needs to write the code to convert the generic interface to the specific interface, and therefore he does not need to have the complete development environment.

A disadvantage may be that the interface adaptations (from generic to specific and back) introduce additional overhead and delays in the communication.

Reduction of overhead caused by conversions and data transfers

The introduction of Java between the PROOSIS custom library and the actual implementation causes overhead in development time and run-time, due to data conversions and data transfers between platforms when invoking a remote function via a Web Service. Run-time overhead appears with each remote call of the function, which usually requires transfer of data within the machine (i.e. memory copy) or over the network in between machines. Overhead in development time occurs because all the data transfers have to be coded manually for each layer. Tools such JAX-RPC and JACE provide mechanisms to simplify these data transfers, but they do not automate the data transfers themselves. They lack the knowledge to identify the parameter mappings between the layers. The overhead, however, might be reduced by using a pure C++ development environment, such as the C++ implementation of the Axis Web Service engine.

The advantage is less overhead in data conversions between Java and C++.

The disadvantage is that C++ support for Web Services (using Axis) is still in the beta phase, and is not guaranteed to work stable. Also, the support for Web Services in the Java platform is more common nowadays. Another reason to choose for the Java platform is being able to integrate with the "Collaboration HUB" that is being developed in another work package in the VIVACE project. The communication layers of the HUB are based on Java. In supplement to the design of a more generic interface, this workflow management framework can also be re-used for the purpose of calling a remote function instead of a remote workflow.

Reduction of overhead in DLL loading and unloading

The current prototype calculations through Web Services cause overhead for loading the dynamically called calculation procedure (loading DLLs containing the functions). A more advanced implementation should load the DLLs once. When all operating points have been calculated a finalising procedure should take care of disposing loaded DLLs and functions.

Multi-user and security

As in practice several engineers from different parties are involved in distributed simulations, the issues of multi-user access and security must be addressed. The current prototype is not designed to be used by multiple users at the same time. Its behaviour is unpredictable when multiple users would invoke the Web Service operation at the same time.

The aspect of security is not addressed in the current prototype, mainly because it was not yet required. Presently, the input and output variables are transferred unencrypted over the internet. Also, anybody who knows the URL of the Web Service can invoke its operation. There is no login mechanism implemented. Security in Web Services can be achieved by using Web Services Security specification (WS-Security). WS-Security includes SOAP Message Security and Username Token Profile. SOAP Message Security guarantees encryption of the transferred data. Username Token Profile guarantees the authentication of the user. The WS-Security specification is defined by OASIS [30]. WS-Security is also supported by the JBoss application server; see Appendix. Various security aspects are also studied in VIVACE project.

SUMMARY AND CONCLUSION

An object-oriented environment for gas turbine engine performance simulations is being developed to facilitate communication between European partners in cooperative engine projects. It is required to be user-friendly for creating, running, managing and sharing engine models using either the standard or custom libraries of engine components. It must also provide capabilities that are not yet commonly available in European performance departments such as multi-disciplinary, multi-fidelity and distributed simulations. The feasibility of performing such simulations with this tool is demonstrated in this paper.

Using a 1-D compressor stage stacking code as an example, different implementations for integrating high fidelity component analysis in overall engine simulations are presented. The tool's flexible and extensible architecture gives the user the freedom to select the most suitable approach for a particular simulation case. The model of an industrial gas turbine engine is used to exemplify the benefits of this type of analysis.

The stage stacking code is also used to demonstrate distributed simulations. A prototype of a Web Component has been created and successfully tested that remotely invokes the code from an engine simulation, via the internet, using Web Services technology. The reasons for selecting this technology, the steps taken and the tools used in realising the prototype are presented in detail while future improvements to it are discussed.

These demonstrations prove that the tool's architecture is adaptable enough to integrate different modelling methods and its potential to fulfil its role as a shared simulation environment in Europe.

ACKNOWLEDGEMENTS

The work described in this paper has been carried out as part of the Integrated Project VIVACE (AIP3-CT-2003-502917) and financial support of the European Union Commission is gratefully acknowledged. The authors would like also to express their thanks to the partners involved in Work Package 2.4 of VIVACE for development of the PROOSIS simulation. The contribution of Bert Schultheiss, Jacco van Weert, Michiel Jansen, N. Aretakis, I. Roumeliotis and Ph. Kamboukos in parts of the work is greatly appreciated.

REFERENCES

1. NATO Research and Technology Organisation, 2006, "Performance Prediction and Simulation of Gas Turbine Engine Operation for Aircraft, Marine, Vehicular, and Power Generation", RTO-TR-AVT-036.
2. http://www.vivaceproject.com/
3. Lytle, J.K., 2001, "The Numerical Propulsion System Simulation: An Advanced Engineering Tool for Airbreathing Engines", ISABE 2001-1216.
4. Follen, G.J., 2002, "An Object-Oriented Extensible Architecture for Affordable Aerospace Propulsion Systems", RTO-MP-089.
5. Alexiou, A. and Mathioudakis, K., 2005, "Development of Gas Turbine Performance Models Using a Generic Simulation Tool", ASME Paper No. GT-2005-68678.
6. Alexiou, A. and Mathioudakis, K., 2006, "Gas Turbine Engine Performance Model Applications Using an Object-Oriented Simulation Tool", ASME Paper No. GT-2006-90339.
7. Walsh, P.P. and Fletcher, P., 2004, *Gas Turbine Performance*, 2nd Edition, Blackwell Science, Oxford.
8. SAE AS681-H, "Gas Turbine Engine Steady State and Transient Performance Presentation for Digital Computer Programs".
9. SAE AS755-C, "Aircraft Propulsion System Performance Station Designation and Nomenclature".
10. SAE ARP4868, "Application Programming Interface Requirements for the Presentation of Gas Turbine Engine Performance on Digital Computers".
11. SAE ARP5571, "Gas Turbine Engine Performance Presentation and Nomenclature for Digital Computers Using Object-Oriented Programming".
12. Mathioudakis, K. and Stamatis, A., 1994, "Compressor Fault Identification from Overall Performance Data Based on Adaptive Stage Stacking", , J. of Engineering for Gas Turbines and Power, **116**(1), pp. 156-164.
13. Melloni, L., Kotsiopoulos, P., Jackson, A., Pachidis, V. and Pilidis, P., 2006, "Military Engine Response to Compressor Inlet Stratified Pressure Distortion by an Integrated CFD Analysis", ASME Paper No. GT-2006-90805.
14. Turner, M.G., Reed, J. A., Ryder, R. and Veres, J.P., 2004, "Multi-Fidelity Simulation of a Turbofan Engine with Results Zoomed into Mini-Maps for a Zero-D Cycle Simulation", ASME Paper No. GT-2004-53956.
15. Pachidis, V., Pilidis, P., Talhouarn, F., Kalfas, A. and Templalexis, I., 2006, "A Fully Integrated Approach to Component Zooming using Computational Fluid

Dynamics", J. of Engineering for Gas Turbines and Power, **128**(3), pp. 579-584.
16. Follen, G. and auBuchon, M., 2000, "Numerical Zooming between a NPSS Engine System Simulation and a 1-Dimensional High Pressure Analysis Code", NASA/TM-2000-209913.
17. Hall, E.J., 2000, "Modular Multi-Fidelity Simulation Methodology for Multiple Spool Turbofan Engines", NASA High Performance Computing and Communications Computational Aerosciences Workshop, NASA Ames Research Centre.
18. Carchedi, F. and Wood, G.R., 1982, "Design and Development of a 12:1 Pressure Ratio Compressor for the Ruston 6-MW Gas Turbine", ASME Paper No. 82-GT-20.
19. Smooth C product information on http://www.gasturb.de.
20. Stamatis A., Mathioudakis K., Papailiou K., 1990, "Adaptive Simulation of Gas Turbine Performance", J. of Engineering for Gas Turbines and Power, **112**, pp. 168-175
21. Tsalavoutas, A., Stamatis, A. and Mathioudakis, K. and 1994, "Derivation of Compressor Stage Characteristics, for Accurate Overall Performance Map prediction ", ASME Paper No. 94-GT-372.
22. Fujimoto, R.M., 2000, *Parallel and Distributed Simulation Systems*, John Wiley and Sons, Inc., New York, USA.
23. Boer, C.A., 2005, "Distributed Simulation in Industry", PhD Thesis, Erasmus University, Rotterdam.
24. Foster, I. and Kesselman, C., 1998, "The Globus Project: A Status Report", Proceedings of the Heterogeneous Computing Workshop, IEEE Computer Society Press, pp.4-18.
25. Lopez, I., Follen, G.J., Gutierrez, R., Foster, I., Ginsburg, B., Larsson, O., Martin, S., Tuecke, S. and Woodford, D., 2000, "NPSS on NASA's Information Power Grid: Using CORBA and Globus to Coordinate Multidisciplinary", Aeroscience Applications. NASA/TM-2000-209956.
26. Zheng, D., Follen, G.J., Pavlik, W.R., Kim, C.M., Liu, X., Blaser, T.M. and Lopez, I., 2001, "Web-Based Distributed Simulation of Aeronautical Propulsion System", NASA/TM-2001-210818.
27. Henning, M., 2006, "The Rise and Fall of CORBA", in: Component Technologies, **4**(5)
28. http://www.w3.org
29. OASIS, 2006, "Reference Model for Service Oriented Architecture 1.0"
30. http://www.oasis-open.org/specs/index.php#wssv1.0

APPENDIX

The following tools and development environments were used for the implementation of the prototype:

– NetBeans 5.0, by default provided with the Java 5.0 software development kit (SDK), for Java development. It supports Web Services development. In particular, it facilitates interactive declaration of Web Service operations, which is the starting point for the generation of code implementing the Web Services component. NetBeans was used for layers 2 and 3.
– Microsoft Visual Studio C++ 6.0 for C++ development. It was used for layers 1 and 4.
– JBoss application server. An application server is a (software) server program in a computer network dedicated to running certain software applications. These applications (in our case, the stacking function) can be made available using Web Services. In this case, the application server plays the role of so-called Web Service End Point which enables deployment of Web Service operations. JBoss is widely used for deploying JAVA applications and is provided under the LGPL (open source) license and is therefore free to use. JBoss is used as Web Service End Point, which manages layer 3 at run time.
– JAX-RPC is applied to generate the Web Services code in Java, based on declarations of Web Service operations. It is supported by NetBeans and compatible with JBoss, and is still more robust than its successor JAX-WS. JAX-RPC is used in the implementation of layers 2 and 3, to generate a WSDL definition, code skeletons and stubs, both client and server side, enabling the client to invoke the stacking function as a Web Service operation, and enabling the server (including JBoss) to handle the Web Service operation and direct it to the actual stacking function.
– JACE is used as tool to bridge the gap between Java and C++. As mentioned before, the Java Native Interface (JNI) enables to connect C++ and Java code. However, JNI is a rather complex API and requires a lot of manual coding to transfer Java values to and from the stacking function (with about 20 arguments). JACE is a JNI-based collection of C++ and Java libraries that facilitate the integration of C++ and Java code. JACE is used to implement the call of Java code of layer 2 from layer 1 on the client side, and the call of C++ code of layer 4 from layer 3.
– No specific tools were required for calling FORTRAN code (layer 5) from the C++ code (layer 4). This only requires a C-style function declaration on the C++ side that makes it easy to call the stacking function as if it were a C function. Care must be taken that FORTRAN passes all function arguments by reference, and hence that all parameters in the C function must be pointers. Also, because the stacking function is available from a DLL on a Windows system, the stacking function must be used via a function pointer that needs to be assigned runtime and the standard calling convention of Win32 API functions must be used (*__stdcall* directive).