A Domain-Specific Language for Interoperability Between Object-Oriented and Mainframe Systems

Marcos Rodrigo Sol Souza
Unisys Brazil
Belo Horizonte, Brazil
marcos.souza@br.unisys.com

Maria Augusta Vieira Nelson
Institute of Informatics
Pontifical Catholic University of Minas Gerais
Belo Horizonte, Brazil
guta@pucminas.br

Abstract
This work presents a domain-specific language (DSL) to integrate object-oriented applications with legacy systems running on mainframes. The DSL offers abstractions to solve recurring problems encountered when integrating these kinds of systems. It offers to the developer a semantic interface that allows him to concentrate on the implementation of the business process requirements without having to worry about the complex computational details of establishing the interaction between systems. The DSL is implemented using the resources of Ruby, a dynamic object-oriented programming language. In this paper we show the syntax of the DSL, its implementation and a case study of its application.

Keywords interoperability, domain-specific language, mainframe systems

1. Introduction
Information technology is a key factor for the success of many organizations. The dynamic and competitive environments in which they are inserted demand ever-greater agility in building and updating systems. Domain-specific languages (DSLs) can contribute to increase development productivity and keep the focus on the elements that really add value.

Many organizations keep important business processes in legacy systems. These systems were often built in closed mainframe architectures. Migrating these systems to more current platforms is not always of interest (Knight 1994). These migrations can have high costs and demand professionals who understand the legacy system’s implementation and who are not always available. In other cases, the mainframe systems simply still fulfill the majority of functional and non-functional requirements.

These legacy systems are usually inserted in heterogeneous environments and must coexist with other software systems. There are several approaches for interoperability between legacy systems and new object-oriented systems. Service-oriented architectures have been used successfully in recent years, unlike other less successful past attempts like CORBA (Common Object Request Broker Architecture) and the JCA (Java Connector Architecture) specification.

Much of the work for achieving interoperability is related to mapping data and communicating across networks. Specifically, to achieve interoperability between legacy systems and object-oriented systems, one must map untyped text messages that contain the inputs and outputs of programs in the legacy system to classes in the object-oriented system. This activity involves many phases related to the type conversion between the two systems.

Building applications that must deal with interoperability issues takes a long time. The tasks are usually repetitive and draw the developers’ attention away from the problem domain. Using a DSL to abstract away the complexity of achieving interoperability between object-oriented systems and legacy systems is a good approach for faster and more productive development.

Many legacy systems in mainframes were implemented in COBOL. In the case of Unisys mainframes, the interoperability with other systems happens through the exchange of messages. These messages represent a data structure that the COBOL program can understand. The messages must be manually converted to the corresponding object-oriented model. The mapping leads to lowered flexibility to deal with changes in the legacy system. A new process or even a new attribute can have an impact on the object-oriented system that depends on the legacy system.

1.1 SOA approach
Service Oriented Architectures (SOA) are becoming a standard for achieving interoperability among complex systems. However, the DSL presented in this work differs from SOA in some aspects. In many computational ecosystems, service buses help with integration and allow great scalability for the environment (Kraflzig et al. 2004).

SOA usually imply some sort of adaptation in the software systems that wish to publish and consume services.
These implementations can be aided by middleware or by direct intervention over the application (Oberle 2006).

The use of SOA requires a clear mapping of the business rules and the knowledge of where they can be found in the system. This is often not available in legacy systems, because the rules can be scattered across several programs. The approach proposed in this work differs from the service-oriented approach because it sees the legacy system as a black box that has a set of working inputs and outputs, without concerning itself with the business rules behind it (Hurtwitz et al. 2006).

2. Providing abstractions for interoperability
A large part of the difficulty in building applications that need to communicate with legacy systems lies in the lack of good interoperability abstractions. Developers spend a long time writing code that performs simple tasks like mapping input/output messages to object-oriented concepts, instead of dedicating time to effectively solve the real problems related to the application domain. Implementing communication is also an important aspect.

No matter what technologies and standards are used to achieve interoperability, the mapping of data and interfaces between the systems is usually static. This leads to replication of data representation in both systems, even if transient. The replication impacts negatively on the maintainability of the application, because a change that includes new data in one of the system’s interfaces must be replicated across the entire code (Bisbal et al. 1999).

2.1 Communication protocol
Interoperability demands the definition of a communication protocol between the platforms. Though mainframes have their own communication protocols like the Burroughs Network Architecture (BNA), they currently natively support the TCP/IP protocol, which makes this the most natural choice (IBM 1999). In the case of Unisys mainframes, it is also possible to combine the TCP with the Custom Connection Facility (CCF). The CCF provides a way to direct packages that arrive at a specific TCP port to a central program. This program will direct the message to its destination and provide a reply to the caller.

2.2 Message exchange
Communication between programs is done through asynchronous message exchange, as required by the mainframe. These messages carry data in a previously defined format and represent the actions that must be executed in the legacy system. Messages also contain execution parameters.

The main body of the message is composed of a sequence of elements. First comes the program identifier, which in the case of programs managed by the Unisys Transaction Server (COMS) is called TRANCODE, a unique attribute that identifies each program managed by the mainframe central controller. Next comes the action. The action determines what the program must execute when receiving the message. Actions can be of many kinds, from common actions of a CRUD (create-read-update-delete) to complex actions like emitting reports or performing simulation computations. The remainder of the message contains the layout of the message that the program expects to receive. In our case, this is a COBOL segment. The complexity of this kind of data structure must be handled by the object-oriented program that needs to communicate. This implies that compaction algorithms must be implemented, and vector or matrix field groups must be interpreted. In COBOL, these field groups are called occurs(Unisys 1999)(Unisys 1995).

3. Mapping domain expertise
The main goal of this DSL is to provide an interface to integrate programs without the need for the developer to know about the issues involved in interoperability. He must be capable of integrating a legacy system with his object-oriented system with nothing but the syntax offered by the DSL.

3.1 Abstracting the interoperability complexity
The first step towards arriving at a syntax for the DSL was to describe a common code fragment to perform a simple integration between a legacy program and its representation in an object-oriented system. The object model representing the domain of the program that will be used (Krishnamurthi 2003) needs to be described. Listing 1 shows a class that represents the domain of debits in loan contracts. The programming language used in the listings is Ruby (rub 2000). The name of an instance variable starts with @ in Ruby.

<table>
<thead>
<tr>
<th>Listing 1. Problem domain class</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#Debits class with legacy system attributes</code></td>
</tr>
<tr>
<td><code>class Debits</code></td>
</tr>
<tr>
<td><code>@trancode</code></td>
</tr>
<tr>
<td><code>@action</code></td>
</tr>
<tr>
<td><code>@contract_id = 0</code></td>
</tr>
<tr>
<td><code>@customer_id = &quot;&quot;</code></td>
</tr>
<tr>
<td><code>@name = &quot;&quot;</code></td>
</tr>
<tr>
<td><code>@year = 0</code></td>
</tr>
<tr>
<td><code>@fine = 0.00</code></td>
</tr>
<tr>
<td><code>@location_code = 0</code></td>
</tr>
<tr>
<td><code>@status = []</code></td>
</tr>
<tr>
<td><code>end</code></td>
</tr>
</tbody>
</table>

Another class would be necessary to map the messages from each direction into the class defined above. Listing 2 shows how this function would be written.

<table>
<thead>
<tr>
<th>Listing 2. A class to query the legacy system</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class DebitsQuery</code></td>
</tr>
<tr>
<td><code>get @trancode , @action , @contract_id , @customer_id , @name , @year , @fine , @location_code , @status</code></td>
</tr>
</tbody>
</table>

A DSL for interoperability
### 3.2 Defining a grammar

The next step is the definition of a grammar so that the DSL can provide the data required for interoperability without burdening the programmer (Fernandez 2007). The code fragment that resulted from the previous analysis was again examined to find the functions that should be supported by the DSL, according to Listing 4:

```
Listing 4. Functionality of the analyzed code

class Foo

  def main
    # Creates an instance of the class DebitsQuery
    debits = DebitsQuery.new

    # Executes an action in the legacy system
    # passing an input attribute
    obj = debits.findByContract(1234)

    # Gets and manipulates the attributes returned
    # by the legacy system
    puts obj.name

  end

end
```

For the programmer, the DSL should support the following actions:

- Create an instance of the object
- Execute the actions supported by the legacy system
- Inform input parameters for the actions
- Obtain and use the results of executing an action

Based on this analysis, a syntax was defined to simplify the process. First, we define a key that indicates which program must respond to the requests in the legacy system. This key was called `trancode`. By convention, any method with a three-character name represents the execution of an action in the legacy system. The basis for this decision lies on the practices of work environment where the study was performed. In this environment, it is common practice to use three letters to name actions. Finally, it was defined that the input or output attributes will be represented by their own names. When used in the left-hand side of an assignment, the attribute will represent data that must be sent to the mainframe. The same attribute, when used in the right-hand side of an assignment, will represent data to be read from the mainframe.

Based on this analysis, Listing 4 was rewritten, establishing the syntax of the DSL to be as shown in Listing 5:

```
Listing 5. DSL syntax

# Defines the program in the legacy environment
trancode : V034741001

# Defines an input parameter
contract_id = 1234
```

Finally, a new class could be written to achieve interoperability using the above-defined classes. Listing 3 shows these classes being used.

```
Listing 3. Execution of a query in the legacy system

class Foo

  def main
    debitsQuery = DebitsQuery.new
    obj = debitsQuery.findByContract(1234)
    puts obj.name
  end

end
```

Except for the 7 lines of code from listing 3, all code was written solely for the purpose of interoperability. There is code to create a class representing the interface of the legacy system, code to create connections with the mainframe, code to compress and decompress messages, and lots of code to map data from the mainframe to the object.

One of the goals of the DSL is exactly to abstract away most of the complexity of interoperability so that the programmer can focus on the implementation of the business rules (Fowler 2005). Given this premise, the DSL must provide a way for the programmer to access any program in the legacy system by using only the last code fragment; all the rest must be a responsibility of the DSL.
4. Implementation strategy

4.1 Formal definition of the DSL syntax

The DSL used in this work was written in the Ruby programming language. Ruby was selected because it is a dynamic language that allows behaviors to be changed by overloading methods of its kernel. Ruby also offers excellent support for metaprogramming.

The implementation of the DSL follows the approach known as Class macro. In this approach, the DSL is built from methods in some base classes. Subclasses can use these methods to modify their own behavior or that of their own subclasses (Fernandez 2007). Another benefit of this approach that is used by the DSL is the ability to change an object’s behavior dynamically by executing a command.

To use the DSL, the programmer must create a class that inherits from the DSL class. This class contains the grammar defined by the DSL. When executing an instance of the object that inherited from the DSL, the constructor will initialize the necessary data structures and will create a connection with the mainframe. Next, it creates the attributes that are required to store the data during the execution of the DSL, which include the current program on the legacy system that the DSL refers to. The following step is to call the private method that updates the input and output data structures according to the specification found in the domain repository. This repository contains the descriptions of the communication interface of each program from the legacy system. The repository is created automatically by reading reports about the programs available in the mainframe. — O repositório também mantém informações sobre a tipagem dos atributos no sistema legado. A DSL utiliza essas informações para realizar o casting dos dados para sua representação equivalente no sistema legado. —

The update_internal_structures method is shown in Listing 6.

Listing 6. Updating internal structures

```ruby
def update_internal_structures
  @in = @specs
  @sorted_in = @in.sort{|x,y| x[1][0] <=> y[1][0]}
  @sorted_in.delete_if{|e| e==nil}
  @sorted_out = @out.sort{|x,y| x[1][0] <=> y[1][0]}
  @sorted_out.delete_if{|e| e==nil}
end
```

This method is also called every time the system executes a method that substitutes the current DSL program for another. After assigning all message specifications, arrays are created with the fields in the order in which they will be sent and received by the mainframe.

Every time the trancode command is executed, the program name received as parameter will be used in a query to the knowledge repository. If there is information about the program in the repository, a specification will be returned and used to update the data structures, configuring the current instance to the context of the new program. This command is defined in the method shown in Listing 7.

Listing 7. Macro to identify the context

```ruby
def trancode(trancode)
  @specs = Repository.trancodes[trancode.to_sym]
  @trancode = trancode
  update_internal_structures
end
```

Once the current program is defined, the programmer can use the other commands from the DSL syntax, which are assignment of a value to an attribute, reading an attribute, and execution of an action. This is implemented by overriding method Kernel.method_missing. This method belongs to the Ruby language kernel. Once overridden, it will intercept every call to a nonexistent method on the object that inherited from the DSL (rub 2000).

When method method_missing is called, the DSL verifies in the current specifications if the method that failed represents any program attributes. If so, it verifies if the failure was caused by an attempt to assign a value to or read a value from a nonexistent attribute. If it’s an attempt to read, the value corresponding to the attribute that is in the data structure that stores the output values will be returned dynamically to the caller. If the call is an assignment, then the value will be assigned to the data structure that stores the input values so that it can be sent later to the legacy system. This implementation also verifies if the name of the method has only three letters, according to DSL convention. In this case, it will send the corresponding action to the program in the legacy system, passing as argument a message containing all the data that were assigned by the programmer. When a reply message is received from the legacy system, the message is parsed and the values are dynamically assigned to the corresponding attributes.

Overriding the method Kernel.method_missing is what makes this approach possible and flexible. Listing 8 shows the implementation of this method for this DSL (rub 2000).

Listing 8. Implementation of method_missing

```ruby
def method_missing(methodld, *args)
  # verifies if it is a getter
  if methodld.id2name =~ /^(?!)$&
    @in[methodld.id2name[0], methodld.id2name.size - 1].to_sym
  end
end
```

A DSL for interoperability
The overridden method receives as parameter the name of the method that was not found and that represents the attempt to use some DSL command, plus a list of pointers to the arguments to this command. The system checks if a read attempt was made, and if the appropriate attribute exists in the input specification. If so, the current value of the attribute is returned. To achieve this, a regular expression detects whether the last character of the name of the received method is ‘=’, which the Ruby language concatenates by convention to the argument that represents the method name. If it isn’t a read attempt, the system checks if a write attempt was made. If there is a compatible attribute, then the value of the first argument to the method is associated to the key of the hash that stores the input attributes.

If both checks above fail, the system checks if the method name has only 3 letters, in which case it calls a method that detects whether the last character of the name of the received method is ‘=’, which the Ruby language concatenates by convention to the argument that represents the method name. If it isn’t a read attempt, the system checks if a write attempt was made. If there is a compatible attribute, then the value of the first argument to the method is associated to the key of the hash that stores the input attributes.

If none of the above conditions are true, the system considers it a syntax error, raising an exception that will inform the user that the attempted command is not valid in the current DSL context.

The method responsible for sending the executed action to the legacy system assembles an input message to that system according to the specifications. This message will be sent through the connection created by the constructor of the parent object. When a reply to this message is received from the legacy system, it is parsed and the values it contains are assigned to the corresponding attributes.

5. Applying the DSL

We performed a case study consisting of rewriting a real application with the proposed DSL. The chosen application is responsible for simulating real estate loan financing for a financial institution. All business rules for this application already existed in a legacy system, but the client required a web-based interface with high usability, which would allow the institution’s customers to do their own loan simulations. This application was originally implemented using the Java programming language and a large portion of its code was written with the sole purpose of achieving interoperability with the legacy system.

When using this application, the user provides some data such as birth date, income, and some characteristics of the property for which he wishes to acquire a loan. This information is sent to the legacy system, which selects the best type of loan for the client’s profile, from among many available types. At first, the legacy system assumes that the conditions for the loan are those that will maximize the value of the loan. The results of the simulation calculations are displayed to the user. The user can change some of the conditions for the loan such as duration and initial value. In this case, he is taken to a screen where a new form for data entry is shown. The changes are sent to the legacy system and new values are returned to the user.

When using the DSL, all of the code related to interoperability was eliminated, making the implementation fully oriented to the problem domain. Listing 9 shows how the DSL can be used to represent the entire flow described above.

**Listing 9. Modeling the simulator flow**

```ruby
SimulatorFlow < DSL

# Identifies the program in the legacy environment
trancode : simulator

# Initializes attributes
birth_date = '22/02/1983'
income = 2000.00
property_value = 50000.00

# Executes the action (sil) that corresponds to the first step in the simulation flow
sil

# Chooses the first product as the simulation target
product_code = products[0]

# Executes the action (si2) that corresponds to the second step in the simulation flow
si2

# Writes the installment value of the loan
puts "Installment value:$\{installment\}"

# Changes the duration of the loan
term = 180
```
# Executes the action (si2) that corresponds
# to the second step in the simulation flow
si2

# Writes the installment value after the change
# of the loan duration
puts "Installment value: ${installment}"
end

6. Conclusion

This work presented a language specific for the domain of interoperability between object-oriented systems running on microcomputers and legacy systems running on mainframes. The legacy applications are important to organizations, because they contain systems that are essential to their businesses. However, the information in these systems needs to be accessed by users through varied interfaces, including the web. This creates a constant need for new applications that need to communicate with the legacy systems.

We conclude that using a DSL for interoperability between legacy and object-oriented systems is a good approach to increase development speed and improve maintainability of this kind of application.

Our results encourage the use of the proposed approach, because it was possible to abstract away a great portion of the complexity inherent to interoperability, such as communication, message composition and object assembly.

Besides productivity, another important aspect of this approach is the ability given to the developer to describe with greater semantics the transformations in his business processes, using the DSL syntax.

There are a few limitations to the DSL approach presented here. First, the focus of this work was the legacy systems developed in COBOL that run on Unisys mainframes. It may be necessary to adapt the DSL to deal with systems written in other languages or in other hardware platforms from different manufacturers. The second limitation is related to the fact that this DSL is internal to Ruby and its syntax can never abandon the principles of its host language.

However, the gains in productivity, its simple syntax, the transparency in the interoperability and the shift in focus to business rules are significant advances.

References
