A DSL for Enterprise Application Integration

Rafael Z. Frantz

Universidade Regional do Noroeste do Estado do Rio Grande do Sul,
São Francisco, 501. Ijuí, RS 98700-000, Brazil
E-mail: rzfrantz@unijui.edu.br

Abstract: Enterprise Application Integration is one of the big challenges for Software Engineering. According to a recent report published by IBM, for each US dollar spent on developing an application, companies usually spend from 5 up to 20 times more to integrate it. In this paper we propose a Domain Specific Language (DSL) for designing application integration solutions. Contrarily to Apache Camel, our DSL proposal allows to design an integration solution visually, by working with building blocks at a higher level of abstraction, to create a reusable, well documented and independent of technology/platform solutions.

Keywords: building blocks; DSL; domain specific language; EAI; enterprise application integration; EIP; enterprise integration patterns.

Reference to this paper should be made as follows: Frantz, R.Z. (xxxx) 'A DSL for Enterprise Application Integration', *Int. J. Computer Applications in Technology*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Rafael Z. Frantz got his Degree from Universidade Regional do Noroeste do Estado do Rio Grande do Sul (UNIJUÍ) in 2002. During his graduation course he worked at the same university at the Software Development Department developing internet programs. Little after he started working for TargetTrust LTDA, where he was an instructor on Java technology. Currently, he is working on his PhD, which focuses on Enterprise Application Integration.

1 Introduction

Nowadays, many companies run a large number of applications in a distributed environment to carry out their business. These applications are often software packages purchased from third parties, specially tailored to a specific problem, or legacy systems. In this environment often a business process have to be supported by two or more applications. In our experience, it is common that these applications are not prepared to interact among themselves, automatically. This usually happens when at least one application that is part of the process was not designed taking into account integration. Thus knowing the different applications, entering and carrying data from one to another and executing functionalities in each separately, is user responsibility. Is also very frequent the need to add new features to the existing applications, which in many cases may be prohibitive. So, in this case, there are two possibilities: to develop a new application with all the current functions and add the new desired or develop another only with the new features and integrate them. The first option is usually very expensive, the second requires designing an integration solution that should provides the user with a high level view with which he or she can interact.

When we speak about integration, we must consider some restrictions to an integration solution be viable for companies. The first restriction is that after making the integration, applications involved should not change. Changing one of these applications could seriously affect or even completely invalidate the integration solution. According to a recent report by Weiss (2005), for each dollar spent with the development of an application, the cost to integrate it is 5–20 times more. The other restriction is that, after integrated, applications should be kept decoupled, as they were before. The integration solution should not change applications generating dependencies that did not exist before. Finally we can add a third restriction whereby the activity for integration should not be made as part of the developing process, but only when needed.

An integration solution is composed of wrapper(s) and one integration cloud. A wrapper is responsible for connecting an application to the integration cloud and also have a decorator that documents what enterprise application (and layer) are being integrated, see Figure 1. The integration cloud contains processes that can execute one ore more integration tasks and also communicate with another building block through ports and integration links. The integration solution may/may not offers an integration interface that provides a set of ports (entry and exit ports) and besides a functional Application Programming Interface (API) through the one with which other integration solution(s) and/or application(s) can interact.

The integration solution can integrate either data or functionality. It is operational when it represents a functional view, with a data flow and a low-level API for integration. This view is called Enterprise Application Integration (EAI). If the view represents a global data schema providing a high level API to query the integration solution, it is called Enterprise Information Integration (EII).

There are a number of companies that provide software products for designing EAI solutions. Some of those products may be classified as Integration Hubs, Enterprise Service Bus (ESB) or Integration Frameworks. An integration hub usually offers the possibility of building integration flow(s), by means of translators, wrappers and routers. The ESB consists of a larger software infrastructure that allows the companies to realise their own enterprise integration/service bus. This kind of product usually offers (apart from translators, wrappers and routers) an authentication control, a security mechanism, a message transport technology (JMS, MSMQ, MQ Series, ...), works with standards (XML, SOA, ...), etc. Frameworks are a little bit different because they normally consist of an API that should be used by programs. Most such programs are deployed inside a Container, e.g., Spring or as stand-alone applications.

In this paper we introduce our Domain Specific Language (DSL) proposal for EAI. A DSL is a language that can be used to design reusable solutions with a high level of abstraction. It means that during the design we do not need to know the technological details for the solution, latter the DSL model must be 'enriched' with the technological details of the platform we chose to deploy the whole solution. But the description of this latter activity is not an aim of this paper.

The remainder is organised as follows: Section 2 introduces our DSL model and the core elements of it; Section 3 presents an example of an integration solution designed using this DSL; Section 4 makes a comparison between our proposal and the Apache Camel framework for application integration; Section 5 presents our conclusions.

2 The DSL model

An integration solution usually contains in its integration cloud, at least, one flow that integrates one application with another. We call this flow, 'integration flow'. It is basically responsible for transporting messages, but can also translate, enrich, filter or route them. This flow is built, essentially, using four elements from the DSL model: processes, tasks, ports and slots. While a process and a task are considered as 'process units' in the flow, port and slot are used to connect them. In Figure 1, we present these core elements of the DSL model, some specialisation types and their relation. Below, we give an explanation for all of them and also introduce some task types; most of them were inspired in the integration patterns collected and documented by Hohpe and Woolf (2003).

Building block. This is one of the most important element in the model, since that it represents a general construction block used to design an integration solution. A building block has, at least, one task, that is a simple integration task (SimpleTask) or a more complex task (CompositeTask). It means that a building block acts as a container for task(s). Besides, it usually has an entry and an exit port to receive and send messages. There is no limit to the number of ports.

Wrappers are used to connect an application to an integration solution. In our DSL model a wrapper is a building block, with its internal task(s) used to access the application being integrated, plus a decorator. A decorator is composed of an icon of an application and a glyph to represent what layer of the application we are integrating (database, user interface, gateway, etc.), cf. Figure 2.

Figure 1 Core elements of the DSL model

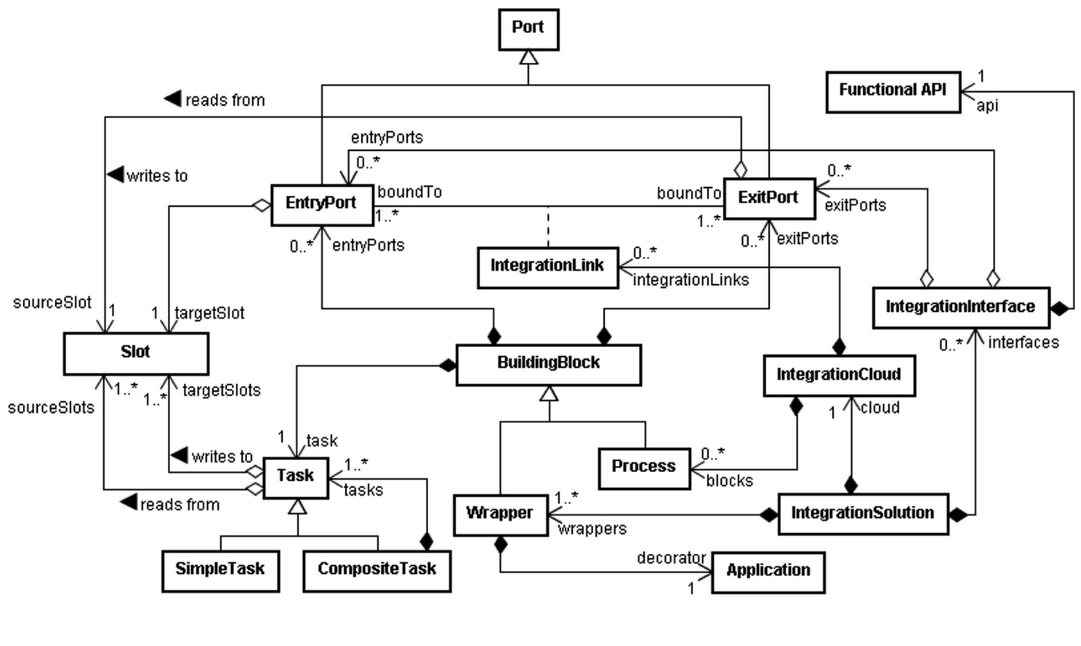
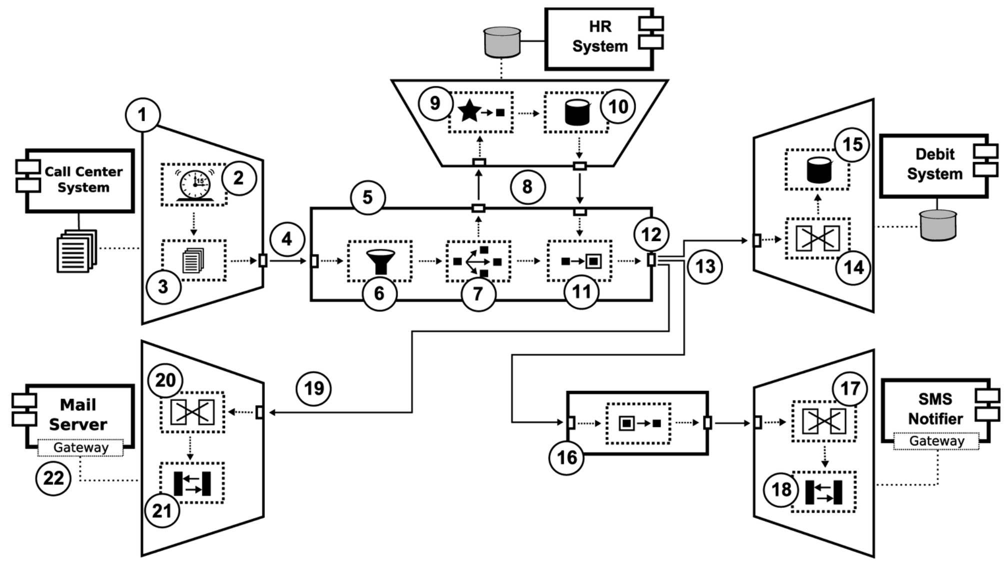


Figure 2 Example of integration solution



The other type of building blocks is processes, which represent blocks used to process integration task(s) across a flow(s). A process contains one ore more tasks and may be connected to another process or wrapper through ports and integration links, cf. Figure 1.

Task. To integrate one application with another we must design an integration flow(s) to transport and/or do some message processing. A task is the element responsible for the process block's internal processing, and, as we can see in Figure 1, every process is composed by one task. A task reads a message from a slot, processes it (according its task type) and writes the result to the next slot, making it available for the next element. This message processing usually consists of translating, filtering, routing, etc. See Section 2.1 for more task types and details.

Simple tasks represent atomic action with a message, otherwise composite task represents a sequence of tasks that can be used to turn a process block into a more complex processing unit.

Slot. Slots are used just inside building blocks in order to allow exchanging messages between ports (entry and exit) and tasks, and also between tasks. Essentially slots are buffers in memory to allow a faster/simpler communication inside a process.

Port. Building blocks have ports through which they can send/receive message(s) to/from another block. An entry port writes an inbound message to an internal slot from which a task can read it. Otherwise an exit port always reads a message from a slot and make it available to the next element(s) in the flow. Entry ports and exit ports are always bound with one another. It happens, for example, when a process block is linked to another process or a wrapper. This relation between ports is represented in Figure 1 with an association called 'IntegrationLink'.

2.1 Task types

We have chosen five groups of task to describe below, they are: Message Constructors, Transformers, Routers, Timing and Interfacing. We also present a pair of a specific task types in each group.

2.1.1 Message constructors

A message constructor, produces new message(s). It means that the message that arrives is not the same message(s) that will continue on the flow. There are two important tasks in this group: aggregator and splitter.

Aggregator. An aggregator is an stateful task that can receive two or more inbound messages and combines their individual content in just one message. Sometimes it may be interesting when we have different messages with individual results that may be combined to be processed as a whole latter.

Splitter. A splitter is the counterpart of an aggregator. This type of task receives an inbound message and produces two or more outbound messages that will be processed individually.

2.1.2 Transformers

Transformers are a group of task that modify the original content of an inbound message. This can be done in several ways, like for example, enriching its content with more information or translating the actual content from one format to another.

Translator. In an integration solution, a very frequent task is translating messages from one format to another. It is necessary because applications that are being integrated usually work with different message formats.

So the translator will receive an inbound message, translate it to the new format and send it to the next element.

Content enricher. Sometimes, it may be necessary to append more information to a message in order to process it latter on in the flow. A content enricher receives an inbound message and computes a new one based on the original message content or data in an external resource.

2.1.3 Routers

Routers are responsible for routing an inbound message to zero, one or more destinations. In this paper we introduce two types of routers: filter and replicator.

Filter. Using a filter we may avoid uninteresting messages from reaching the next task. So a filter task receives an inbound message and based on a certain criteria allows this message to continue or removes it from the flow. All removed messages may be saved in a special integration repository/log in order to keep this information.

Replicator. A replicator task makes copies of an inbound message and sends them to two or more destinations. It does not change the original content and there is no limit for message copies. However the number of messages must be equal to the number of slots that the replicator can write to. This task type should be used, for example, to duplicate a message in order to execute a query in another system and latter aggregate the result of this query with the other copy to distribute copies of the original message for two or more applications.

2.1.4 Timing

There are situations in which we need to have a certain 'control of the time' in an integration flow. This group contains tasks that may, for example, start or delay an execution of a flow.

Timer. Timer is a type of task that we can configure to run (every now and then) frequently, usually in a wrapper. This task instead of receiving inbound message, will produce outbound messages. The message produced is normally used to activate the next task.

Delayer. An integrated application or a process block may get overloaded because it receives more messages it can process, we should add a delayer in front of it to delay the delivering of messages.

2.1.5 Interfacing

To integrate an application we have to design, at least, a wrapper for it. This wrapper is responsible for, among other things, reading information from the application and sending it to the integration solution and/or writing information from the integration solution to the application. Actually, a wrapper interfaces a layer of the application; in most of the cases, it is the database, gateway, a messaging channel (of a messaging system where the application writes messages), or even the user interface (with a scrapper). Below we describe two types of interfacing task: DBDataSource and RPCAccess.

DBDataSource. A database data source is a task to provide access to the database layer, in other words a set of tables. This task, basically, allows reading and writing to the application's database.

RPCAccess. There are applications 'better designed' for integration. These applications usually provide a public gateway that may be used to access its functionalities and even data. A wrapper should contain a RPC access task in order to interact with the gateway. As any other interfacing task, a RPC access will send/receive messages from the integration solution and send/receive data to the application.

3 Integration example

Figure 2 shows an integration solution of five applications, which initially were not designed taking integration into account; the solution gets inspiration from a real system used at UNIJUÍ. They are very different applications and developed with different technologies. The aim of this solution is to make that all phone calls registered by the 'Call Center System' (CCS) in its database and which have some cost for the university, also be registered in the 'Debit System' (DS). In addition to store these calls on the DS, some information from the call (e.g., cost, time of the call, city and number of destination) are sent by SMS and/or email to the user who made the call. At this university, employees who have a personal key with which they can use any phone terminal, in any of the cities where the university is, and make a call. All calls are registered, and at the end of the month the employee has to tick which were work calls and which were private calls. Private phone calls will have to be paid by the employee.

All applications are connected to an integration solution through a wrapper (1). A wrapper contains tasks to interact with the application's interface layer (database, gateway, user interface, etc.), send and/or receive messages from the integration solution. The decorator (22) simply indicates which application and interfacing layer are being integrated/accessed.

There are two types of arrow: solid arrows and dotted arrows. Solid arrow represents integration links while dotted arrow represents slots. As we have already explained, an integration link (4) usually connects two process or a wrapper to a process (and vice versa), and are used to desynchronise these elements in the solution, if necessary.

The integration flow for the solution presented in this example begins in the wrapper of the application CCS, with a timer task (2). This task creates an activation message every two minutes and sends it, through a slot to the next task, a file data source (3). This task is responsible for accessing the files where the application CCS writes the phone calls, creates a message for each call and sends these messages one-by-one to the next slot. The exit port of this wrapper reads each message from this slot and then send them to the integration link (4), making it available for the entry port of the central process block (5).

The central process (5) contains a composite task that starts with a filtering task (6). This task filters out messages that do not have a cost for the university, and allow just tall calls to remain in the flow. Those messages are written to the next slot, and will be read by the next task, a replicator (7). The replicator makes copies of the original message. In this case one copy is sent to the wrapper of the application Human Resource System (HRS) and the other to the next element in the current process. In this integration solution we need to append missing information about the employee to the message, like: name, department, email and mobile phone. These information is in the HRS, that is why it is also integrating our solution. The message copy received by the wrapper of HRS, through the ports (8), will be processed by a custom task (9). This task produces an outbound message that represents a database query to be executed by the database data source (10). After that the content enricher (11) receives the result from the HRS's wrapper and enriches the original message with it. Now the enriched message is sent to the next slot, the one that connects with the exit port (12). This port is also connected with three integration links that allow sending a message copy to DS, SMS and MS. In this case the port acts like a recipient list described by Hohpe and Woolf (2003), distributing message copies.

The first integration flow after the exit port (12) connects the process (5) to the wrapper of the DS application. This wrapper receives the message through its entry port and makes it available for the first internal task of the wrapper, the translator task (14). The translator is responsible for translating the current message format into a new format that the DS can understand, and then immediately writes the message into the slot between the translator and the database data source (15). This task accesses the database of the DS and stores the message.

The second flow connects the same exit port (12) to the other process (16) which have a unique internal task, a slimmer task. A slimmer is responsible for removing some information from the message in order to make it smaller before sending it to the SMS. The SMS is an external application that allows sending messages to mobile phones. In its wrapper, there is a translator (17) that receives the inbound message and translates it into a special format that the SMS can understand. Once the SMS offers a public gateway the interaction can be done by a RPC access (18), that forwards the inbound message.

The last copy of the message goes to the flow (19) that now connects the process (5) with the wrapper of the MS. This wrapper integrates the application allowing the solution to send emails with all the details about the employee's call. As in the other wrappers it is important to translate the inbound message into a message format that the MS can understand. This is done using a translator (20) inside the wrapper, just after the entry port. The translated message now goes, through a slot, to the next task, the RPC access (21), and then to the MS.

4 Comparison with Apache Camel

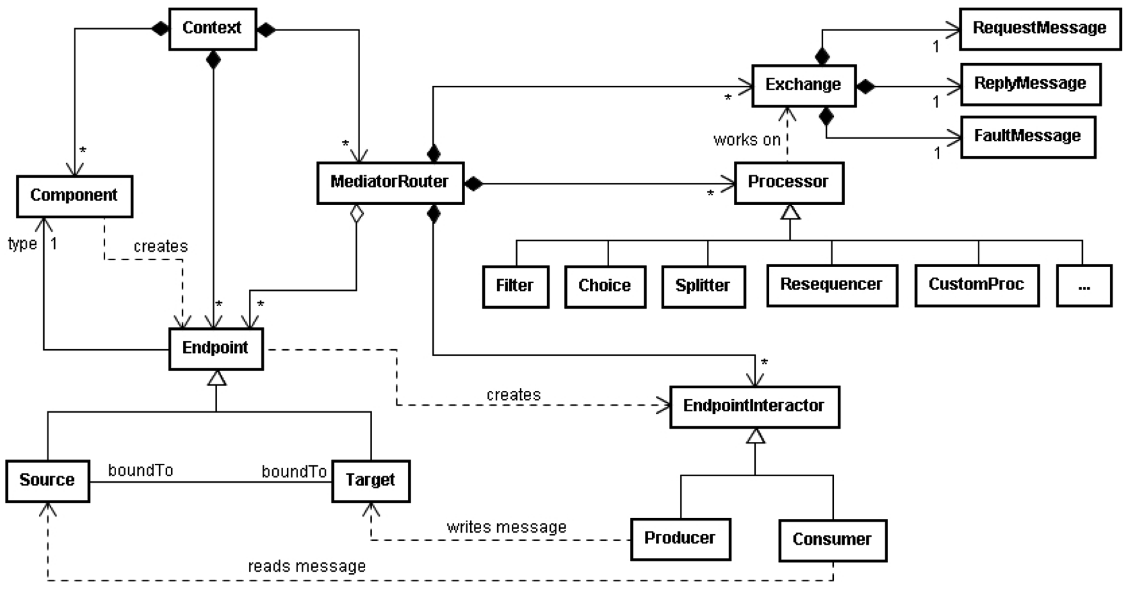
Apache Camel is a Spring based integration framework for building EAI solutions. This framework, as our DSL model, supports a number of the Enterprise Integration Patterns documented by Hohpe and Woolf (2003). However we do believe there are important differences between both proposals. Below we describe some of the differences that we consider more relevant for this paper.

Our DSL model is based on the concept of 'building block', what allows design an integration solution visually by connecting those blocks through their ports and adding integration task(s) to do a specific message processing inside the block. For each element, that is part of our model, we provide an icon that identifies it. For example, every type of task has a graphical representation, also for the process, port, integration link, wrapper, application being integrated and slot (see Figure 2). It means you can work at a high level of abstraction to design the integration solution. Besides applications, integration flow(s) and tasks are clearly documented and self-explanatory in a visual way. On the other hand, Apache Camel's proposal does not have the concept of building block and also does not provide a graphical language for designing an integration solution. In Camel designing an integration solution imposes working directly at code-level. In version 1.2.0, they provide a tool to render 'Camel Routes' (integration flows) in order to have a graphical view of the routes, but it is still a poor visual documentation.

In Camel any application/component that is being integrated (or used in the integration solution) is considered an endpoint. For example, Camel allows using an external application, as for example Velocity, as a message translator task in a flow (cf. Apache-Foundation, 2008 for more details). Because of it we cannot see which are really those applications being integrated and which are those that are just being used to do a special processing task over a message. Also it is difficult to recognise what an endpoint is and what a wrapper is. These documentation must be done externally, whereas in our proposal it is very clear and well documented in the visual design of the integration solution as one can see in Figure 2. But in Camel, since everything is an endpoint, it is easier to add a new task (that can externally process a message) to the flow. It is also possible to dynamically configure those endpoints through an URI, which we have not explored so far, cf. Apache-Foundation (2008).

As we already know, an integration flow usually connects one or more applications. It is important to know (and have documented) in an integration solution where the flow starts, which are the integration tasks that are executed across this flow, and where it ends. As we can see, in Figure 2, in our DSL model the integration flow

Figure 3 Partial model of Camel



always starts in a wrapper and also ends in a wrapper. There is no central element to create the flow. Each process executes one or more task(s) over a message and forwards it to the next element. The flow is clearly documented graphically. On the other hand, in Camel, there is a central element (Mediator Router) to create the integration flow (see Figure 3). It connects a source endpoint (where the flow starts) to a target endpoint (where the flow ends) through the flow. Across this flow one ore more processors (filter, choice, splitter, another endpoint that represents an external application for executing a task over a message, etc.) may be executed. After executing, the processor is responsible for calling the next one.

Concerning the communication between tasks in Camel, tasks have just one entry/exit, except for the routers that may have more than one exit. In our DSL model, all building blocks may have more than one entry/exit port (see Figure 2, number 5); a task may also read/write to one or more slots (see Figure 2, numbers 7 and 11). It allows, for example, having a task that can do a processing with messages from two or more different sources. Or, for example, a task to make a copy of the original message, send one copy to another application that should return information to latter enrich the other copy (see Figure 2, numbers 7–11). As we know, this query for information may take a lot of time, so the thread that is executing the process (see Figure 2, number 5) can be stopped and another thread may run while the first process waits it. On the other hand, in Camel, consulting an external resource would block the current thread (for all the integration flow) until it receives the result.

Sometimes, we have to execute part of the integration flow (one or more task) in different machines on the network. It means the flow's execution is distributed and then its integration tasks needs to communicate with each other to forward the message. In Camel, all the integration flow is executed in memory, it means in the same machine. To distribute this flow in Camel we have to create special 'small applications' for each part of the flow that should be executed in a different machine and then integrate them as an endpoint. The whole flow is stopped as long as the external task in executing. We need external documentation to indicate the endpoint's intention, in order to know which endpoints represent the original applications being integrated and witch represent the applications specifically created to execute the task(s) of the flow. In our DSL model, we provide the following elements to solve this problem: building block and integration link. Remind that an integration task is executed inside a building block (see Figure 2, number 4). Building blocks have, among other properties, deployment properties (please have in mind we are designing the DSL model, we do not have to worry about this deployment properties). Later, this allows choosing where and how the block should be deployed, for example, as a web service in a Java Web Container or as a stand-alone application in a certain machine, always identified by an URI. While building blocks contain integration task(s) the integration link (see Figure 2, number 3) is the element that give us all the flexibility to design the integration solution without having to concern (at design time) if the building blocks should or should not be executed in different machines and how to distribute them. Integration links have properties that allow us to choose how they should be deployed and how they will connect the building blocks in an integration flow. An integration link can be deployed in memory (like Camel), in a messaging system's channel, in a database, or in a file for instance.

5 Conclusion

Application integration is a growing up activity in companies and, according to the report published by Weiss (2005), is very expensive, apart from that demands much more resources than the regular software development process. Knowing these, it is very important to have engineering technologies (languages, tools, frameworks, etc.) that can support this activity helping to reduce the cost and resources usually spent in. The DSL model proposal and the Apache Camel framework presented in this paper contribute with it once they are both proposals to realise EAI besides based in the notorious patterns from the reference book by Hohpe and Woolf (2003).

Our proposal is based in the concept of building block, what allows to design an integration solution visually by working at a higher level of abstraction, creating reusable, well documented and independent of technology/platform solutions. On the other side Camel's framework is a low level option to design integration solution by using the framework inside applications and coding the solution. However Camel already provides an implementation of the framework, which we do not provide yet. We do believe that defining a DSL model that can be used to design and latter generate executable solutions is a good way to minimise the costs, time and the need for extra resources in the integration activity.

Acknowledgement

Supported by the Evangelischer Entwicklungsdienst e.V. (EED)

References

Apache-Foundation (2008) *Camel Book in One Page*, Freely available at http://activemq.apache.org/camel/book-in-one-page.html

Hohpe, G. and Woolf, B. (2003) *Enterprise Integration Patterns–Designing, Building, and Deploying Messaging Solutions*, The Addison Wesley Signature Seies, Addison-Wesley, Boston.

Weiss, J. (2005) *Aligning Relationships: Optimizing the Value of Strategic Outsourcing*, Global Service Report, IBM.