Model-Driven Development in implementing integration flows

Tomasz Górski
Institute of Computer and Information Systems, Faculty of Cybernetics, Military University of Technology, Warsaw, Poland
tomasz.gorski@wat.edu.pl

Abstract: Integration of many different IT systems makes the integration project highly complex. The process of constructing architectural models and source code can be automated through the application of transformations. As a result, the duration time of designing or implementation, as well as the work input involved can be reduced. The purpose of the paper is to present an approach to automation of designing one of the key elements of an integration platform, namely, integration flows. The author proposes model-to-code transformation IntegrationFlow-to-Java which automates the implementation of integration flows applications for selected mediation patterns. The integration flows generator has been incorporated as a plug-in into the IBM Rational Software Architect (RSA). The RSA plugin which generates complete Java EE application of integration flow from mediation flows diagram. Thus eliminates design and programming stage in WebSphere Integration Developer which reduces development time and costs of licenses. Model-Driven Development is approach which can lead to automation of design and programming stage in software development. The IntegrationFlow-to-Java transformation offers an opportunity to reduce the duration time of the integration flows implementation forty times (with one hundred flows to be implemented). The outcomes support the significance of using transformations when designing complex IT systems, especially when integration solutions are developed.

Keywords: model-driven development, transformations, enterprise application integration, service-oriented architecture

1. Introduction

The ability to promptly provide services or products meeting clients’ needs is a matter of key importance to companies. Most commonly, an organisation operates a variety of IT systems. This implies the need to build integration solutions composed of IT systems and a communication layer that enables cooperation between these systems [1, 2]. A solution of this type is called an integration platform or an integration solution. When designing integration solutions, it is essential to be able to model their complete architectural description. For this purpose, an architectural views model is needed which allows to model integration platforms and a set of model constructs enabling the presentation of the integration platform’s entire architecture [3, 4]. Integration of many different IT systems makes the integration project highly complex [5]. The process of constructing architectural models can be automated through the application of transformations [6]. As a result, the duration of designing or implementation can be reduced, as well as the effort and the number of errors involved [7, 8]. Transformations are elements of the Model-Driven Engineering approach (MDE) [9, 10, 11].
The paper proposes an approach to the automation of designing and programming one of the key elements of an integration platform, namely, integration flows. The architectural views model ‘1+5’ is used [12]. The remainder of the paper is structured as outlined below. Section ‘Architectural views model 1+5’ describes views, models and diagrams of the architectural views model ‘1+5’. Section 3 contains an overview of publications dedicated to similar problems. The problem of integration flows, as well as mediation patterns and examples of mediation patterns application are presented in the section ‘Integration flows’. In that section, mediation patterns to be used for automation are also selected. Thereafter, a design of the automation of implementing integration flows applications for selected mediation patterns is proposed. Furthermore, the paper presents structure of the integration flow application and design of the integration flow generator. Subsequently, it is shown an exemplary application of the solution developed for automation of the integration flows implementation. In addition, the paper discusses the benefits of the automation facility application. Section 11 concludes the paper, summing up the subject and outlining directions for further work.

2. Architectural views model ‘1+5’

The software architecture of an IT system is the result of a set of architectural decisions. The topic of architectural decisions in software engineering has received significant attention in recent years [3, 4, 13]. A variety of models exist, with differing sets of architectural views, such as, for example: ‘4+1’, RM-ODP, Siemens, SEI views [14]. Yet, they do not allow for a complete description of the integration solutions architecture. The ‘1+5’ model of architectural views proposed here has been accommodated to suit the process of an integration platform design. The following architectural views have been distinguished within the model: Integrated Processes, Use Cases, Logical, Integrated Services, Contracts, Deployment. The view of Integrated Processes is the basic architectural view here. In this view, business processes to be automated on the integration platform are modelled. The next four views (Use Cases, Logical, Integrated Services and Contracts) present the integration platform design. The Use Cases view contains functional requirements for the system being integrated within the platform. The view of Integrated Services presents services exposed by IT systems, and the way in which they are connected to the service bus. The Contracts view shows components representing IT systems and the contracts defined between them. This view also encompasses mediation flows for each contract. The last view, Deployment, shows the way the integration platform elements are deployed in a specific runtime environment. Figure 1 illustrates the architectural views model ‘1+5’.
Table 1. Elements for modelling the integration platform architecture

<table>
<thead>
<tr>
<th>Model</th>
<th>View</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes</td>
<td>Integrated Processes</td>
<td>(BPMN) Business Process</td>
</tr>
<tr>
<td>Use Cases</td>
<td>Use Cases</td>
<td>(UML) Use Case</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(UML) Activity</td>
</tr>
<tr>
<td>Design</td>
<td>Logical</td>
<td>(UML) Sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(UML) Communication</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(UML) Class</td>
</tr>
<tr>
<td>Services</td>
<td>Integrated Services</td>
<td>(UML) Component</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(UML) Activity – mediation flow diagram</td>
</tr>
<tr>
<td>Contracts</td>
<td></td>
<td>(UML) Component</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(UML) Composite Structure</td>
</tr>
<tr>
<td>Deployment</td>
<td>Deployment</td>
<td>(UML) Deployment</td>
</tr>
</tbody>
</table>

In the approach analysed here, models and diagrams of BPMN [15, 16] and UML [17] languages, with an extension of Service oriented architecture Modeling Language stereotypes [18] have been used for modelling the integration platform architecture (Table 1). In Integrated Services view a mediation flow diagram was used which is extension of UML activity diagram [19].

3. Related studies

Literature on the subject is rich in Model-Driven Development problems. In the paper transformation IntegrationFlow-to-Java was proposed. Details of the model transformation taxonomy are presented in [6]. Publications describe various languages that can be used for generating transformations: Query/View/Transformation (QVT) [20] and ATL (Atlas Transformation Language) [21, 22]. In publications of recent years, special focus has been on the topical and vital issues of model-to-model transformations verification and validation. An up-to-date, comprehensive overview of the model transformation verification problems is provided by Calegari and Szasz [23]. A systematic review of formal verification of static software models in MDE was presented by González et al. [24]. One of the most important conclusions obtained by González is that current model verification approaches are strongly influenced by the support given to Object Constraint Language (OCL). Another important finding is that in general, current verification tools present important flaws like the lack of integration into the model designer tool chain or the lack of efficiency when verifying large, real-life models. Furthermore, a method for UML class diagrams verification with OCL was proposed by Cabot et al. [25]. The method checks compliance of the diagram with respect to several correctness properties including weak and strong satisfiability or absence of constraint redundancies among others. Moreover, Guerra et al. [26] propose how the QVT language can be used for constructing a declarative language for the specification of visual contracts, which enables verification of transformations defined in any language of transformation. Model-Driven Development finds its applications in different aspects of software
Model-Driven Development in implementing integration flows

engineering. Security is one of the vital types of non-functional requirements. An extensive systematic review on the Model-Driven Development of secure systems was provided by Nguyen [27].

As a result of applying transformation, it is possible to significantly reduce the time for creating architectural description and software development. Analysis available for computer systems designed in the medical sector show that in designs where model-driven architecture is used, there has been a three-fold shortening in software development reported [7, 28]. When designing solutions for IT solutions for use on the Internet, savings from the use of model-driven architecture are more than two-fold (shortening software development time by 59%) [9]. In addition, MDA enables to save time spent on the various stages of an IT project’s life cycle [29]. Studies show that by using transformation there a 10-fold decrease in the number of reported errors in software has been recorded [7, 28]. On the other hand, Panach et al. [30] claim that for small systems and less programming-experienced subjects, MDD does not always yield better results than a traditional method, even regarding effort and productivity. But, results of Lytra et al. [13] show that the use of reusable decision models can significantly increase both the efficiency and the effectiveness of novice architects.

The paper deals with problem of designing integration solutions especially in service-oriented architecture. Especially, integration flows are under scrutiny which are subject of recent studies [31]. The state of the art in the topic of development of service-oriented architectures using model-driven development was provided by Ameller et al. [32]. The analysis reveals, among others, predominance of top-down transformation in software development activities, in-existence of consolidated methods and significant percentage of works without tool support. As far as the paper is concerned, the author proposes top-down transformation IntegrationFlow-to-Java, placed within the ‘1+5’ views model and with tool support in IBM Rational Software Architect.

4. Integration flows

Transformations automating the process of integration platform model design have been discussed earlier, as an element of the architectural views model ‘1+5’. The following transformations have been previously proposed:

- the Business Process Modeling Notation to the Use Cases view (BPMN2UC),
- the Use Cases view to the Logical view (UC2Logical),
- the Use Cases view to the Integrated Services view (UC2IS),
- the Use Cases view to the Contracts view (UC2Contracts).

The present paper presents transformation of a model into a software code. The focus here is on the Integrated Services view and, more specifically, on special form of UML activity diagram – mediation flow diagram. Example of mediation flow diagram, with application of stereotypes from ‘UML Profile for Integration Flows’ proposed by Górski [33], was presented in the Figure 2.
The activity diagram showing the integration flow is used as a basis for generating an operational module of integration flow. The diagram is detailed enough to enable definitive identification of: supplier of the integration flow’s message, integration flow type which determines how the message should be delivered to the recipient, recipient of messages participating in the integration flow.

The transformation presented here is a vertical one, which means that the source element and the target element represent different levels of abstraction. Moreover, the transformation is exogenous, as it transforms models expressed in various notations: UML and Java. A comprehensive classification of transformations was presented by Mens et al. [6].

The key role in integration flows belongs to mediation patterns. Mediation patterns are used to receive the supplier’s message, to perform certain actions and to send a message to the recipient. Messages can be received and sent further via various channels, depending on the technology chosen (e.g. FTP, HTTP, MQ). Mediation patterns are a set of good practices checked, pre-tested and used in the process of integration. Each pattern defines its basic action it performs within the integration flow. The message channel is one of the simplest mediation patterns – its role is limited to sending the message received further on to the recipient [34]. Selected mediation patterns are summarised in Table 2.

Two examples was presented to illustrate the practical utility of Content enricher and Content-based router mediation patterns. The first example refers to sales reports being generated to show the correlation between the sales volume and the weather. We are looking at a chain of retail stores offering a variety of household and daily use products. With the reports, the personnel can view daily sales figures of the chain in a breakdown by articles sold. Information is itemised by article, date and shop, thereby enabling identification of trends. Let us say that a need arises to find out, how the weather affects the sales volume. To this end, a track of sales by categories of articles and sunny or rainy days has to be kept. Obviously, this will not be possible without adding weather details to the body of data sourced by the system. This is where a Content enricher can be applied within the integration flow. Hence, the sales data record is augmented with the weather data by means of an appropriate flow with the Content enricher pattern defined (Figure 3).
Table 2. Selected mediation patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Graphic representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message channel</td>
<td><img src="message_channel.png" alt="Diagram" /></td>
<td>Defines how the supplier communicates with the client. This is the simplest mediation pattern.</td>
</tr>
<tr>
<td>Publish-subscribe</td>
<td><img src="publish_subscribe.png" alt="Diagram" /></td>
<td>Multiplies the message in such a manner that its copy is received by each of the subscribers. Ensures that the message is received by each recipient only once.</td>
</tr>
<tr>
<td>Content-based router</td>
<td><img src="content_based_router.png" alt="Diagram" /></td>
<td>Selects message recipients based on message contents. Routing parameters are defined by means of flow configuration.</td>
</tr>
<tr>
<td>Content enricher</td>
<td><img src="content_enricher.png" alt="Diagram" /></td>
<td>Describes how the message will be enhanced with additional information.</td>
</tr>
</tbody>
</table>

Now, let us consider a case where an organisation uses gift cards issued by external suppliers. Each gift card has its own, unique ID number with the first four digits identifying the supplier. Our intention is that whenever a client attempts to use a card, the spending is reported to the supplier concerned. Here, a content-based router can help. The integration flow would simply read the card number, decode the first four digits and identify the target service address this way (Figure 4).

![Diagram](content_enricher.png)

Figure 3. Content enricher
5. The aim of automating the implementation of integration flow

The purpose of automating the implementation of integration flow is to reduce the time needed to implement the integration flow application being installed on the service bus. The following tools have been used to design and implement integration flows automation:

- Eclipse Kepler,
- IBM WebSphere Enterprise Service Bus (ESB) – the environment for the execution of integration flow application,
- IBM WebSphere Integration Developer (WID) – the integration flows design and implementation environment,
- IBM Rational Software Architect (RSA) – the environment for designing the IT solutions architecture, in particular modelling and designing integration flows,
- SoapUI – testing environment for services.

The process of automation could not take place without understanding the structure of applications to be executed in the service bus environment. Using WID, a simple application consisting of one mediation module with one flow has been developed for the runtime environment. The message channel has been used within the integration flow, with the task of sending the message received on the integration platform. The functionality of RSA enables integration flow modelling. The integration flow is both designed and implemented by means of WID. The purpose is to expand the RSA set of functions in a manner enabling applications to be generated from an adequately created activity diagram (Figure 5).
The integration flow application developed using WID is an Enterprise ARchive (EAR) file, which can be installed with an ESB console. The EAR archive has to contain XML files describing the module implementation method. EAR is a standard Java ARchive (JAR) file containing additionally a representation of modules comprised in the implementation descriptors. The archive of EAR applications includes JAR files and WAR files, as well.

The automation of the integration flows implementation covers the following mediation patterns:

- Message channel – the simplest flow type, but a mandatory one for implementation, since all other patterns are its derivatives,
- Content enricher – its task is to augment the message,
- Publish-subscribe – multiplies the message and broadcasts it to subscribers.

6. Structure of the integration flow application

The structure of the integration flow application archive is described with the application AplikacjaApp.ear which realises the message channel mediation pattern. Most files inside the application are stored in the XML format. The files inside the application archive (ApplicationApp.ear) are structured as follows:

- META-INF – a folder with files describing the application and the method of implementation: sca-module-properties.xml – a file containing environmental variables for the Service Component Architecture (SCA) module, deployment.xml – a file defining the location of libraries and resources, application.xml – a file defining the application name and exchanging additional modules of the application,
- Application.jar – an archive containing the module implementation,
- ApplicationWeb.war – a web module archive.

Such a structure of the application makes it possible to determine parameters needed for a complete mediation module to be generated. At this stage of work, the activity diagram modelled in RSA is left out. The generator is a separate element and can be used by other applications and extensions. Web Services Description Language (WSDL) files defining the supplier and recipient contract are the main and the most important components of the application. These files are selected by the user before the code generating process commences. The application structure analysed above constitutes a template, which has become an integration flow generator resource.

7. Structure of the mediation flow generator

The generator is implemented using the classes and the interface shown in Figure 6. The IMediationModule interface is independent on the type of mediation being generated. The MediationModuleBase abstract base class implements the basic methods defined in the interface.

The IMediationModule interface defines the following operations:

- `getAppName()` – gets the name of the application being generated,
- `setAppName(String name)` – sets the application name,
- `addExport(Definition def)` – sets the WSDL file definition. Data obtained from this definition will be used for determining the suppliers’ interface,
- `addImport(Definition def)` – data obtained from the import definition will be used for determining the message recipient’s interface,
− *generate()* – an application generating method, executed by taking the following steps: creating an output folder, reading the WSDL files indicated by the user, copying the template to the temporary output folder, copying the WSDL files, renaming the files, supplementing the files contents with the values of variables,

− *publishApp(String path)* – archives the application and copies it to the pre-set location.

![Class diagram for generator of integration flow application](image)

Since additional Import files need to be generated, the *PublishMediation* class shuts the base implementation of the *generate()* method out. The overloaded method generates additional parameters to be arranged in a structure of interconnected elements (in the form of correct records to the application’s XML files) receiving the message and broadcasting it to multiple recipients. Furthermore, additional.import files are created, which indicate target addresses of services receiving the message.

8. A plug-in for IBM Rational Software Architect

The generator is linked to the graphic representation of the integration by a plug-in which expands the RSA tool functionality. The extension adds new items to the activity diagram context menu. The extension point is the *org.eclipse.ui.popupMenus* type, responsible for the context menu. The plug-in’s visibility in menu has been determined by a specific object type – object contribution. The option added will be visible in all types of editors or views, as long as the selected object meets the pre-defined type criteria. Figure 7 shows a general outline of the approach. At this stage of work, the activity diagram has been modelled using RSA.
The plug-in has been designed to read the activity diagram values and to create an adequate generator class instance. The extension supplies the application name information to the generator. The mediation type is defined upon the creation of an adequate generator class type. To achieve this, the plug-in reads the selected element’s stereotype (<<Channel>>, <<PublishSubscribe>>, <<ContentEnricher>>).

Reading the value occurs through referring to elements of the structure describing the activity diagram. The path begins with importing the active editor (diagram edition window) instance. The transition from the editor instance to elements containing data is represented by the code below.

```java
// Getting the active editor
IEditorPart part = PlatformUI.getWorkbench()
    .getActiveWorkbenchWindow().getActivePage().
    getActiveEditor();

// Getting the selected element
ISelectionService service = part.getSite().getWorkbenchWindow()
    .getSelectionService();

IStructuredSelection sel = (IStructuredSelection)service.getSelection();
ActionNodeEditPart actionNod = (ActionNodeEditPart)sel.getFirstElement();
UMLShapeImpl shape = (UMLShapeImpl)actionNod.getModel();
UMLShapeCompartmentImpl container = (UMLShapeCompartmentImpl)shape.eContainer();
ActivityImpl activityContainer = (ActivityImpl)container.getElement();

// Getting the application name
ApplicationName = activityContainer.getName();

OpaqueActionImpl actionElement = (OpaqueActionImpl)shape.getElement();
DynamicEObjectImpl stereotype = (DynamicEObjectImpl)actionElement
    .getStereotypeApplications().get(0);

ApplicationType = stereotypeClass.getName();
```
Thereafter, based on the given mediation type, an adequate generator instance is created.

```java
switch(applicationType)
{
    case "Channel":
        moduleGenerator = new SimpleMediation();
        break;
    case "ContentEnricher":
        moduleGenerator = new ContentEnricherMediation();
        break;
    case "PublishSubscribe":
        moduleGenerator = new PublishMediation();
        break;
    [...]
}
```

Starting the process of generation is the last operation performed by the plug-in. The generated application is recorded on the disc.

```java
moduleGenerator.setAppName(ApplicationName);
moduleGenerator.generate();
```

9. Exemplary application of the transformation

For the sake of verification, the transformation has been applied to a specific business situation. We are looking at the Good Shop chain of stores and its process of selling a variety of groceries and household products. The chain consists of several dozen shops, including facilities owned previously by the chain’s former competition, the Cheap Stores chain. Both the original and the new stores of the Good Shop chain use a dedicated IT system. Let us analyse some of its utilities. javaPOS is the program responsible for recording sales operations in shop cash registers. The number of cash registers per shop varies. The Cheap Stores shops use some other type of software. All elements of the IT system are integrated by means of an integration platform. Let us assume that some new regulations require that sales receipts must be stored electronically, in a manner enabling the organisation to view them and make accessible to customers and to competent institutions, on their request. Hence, the organisation is facing a new system requirement. Files representing sales operations will be recorded on a dedicated server’s disc in XML format. The service executing the new functionality has been deployed and implemented. The only element missing is the mediation module deployed on the ESB environment. The following section presents how the process of the mediation module implementation can be automated using the approach proposed in the paper.

The need to archive sales receipts has been imposed externally, therefore, each shop in the organisation’s chain has to be integrated. The UML use case diagram (Figure 8) illustrates need for communication with external IT system ‘Receipts archive’, an actor with the <<IntegratedSystem>> stereotype.

The ESB code executing the new functionality will be generated automatically, based on the activity diagram. The receipt changes its status upon completion of the sales recording process and is sent for archiving as an XML file. The diagram of UML components (Figure 9) shows the components participating in the process of sales receipts archiving. Each time, having recorded the payment, the javaPOS application triggers a service archiving the receipt using the ESB service bus. A solution like this allows for a virtually indefinite number
of cash register applications to be connected to the service bus, with the archiving service provider always remaining accessible to them, owing to the service bus.

![Image](image-url)

**Figure 8. “Payment recording” use case**

The mediation flow is responsible for transmitting each sales receipt to the archiving service. Both paid and cancelled receipts are transmitted. With a solution like this, shops using other software (the Cheap Stores chain) can also be integrated into the system. The flow executing the message channel patterns is illustrated with the activity diagram in Figure 10.

![Image](image-url)

**Figure 9. Diagram of components presenting the sales receipts archiving operation.**

**Figure 10. Activity diagram for the “Sales receipt archiving” integration flow.**

The following steps have been taken to generate the mediation flow:

- Step 1: Starting the automation through triggering an action from the context menu of the message channel element (Figure 11),
− Step 2: Selecting the WSDL file which describes the contract made available by the mediation module,
− Step 3: Selecting the WSDL file which describes the recipient’s contract,
• Step 4: Generating the application (Figure 12),

Figure 11. Plug-in’s context menu.

Figure 12. Message informing of the integration flow application generation.
– Step 5: Installing the application in the ESB environment,
– Step 6: Starting and testing the application.

It should be emphasised that the example shows application of the transformation on a very simple flow, containing only one action with <<Channel>> stereotype. Further work will encompass extending the transformation for generating complex flows.

10. Benefits of the transformation

The transformation described in this paper eliminates the process of simple flows implementation altogether. The only thing that remains to be done is to design the integration flow. Nevertheless, for more complex integration flows, the process of implementation cannot be avoided in the current version of the automation mechanism. In order to analyse the benefits of the automation solution proposed here, some assumptions need to be made. The manual implementation of integration flows has been performed using WID. In the course of implementation, the duration of three successive attempts of implementing three different flows was measured: the Message channel, the Content enricher and the Publish-subscribe flow. Assuming that the flow has already been designed and that implementation is the only operation that needs to be performed, this time may be discounted for both attempts. The tables 3, 4 and 5 summarise the duration of integration flows manual implementation for each individual mediation pattern depending on the number of implementations (verified by students).

<table>
<thead>
<tr>
<th>Attempt no</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>4h</td>
<td>1h</td>
<td>40min</td>
<td>30min</td>
<td>23min</td>
</tr>
</tbody>
</table>

Table 3. Duration of the Message channel flow manual implementation

<table>
<thead>
<tr>
<th>Attempt no</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>5h</td>
<td>2h</td>
<td>40min</td>
<td>30min</td>
<td>29min</td>
</tr>
</tbody>
</table>

The Content enricher pattern requires a non-standard flow to be implemented using Java.

<table>
<thead>
<tr>
<th>Attempt no</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>6h</td>
<td>3h</td>
<td>40min</td>
<td>40min</td>
<td>40min</td>
</tr>
</tbody>
</table>

Table 5. Duration of the Publish-subscribe flow manual implementation

In order to visualise the scale of benefits that can be achieved from the code generating automation, let us assume that it takes one minute to generate an application and to install it on the service bus.

Differences between the automated and manual implementation in extensive systems, where, for example, a hundred flows need to be generated for each of the mediation types
described, are summarised in Table 6. The minimum achieved in one of the earlier attempts has been taken as the time of manual implementation.

<table>
<thead>
<tr>
<th>Mediation type</th>
<th>Implementation of 100 integration flows</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Automated</td>
<td>Manual implementation</td>
</tr>
<tr>
<td>Message channel</td>
<td>1h 40min</td>
<td>38h 20min (almost 5 days of work)</td>
</tr>
<tr>
<td>Content enricher</td>
<td>1h 40min</td>
<td>48h 20min (more than 6 days of work)</td>
</tr>
<tr>
<td>Publish-subscribe</td>
<td>1h 40min</td>
<td>66h 40min (more than 8 days of work)</td>
</tr>
</tbody>
</table>

Moreover, it should be emphasised that in real integration flows there may be many actions (e.g. flow in Figure 2). Therefore, the benefits of automation of implementing integration flows, in terms of implementation time, can be even greater. An analysis of the data reveals that automation of the integration flows implementation is a matter of great importance in complex systems.

Furthermore, application of the transformation, although only for very simple flows, limits the number of necessary tools, it eliminates WID, which reduces the cost of procurement of IT tools for the project team.

11. Conclusions and further work

As was mentioned previously, analysis of literature reveals, among others, predominance of top-down transformation in software development activities, in-existence of consolidated methods and significant percentage of works without tool support. As far as the paper is concerned, the author proposes top-down transformation IntegrationFlow-to-Java, placed within the ‘1+5’ views model and with tool support in IBM Rational Software Architect. Model-to-code transformation IntegrationFlow-to-Java automates the implementation of integration flows applications for selected mediation patterns: <<Channel>>, <<PublishSubscribe>>, <<ContentEnricher>>. The transformation has been incorporated as a plug-in into the IBM Rational Software Architect (RSA). The RSA plug-in generates complete, fully operational, Java EE application of integration flow from mediation flow diagram. Thus eliminates design and programming stage in WebSphere Integration Developer which reduces development time and costs of licenses. The IntegrationFlow-to-Java transformation offers an opportunity to reduce the duration time of the integration flows implementation forty times (with one hundred flows to be implemented). It should be emphasised that the example shows application of the transformation on a very simple flow, containing only one action with <<Channel>> stereotype. Further work will encompass extending the transformation for generating complex flows. Moreover, it should be emphasised that in real integration flows there may be many actions (e.g. flow in Figure 2). Therefore, the benefits of automation of implementing integration flows, in terms of implementation time, can be even greater. Furthermore, application of the transformation, although only for very simple flows, limits the number of necessary tools, it eliminates WID, which reduces the cost of procurement of IT tools for the project team. The outcomes support the significance of using transformations when designing complex IT systems, especially when integration solutions are developed.
The transformation proposed here is capable of generating three mediation patterns. Improvements in the following areas are planned in order to enhance its efficiency:

- Class-based generation – the generation has been performed based on a file template. The generator needs to be expandable, therefore, automation should be based on objects. For each file required for the flow to take place there should be a corresponding class generating an XML structure,
- Design generation – in addition to generating an operable application, the solution should allow to generate a mediation module design, which might be imported in WID,
- Additional functionalities such as logging messages in the ESB environment,
- Supporting other protocols – the user performing the automation should be able to choose some other protocol than SOAP,
- RSA profile – the more information the activity diagram would be able to provide, the greater number of patterns could be implemented.

The transformation generates Java EE application on IBM Enterprise Service Bus (now part of IBM WebSphere Application Server). Further work will encompass generating from one UML model, mediation flow diagram, in RSA complete application flow in: BPEL (OpenESB), Java DSL (Apache Servicemix) and Mule flow (Mule ESB). Constructing a bidirectional transformation between UML model and BPEL is also within the range of further studies.

References


