Advances in a DSL for Application Integration*

Rafael Z. Frantz[1], Rafael Corchuelo[2], Jesús González[3]

[1] Universidade Regional do Noroeste do Estado do Rio Grande do Sul (Unijuí)
São Francisco, 501. Ijuí 98700-000RS (Brasil)
`rzfrantz@unijui.edu.br`
[2] Universidad de Sevilla, ETSI Informática
Avda. Reina Mercedes, s/n. Sevilla 41012
`corchu@us.es`
[3] Intelligent Dialogue Systems, S.L. (INDISYS)
Edificio CREA, Avda. José Galán Merino, s/n. Sevilla 41015
`j.gonzalez@indisys.es`

Abstract. Enterprise Application Integration (EAI) is currently one of the big challenges for Software Engineering. According to a recent report, for each dollar spent on developing an application, companies usually spend from 5 to 20 dollars to integrate it. In this paper, we propose a Domain Specific Language (DSL) for designing application integration solutions. It builds on our experience on two real-world integration projects.

1 Introduction

Nowadays, many companies run a large number of applications in a distributed environment to carry out their businesses. These applications are often software packages purchased from third parties, specially tailored to solve a specific problem, or legacy systems. In this environment often a business process has to be supported by two or more applications. In our experience, it is common that these applications are not prepared to interact among themselves. This usually happens when at least one application that is part of the process was not designed taking integration into account. Worse than that, it is very common that the applications have been developed using very different technologies and platforms. Such systems are commonly referred to as software eco-systems [10].

In such software eco-systems it is common that entering and carrying data from an application to another and executing functionalities is user's responsibility. Also, is very frequent the need to add new features to the existing applications, which may be prohibitive. So, in this case, there are two possibilities: to develop a new application with all the current functions and then add the new desired functions to it, or to develop another application only with the new features and integrate them all. The first choice is usually very expensive; the second requires designing an integration solution that should provide the user with a high-level view of the problem. According to a recent report by IBM, for each dollar spent with the development of an application, the cost to integrate it is from 5 to 20 times more expensive [15]. These figures make it clear that integrating business applications is quite a serious challenge.

* Partially funded by the Spanish National R&D&I Plan under grant TIN2007-64119, and the Andalusian Local Government under grant P07-TIC-02602. The work by R.Z. Frantz was funded by the Evangelischer Entwicklungsdienst e.V. (EED). We are deeply indebted to Abdul Sultán from Sytia Informática, S.L., Hassan A. Sleiman, Raúl Sánchez, and Francisco J. Dominguez from Indevia Solutions, S.L.L., for their extremely helpful collaboration to implement the DSL described in this paper.

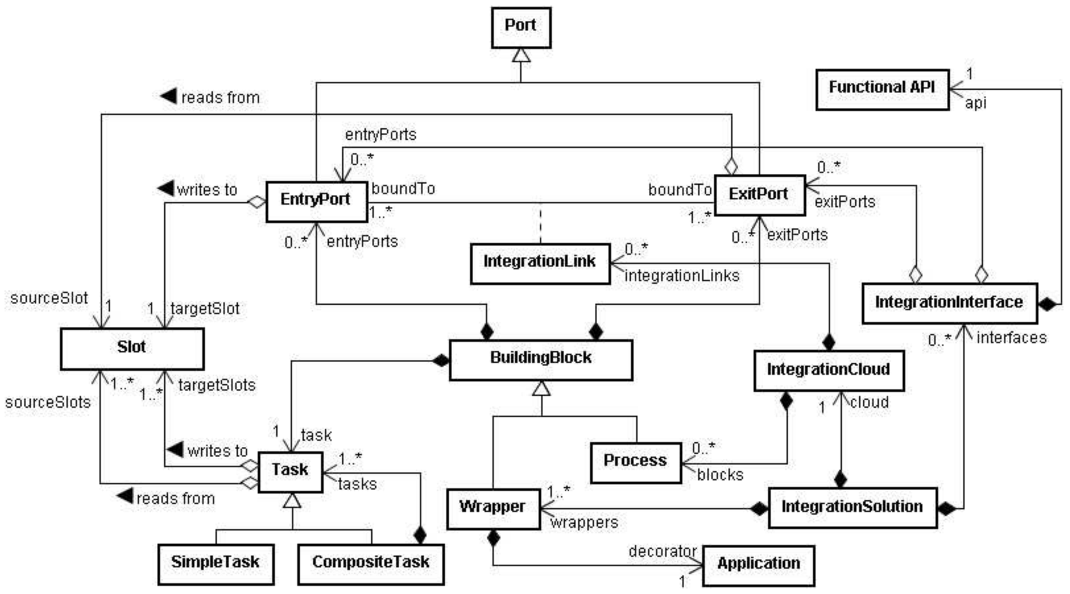

Fig. 1. Core elements of the DSL model.

When considering integration, we must consider some constraints so that a solution is viable for a company [9]. The first constraint is that integrated applications should not change, since a change might seriously affect or even break down other business processes. The other constraint is that, after integrated, the applications should be kept decoupled, as they were before. The integration solution should not change the applications and introduce dependencies that did not exist before, but just coordinate them in an exogenous way by means of building blocks. There is another constraint: integration must be performed on demand, as new business requirements emerge and require new services to be created building on the existing applications [2].

Enterprise Service Buses (ESBs) range among the most usual tools used to devise and implement integration solutions [4,5,6,12,17]. Such tools commonly rely on the well-known Pipes&Filters architectural pattern [7], according to which an integration solution is designed as a flow of messages through a series of filters that communicate by means of pipes. It is not surprising then that there are many proposals for Domain Specific Languages (DSLs) that help engineers to design both pipes and filters, within the context of ESBs. In this paper, we introduce a DSL to design filters that builds on our conclusions working on the design of two real-world projects in quite different scenarios. The remainder is organized as follows: Section 2 introduces the core elements of our DSL model; Section 3 reports two real-world integration scenarios used to validate our proposal; Section 4 compares our proposal to others; finally, Section 5 presents our conclusions.

2 The DSL Model

In Figure 1, we present the main ingredients of our DSL and their relationships. Roughly speaking, an integration solution comprises at least one flow that integrates one application with another; we refer to such flows as integration flows. They are responsible for transporting messages, but can also translate, enrich, filter or route them. Flows are built from four elements: processes, tasks, ports and slots. Processes and tasks are considered process units in the flows; contrarily, port and slots connection units.

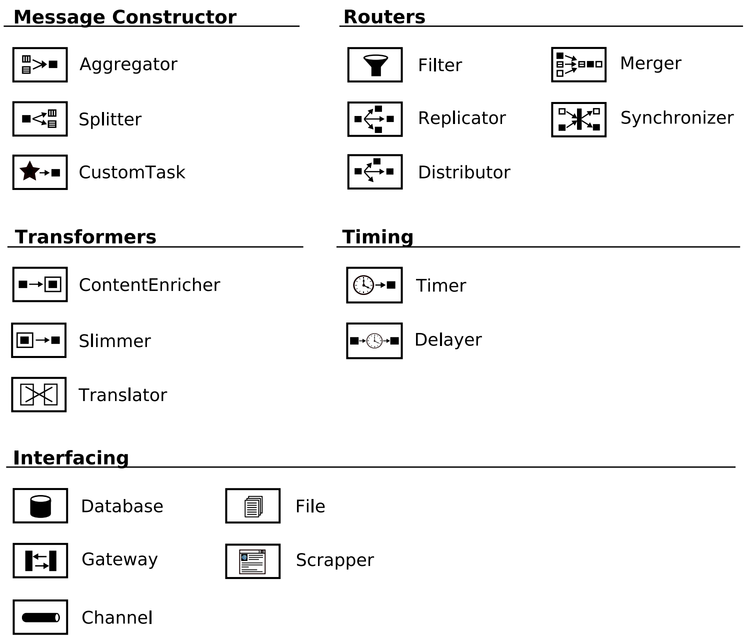

Fig. 2. Task taxonomy.

Below, we provide a short description of the main elements of our DSL.

Building Blocks: This is one of the most important elements in the model, since it represents a general construction block used to design an integration solution. A building block has, exactly, one task, that is a simple integration task (simple task) or a more complex task (composite task). Building blocks can be either wrappers or processes. 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 the layer being integrated is. Processes allow to implement integration-specific services across a flow. They contain one or more tasks and may connect to other processes or wrappers through ports and integration links.

Tasks: A task is the element responsible for performing a simple, atomic step within a building block. Note that every process and wrapper are composed of exactly one task, which is commonly a composite task that has other inner tasks. A task reads a message from a slot, processes it, and writes the result to the next slot.

Slots: Slots are used inside building blocks to allow exchanging messages between ports and tasks, and also between tasks. Essentially slots are in-memory buffers that allow tasks, of which a building block is composed, to communicate asynchronously.

Ports: Building blocks have ports through which they can send/receive message(s) to/from another building block. An entry port writes an inbound message to an internal slot from which a task can read it. On the contrary, an exit port always reads a message from a slot and makes it available to the next port in the flow. Entry ports and exit ports are always bound with one another. It happens, e.g., 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 integration link.

Our DSL provides five types of tasks, namely: Message constructors, transformers, routers, timing and interfacing tasks, cf. Figure 2 and [9]. Below, we report on them:

Message Constructors: A message constructor creates new messages. There are three important tasks in this group: aggregator, splitter and custom task. 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. 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. A custom task allows users to create a message using custom code. Sometimes it may be interesting when from an inbound message we want to create a complete different message, like e.g. a message to query a database and latter enrich the original message with the result.

Transformers: Transformers modify the original content of an inbound message. There are three important tasks in this group: content enricher, slimmer and translator. 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. On the other hand, a slimmer may reduce the message size by removing part of its content. When considering 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; thus, a translator receives an inbound message, translates it to the new format and send it to the next task.

Routers: Routers are responsible for routing an inbound message to zero, one or more destinations, and here we focus on filter, replicator, distributor, merger and synchronizer. Using a filter we may avoid uninteresting messages from reaching the next task; a filter task receives an inbound message and based on a certain criteria allows this message to continue or removes it from the flow. A replicator task makes copies of an inbound message and sends them to two or more destinations; it does not change the original contents and there is no limit for message copies; however the number of messages must be equal to the number of slots to which the replicator can write; this task type should be used, e.g., 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. A distributor routes an inbound message to zero, one or more destinations. Note that, in the worst case, a distributor may behave exactly as if it was a replicator. Sometimes it may be necessary to merge messages from two or more slots into just one, like e.g. when the next task just can read from a single slot. To perform this, merger task can be used. Finally, when a task needs to receive a group of two or more messages that must follow a given pattern, and this pattern may take time to be fulfilled, a synchronizer task can be used. In this case the synchronizer permanently checks its entry slots and when a message fulfils the pattern, all messages of the group are forwarded. It is common in those cases where a message is used to create another query message to query an external resource and then both messages (the resulting message and the original message) need to be forwarded together to be processed by another task, like e.g. an aggregator task. Note that, the number of entry and exit slots for a synchronizer must be the exactly the same.

Timing: This group includes timers and delayers. A timer is a type of task that we can configure so that it produces an outbound message at fixed times. A delayer, on the contrary, is used to delay delivering a message for a fixed number of seconds; it may be used, for instance, to avoid flooding a process that runs on a slow machine.

Interfacing: To integrate an application we need to design at least one wrapper that must be responsible for reading information from the application and sending it to the integration solution and/or writing information from the integration solution to the appli-

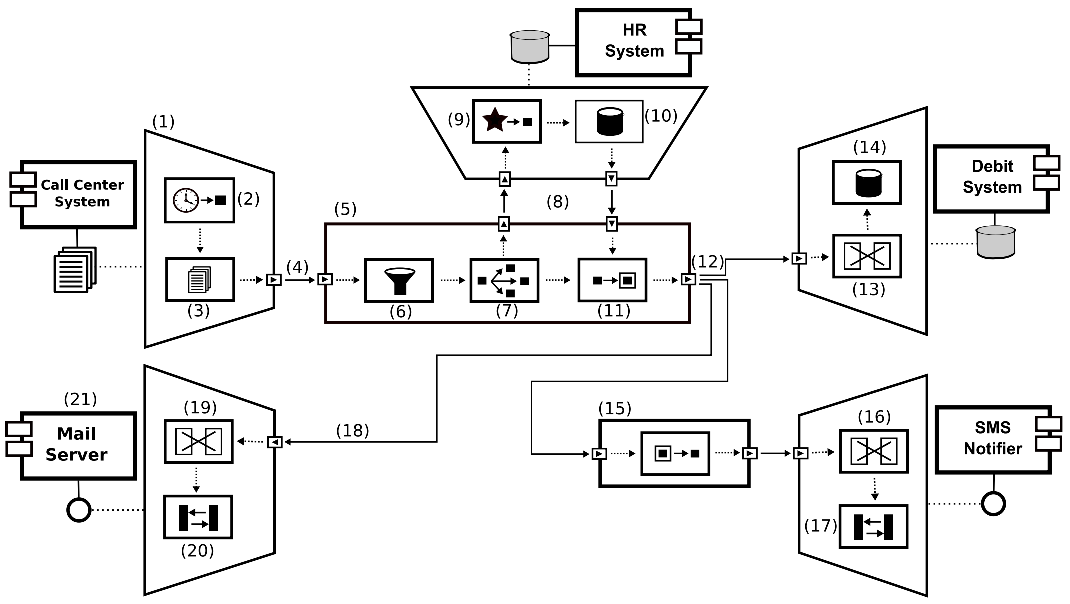

Fig. 3. Unijuí's integration solution.

cation. Actually, a wrapper interfaces a layer of the application; in most of the cases, it is a database, gateway, a messaging channel, file or even the user interface (with a scrapper [1]). Interfacing tasks relieve the programmer from the burden of performing such low level operations.

3 Integration scenarios

We have validated our proposal working on the design of two real-world integration projects. The former provides an effective solution to automate the invoicing of phone calls at Unijuí; the latter provides an intelligent interface to a number of information servers owned by a public institution. Below, we report on the details.

3.1 Call System at Unijuí

Unijuí has five applications involved in a hand-crafted process whose goal is to invoice their employees of the private phone calls they make using the University's phones. Each application runs on a different platform and was designed without integration concerns in mind. There is a Call Center System (CCS) that records every call every employee makes from one of the telephones this university provides to them. Every month, an analysis is performed to find out what calls have a cost and are not related to the work activities of the employees; such calls are debited to the employees by using a Debit System (DS). There is also a Human Resources System (HRS) that provides personal information about the employees, including their names, phone numbers, social security numbers, and so forth. There are two additional systems to send mail or SMS messages. The goal of the integration solution is to automate the invoicing of the calls that an employee makes and are not related to his or her work activities.

Figure 3 depicts our integration solution. The applications are connected to an integration solution through wrappers (1). A wrapper contains tasks to interact with the application's interface layer (database, gateway, user interface, and so on), send and/or receive messages from the integration solution. The decorator (21) simply indicates which application and interfacing layer are being integrated/accessed.

The integration flow for the solution presented in this example begins in the wrapper of 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, which is a file interfacing task (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 them to the next slot one-by-one. The exit port of this wrapper reads each message from this slot and then sends it to the integration link (4), making it available for the entry port of the central process block (5). This process 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 toll 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, e-mail and mobile phone. This 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 to three integration links that allow sending a message copy to DS, SMS and MS.

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 (13). 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 (14). 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 (15) 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 (16) 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 (17), that forwards the inbound message.

The last copy of the message goes to the flow (18) that now connects the process (5) with the wrapper of the MS. This wrapper integrates the application allowing the solution to send e-mails 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 (19) inside the wrapper, just after the entry port. The translated message now goes, through a slot, to the next task, the RPC access (20), and then to the MS.

3.2 Job Advisor for a Public Institution

Indisys is a spin-off of the University of Seville that works on the development of interactive virtual assistants (IVAs). Generally speaking, an IVA is an application that allows a user

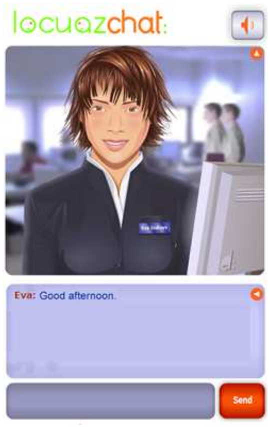

Fig. 4. A screenshot of Indisys's interactive virtual assistant.

to talk to a computer system using natural language through a rich web interface. Such applications are starting to sprout out since they help call centers to be more effective; the applications range from answering general user information queries to on-line vending services, e.g., flight reservations, cinema tickets, and so on. Since IVAs are quite a new field from a commercial point of view, their features are quite heterogeneous. Our motivating example builds on the development of an IVA whose user interface is presented in Figure 4, which is a job advisor for a public institution.

The modules this system integrates are as follows:

NLU: This is the Natural Language Understanding module, and it is responsible for translating natural language sentences like "Where can I find a job?" into semantically rich structures that are machine understandable.

NLG: This is the Natural Language Generation, which is complementary to the NLU, that is, it transforms computer structures into natural language sentences.

TTS: This module is responsible for translating textual sentences in natural language into voice.

KM: This is the Knowledge Manager module, which is a facade to external information sources to which the system can connect by means of a plethora or communications adapters, including database drivers, web services, and so on.

NLU, NLG, and TTS are legacy systems, i.e., systems that were not designed together for this project but must be reused as they are; designing new modules that are better prepared to be integrated was considered unaffordable so they have to be reused as is. Modules, NLU and NLG are owned by Indisys, so we can have access to them freely. Contrarily, module TTS was provided by a third party, and it is proprietary. The KM module is being developed by another company.

Figure 5 depicts the complete integration solution, which is composed of process CORE, and five wrappers to the existing modules. CORE is the central process since it is responsible for coordinating the activities of the applications being integrated. It is also the cornerstone of our IVA, which just provides a user interface to this process.

Note that modules KM and TTS provide a programmatic interface; so their wrappers are the simplest ones since they just require a gateway interfacing task to integrate them (1 and 13). The wrapper to the web client is referred to as User Interface in Figure 5. It is responsible for gathering user input by means of a gateway interfacing task; however, producing an output is far more complex since this requires a replicator task (3) it is first

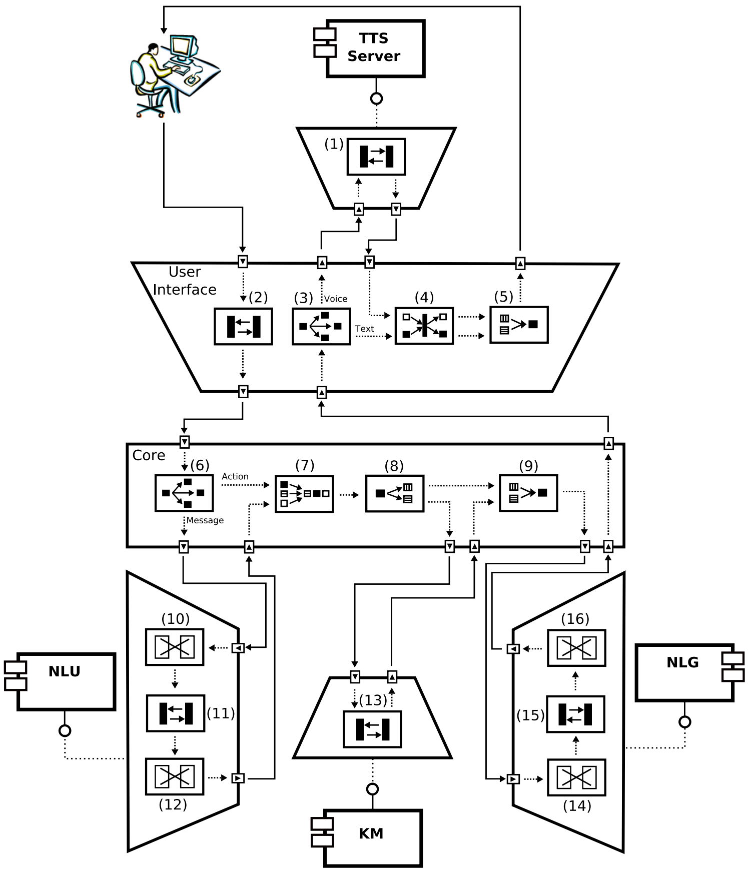

Fig. 5. Indisys's integration solution.

necessary to decide if it is necessary to use the TTS module to synthesise a voice message in response to the client's input (note that the client may switch the voice off, in which case it does not make sense to waste TTS resources on synthesising a stream of voice that is going to be discarded by the client). Next, it is necessary to use synchronizer (4) and an aggregator (5) to merge both the textual and the voice response, if any, before the results are sent to the user.

Process CORE must initially determine if the client's input is a sentence in natural language or just a click on the user interface. It must use the NLU module only in the first case. Again, this is implemented using a replicator (6) that is followed by a merger (7). Later, it is checked if it is necessary to have access to external information sources, in which case it is necessary to separate the internal structure the CORE process handles into a number of messages to request to the KM the information requested by the user. This is accomplished by using a splitter (8). Later, the information returned by the KM module is merged using an aggregator (9). In the end, the whole semantic structured is passed on to the NLG module to generate the appropriate output in natural language.

Last, but not least, the wrappers to NLU and NLG require a number of translators (10, 12, 14, and 16) to deal with a number of problems related to the fact that they are actual legacy modules, in addition to their respective gateway interfacing tasks.

4 Related Work

We compare our proposal to Camel [6], Spring Integration [12], and BizTalk 2006 [17] since they also provide DSLs for building integration filters. Tools such as Mule [5] or ServiceMix [2] are also related, but their focus is on the design of pipes, which is not ours in this paper. We have built a comparison framework out of a number of 47 objective properties that are classified into scope, modelling, and technical properties. Below we describe five properties of each group and also compare these properties with the chosen tools and our proposal.

4.1 Scope Properties

Property	**Camel**	**Spring Int.**	**BizTalk**	**Ours**
Architectural pattern	Filters	Filters	Pipes/Filters	Pipes/Filters
Context	EAI	EAI	EAI/B2BI	EAI/B2BI
Abstraction level	PSM	PSM	PSM	PIM/PSM
Transactions	ST-F	–	ST-F/LT-F	ST-F/LT-F/ST-S/LT-S
Kind of model	O/D-IoC	O/D-IoC	D-Graph. and XML	D-Graph. and XML

Table 1. Scope properties.

Scope properties (cf. Table 1) represent properties whose absence can greatly hinder or even invalidate a proposal, and to provide them it is necessary to implement extensions whose the cost for developing we believe may be prohibitive in most cases.

Architectural Pattern: As we already know, Pipes&Filters is the standard for excellence in our field of interest. Therefore, it seems reasonable to expect that the tools for building ESBs provide domain specific language for designing both pipelines as filters.

Context: Normally we discern among the following integration contexts: Enterprise Application Integration (EAI), where the emphasis is on integrating applications aiming to synchronize their data or implement new functionalities. Enterprise Information Integration (EII), whose emphasis is on providing a live-view of the data handled by the integrated applications; Extract, Transform, and Load (ETL), which intends to provide materialized views of the data on which we can apply knowledge extraction techniques [16]. In all previous cases, we implicitly assume that the integrated applications belong to the same organization. Lately the activity of integrating applications from different organizations are requiring more attention, aiming to implement inter-organizational business processes; this context is known as Business to Business Integration (B2BI). Also recently, the so-called mashups are requiring attention. They are applications that usually run on a web browser and integrate data from various sources.

Abstraction level: Working with platform independent models (PIM) allows to design stable solutions as independent as it is possible from the technology used to implement them, and its inevitable evolution. By working with PIM models, it is also important to be supported by tools that can transform them into a platform specific model in which we want to run.

Transactions: Transactions allow to design robust solutions capable of facing the failures that may occur during the execution of an integration solution. It is usual to discern between transactions in the short term (ST) and long term (LT): The former usually is implemented using the known protocol Two-Phase Commit that basically consists of carrying out those actions that require state changes only when all parts involved have confirmed that can carry them out without any problem [11]; the latter, executes the actions when they are possible and, in case some of them fails, then executes compensation actions aiming to undo the effects of the previous actions, or in case that this is not possible anymore, try to lessen their impact. In the context of application integration it is also useful to classify transactions depending on its scope, resulting in filter transactions (F) or solution transactions (S): the first are those which guarantee that a filter achieves its goal, otherwise any change that has been done up to the moment is invalidated; the second are those that ensure this property for the whole integration solution.

Kind of model: According to the kind of model of the tool, it allows to design integration solutions operatively (O) or declaratively (D). When the design is operative, the tool provides a library that developers can use to create their integration solutions; on the contrary, when declarative developers can work at a higher level of abstraction. The first way is by using a domain specific language with graphic or XML representation, that will be automatically translated into executable code; the second is using XML configuration files, which are then interpreted by an Inversion of Control (IoC) [8] engine, e.g., Spring [14].

Our proposal allows for both the design of pipes and filters, although in this paper we have focused on filters only. We also make a clear distinction between platform independent models and platform specific models; the latter are generated automatically by means of a model transformer that builds on DSL Tools [3]. Regarding transactions, we support all types of transactions, which is partially due to the fact that we can track messages and allow for compensation actions to be defined at each building block.

4.2 Modelling Properties

Property	**Camel**	**Spring Int.**	**BizTalk**	**Ours**
Card. of tasks	$1 : N$	$1 : N$	$1 : 1$	$N : M$
Card. of locations	$N : M$-Comp	$N : M$-Comp	$1 : 1$	$N : M$-Comp/Repl
Correlation	No	No	Yes	Yes
Port adapters	$1 : 1$	$1 : 1$	$N : 1$	$N : M$
Stateful filters	No	No	No	Yes

Table 2. Modelling properties.

The modelling properties (cf. Table 2) are not as critical as the previous ones, and that if lack of them is still possible to design a solution for effective integration at a reasonable cost, but perhaps the design is much more complex and less intuitive. Obviously, this can result in a negative effect on the subsequent maintenance.

Cardinality of tasks: The filters are made up of simpler tasks that allow to build messages, transform them, route them or interact with pipes. In our experience, it is usual that some tasks require contextual information from any external source, to deal with messages that arrive; a clear example is when a message to be processed includes some identifier and it is necessary to query an application to know which data are related with this identifier.

Cardinality of locations: The term location refers to the physical location on which a pipe is implemented, e.g., a folder in a file system or a mail server. Generally it is interesting to discern between two types of reading: with competition (Comp), in which only one of the filters can read at a time from a particular location or replication (Repl), in which all filters can read at the same time.

Message correlation: Stated that we can not assume synchronicity among the integrated applications, is very common for messages to arrive out of sequence in the filters, so it is the filter's responsibility to correlate them in such a way that those complementary messages always must be processed together. This need is more bounden in cases where it is possible to create filters or tasks with multiple entries.

Port adapters: The possibility of having multiple adapters in a port, allows it to receive/send information from/to two or more locations. The binding of a port with multiple locations helps keeping the model simpler and more intuitive, once that, in the case only one port can be bound to a location, modelling a filter may be more complex.

Stateful filters: A filter may want to store useful information for its next executions. This allows, e.g., storing a list of those last received messages to avoid processing when receiving future messages that are semantically equivalent; in other situations can be used to store results that are costly to calculate, thus avoiding to execute repeatedly the same processing for similar messages.

Our proposal is the most general out of the tools surveyed in this paper, since it allows for multiple input and output filters and tasks, it allows to configure exit ports in both competition and replication modes, and it supports multiple adapters per port. We also provide explicit support for stateful filters.

4.3 Technical Properties

Finally, we present some technical properties (cf. Table 3), as its absence could affect the facility of programming, the performance or the management of solutions integration, also may hinder the deployment and the operation of them.

Property	**Camel**	**Spring Int.**	**BizTalk**	**Ours**
Execution model	1 : 1	1 : 1	1 : 1	$N : M$
Typed messages	No	No	Yes/No	Yes
Communication patterns	Yes*	No	No	Yes
Attachments	No	No	Yes	Yes
Abnormal messages	Yes	Yes	Yes	Yes

* No new MEPs can be defined.

Table 3. Technical properties.

Execution model: The filter's execution model may seriously affect the performance of an integration solution. The simplest model consists of assigning a thread to each message or set of messages that must be treated as a whole by a filter; of course, the threads can be taken from a pool to keep under control the total workload of the server. We believe that this model has a shortcoming that can damage the scalability of the solutions. The problem is related to the fact that when an instance of a filter reaches a point where it can not continue running tasks, e.g, because it is necessary to wait for a message arrival, the thread is idle for a completely undetermined time. From our experience we concluded that a model capable of running on an asynchronous way multiple instances of the same filter on a pool of threads would be much more effective. Currently we are working and evaluating both alternatives in order to obtain more conclusions.

Typed messages: A typical integration solutions usually performs a lot of message transformations; any error in one of them can lead to an incoherent message, so it is highly desirable that these messages are typed.

Communication pattern: The Message Exchange Pattern (MEPs) allows to define specific communication types [13]. An integration solution can use different types of MEPs. Using MEPs facilitates the correlation between messages that reach a filter and the responses produced. For this reason, from a technical point of view, it is desirable that the tool offers support to this pattern and, besides, allows to define new types of MEPs, in addition to those that are already available.

Messages with attachment: A message can also carry, in addition to the header and body information, other objects with attached information. The attached information should not be processed in the integration solution, in other words, it only represents additional information that is transmitted with the message. We believe that the separating attached information from body, allows to reduce its processing time, once that when a task access the body to process it, will not have to deal with the attaches. Moreover, if the attaches are separated from the body, it is possible to use the ClaimCheck pattern [9] to store the attaches in a persistent repository, while the message is being processed. The attaches can be recovered latter.

Abnormal messages: When a message presents some kind of anomaly that makes it impossible to be processed by a building block, the norm is that it produces an exception and that the message in question is stored so that it can be analyzed by the system administrator. In addition, it is highly desirable that these messages can also be processed automatically so that we can try to carry out some sort of corrective action at the same time in which it was detected.

Regarding technical properties, our proposal is also quite complete since it allows for typed messages, it allows to define arbitrary message exchange patterns, and messages can have attachments to improve efficiency. The asynchronous, multi-threaded execution model is still under evaluation.

5 Conclusions

Application integration is a growing activity in companies and, according to the report published by [15], it is very expensive since it demands much more resources than the regular software development process. Knowing these, it is very important to have engineering technologies (languages, tools, frameworks, and the like) that can support this activity helping to reduce the cost and resources usually spent in. The DSL proposal presented in this paper contributes to helping engineers implement integration solutions with less effort. Our proposal is based on the concept of building block, which 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. It has been validated in two real-world integration projects in quite different scenarios.

References

1. C. Chang, M. Kayed, M.R. Girgis, and K.F. Shaalan. Survey of web information extraction systems. *IEEE Transactions on Knowledge and Data Engineering*, 18(10):1411–1428, 2006.
2. B.A. Christudas. *Service Oriented Java Business Integration.* Packt Publishing, 2008.
3. S. Cook, G. Jones, S. Kent, and A.C. Wills. *Domain-Specific Development with Visual Studio DSL Tools.* Addison-Wesley, 2007.
4. J. Davies, D. Schorow, and D. Rieber. *The Definitive Guide to SOA: Enterprise Service Bus.* Apress, 2008.
5. P. Delia and A. Borg. *Mule 2: Official Developer's Guide to ESB and Integration Platform.* Apress, 2008.
6. Apache Foundation. Apache Camel home. Available at http://activemq.apache.org/camel.
7. M. Fowler. *Patterns of Enterprise Application Architecture.* Addison–Wesley, 2002.
8. Martin Fowler. Inversion of Control Containers and the Dependency Injection Pattern, 2004.
9. G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley, 2003.
10. D. Messerschmitt and C. Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry.* MIT Press, 2003.
11. D. Skeen. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, 9(3):219–228, 1983.
12. Inc. SpringSource. Spring integration home. Available at http://www.springframework.org/spring-integration.
13. W3C. Web services message exchange patterns. Available at http://www.w3.org/2002/ws/cg/2/07/meps.html#id2612442.
14. C. Walls and R. Breidenbach. *Spring in Action.* Manning Publications, 2004.
15. J. Weiss. Aligning relationships: Optimizing the value of strategic outsourcing. Technical report, IBM, 2005.
16. I.H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, 2005.
17. D. Woolston. *Foundations of BizTalk Server 2006.* Apress, 2007.