LCDL: An Extensible Framework for Wrapping Legacy Code

Ernst Juhnke, Dominik Seiler, Thilo Stadelmann, Tim Dörnemann, Bernd Freisleben
Department of Mathematics & Computer Science, University of Marburg
Hans-Meerwein-Str. 3, D-35032 Marburg, Germany
{ejuhnke,seiler,stadelmann,doernemt,freisleb}@informatik.uni-marburg.de

ABSTRACT

If legacy code has to be integrated into an application, it is often necessary to call this code available as source code written in a particular programming language or available in binary format for a particular computing platform from another programming language or from a remote machine. For this reason, wrapping code has to be developed for each source code library or binary code to be integrated. This paper presents an extensible framework that supports legacy code integration by modeling legacy code not only in a way that is programming (language) independent, but also by supporting different input and output types and bindings. This aim is achieved by the use of an integrated plug-in mechanism.

Keywords

Legacy code wrapping, Interface definition languages

1. INTRODUCTION

Integrating existing software available as source code written in a particular programming language or in binary format for a particular computing platform into other applications is a tedious task that has been considerably simplified by the introduction of the service-oriented architecture (SOA) paradigm. It offers a standardized way to call services and to exchange data between services possibly written in different programming languages and running on distributed computing platforms. In particular, a SOA based on web service technology offers standardized means that cope with security, data management, or stateful resources.

The juncture between existing code (which we will refer to as “legacy code” later on) and a web service based computing environment platform requires the development of wrapper code in the programming language of the target platform for the legacy code. This has to be repeated for all legacy code and all platforms.

In this paper, we present an approach that is capable of wrapping different types of legacy code on several levels of abstraction. The approach is based on an extensible framework providing a legacy code description language for realizing the wrapping functionality. Neither its set of data types nor the bindings are subject to any restrictions. Both can be extended using a plug-in mechanism, e.g., to support new transportation types such as Flex-SwA [9] or new binding types like RESTful services [5]. Based on a reusable model the integration of new abstraction levels only requires the declaration of that level. For this aim it is neither necessary to learn nor to use a different framework or language. A use case from the area of speech processing is presented to illustrate our approach.

The paper is organized as follows. In Section 2, a brief overview of existing frameworks and approaches is given. Section 3 describes the design of the presented approach. Section 4 provides some implementation details. The evaluation of the framework is carried out by a use case in Section 5. Section 6 concludes the paper and outlines areas for further research.

2. RELATED WORK

The integration of legacy code into other applications has a long tradition. Thus, several approaches supporting this task have been proposed in the literature. Typically, they map a special type of legacy code to a higher software component level, which can be a web service, a Grid service or something similar. Here is the difference to and the advantage of our approach outlined in this paper.

Glatard et al. [7] present a service wrapper that is able to integrate existing code into a service-based framework in an application independent manner. Their service exposition approach is based on a factory pattern to optimize services dynamically by grouping them according to different strategies. However, this interface of service changes can be considered harmful in a – contract-first – service-oriented architecture, because a third-party trying to execute the original service may be unable to find it due to the optimization.

The “Grid Execution Management for Legacy Code Architecture” (GEMLCA) introduced by Delaitre et al. [2] also makes use of a factory pattern. Legacy code deployed on the head node of a Globus Toolkit installation and executed on the computed nodes managed by this head node is wrapped. Delaitre et al. utilize a so-called “Legacy Code Interface Description”, an XML based format. For service invocation, a portal-based solution using Gridsphere is proposed. Since user interfaces and their generation depend on the user’s experience, there is no need to mix up service provisioning and user interfacing. The authors use the power of stateful services primarily to support a multi-user environment, but not for the life cycle management of native code.

For the integration of legacy code into Triana, a scientific workflow system, a two-stage approach has been presented by Huang et al. [10]. In the first step, a Java wrapper is created by the "Java-C Automatic Wrapper" (JACAW). In the second step, this Java class is mediated into the Triana workflow system using the "Mediation of Data and Legacy Code Interface".

Zou and Kontogiannis [15] describe a framework that wraps legacy code as CORBA objects. Furthermore, the authors design a concept for a service repository, including the registration and localization of services. In a way, their approach somehow resembles UDDI. Although the authors additionally introduce an "extensible service description language", extensibility is limited to environmental descriptions, e.g., operating systems etc.

3. DESIGN

The design of the presented framework called LCDL (Legacy Code Description Language) is based on the internal model shown in Figure 1. The model is taken and transformed into executable wrapper code. The model shows which information is needed to – declaratively – describe legacy code both in binary and source code library form. One of the main objectives of this design is to keep the model extensible. This includes new types of bindings as well as new input and output sources and types, but is not limited to those.

The main element is the `Service` element. It is the container for the `Operation`s and `Binding`s. The first ones contain an `Execute` element including its `Input` and `Output` parameters and a definable environment as well. The `Operations` describe which type this service should be exposed by. A more detailed description of the elements is given below.

A `Service` contains one or more `Operation`s that in turn represent methods of the legacy code. An `Operation` is identified by a symbolic name which other elements can refer to. An `Operation` possesses an `Execute` element that is the generalization of the `Library` and `Binary` type of legacy code. Both types need the path information of the legacy code. In some cases, it is necessary to specify auxiliary information for the `Binary` type. It might be insufficient to pass a given argument directly to the binary, since a parameter specification like *-preEmphasizeFactor=0.5* is needed. To map this, *-preEmphasizeFactor* is the prefix attribute and = is the infix attribute of the parameter element of the `Binary`.

`Input` arguments are determined by a type attribute, referring to an XML Schema Type [14] that itself is identified by its qualified Name (QName). To reflect the call-by-reference pattern, the mode attribute can be used to set an input parameter either as `in`, `in-out` or `out`. The binding has to take care of handling this correctly.

As an `Input`, the following elements are realized as a specialization. The `ElementInput` models a parameter that takes an arbitrary argument with a given type and passes it to the `Binding`. An `OptionInput` models an optional flag. This may have a value that can be set by a caller. The `StaticInput` models a parameter that is always set. Like the `OptionInput`, the `StaticInput` can have a settable value. `FileInput` models a file as an input parameter. The concrete treatment of this file is defined by the `Binding`.

The `Output` consists of two elements. One is the `ReturnSource` element that declares the source for the return value. Common sources are standard out (`StdOutSource`), standard error (`StdErrSource`) and the return code of a binary (`ReturnCodeSource`) or a file that has been created by the legacy code (`FileSource`). Sometimes, legacy code prints its information on standard out as well as standard error, without printing any error information. In this situation, the `StdComposite` can be used. It merges standard out and standard error and ignores the error semantics of standard error. Besides the `ReturnSource` element, the `Output` is endowed with the `ReturnType` element. It defines how the `ReturnSource` data should be passed back, i.e., returned. In most cases, the `ElementReturn` may be used for the `ReturnType`. For the Java binding, this is the return type of a method. Another type is the `FileReturn` that turns the return type into a file type. The concrete return for the `FileReturn` depends on the `Binding`. For example, the `JavaProxy` may use the *java.io.File*, whereas the `Webservice` may use a Base64 encoded string. An `Operation` can be equipped with an `Environment` containing information about a working directory or an environment variable. The latter one may needed to be set to execute the legacy code, like *LD_LIBRARY_PATH*.

The `Binding` element that can occur multiple times within a `Service` element defines the (so-called) binding of the legacy code. So far, the LCDL model has neither information nor references about the type of wrapper that will wrap this legacy code. The mapping is specified by the `Binding` element. A `JavaProxy` binding generates a Java interface for the legacy code, whereas a `WebService` binding generates a web service (for a specific web service container).

As already mentioned above, the extensibility of this framework is an important issue. In Figure 2, an extension of the `Input` is shown. As a new input type, Flex-SwA [9] is introduced. Flex-SwA serves as an input type for the wrapper code in order to manage the data handling. When using Flex-SwA instead of directly passing huge binary objects, only small references have to be exchanged. In conjunction with service orchestration, this allows for speeding up the whole application, because the usually emerging bottleneck at the orchestration node is circumvented.

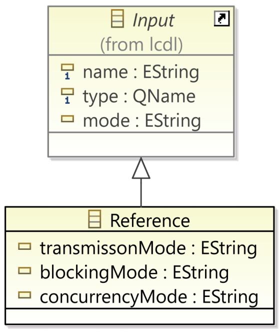


Figure 2: Extension of the Binding by a Flex-SwA element

4. IMPLEMENTATION

The UML diagram in Section 3 was created by dint of the Eclipse Modeling Framework (EMF) [4]. The EMF plug-in

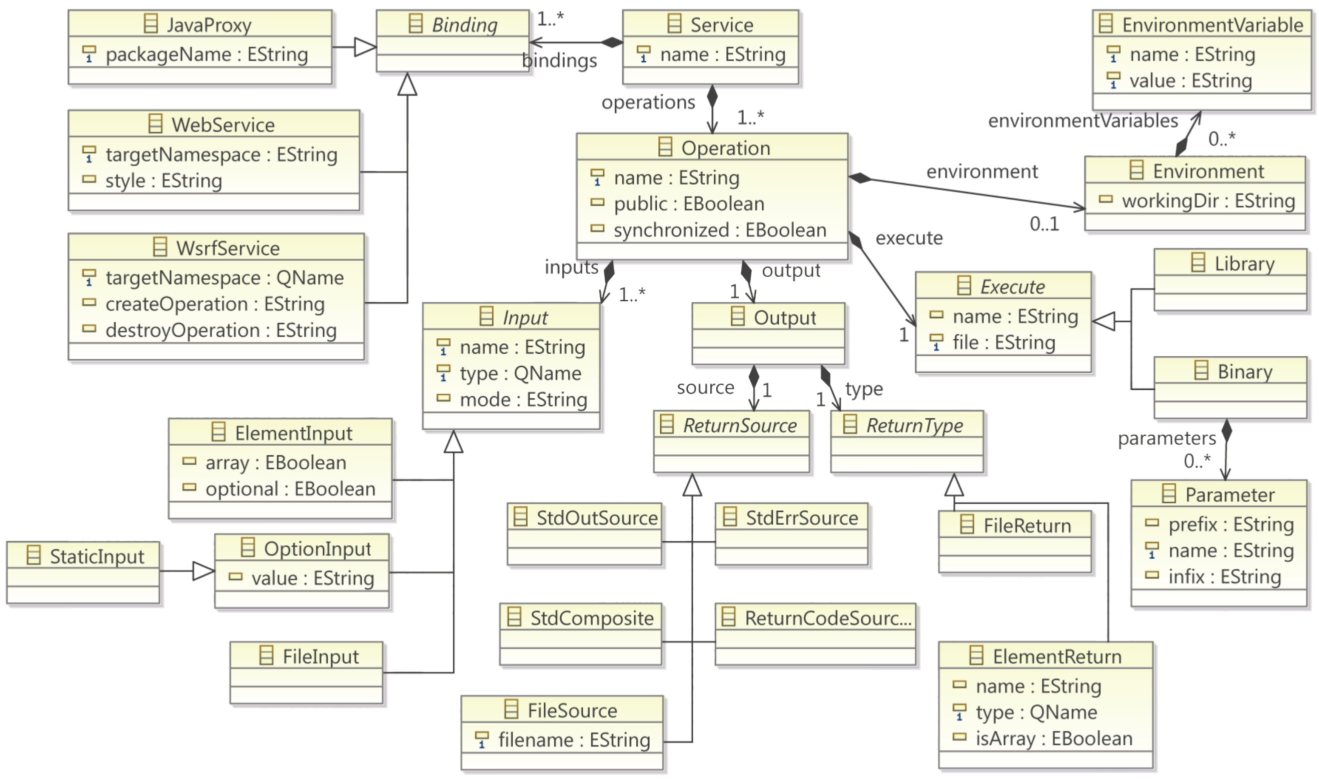


Figure 1: UML Diagram of the LCDL

permits a graphical modeling of the UML diagram as well as the generation of the corresponding model classes and a simple table based editor (for testing purposes). In addition to the editor, the EMF creates a (basic) validator that is capable of checking the (syntactical) constraints given by the UML diagram.

Based on this approach, a prototypical implementation has been developed. This implementation takes the XML representation of the LCDL model, uses the Eclipse EMF API for loading and validating and then generates the service or proxy depending on the actually defined binding(s).

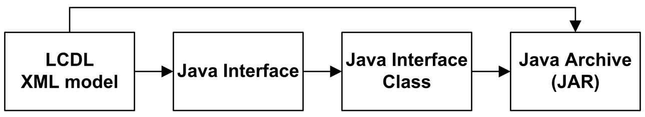


Figure 3: Generator process

The generator process of a `JavaProxy` binding is shown in Figure 3. To generate a Java interface, all `Operation` elements are mapped to Java methods and the XML Schema types of the input and output types are also mapped to their corresponding Java types. This mapping is based on the mapping defined in Apache Axis ([1]). Finally, the generated interface is annotated with the XML model file, and both the model file and the compiled interface are packed into a Java archive (jar) file.

At runtime (in the Java case), a factory implementation `LcdlFactory` is the main component for invoking legacy code via the LCDL. In the `JavaProxy` case, the *LCDL factory* is called by a Java class that will be calling the legacy code. This class gets an instance of the `JavaProxy` by using the factory. This instance is created in the following way. At first, the model file is loaded, by using the Annotations within the Java interface. With the aid of the Dynamic Proxy concept of Java, an invocation of a method of the `JavaProxy` is then redirected to an `AInvocation` implementation. This one analyzes the called method and – together with the LCDL model – creates a `Runnable` that in turn takes the given arguments and executes either the binary or makes a library call. For optimization purposes, this `Runnable` is cached within the factory. This is illustrated in Figure 4.

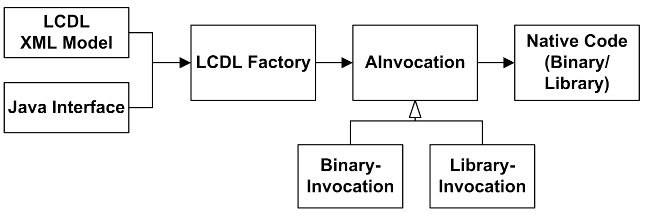


Figure 4: Runtime behavior

For library calls, the *Java Native Access* (JNA) [11] API is used. This offers a platform independent way of calling library code and – in contrary to the Java Native Interface (JNI) [12] – JNA works on a given library, on top of which the adequate Java code can be generated. Anyway, JNI takes a Java class prepared with special keywords (*native*) and generates a header file for C/C++, which in turn has to be implemented. In the extreme case, an appropriate JNI wrapper has to be implemented for each single Java wrapper. The use of JNA together with the LCDL description avoids this recurring (and costly) activity.

The implementation of the `Binding` element works as follows. The `JavaProxy` creates a Java interface within the specified package, whereas the `Webservice` creates an Axis web service [1] with the given target namespace and the RPC or document style. If the legacy code has a state, like libraries that need to be initialized with certain parameters, the usage of a `WsrfService` binding is reasonable. Unlike web services, which represent stateless resources, WSRF-Services has a state, which is preserved between several invocations. One of them might be an initialization and the other one(s) might be the actual call. The create and destroy operations of the factory pattern are reflected by both the createOperation and the destroyOperation attribute of the `WsrfService`. They reference to the symbolic name of the operation, e.g., to perform an initialization. This enables the framework to create a WSRF Service for the Globus Toolkit [6] that is capable of deploying and executing WSRF services.

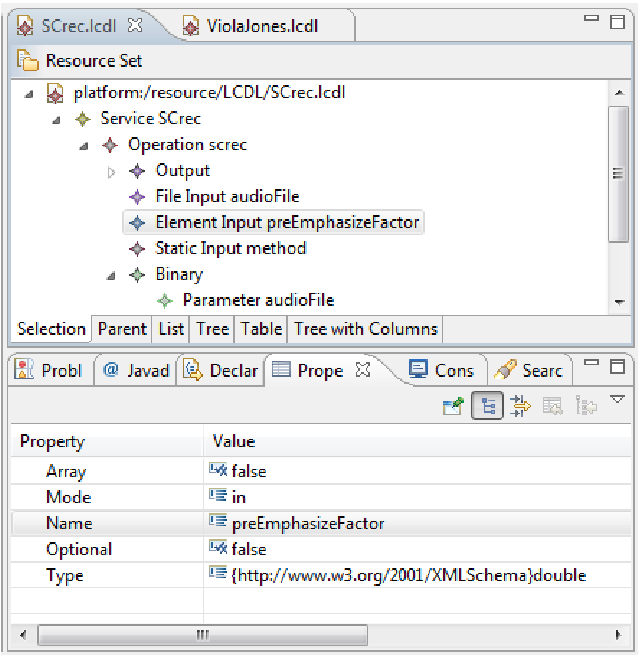
5. USE CASE: SPEECH PROCESSING

Typical use cases for the LCDL arise frequently in a multimedia analysis project conducted in our research group: there, new algorithms are often implemented in C/C++ for the sake of speed of processing and compatibility with existing libraries; when coupling them with our Java based user interfaces (or making prototypes available for other researchers via the web), legacy code wrapping for a certain binding becomes necessary.

A specific example is speech processing: recently, we have developed a tool that is able to make intermediate results of a typical speech processing chain audible. In this way, a user can perceive the effects of different parameter settings. This fosters rapid understanding and thus quick and accurate application of audio processing techniques like feature extraction [13].

Previous publications focused on the signal processing aspects of the problem as well as on how the code, exposed as a web service, can be accompanied with an easy-to-use client for prospective users. Here, we focus on the prior step of wrapping the application as a Java application. The software has originally been implemented as a MS Windows binary called `SCrec` written in C++ and contains different methods related to speaker recognition.

The first step is the modeling of the LCDL information. This can be done using a basic LCDL editor, as shown in Figure 5. All information is entered there and a basic validation can also be performed, in order to check whether all needed elements and attributes are set. The corresponding XML file is shown in Listing 1.


SCrec.lcdl ☒ | ViolaJones.lcdl

Resource Set
- platform:/resource/LCDL/SCrec.lcdl
  - Service SCrec
    - Operation screc
      - Output
      - File Input audioFile
      - Element Input preEmphasizeFactor
      - Static Input method
      - Binary
        - Parameter audioFile

Selection | Parent | List | Tree | Table | Tree with Columns

Probl | Javad | Declar | Prope ☒ | Cons | Searc

Property	Value
Array	false
Mode	in
Name	preEmphasizeFactor
Optional	false
Type	{http://www.w3.org/2001/XMLSchema}double

Figure 5: Basic LCDL editor

```
1 <lcdl:Service xmlns:lcdl="http://fb12.de/lcdl↘
  →/1.1" name="SCrec">
    <operations name="screc">
3     <output>
        <source xsi:type="lcdl:StdOutSource"/>
5       <type xsi:type="lcdl:ElementReturn"
        name="returnValue"
7       type="{http://www.w3.org/2001/XMLSchema↘
        →}string"/>
      </output>
9     <inputs xsi:type="lcdl:FileInput"
      name="audioFile" mode="in"
11    type="{...}any" />
      <inputs xsi:type="lcdl:ElementInput"
13    name="preEmphasizeFactor" mode="in"
      type="{...}double" />
15    <inputs xsi:type="lcdl:StaticInput"
      name="method" mode="in" value="28"
17    type="{...}int" />
      <execute xsi:type="lcdl:Binary"
19      file="SCrec">
        <parameters name="audioFile"/>
21      <parameters name="method"/>
        <parameters prefix="-featureMfcc.↘
        →preEmphasizeFactor"
23      name="preEmphasizeFactor" infix="="/>
      </execute>
25  </operations>
    <bindings xsi:type="lcdl:JavaProxy"
27  packageName="de.fb12.sclib"/>
  </lcdl:Service>
```

Listing 1: LCDL model (namespaces are omitted for reasons of readability)

The particular method under consideration (`method=28`) takes as an additional parameter the filename of an audio file for which standard speech features are extracted. The LCDL model describing it is depicted in Listing 1. Virtually any parameter of the feature extraction procedure can be controlled. We only take the `featureMfcc.preEmphasis` parameter as an example here – it controls the high frequency boost prior to further signal processing, which is clearly audible in the result: an audio file containing the resynthesized features. The method we have defined above is reflected in the `StaticInput` element in lines 14 – 16. The name of the file is written to standard out by the program. This information is captured and returned by the LCDL framework as a string.

```
  package de.fb12.sclib;
2 import de.fb12.lcdl.runtime.java.↘
  →LcdlAnnotation;

4 @LcdlAnnotation(model="SCrec")
  public interface ISCrec {
6   public java.lang.String screc(
      java.io.File audioFile,
8     java.lang.Double preEmphasizeFactor);
  }
```

Listing 2: Generate Java Interface

```
1 // ...
  File audioFile = new File("input.wav");
3 double preEmphasize = 0.97;

5 ISCrec screcService = (ISCrec)
   LcdlFactory.getInstance(ISCrec.class);
7
  String filename = screcService.
9  screc(audioFile, preEmphasize);
  // ...
```

Listing 3: Usage of the generated interface

To use the legacy code in a Java program, the code snippets shown in Listing 2 and 3 come into play. In Listing 3, it is shown how the LCDL factory is utilized to get an instance of the SCrec service (lines 4 – 5). To invoke a method of the generated interface in Listing 2, a simple Java method call is necessary (lines 6 – 7).

6. CONCLUSIONS

In this paper, we have presented the LCDL framework, an extensible approach for wrapping legacy code. It supports different types of legacy code and is based on a plug-in mechanism to support several input and output types as well as binding mechanisms. A use case from the area of speech processing has been presented to illustrate our approach.

There are several areas for future work: (1) the integration of the complete XML Schema for representing complex data types as they are used in library calls; (2) the extension of the Environment element to cope with JSDL information, i.e., hardware requirements etc., and to facilitate an automated deployment of the legacy code on appropriated computing nodes, such as Cloud computing nodes [3], or on desktop pools [8]; (3) the incorporation of additional interfaces – bindings for Restful Services, Matlab interfaces or an integration into (Java) Message Bus Systems are conceivable.

7. ACKNOWLEDGEMENTS

This work is supported by the German Ministry of Education and Research (BMBF) (D-Grid Initiative) and by the German Research Foundation (DFG) (SFB/FK 615).

8. REFERENCES

[1] Apache Foundation. Apache Axis. http://ws.apache.org/axis/.

[2] T. Delaitre, T. Kiss, A. Goyeneche, G. Terstyanszky, S. Winter, and P. Kacsuk. Gemlca: Running legacy code applications as grid services. *Journal of Grid Computing*, 3(1):75–90, 2005.

[3] Amazon Web Services LLC, Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`.

[4] Eclipse Foundation. Eclipse modeling framework project. http://www.eclipse.org/modeling/emf/.

[5] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, UNIVERSITY OF CALIFORNIA, 2000.

[6] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing*, pages 2–13. Springer-Verlag, 2006.

[7] T. Glatard, D. Emsellem, and J. Montagnat. Generic web service wrapper for efficient embedding of legacy codes in service-based workflows. In *Grid-Enabling Legacy Applications and Supporting End Users Workshop (GELA'06), Paris, France*, 2006.

[8] M. Heidt, T. Dörnemann, K. Dörnemann, and B. Freisleben. Omnivore: Integration of Grid Meta-Scheduling and Peer-to-Peer Technologies. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '08)*, pages 316–323. IEEE Press, 2008.

[9] S. Heinzl, M. Mathes, T. Friese, M. Smith, and B. Freisleben. Flex-SwA: Flexible Exchange of Binary Data Based on SOAP Messages with Attachments. In *Proc. of the IEEE International Conference on Web Services, Chicago, USA*, pages 3–10. IEEE Press, 2006.

[10] Y. Huang, I. Taylor, D. Walker, and R. Davies. Wrapping legacy codes for grid-based applications. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 7, 2003.

[11] Java Native Access (JNA). `https://jna.dev.java.net/`.

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

[13] T. Stadelmann, S. Heinzl, M. Unterberger, and B. Freisleben. WebVoice: A Toolkit for Perceptual Insights into Speech Processing. In *Proc. of 2nd International Conference on Image and Signal Processing CISP'09*, page to appear, 2009.

[14] W3C. Xml schema. http://www.w3.org/XML/Schema.

[15] Y. Zou and K. Kontogiannis. Web-based specification and integration of legacy services. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research.* IBM Press, 2000.