A DOMAIN-SPECIFIC LANGUAGE TO DESIGN ENTERPRISE APPLICATION INTEGRATION SOLUTIONS

RAFAEL Z. FRANTZ [*]
Department of Technology, Unijuí University.
Rua do Comércio, 3000. Ijuí 98700-000, RS. Brazil.

ANTONIA M. REINA QUINTERO [†]
ETSI Informática, University of Seville.
Avda. Reina Mercedes, s/n. Sevilla 41012. Spain.

RAFAEL CORCHUELO [‡]
ETSI Informática, University of Seville.
Avda. Reina Mercedes, s/n. Sevilla 41012. Spain.

Enterprise Application Integration (EAI) solutions cope with two kinds of problems within software ecosystems, namely: keeping a number of application's data in synchrony or creating new functionality on top of them. ESBs provide the technology required to implement a variety of EAI solutions at sensible costs, but they are still far from negligible. It is not surprising then that many authors are working on proposals to endow them with domain-specific tools to help software engineers reduce integration costs. In this article, we introduce a proposal called Guaraná. Its key features are as follows: it provides explicit support to devise EAI solutions using enterprise integration patterns by means of a graphical model; its DSL enables software engineers to have not only the view of a process, but also a view of the whole set of processes of which an EAI solution is composed; both processes and tasks can have multiple inputs and multiple outputs; and, finally, its runtime system provides a task-based execution model that is usually more efficient than the process-based execution models in current use. We have also implemented a graphical editor for our DSL and a set of scripts to transform our models into Java code ready to be compiled and executed. To set up a solution from this code a software engineer only needs to configure a number of adapters to communicate with the applications being integrated.

Keywords: Domain-Specific Language; Enterprise Application Integration.

1. Introduction

Nowadays, most companies run many applications in their software ecosystems [20] to carry out their business activities. These applications are frequently software packages purchased from third parties or legacy systems.

[*]rzfrantz@unijui.edu.br
[†]reinaqu@us.es
[‡]corchu@us.es

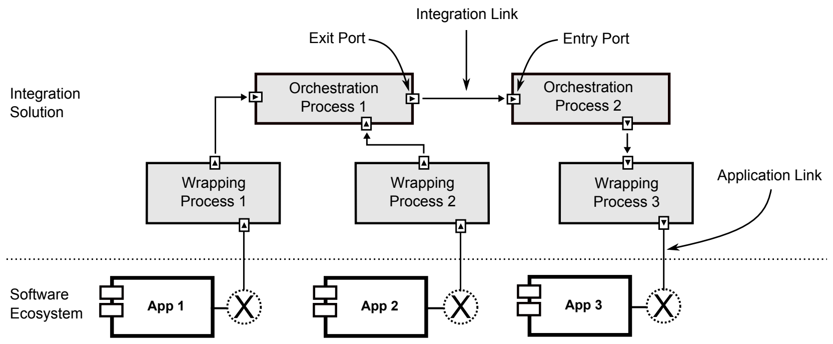

Figure 1. A typical EAI solution.

A recurrent challenge is to make these applications interoperate with each other to keep their data synchronised or to create new functionality [15]. It is not uncommon that the integration is performed by a user who copies data from an application and pastes it into another. This is obviously not scalable, which motivates many companies to invest in automated EAI solutions.

Unfortunately, applications are not usually easy to integrate due to many reasons, e.g., the technologies on which they rely are different, their APIs are not compatible from a semantic point of view, or they might not provide an API at all, which is the case of many web applications. Additionally, EAI solutions must take three important constraints into account [28], namely: first, the applications being integrated should not be modified at all since a change might seriously affect or even break down other business processes; second, they must keep running independently from each other since they were designed originally without taking integration concerns into account, i.e., no additional coupling must be introduced; finally, integration must be performed on demand, as new business requirements emerge and require new services to be created on top of the existing applications [2].

The previous problems usually turn into costs that are far from negligible. According to a recent report, the cost of integrating a new application into a software ecosystem is from 5 to 20 times higher than developing it [31]. These figures make it clear that integrating business applications is quite a serious challenge that has promoted a research field known as Enterprise Application Integration (EAI).

From an abstract point of view, a typical EAI solution is composed of several wrapping processes that are responsible for interacting with the applications, and several orchestration processes that are responsible for managing a flow of messages amongst them, cf. Figure §1. Processes use ports to communicate with other processes or with applications by means of links; ports abstract away from the actual communication mechanism, which may range from an RPC-based protocol over HTTP to a document-based protocol implemented on a database management system.

Enterprise Service Buses (ESB) [1, 5] provide the technology required to implement the previous abstractions and cope with the problems and constraints that were mentioned previously. Typical ESBs provide so-called adapters or connectors, which are used to communicate with the applications being integrated, and an orchestration language to define workflows of messages amongst processes and/or applications. Java Business Integration (JBI) is an specification of a pluggable architecture of services [2] to which several ESBs have adhered, e.g., Open ESB, Petals ESB, ServiceMix or its commercial version, Fuse ESB [24]. In JBI, adapters and connectors are referred to as binding components, and the orchestration language used is the Business Process Execution Language (BPEL) [25]. There are many types of binding components available nowadays, which allows solutions to connect to almost any existing application. The catalogue includes binding components for databases, files, SOAP/HTTP, RSS, SMTP, RMI/IIOP, JMS, HL7, LDAP, DCOM, and so on. Our research focuses on Open ESB.

Although BPEL is quite a common orchestration language, it is too general. It provides only general constructs like invoke, receive, reply, assign, throw, wait, or compensate. It does not provide explicit support to the whole set of the well-known Enterprise Integration Patterns (EIPs) [15] that are used extensively in the EAI Community. A few EIPs can be implemented directly as part of an orchestration process, e.g., filters, routers or mergers; many others, however, need to be implemented separately as independent services that need to be invoked by an orchestration [33]. This results obviously in inefficiency, but also in difficulties to understand and maintain the resulting solutions. This has motivated many software engineers to use Camel [16] or Spring Integration [8], since these tools help use many EIPs inside orchestration processes; obviously, they do not rely on BPEL.

Model Driven Development (MDD) is an approach to software development in which models are first-class citizens and the centre of the development process [13]. Domain-Specific Languages (DSLs) play an important role in MDD. Every DSL is supported by an abstract syntax, aka, metamodel, that provides a number of concepts that can be used to devise solutions to problems in a given domain; the solutions are specified by means of a concrete syntax that allows to represent the concepts in the metamodel textually or graphically; such solutions are usually referred to as models, and the key is that they must be accompanied by a number of transformations to translate them into code written in a programming language. Overall, using models allows software engineers to devise solutions to their problems at a proper abstraction level and relieves them from the burden of dealing with the rather low-level constructs provided by programming languages.

In this article, we present a proposal called Guaraná. Its main advantages with regard to similar proposals are as follows: it provides explicit support to devise EAI solutions using enterprise integration patterns by means of a graphical model; its DSL is graphical, and it enables software engineers devise EAI solutions at a high-level of abstraction; not only provides the DSL the view of a process, but also a view of the whole set of processes of which EAI solutions are composed; both processes and tasks can have multiple inputs and multiple outputs; and, finally, its runtime system provides a task-based execution model that is usually more efficient than the process-based execution models in current use. We have also implemented a graphical editor for our DSL a set of scripts to transform our models into Java code ready to be compiled and executed. To set up a solution from this code a software engineer only needs to configure a number of adapters to communicate with the applications being integrated. Our results were implemented in the laboratory using the following tools: Eclipse Helios as the workbench, the Eclipse Modelling Framework (EMF) [29] to implement the metamodel, the OCLInECore plug-in [32] to implement constraints in the metamodel, the EuGENia plug-in [18] and the Graphical Modeling Framework (GMF) [9] to implement the editor, the Dresden OCL Toolkit [6] to validate the OCL specification, and MOFScript [22] to implement our model-to-text transformations. This implementation helped us validate our proposal on a non-trivial, real-world case study.

The rest of this article is structured as follows: Section §2 presents the related work and compares other proposals to ours; Section §3 presents Guaraná in a nutshell so that readers can have an overall idea of the proposal; Section §4, presents the abstract syntax of our DSL, and Section §5 complements it with a concrete syntax; Section §6 reports on the runtime system that supports our proposal; Section §7 summarises the transformations we have devised to translate Guaraná into Java code; Section §8 summarises one of the task toolboxes we have devised for Guaraná; Section §9 reports on a validation we have performed using a non-trivial, real-world integration problem; finally, Section §10 reports on our main conclusions.

2. Related work

The catalogue of Enterprise Integration Patterns (EIPs) proposed by Hohpe and Woolf [15] puts a foundation to the majority of proposals in the field of EAI, including ours. This catalogue describes a number of recurring problems in EAI and a number of proposals to solve them. The catalogue is informal, in the sense that it is not intended to provide a tool by means of which a software engineer can devise, build and deploy an EAI solution; contrarily, it can be viewed as a cookbook that a software engineer can use to have a better understanding of common solutions to common EAI problems, without committing to a particular technology or tool.

Scheibler et al. [27] presented an approach to enable the use of EIPs for EAI in the context of the Software as a Service (SaaS) business model. They claimed that this model relieves software engineers from setting up the environment where an EAI solution is executed, and the same pattern implementation can be reused by different solutions once they are provided as services. Although it is an interesting approach for EAI, it differs from ours because it is narrowly focused on the SaaS business model, which is not our focus.

Dirgahayu et al. [7] introduced a language and a set of patterns to design EAI solutions in the context of web services. Their focus was on the design of orchestration processes and how to represent their interactions. They do not provide explicit

Property	**Camel**	**Spring**	**BizTalk**	**Guaraná**
Context	EAI/B2BI	EAI/B2BI	EAI/B2BI	EAI/B2BI
Model	API/Textual	API/Textual	Textual/Graphic	API/Graphic
Kind of DSL	Internal	Internal	External	External
Routing slip	Dynamic	Static	Static	Static

Table 1. Scope properties.

support for EIPs, but model EAI solutions as compositions of processes that interact amongst themselves and/or with the applications being integrated. The authors mentioned that this is important to increase the level of abstraction and to enable business analysts to participate actively in the design of EAI solutions.

Yuan [33] presented a tool that helps generate code for EIPs automatically. EIPs are configured with their inputs and outputs and the tool maps them onto BPEL processes. The author stresses that it is not usually possible to implement EIPs relying solely on BPEL constructs; in such cases, a part of the EIP must be implemented as independent web services. Unfortunately, this proposal does not provide a DSL like ours, but only a tool to translate some EIPs.

Camel [16] and Spring Integration [8] provide fluent APIs [10] by means of which it is possible to implement solutions using constructors that are very close to the EIPs; however, they are totally bound to the Java technology. Therefore, they do not provide the same level of abstraction as our DSL.

BizTalk [21] and IBM WebSphere [17] are related commercial tools. The former provides a BPEL-like DSL, whereas the latter relies fully on BPEL. Hohpe & Tham [14] and Scheibler & Leymann [26] devised a cookbook with many hints to implement EIPs using these tools.

From the analysis of the literature and tools, we conclude that Camel, Spring Integration, and BizTalk are the most closely related proposals. To compare Guaraná to these tools, we have built a comparison framework with a set of objective properties that are classified into scope, modelling, and technical properties. We report on the comparison framework in the following subsections.

2.1. *Scope Properties*

A scope property represents a feature whose absence can greatly hinder or even invalidate a proposal for a particular purpose. Table §1 summarises the scope properties that we have identified, namely:

Context: We distinguish amongst the following contexts: Enterprise Application Integration (EAI), in which the emphasis is on integrating applications to keep their data in synchrony or to implement new functionalities on top of them; Enterprise Information Integration (EII), whose emphasis is on providing a live-view of the data handled by the integrated applications; and Extract, Transform, and Load (ETL), whose goal is to provide a materialised view on which we can apply data mining techniques. In the previous cases, we implicitly assume that the integrated applications belong to the same organisation. Recently, the integration of applications that belong to different organisations is becoming more and more widespread; this context is known as Business to Business Integration (B2BI), if the integration is performed on the server side, or Mash-up, if the integration is performed on the client side, usually on a web browser.

Model: There are several kinds of models in widespread use, namely: APIs, textual or graphic models. An API-based tool provides a library that software engineers can use to create their EAI solutions. The resulting models are thus written in general-purpose programming languages and rely on calls to the API. Text-based and graphic-based tools provide a complete DSL to design EAI solutions, i.e., they minimise the need to use a general-purpose language.

Kind of DSL: There are two kinds of DSLs, namely: internal and external [10]. An internal DSL consists of a language which is defined using a general-purpose language as host, and thus conforms to its syntax; many such internal DSLs rely on so-called fluent APIs. External DSLs do not follow any general-purpose language's syntax, instead they are defined in a separate language.

Routing slip: This term is generally used to refer to the routes that a message in an EAI solution follows. A route can be designed either statically or computed dynamically. Proposals that support dynamic routing slips are appropriate in settings in which the kind of applications to be integrated are known beforehand, but the actual instances may change at run time. For instance, think of a company that provides stock exchange forecasts and has a number of applications that deliver similar information; if dynamic routing slips are supported, the decision on which application should be used and the exact route that the messages needs to follow may be delayed until the EAI solution is running.

2.2. *Modelling Properties*

A modelling property is a feature that is not so critical as a scope property. Lacking a modelling property does not make it impossible to use a tool, but may lead to a design that is more complex and less intuitive than it should be; this obviously may have an impact on maintenance costs. Table §2 summarises the modelling properties we have identified, namely:

Views: The most common view is the process view. It allows to model processes that act as centralised orchestrators, i.e., they help co-ordinate the activities of other processes or applications [4]. This is an imperative view, in the sense that processes make it explicit what has to be done at each time. BPEL

Property	**Camel**	**Spring**	**BizTalk**	**Guaraná**
Views	Process	Process	Process	Process/Solution
EIP support	Yes	Yes	No	Yes
Stateful tasks	Yes	Yes	No	Yes
Task cardinality	$1 : N$	$1 : N$	$1 : N/N : 1$	$M : N$
Process cardinality	$1 : N$	$1 : N$	$M : N$	$M : N$

Table 2. Modelling properties.

is the most prominent language used to design orchestration processes. On the contrary, a solution view is a choreography view in which one can have an overall picture of what is going on, but there is not a centralised process that co-ordinates it all; contrarily, the behaviour of the solution can be seen as the co-ordinated behaviour of the processes of which it is composed. This is the reason why these views are usually referred to as declarative. WS-Choreography [30] is the most prominent language to specify choreographies.

EIP support: The catalogue of EIPs by Hohpe and Woolf [15] is a de-facto standard. It is then desirable for a tool to provide explicit support for at least the most common EIPs, since this saves software engineers from the burden of implementing them from scratch.

Stateful tasks: A task may require to store its state throughout different executions. It is useful in situations in which a task performs an incremental computation that involves a series of messages. Examples of such tasks include aggregating a sequence of messages or filtering out a message that is semantically equivalent to a previous one.

Task cardinality: Most common tasks get one message as input and produce one message as output, e.g., filters, translators, or mappers. There are others that require multiple inbound messages and produce multiple outbound messages. A typical such task is a message enricher, which gets a data message and a context message as input and enriches the former with information provided by the latter. In Table §2, task cardinalities are represented as $a : b$, where a and b denote the maximum number of inbound and outbound messages allowed, respectively; N and M denote 'many'.

Process cardinality: Similarly to tasks, a process may require one or multiple inbound messages to work and may produce one or multiple outbound messages. In Table §2, process cardinalities use the same notation as task cardinalities.

2.3. *Technical Properties*

Technical properties are features whose absence might have an impact on how easy programming is, on the performance of a solution, or on how easy managing it is.

Property	**Camel**	**Spring**	**BizTalk**	**Guaraná**
Execution model	Process-based	Process-based	Process-based	Task-based
Typed messages	No	No	Yes	No
Architecture for adapters	Yes	Yes	Yes	No

Table 3. Technical properties.

Table §3 summarises the technical features we have identified, namely:

Execution model: A typical approach is to use a database to store inbound messages until all of the messages needed to start a process arrive. Then, the runtime system assigns one thread to execute its tasks. We refer to this model as a process-based execution model. Note that if a task makes a request to an external resource, the assigned thread remains blocked until it receives the response. It is common that messages from EAI solutions are handled with a low priority by the integrated applications, so as not to disturb them; there are cases in which a message received by an application requires the intervention of a person; in other cases, an application is actually an already-deployed EAI solution. Therefore, the response to a request is not usually instantaneous, but may take a time that ranges from seconds to minutes, hours or even days [12]. The problem with the process-based execution model is that the thread assigned to a process may remain blocked and inactive for a long time, which amounts to wasting system resources. The task-based execution model does not suffer from this problem. In this model, threads are assigned to tasks, instead of processes; this allows to use system resources more efficiently [11, 19]. This lower level of granularity allows tasks to be executed the sooner as there is a message available to them, without having to wait for all of the messages needed to start a process.

Typed messages: Messages must usually go through a series of tasks that route, transform or modify them. It is then desirable that messages are typed, so that the runtime system can catch messages that are routed incorrectly the sooner as possible.

Architecture for adapters: It is desirable for a mechanism to design new adapters to exist, since this makes it possible to integrate applications that use new technologies as necessary. A shared repository of such adapters is also a must. JBI provides a specification to design new adapters, i.e., binding components, and there are several repositories available on the Web.

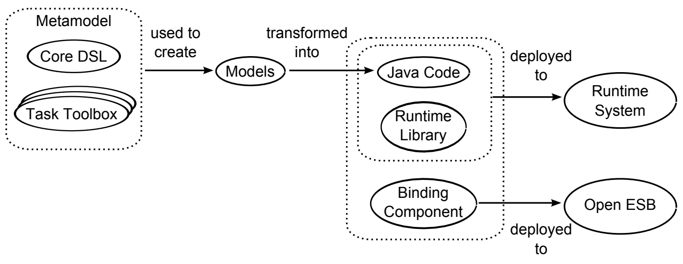

Figure 2. Overall picture.

3. Guaraná in a nutshell

In this section, we first provide an overview of our proposal and then delve into the main concepts supported by our DSL.

3.1. *The overall picture*

Guaraná refers to a project whose goal is to provide software engineers with tools they can use to devise and implement EAI solutions at sensible costs. Figure §2 presents the overall picture.

Guaraná provides a metamodel that supports a number of concepts that software engineers can use to devise their EAI solutions. Note that the metamodel is divided into two parts, namely: a core, which supports a subset of concepts that are assumed to be useful across a wide range of EAI solutions, and a series of task toolboxes, which support subsets of tasks that are assumed to be specific to a given domain of integration. We provide additional details on the metamodel in the following subsection.

A software engineer can use the concepts defined in the metamodel to create his or her own models, which are specific solutions to specific integration problems. Such models are graphic and allow to devise EAI solutions at a high level of abstraction. Guaraná also provides a set of transformations by means of which a software engineer can translate his or her models into Java code. This code relies on a runtime library that provides base classes to implement the concepts supported by the metamodel.

Note that the Java code plus the runtime library are not enough to implement an EAI solution; it is also necessary a number of binding components. Whereas the Java code must be compiled and deployed to the runtime system that supports Guaraná, the binding must be configured and deployed to Open ESB independently. The processes of which the solution is composed shall use these binding components to interact with the applications being integrated or with other processes.

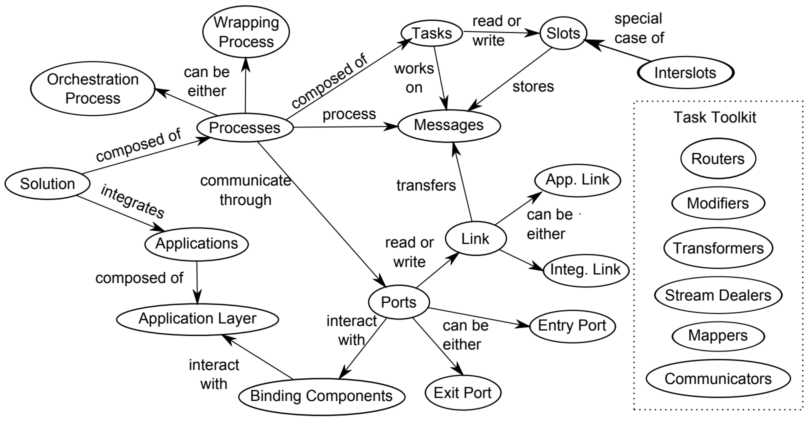

Figure 3. Conceptual map.

3.2. *The metamodel*

Figure §3 presents a conceptual map in which we present the concepts with which we deal to model EAI solutions. The root concept is solution, which represents a collection of processes that co-operate to integrate a number of applications.

Processes serve two purposes, namely: there are processes that allow to wrap applications and processes that allow to integrate them. The former are reusable processes that endow an application with a message-oriented API that simplifies interacting with it. Implementing such a wrapping process may range from using a JDBC driver to interact with a database to implementing a scrapper that emulates the behaviour of a person who interacts with a user interface [3]. Orchestration processes, on the contrary, are intended to orchestrate the interactions with a number of wrapping processes and other orchestration processes.

Processes rely on tasks to perform their wrapping or their orchestration activities. Simply put, a process can be viewed as a message processor. A message is an abstraction of a piece of information that is exchanged and transformed across an EAI solution. The structure of messages depends completely on the solutions in which they are involved.

Tasks are extremely dependent on the context, which makes it of little interest to think of a general-purpose collection of tasks. Instead, we decided to provide different task toolboxes for different integration contexts, e.g., HL7-oriented tasks in health contexts, HIPPA-oriented tasks in insurance contexts, RosettaNet-oriented tasks in business-to-business contexts, or SWIFT-oriented tasks in financial contexts, to mention a few. Each toolbox results in a different version of the DSL and a specific editor. The toolbox on which we report in this article is the most general one, since it provides a collection of general-purpose tasks that provide the foundations for many other special-purpose task toolboxes. In Figure §3, we illustrate the main categories of tasks only; cf. Section §8 for a complete description.

Note that our proposal does not preclude several tasks (including several instances of the same task) from executing in parallel. This makes it impossible for tasks to communicate directly to each other. Instead, they communicate indirectly by means of slots. A slot acts as a buffer in-between tasks, i.e., they allow a task to output messages that shall be processed asynchronously by another task.

Processes use ports to communicate with each other or with the applications involved in an EAI solution. Simply put, the purpose of a port is to abstract away from the details required to interact with a binding component, which, in turn, abstracts away from the details required to interact with an application or with a process. Binding components are used by means of a special kind of task, referred to as communicator. The interaction with an application occurs at one or more layers, i.e., data layer, data access layer, business logic layer, and user interface layer. Ports can be either entry or exit ports, depending on whether they were designed to read messages from a process or an application, or to write messages to them.

Note that ports usually need to transform the messages they transfer, which implies that they are composed of tasks, as well. This means that they also need slots to help their tasks work as much asynchronously as possible. Another subtle implication is that there must be a slot to communicate a task in a port to a task in the process to which the port belongs. We refer to such slots as interslots.

4. Abstract syntax

This section describes the part of our metamodel, aka abstract syntax, that is related to the core DSL. Figure §4 provides an overall picture to guide the reader through the following subsections.

4.1. *EAI solutions*

Solution is the root class of our metamodel, and it represents an EAI solution. A Solution has a name property, which is used for documentation purposes only, and consists of one or more Processes, one or more Applications, and one or more Links. A Solution must fulfill the following invariants:

```
context Solution
   inv: applications->isUnique(name)
   inv: processes->isUnique(name)
   inv: links->isUnique(name)
```

They state that the names of the applications, processes and links must be unique. Note, however, that an application and a process may have the same name, since there are not any chances to mistake them.

4.2. *Processes*

Class Process represents either a wrapping or an orchestration process. A Process is composed of at least one EntryPort, at least one ExitPort, at least one Task, and at

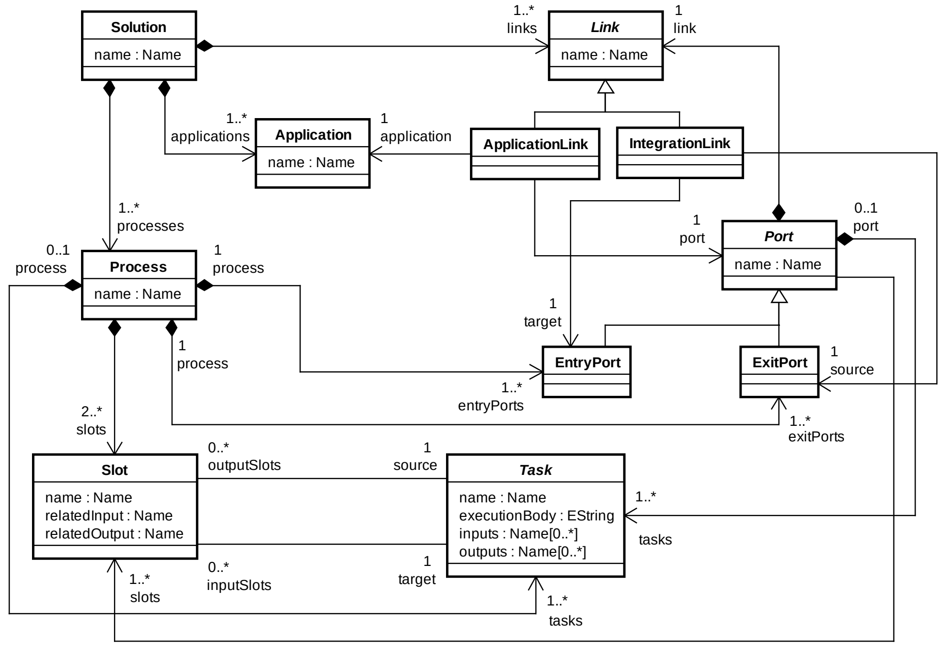

Figure 4. Main constructors of Guaraná.

least two Slots. A Process has to fulfill the following invariants:

```
context Process
   inv: tasks->union(entryPorts.tasks->union(exitPorts.tasks))->isUnique(name)
   inv: slots->isUnique(name)
   inv: entryPorts->union(exitPorts)->isUnique(name)
   inv: tasks->select(oclIsKindOf(Communicator))->size() = 0
   inv: let interslots: Set(Slot) = slots->select(s: Slot |
         not self.tasks->includes(s.source) and self.tasks->includes(s.target) or
         self.tasks->includes(s.source) and not self.tasks->includes(s.target)) in
      interslots->size() = self.entryPorts->size() + self.exitPorts->size()
```

These invariants state that tasks, slots and ports must have unique names, that a process cannot contain any tasks of kind Communicator because these tasks are specific to ports, and that there can be only a interslot per port.

To understand the first invariant, recall that both processes and ports can contain tasks, and they all must have different names. Thus, for each process, we need to calculate the set of tasks of which it is directly composed, union the set of tasks in its entry and exit ports.

Note, however, that the invariant regarding slots is slightly different, since we do not need to calculate the slots of a process, union the slots of its entry and exit ports; instead we can simply write slots->isUnique(name). The reason is that, at least in theory, interslots belong to both a process and a slot; unfortunately, EMF does not allow to model such situations. The only solution is that property slots holds all of the slots involved in a process, including the slots in its entry and exit ports.

The last invariant also deserves an explanation. It states that there can be only one slot connecting the tasks that are contained in a port to the tasks that are contained in the corresponding process, i.e., there can be at most one interslot per port. The most difficult part of the invariant is the identification of interslots: they are calculated as the set of slots whose source task is not included in the set of tasks of which a process is directly composed, but the target is, or vice-versa. Note that if a source or a target task in a slot does not belong to a process itself, it must belong to one of its ports, which implies that the original slot is actually an interslot.

4.3. *Ports and links*

Ports are composed of tasks and slots, that get connected by Links; these, in turn, can be either ApplicationLinks, which connect Applications to Ports, or IntegrationLinks, which connect EntryPorts to ExitPorts. Recall, however, that the inability to represent interslots as shared objects prevented us from modelling the slots of which a port is composed as a proper containment property. Instead, we need to calculate the slots of which a port is composed by means of a derivation, namely:

```
context Port::slots: Set(Slot)
  derive: tasks->collect(outputSlots)->union(tasks->collect(inputSlots))
```

Another derivation is property link; recall that every port must be connected to a link so that messages can be transferred. The problem is that links should belong to both a solution and some of its ports, which is not possible. This is the reason why property link is also derived, namely:

```
context Port::link: Link derive:
  let appLink: Link = ApplicationLink.allInstances()->any(port = self) in
  let intLink: Link = IntegrationLink.allInstances()->any(source = self or target = self) in
    if not appLink.oclIsUndefined() then appLink else intLink endif
```

Note that the derivation is defined in class Port. This is the reason why the formula tries to find both an application and an integration link whose port is the current context; depending on whether the port is actually connected to an application or to a process, either appLink or intLink shall not be undefined.

Furthermore, ports must fulfill the following invariants:

```
context Port
  inv: tasks->isUnique(name)

context EntryPort
```

```
    inv: tasks->one(oclIsKindOf(Communicator))
    inv: tasks->one(oclIsKindOf(InCommunicator))

context ExitPort
    inv: tasks->one(oclIsKindOf(Communicator))
    inv: tasks->one(oclIsKindOf(OutCommunicator))
```

The previous invariants state that the tasks in a port must have unique names, that an EntryPort must have one Communicator of kind InCommunicator, and that an ExitPort must also have one Communicator of kind OutCommunicator. Whilst the former kind of communicator is used to read messages from a binding component, the latter is used to write messages to the binding component.

There is a final invariant: we do not allow for 'looping' processes, i.e., processes in which a port is connected to another port in the same process. To avoid this kind of anomaly, we introduced the following invariant in our metamodel:

```
context IntegrationLink:
    inv: not (source.process = target.process)
```

4.4. *Tasks and slots*

Every task has a name, a set of inputs, a set of outputs, and an executionBody. Both inputs and outputs are connected to slots at run time and hold messages; the execution body is a piece of Java code that implements the activities that must be carried out. Inside the execution body, a software engineer may reference the messages held in the inputs and outputs.

Every task must fulfill the following invariants:

```
context Task
    inv: inputs->union(outputs)->isUnique(n: Name | n)
    inv: inputSlots->collect(s: Slot | s.relatedInput) = inputs
    inv: outputSlots->collect(s: Slot | s.relatedOutput) = outputs
```

These invariants state that both inputs and outputs must have unique names, that no input or output can be disconnected from a slot, and that no input or output is connected to more than one slot. Note that every slot has a property called relatedInput and a property called relatedOutput; they indicate to which task inputs and outputs they are connected, respectively. Thus, our invariants require that the set of related inputs of the input slots must coincide with the set of inputs of every task; similarly, the set of related outputs of the output slots must coincide with the set of outputs of every task. This guarantees that every input or output is connected to one and only one slot.

Every slot must fulfill the following invariants:

```
context Slot
    inv: not (target = source)
```

Datatype	**Base**	**Constraint**
Name	String	[a-zA-Z_]([a-zA-Z_0-9])*
HostName	String	[a-zA-Z0-9\-]{1,62}((\.[a-zA-Z0-9\-]{1,62})+\.[a-zA-Z]{2,6})?
JndiName	String	[a-zA-Z_@$]([a-zA-Z_@$0-9])*(/[a-zA-Z_@$0-9]+)*
PositiveInteger	int	minInclusive = 0 and maxInclusive = 65534

Table 4. Datatypes used in our metamodel.

```
inv: target.inputs->includes(relatedInput)
inv: source.outputs->includes(relatedOutput)
inv: let sourceProcess: Process = Process.allInstances()->any(p: Process |
        p.tasks->union(p.entryPorts.tasks)->
          union(p.exitPorts.tasks)->includes(self.source)) in
   let targetProcess: Process = Process.allInstances()->any(p: Process |
      p.tasks->union(p.entryPorts.tasks)->
          union(p.exitPorts.tasks)->includes(self.target)) in
   sourceProcess = targetProcess
```

These invariants state that every slot must connect different tasks, that they must be properly connected to the inputs and outputs of the corresponding tasks, and that they cannot connect tasks in different processes. Note that the association between Process and Slot is not backwards navigable because of the problem to model interslots; this implies that we need to calculate explicitly the process to which the source and the target tasks of every slot belong. To calculate it, we need to iterate over the whole set of process instances to find a process whose tasks, union the tasks of its entry and exit ports contain the source or the target task of every slot.

4.5. *Datatypes*

In the metamodel, we refer to the following data types: Name, which represents a subset of Java identifiers; HostName, which represents DNS host names; JndiName, which represents a subset of JNDI names; and PositiveInteger, which represents 16-bit positive integers. They allow us to constraint some properties of the metamodel that need to be copied verbatim by our transformations. Table §4 summarises the definitions of the previous datatypes.

5. Concrete syntax

Table §5 shows the concrete syntax we use to represent the classes provided by our abstract syntax.

Since tasks are provided in toolboxes that are not part of the core language, the symbol that we depict in Table §5 to represent them is generic. (Cf. Section §8 for a

Icon	**Class**	**Icon**	**Class**
⧉	Application	→	IntegrationLink
▭	Process	—	ApplicationLink
▶	EntryPort	⇢	Slot
◀	ExitPort	▭	Task

Table 5. Concrete Syntax.

complete description of our general-purpose task toolbox.) Note the small bulges on the sides of the icon; they represent the inputs and the outputs. Slots are connected to tasks using these bulges.

Note that the syntax regarding processes and ports is abbreviated, i.e., this is the syntax used to hide the details; they both are containers, which implies that they can be represented making it explicit their internal structure, as well.

6. Runtime system

In this section, we describe the semantics of the runtime system that supports Guaraná. The runtime system adheres to the metamodel in Figure §5, where Runtime represents the root class. Roughly speaking, a runtime system is composed of the following objects:

- At least two BindingComponent objects that represent the binding components with which a solution must interact. Note that in cases in which an EAI solution is used to provide new functionality to an existing application, two binding components are still required to interface with it: one to retrieve information and another to write information. This justifies the need for both the runtime system and its binding components to know each other (note that the association between both classes is bidirectional).
- A ThreadPool, which is basically a container of threads the runtime system manages to run tasks when they are ready to be executed. Note that the Runtime itself runs on a thread that is not part of this pool.
- A ReadyQueue, in which the RunTime references the tasks that are ready to be executed, but cannot run because there is not a free thread in the ThreadPool. Note that every Runtime also has a container called tasks that is intended to store references to all of the tasks for which it is responsible, i.e., the tasks in ReadyQueue are a subset of the tasks in container tasks.
- A ChannelBoard, which is responsible for managing a set of Channels. In the previous sections, we made an explicit difference between slots and links. However, strictly speaking, both can be viewed as buffers that allow to decouple tasks from each other. This allows us to model them both

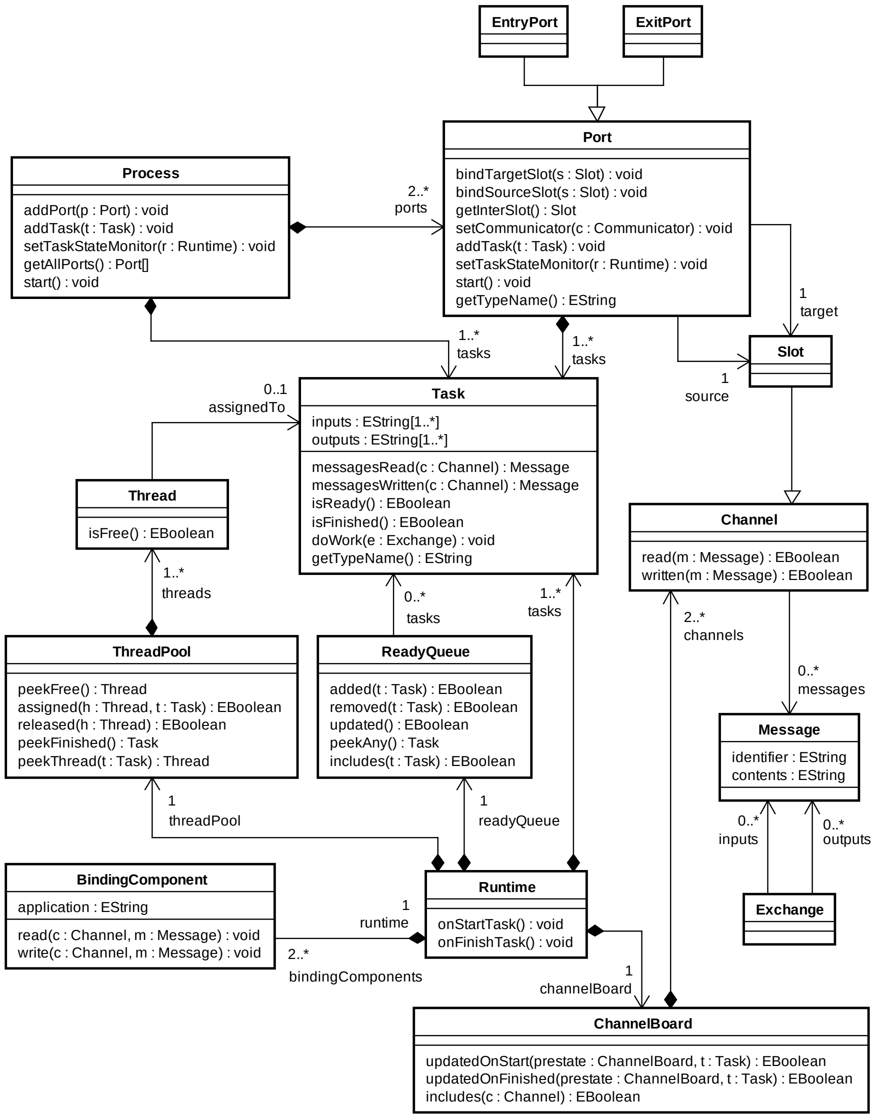

Figure 5. Guaraná's runtime system metamodel.

as abstract communications channels in our RunTime. A Channel stores a collection of Messages; note that they are not queues since tasks are free to select the messages in the order that best suits their semantics. We provide a class called Slot to implement in-memory slots, and the task toolbox must provide classes InCommunicator and OutCommunicator to read from or write to a link it by means of the appropriate binding component, cf. Section §8.

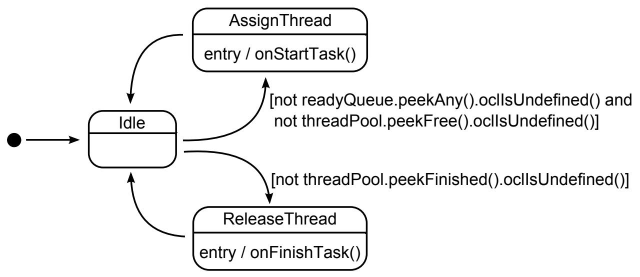
AssignThread
entry / onStartTask()
Idle
[not readyQueue.peekAny().oclIsUndefined() and not threadPool.peekFree().oclIsUndefined()]
[not threadPool.peekFinished().oclIsUndefined()]
ReleaseThread
entry / onFinishTask()

Figure 6. Guaraná's runtime statechart.

An interesting point is that our runtime system does not deal with processes or ports themselves, but with the tasks and slots of which they are composed. Processes are organisational units that Guaraná supports to facilitate modularising and deploying EAI solutions. However, it is not necessary to deal with them in the runtime system except to make the previous organisation explicit in the code; for such a purpose, we provide classes Process, Port, EntryPort, and ExitPort in our runtime.

When a process is deployed to a runtime system, its tasks and slots, including the tasks and slots in its ports, are added to the list of tasks managed by the runtime system, and the slots are added to the channel board. In other words, the same runtime system can deal with multiple tasks and channels involved in different solutions. Note that this deviates significantly from the runtime systems used in many ESBs since threads are allocated on a per-task policy, instead or a per-process policy; this has an impact on the efficiency since system resources can be used more efficiently, cf. Section §2.

In the following subsections, we present the semantics of our runtime system and the binding components. Note that the remaining components are active, too, but not autonomous, i.e., the activities of the thread pool, the ready queue or the channel board are monitored and controlled by the previous components. Our description of the semantics builds on the well-known Structural Operational Semantics method [23], which is particularly well-suited to describe the semantics of concurrent systems like ours.

6.1. *Semantics of the runtime system*

Figure §6 presents a state chart that provides a bird's eye view of the semantics of our runtime system. Roughly speaking, the thread in which the system runs is idle until any of the following conditions hold: a task is available in the ready queue and a thread is free in the thread pool, in which case operation onStartTask is executed, or a task that is currently assigned to a thread has finished working, in which case operation onFinishTask is executed. The rest of the time, the thread assigned to the runtime system is idle, which maximises the resources available to execute productive tasks.

The previous operations constitute the core of the runtime system. The formal specification of operation onStartTask is as follows:

```
context Runtime::onStartTask(): void
   pre: not readyQueue.peekAny().oclIsUndefined()
   pre: not threadPool.peekFree().oclIsUndefined()
  post: let t: Task = readyQueue@pre.peekAny() in
        let h: Thread = threadPool@pre.peekFree() in
          readyQueue.removed(t) and
          threadPool.assigned(h, t) and
          channelBoard.updatedOnStart(channelBoard@pre, t)
```

The precondition of this operation makes it sure that there must exist a task ready to be executed and a free thread to run it. In the postcondition, we simply fetch that task and that thread and make it sure that the task was removed from the ready queue, the thread was assigned to run the task, and the channel board is updated to make it sure that the messages that the task requires are removed from the corresponding channels. The reason why operation updatedOnStart requires the pre-state of the channel board to be passed as a parameter shall become clear later.

The specification of operation onFinishTask is as follows:

```
context Runtime::onFinishTask(): void
   pre: not threadPool.peekFinished().oclIsUndefined()
  post: let t: Task = threadPool@pre.peekFinished() in
        let h: Thread = threadPool@pre.peekThread(t) in
          threadPool.released(h) and
          channelBoard.updatedOnFinished(channelBoard@pre, t) and
          readyQueue.updated()
```

In this case, the precondition requires that the thread pool must contain a thread that is assigned to a task that has finished executing. In this case, the runtime system must release the thread, the channel board must be updated so that the messages produced by the task that has finished executing are written to the appropriate channels, and, finally, the ready queue must be updated since new tasks might have become ready to execute on account of the previous messages.

6.2. *Semantics of the binding components*

Binding components are autonomous, and they run inside an ESB, i.e., they run in parallel and outside the control of our runtime system. Note that a complete specification of binding components is beyond the scope of this article, cf. Christudas's [2] book for further details. In the model in Figure §6, we have abstracted away from the many details involved in configuring the many types of existing binding components, and we just highlight the operations the runtime system requires, namely read and write.

The read operation is intended to model situations in which an application or a process reads a message from a channel. (Note that in these cases, the channel must necessarily be a link.) The specification is as follows:

```
context BindingComponent::read(c: Channel, m: Message): void
   pre: runtime.channelBoard.includes(c)
  post: c.read(m)
```

The precondition is a sanity check that requires that the channel was included in the set of channels managed by the runtime system with which the binding component is associated. The postcondition makes it sure that messages are removed from the channel after reading them.

The specification of operation write is similar, namely:

```
context BindingComponent::write(c: Channel, m: Message): void
   pre: runtime.channelBoard.includes(c)
  post: c.written(m) and runtime.readyQueue.updated()
```

The precondition is also a sanity check to make it sure that the channel to which the message is being written is actually a channel managed by the runtime system to which the binding component is associated. The postcondition makes it sure that the message was actually written to the channel, i.e., it is stored in the channel, and that the queue of tasks that are ready for execution is updated. Note that, contrarily to reading from a channel, writing to a channel may be ready for execution the task that reads from that channel.

6.3. *Semantics of tasks*

Note that tasks are not autonomous since they cannot decide when to start executing. Instead they provide an interface to the runtime system by means of which it can schedule them for execution.

The interface a task must implement provides the following operations:

- ▷ isReady: This is a boolean operation by means of which a task can report that it is ready to be executed, i.e., the channels from which it reads have enough messages for the task to run.
- ▷ isFinished: This is a boolean operation that helps a task report that it is finished, i.e., the thread to which it was assigned can be released and reused to run other tasks.
- ▷ messagesRead: This operation takes a channel as input and must return the set of messages a task can read from that channel. Note that messages are not actually read until the task starts executing.
- ▷ messagesWritten: This operation takes a channel as input and returns the set of messages a task can write to that channel. Note that calling this

operation does not make sense unless the corresponding task has finished executing, and that the messages are not actually written until the runtime system releases the corresponding thread.

▷ doWork: This is the operation threads call on the tasks to which they are assigned. It is assumed to implement a specific-purpose activity which depends on the task toolbox under consideration. In this article, we report on the most general toolbox, in which the semantics of this operation is described by property executionBody in our metamodel, cf. Figure §4. Note that this operation gets an Exchange as parameter. An exchange wraps the inbound messages that the task must process, and the outbound messages the task produces when the doWork operation finishes executing. Inbound messages are loaded from the inputs of the task and the outbound messages are written to the outputs.

6.4. *Ancillary semantics*

In this section, we describe the semantics of the remaining classes in the runtime system. Note that most of the operations have already being used intuitively in the previous subsections. Here, we describe them formally.

The thread pool. Class ThreadPool abstracts away from the intricacies of managing a pool of threads. The interface of this class provides the following operations:

▷ peekFree: This operation returns a thread in the pool that is not assigned to a task. We assume that the implementation of this method is deterministic, i.e., consecutive invocations return the same thread as long as the state of the thread pool does not change. Its specification is as follows:

```
context ThreadPool::peekFree(): Thread
   body: threads->any(isFree())
```

▷ assigned: This operation simply checks whether a given thread is assigned to a given task. Its specification is as follows:

```
context ThreadPool::assigned(h: Thread, t: Task): EBoolean
   body: h.assignedTo = t
```

▷ released: This operation complements the previous one, since it checks whether a given thread was released. Its specification is as follows:

```
context ThreadPool::released(h: Thread): EBoolean
   body: h.isFree()
```

▷ peekThread: This operation takes a task as input and returns the thread to which it is assigned, if any. Its specification is as follows:

```
context ThreadPool::peekThread(t: Task): Thread
    pre: threads->one(assignedTo = t)
   post: result = threads->any(assignedTo = t)
```

Class Thread abstracts away from the details of the Java threads we used to implement our runtime system. The specification is simple, since we just need an operation to determine whether a given thread is free or not. The specification of this operation is as follows:

context Thread::isFree(): EBoolean
  body: assignedTo.oclIsUndefined()

The ready queue. Class ReadyQueue models a queue of tasks, and it provides the runtime system with the following operations:

▷ peekAny: This operation returns any of the tasks that are waiting in the queue. We assume that the implementation of this method is deterministic, i.e., consecutive invocations return the same task as long as the state of the ready queue does not change. Its specification is as follows:

  context ReadyQueue::peekAny(): Task
    body: tasks->any(true)

▷ removed: This operation checks if a given task was removed from the queue. Its specification is as follows:

  context ReadyQueue::removed(t: Task): EBoolean
    body: tasks->excludes(t)

▷ added: This operation complements the previous one, since it checks if a given task was added to the queue. Its specification is as follows:

  context ReadyQueue::added(t: Task): EBoolean
    body: tasks->includes(t)

▷ updated: This operation checks if the queue was updated; recall that this must happen after a task finishes executing and writes a number of messages to the appropriate slots, or every time an application or a process writes a message to a link. Its specification is as follows:

  context ReadyQueue::updated(): EBoolean
    body: Task.allInstances()->select(isReady())->forAll(t: Task |
    self.tasks->includes(t))

The channel board. Objects of class ChannelBoard help manage the channels that are associated with a given runtime system. It provides the following operations:

▷ includes: This is a simple operation to check if a channel is managed by a given channel board. Its specification is as follows:

  context ChannelBoard::includes(c: Channel): EBoolean
    body: channels->includes(c)

▷ updatedOnStart: This operation helps the runtime system check that all of the messages read by a given task are removed from the corresponding channels when it starts executing. Its specification is as follows:

```
context ChannelBoard::updatedOnStart(prestate: ChannelBoard, t: Task): EBoolean
  body: prestate.channels->forAll(
    c1: Channel | t.messagesRead(c1)->forAll(
      m: Message | self.channels->one(c2: Channel |
      c1 = c2 implies c2.read(m))))
```

Not only requires this operation a reference to the task that is going to be executed, but also the previous state of the channel board. Recall that this operation is used in the postcondition of operation onStartTask, which implies that task t has been scheduled for execution and that the messages it processes must have been read from the corresponding slots. Thus, the specification requires that every message that was about to be read by t in the prestate of operation onStartTask, must have been effectively read in the poststate.

▷ updatedOnFinish: This operation complements the previous one since it checks that all of the messages produced by a task are effectively written to the corresponding channels. Its specification is as follows:

```
context ChannelBoard::updatedOnFinished(prestate: ChannelBoard, t: Task):
EBoolean
  body: prestate.channels->forAll(
    c1: Channel | t.messagesRead(c1)->forAll(
      m: Message | self.channels->one(c2: Channel |
      c1 = c2 implies c2.written(m))))
```

As it was the case before, this operation requires both a reference to the task that has just been executed and a reference to the prestate of the channel board. This is required to make it sure in the postcondition of operation onFinishTask that all of the messages produced by this task are effectively written to the appropriate channels.

7. Transformations

In this section, we describe the transformations we have devised to translate Guaraná models into Java code. Unfortunately, the original transformations are very verbose, which makes them not appropriate for an article. In the sequel, we have resorted to a simplified notation in which the executable code is enclosed within angle brackets; the remaining text is assumed to be copied verbatim.

7.1. *Transforming processes*

This transformation is executed on every process found in a model, and it produces a Java class that includes slots, tasks, entry and exit ports declarations, plus a constructor that initialises them all. The transformation is as follows:

```
1: package ⟨Process.name⟩;
```

```
 2: ⟨Import classes⟩
 3: public class ⟨Process.name⟩ extends Process {
 4:     ⟨Slots declaration⟩
 5:     ⟨Tasks declaration⟩
 6:     ⟨EntryPorts declaration⟩
 7:     ⟨ExitPorts declaration⟩
 8:
 9:     public ⟨Process.name⟩() {
10:         ⟨Slots initialisation⟩
11:         ⟨EntryPorts initialisation⟩
12:         ⟨ExitPorts initialisation⟩
13:         ⟨Tasks initialisation⟩
14:     }
15: }
```

Line §1 declares a package with the name of the process being transformed; the classes that correspond to the ports shall also be placed within this package to avoid name clashes with other ports in other processes.

At line §2, the transformation constructs the import statements required to have access to the classes the runtime system provides. Line §3 declares the class for the process, which extends the Process class provided by the runtime system. Inside this class, lines §4–§7 introduce a number of attributes that shall reference the slots, tasks, and ports of which the process being transformed is composed. Lines §9–§14 provide a constructor that initialises the previous objects.

Note that none of the previous declarations or initialisations are difficult, since they just need to iterate over the appropriate properties of a model and output Java declarations or initialisations. The only part that requires a little more explanation is the piece of transformation to initialise ports and tasks. Here we report on the former; the latter is complex enough to deserve a new subsection.

What follows is the piece of transformation to initialise the entry ports:

```
1: ⟨Process.entryPorts->forEach(p: EntryPort) {⟩
2:     ⟨p.name⟩ = new ⟨p.name⟩();
3:     addPort(⟨p.name⟩);
4: ⟨}⟩
```

The loop at line line:script-entry-port-initialisation:1 iterates over the collection of entry ports of a process. At line §2, it outputs a new statement to create a port; recall that every port results in a class with the same name. The following line binds the port and the process to which it belongs by means of the corresponding interslot. The transformation of exit ports is equivalent.

7.2. *Transforming ports*

This transformation is executed on each port independently, and it results in a Java class that includes slot and task declarations, plus a constructor that initialises them all. The transformation is as follows:

```
 1: package ⟨Process.name⟩;
 2: ⟨Import classes⟩
 3: public class ⟨Port.name⟩ extends ⟨Port.getTypeName()⟩ {
 4:     ⟨Slots declaration⟩
 5:     ⟨Tasks declaration⟩
 6:
 7:     public ⟨Port.name⟩() {
 8:         ⟨Slots initialisation⟩
 9:         ⟨Communicator initialisation⟩
10:         ⟨Tasks initialisation⟩
11:     }
12: }
```

Lines §1 and §2 introduce the package declaration and import the classes that are required. In the class declaration at line §3, the operation getTypeName is used to discover the parent class. Inside the class, lines §4 and §5 declare an attribute per slot and task, respectively. Lines §8–§10 inside the constructor deal with the initialisation of the previous declarations. Recall that every port must have a communicator so that it can interact with the corresponding binding component. These tasks are dealt by the runtime system like every other tasks; however, they must be initialised in a way that deviates from the rest. We report on how to initialise tasks and communicators in subsequent subsections.

7.3. *Transforming tasks*

We have grouped tasks into five groups. The following transformation illustrates how to initialise tasks that have several inputs and several outputs. The rest of the groups, except for communicators and a few other tasks, are special cases.

```
1: ⟨Task.name⟩ = new ⟨Task.getTypeName()⟩()
2:     ("⟨Task.name⟩", ⟨Task.inputs.size()⟩, ⟨Task.outputs.size()⟩) {
3:     @Override
4:     public void doWork(Exchange e) {
5:         ⟨Task.executionBody⟩
6:     }
7: };
8: ⟨Bind input slots⟩
9: ⟨Bind output slots⟩
```

```
10: addTask(⟨Task.name⟩);
```

Note that we create anonymous classes to initialise tasks. Each concrete task is derived from a class that is provided by a toolbox, which is in turn discovered by means of a call to operation getTypeName. As a consequence, we need override the doWork operation only.

The piece of transformation at line §5 writes the execution body inside this operation. Lines §8 and §9 deal with binding the input and output slots to the corresponding inputs and outputs of this task. Finally, line §10 adds the task that has been initialised in the previous lines to the enclosing port or process.

7.4. *Transforming communicators*

Transforming a communicator is different from transforming other tasks because they have to interact with binding components. Next, we present the transformation that deals with InCommunicators in entry ports:

```
 1: ⟨Communicator.name⟩ = new InCommunicator(
 2:     "⟨Communicator.jndiName⟩",
 3:     "⟨Communicator.host⟩",
 4:     ⟨Communicator.portNumber⟩);
 5: ⟨Bind output slot⟩
 6: setCommunicator(⟨Communicator.name⟩);
```

InCommunicators are published as remote objects in an RMI registry so that they can be invoked by binding components. This is the reason why the constructor gets a JNDI name which identifies the communicator inside the registry, as well as the host name and the port number where the RMI registry is running. Line §5 binds the single slot connected with this communicator to its output. Finally, line §6 sets the communicator to the enclosing entry port.

OutCommunicators are a little more cumbersome since they need to invoke binding components to write messages. The script to transform them is as follows:

```
 1: Properties props = new Properties();
 2: props.setProperty("java.naming.factory.initial",
 3:     "com.sun.enterprise.naming.SerialInitContextFactory");
 4: props.setProperty("java.naming.factory.url.pkgs",
 5:     "com.sun.enterprise.naming");
 6: props.setProperty("java.naming.factory.state",
 7:     "com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl");
 8: props.setProperty("org.omg.CORBA.ORBInitialHost",
 9:     "⟨Communicator.host⟩");
10: props.setProperty("org.omg.CORBA.ORBInitialPort",
11:     "⟨Communicator.portNumber⟩");
```

```
12:
13: ⟨Communicator.name⟩ = new OutCommunicator(
14:     "⟨Communicator.name⟩",
15:     new JbiExitResourceAdapter("⟨Communicator.jndiName⟩", props));
16: ⟨Bind input slot⟩
17: setCommunicator(⟨Communicator.name⟩);
```

Note that one can have access to a binding component as if it were a regular EJB. This is why lines §1–§11 set up a Properties object with most of the properties required to configure a connection with an EJB. Line §13 initialises the out communicator, which requires to create a JbiExitResourceAdapter; this object implements an adapter to connect to a binding component given its JNDI name and the previous properties. Later, the out communicator is bound to its input slot in line §16 and it is registered with the enclosing exit port in line §17.

7.5. *Starter transformation*

The previous transformations deal with creating the classes that implement the processes and the ports of which a solution is composed. These are the pieces that need now put together by means of the following starter transformation:

```
 1: package ⟨Solution.name⟩;
 2: ⟨Import classes⟩
 3: public class ⟨Solution.name⟩ {
 4:     public static void main(String[] args) {
 5:         Runtime r = new Runtime(⟨number of threads⟩);
 6:
 7:         ⟨processes->forEach(p: Process) {⟩
 8:             Process ⟨p.name⟩ = new ⟨p.name⟩();
 9:             ⟨p.name⟩.setTaskStateMonitor(r);
10:
11:             Collection<Port> ⟨p.name+"Ports"⟩ = ⟨p.name⟩.getAllPorts();
12:             for (Port pt: ⟨p.name+"Ports"⟩) {
13:                 pt.setTaskStateMonitor(r);
14:             }
15:         ⟨}⟩
16:         r.start();
17:     }
18: }
```

Line §5 instantiates the runtime system with a given number of threads. The loop in lines §7–§15 iterates over every process in the model, initialises an instance, and binds it to the runtime system we have created previously by means of operation

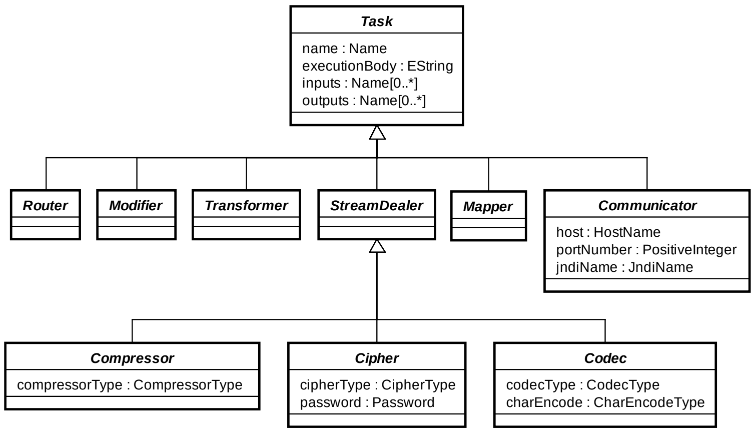

Figure 7. Partial view of Guaraná's general-purpose task toolbox.

setTaskStateMonitor; furthermore, the ports are retrieved and registered with the runtime system as well. Registering a process or a port with a runtime system allows it to have access to their tasks and slots and to initialise its internal data structures. Finally, the runtime system is started at line §16. From this point on, the runtime system behaves as an autonomous object according the semantics we described in Section §6. The generated Java code from all model-to-text transformations is ready to be compiled and executed on the runtime system. Software engineers only need to configure a number of binding components, in Open ESB; they shall be used to communicate with the applications being integrated.

8. General-purpose task toolbox

In the previous sections, we have dealt with tasks in an abstract manner. In this section, we provide an insight into a general-purpose task toolbox that accompanies Guaraná. Figure §7 sketches the abstract classes that help classify tasks according to their intended semantics, namely: routers, which do not change the state of the messages they process, but route them through a process; modifiers, which help add or remove data from messages, but do not alter their schemata; transformers, which help transform one or more messages into a new message with a different schema; stream dealers, which allow to compress, cipher, or encode messages; mappers, which change the format of the messages they process, e.g., from a stream of bytes into an XML document; and communicators, which are used to interact with binding components.

The remaining classes in the toolbox are concrete classes, cf. Tables §6 - §11.

Icon	Class	Description
	Correlator	Analyses inbound messages and outputs sets of correlated ones.
	Merger	Merges messages from different input slots into one output slot.
	Resequencer	Reorders messages into sequences with a pre-established order.
	Filter	Filters out unwanted messages.
	IdempotentTransfer	Removes duplicated messages.
	Dispatcher	Dispatches a message to exactly one slot.
	Distributor	Distributes messages to one or more slots.
	Replicator	Replicates a message to all of the output slots.
	SemanticValidator	Validates the semantics of a message.

Table 6. Router tasks.

Icon	Class	Description
	Slimmer	Removes contents from the body of a message.
	ContentEnricher	Adds contents to the body of a message.
	HeaderEnricher	Adds contents to the header of a message.
	HeaderPromoter	Promotes a part of the body of a message to its header.
	HeaderDemoter	Demotes a part of the header of a message to its body.

Table 7. Modifier tasks.

9. Validation

The validation was carried out by means of a series of case studies, some of which were carried out in co-operation with local companies. Due to space constraints we report on only one case study. It consists of a non-trivial, real-world integration problem that builds on a project to enhance the functionality of the call centre system at Unijuí University. The goal of this EAI solution is to automate the invoicing of personal phone calls that employees make using the University's phones.

This EAI solution involves three applications, namely: a Call Centre System (CCS), a Payroll System (PS) and a Mail Server (MS). Each application runs on a different platform and was designed without integration concerns in mind. The CCS records every call every employee makes from any of the University's phones;

Icon	Class	Description
■→▥	Translator	Transforms the body of a message from one schema into another.
■→▥▤	Splitter	Splits a message into several messages.
▤▥→■	Aggregator	Constructs a new message from several messages produced previously by a Splitter.

Table 8. Transformer tasks.

Icon	Class	Description
◇→◊	Zipper	Compresses a message.
◊→◇	Unzipper	Decompresses a message.
◊→♦	Encrypter	Encrypts a message.
♦→◊	Decrypter	Decrypts a message.
◊→◈	Encoder	Encodes a message.
◈→◊	Decoder	Decodes a message.

Table 9. Stream-dealer tasks.

it can identify who the employee is because they have a personal number that they have to enter before dialling the number they wish to call. This number is used to correlate phone calls with the information in the PS. The MS runs the University e-mail service, and is used for notification purposes. Note that due to space limitations, the presentation omits a number of auxiliary systems, e.g., a Human Resource System that provides personal information about the employees or an SMS notification system Unijuí uses to text their employees, and, besides, some implementation details, e.g., the internal structure of the ports or the schema of the messages.

Figure §8 shows the EAI solution we have devised using Guaraná. It is composed of three wrapping processes and one orchestration process. The integration flow begins at entry port (1), which periodically reads the CCS log to find new phone

Icon	Class	Description
◊→■	Stream2XMLMapper	Maps a stream of bytes onto an XML message.
■→◊	XML2StreamMapper	Maps an XML message onto a stream of bytes.

Table 10. Mapper tasks.

Icon	**Class**	**Description**
→◊	InCommunicator	Reads messages from a binding component.
◊→	OutCommunicator	Writes messages to a binding component.

Table 11. Communicator tasks.

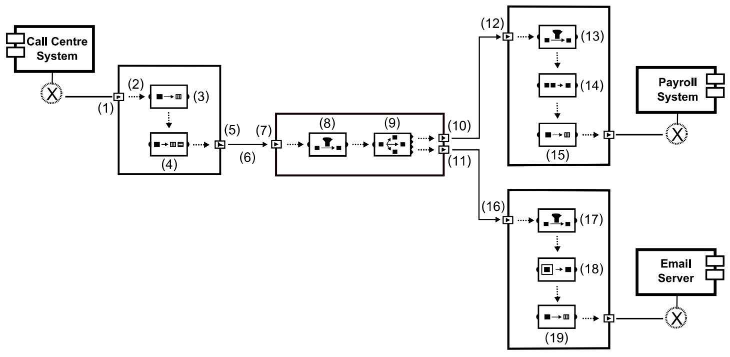

Figure 8. The EAI solution designed with Guaraná.

calls. This port produces messages that contain information about several phone calls, and transfers them to slot (2). Task (3) is a translator that transforms inbound messages into outbound messages that conform to the canonical schema that was defined for this EAI solution. Task (4) is a splitter that breaks inbound messages into several messages, each of which is related to one call only. Exit port (5) writes these messages to integration link (6), from which the orchestration process can read them by means of entry port (7). The goal of this process is to filter out (8) messages about toll-free calls, i.e., only messages that have a cost for the university are allowed to remain in the workflow. Task (9) is a replicator that copies the messages it receives to exit ports (10) and (11).

The messages that are sent to port (10) are in turn transferred to port (12), which is a part of the wrapping process of application PS. First, this message goes through filter (13) to ensure that only messages with a debit amount remain in the workflow. Task (14) is an idempotent transfer that prevents duplicated messages to be sent to the application. Task (15) is a translator that transforms the messages it receives into new messages for application PS.

The messages that are sent to port (11) are in turn transferred to port (16), which is a part of the wrapping process of application MS. Filter (17) ensures that only messages that have an employee e-mail remain in the workflow. This task prevents application MS from receiving messages that cannot be notified. The notification summarises the most useful information of every call, e.g, destination number, call duration, cost, destination city, or state; however, it does not contain

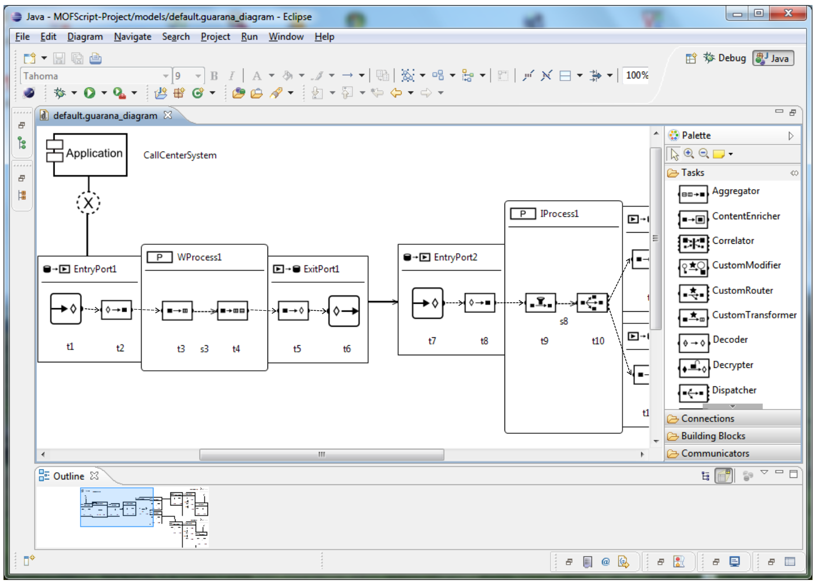

Figure 9. Screenshot of the Eclipse Guaraná Plug-in.

any technical details since they are removed by task (18), which is a slimmer. Task (19) is a translator that transforms the messages it receives into outbound messages that conform to the schema used by application MS.

Figure §9 shows an screenshot of the Eclipse Guaraná plug-in we have developed in the laboratory. The canvas shows part the EAI solution we have described above.

10. Conclusions

Companies are increasingly relying on ESBs to implement EAI solutions. They provide a number of binding components software engineers can use to connect to almost every existing application and a language to design orchestration processes. The most common such language is BPEL; unfortunately, it is rather low-level, since it does not support directly many common enterprise integration patterns. This has motivated many authors to work on domain-specific tools to endow ESBs with the appropriate abstraction level to model EAI solutions. There are many proposals in the literature, of which Camel, Spring Integration and BizTalk are the most closely related to ours.

In this article, we have introduced Guaraná, which is a proposal that aims at increasing the level of abstraction of ESBs, in an attempt to make it easier for software engineers to devise, implement, and deploy their EAI solutions. Guaraná is classified as an external DSL. We have presented Guaraná's abstract and concrete syntaxes, we have described the runtime system that supports the proposal, the transformations that translate Guaraná models into Java code ready to be compiled and executed, and a generic task toolbox; we have also reported on the results of a project we have carried out to validate our proposal. We have also compared our proposal to others in the literature, and our conclusion is that Guaraná's key features are the following: it provides explicit support to devise EAI solutions using enterprise integration patterns by means of a graphical model; its DSL is graphical, and it enables software engineers devise EAI solutions at a high-level of abstraction; not only provides the DSL the view of a process, but also a view of the whole set of processes of which EAI solutions are composed; both processes and tasks can have multiple inputs and multiple outputs; and, finally, its runtime system provides a task-based execution model that is usually more efficient than the process-based execution models in current use. We have also implemented a graphical editor for our DSL a set of scripts to transform our models into Java code ready to be compiled and executed.

11. Acknowledgments

The work on which we report in this article was supported by the European Commission (FEDER), the Spanish and the Andalusian R&D&I programmes by means of grants TIN2007-64119, P07-TIC-2602, P08-TIC-4100, TIN2008-04718-E, and TIN2010-21744). Rafael Z. Frantz was also supported by the Evangelischer Entwicklungsdienst e.V. (EED). Antonia M. Reina was also supported by grant TIN-2007-67843-C06.

Bibliography

1. David Chappel. *Enterprise Service Bus: Theory in Practice.* O'Reilly, 2004.
2. Binildas A. Christudas. *Service-Oriented Java Business Integration.* Packt, 2008.
3. Rafael Corchuelo, José Luis Arjona, and David Ruiz. Wrapping web data islands. *Journal of Universal Computer Science*, 14(11):1808–1810, 2008.
4. Bertrand David, René Chalon, and Bertrand T. David. Orchestration modeling of interactive systems. In *Human-Computer Interaction*, pages 796–805, 2009.
5. Jeff Davies, David Schorow, Samrat Ray, and David Rieber. *The Definitive Guide to SOA: Enterprise Service Bus.* Apress, 2008.
6. Birgit Demuth. The dresden OCL toolkit and its role in information systems development. In *Int. Conf. on Information Systems Development*, pages 1–12, 2004.
7. Teduh Dirgahayu, Dick Quartel, and Marten van Sinderen. Designing interaction behaviour in service-oriented enterprise application integration. In *ACM Symposium on Applied Computing*, pages 1048–1054, 2008.
8. Mark Fisher, Jonas Partner, Marius Bogoevici, and Iwein Fuld. *Spring Integration in Action.* Manning, 2010.
9. The Eclipse Foundation. Eclipse Modeling Project. http://www.eclipse.org/modeling/gmp.
10. Martin Fowler. *Domain-Specific Languages.* Addison-Wesley, 2010.
11. Christoforos N. Hadjicostis and George C. Verghese. Monitoring discrete event systems using Petri Net embeddings. In *Int. Conf. on Applications and Theory of Petri Nets*, pages 188–207, 1999.

12. Claus Hagen and Gustavo Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, 2000.
13. Brent Hailpern and Perry L. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–462, 2006.
14. Gregor Hohpe and Hsue-Shen Tham. Enterprise integration patterns with BizTalk Server 2004. Technical report, ThoughtWorks, Inc., 2004.
15. Gregor Hohpe and Bobbie Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.
16. Claus Ibsen and Jonathan Anstey. *Camel in Action*. Manning, 2010.
17. Ashok Iyengar, Vinod Jessani, and Michele Chilanti. *WebSphere Business Integration Primer: Process Server, BPEL, SCA, and SOA*. IBM Press, 2008.
18. Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, and Fiona A.C. Polack. Raising the level of abstraction in the development of GMF-based graphic model editors. In *ICSE Workshop on Modeling in Software Engineering*, pages 13–19, 2009.
19. Lingxi Li, Christoforos N. Hadjicostis, and Ramavarapu S. Sreenivas. Designs of bisimilar Petri Net controllers with fault tolerance capabilities. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 38(1):207–217, 2008.
20. David Messerschmitt and Clemens A. Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press, 2003.
21. Sergei Moukhnitski, Harold Campos, Stephen Kaufman, Peter Kelcey, David Peterson, and George Dunphy. *Pro BizTalk 2009*. Apress, 2009.
22. Jon Oldevik, Tor Neple, Roy Grønmo, Jan Øyvind Aagedal, and Arne-Jørgen Berre. Toward standardised model to text transformations. In *European Conference on Model Driven Architecture*, pages 239–253, 2005.
23. Gordon D. Plotkin. The origins of structural operational semantics. *Journal Logic and Algebraic Programming*, 60–61:3–15, 2004.
24. Tijs Rademakers and Jos Dirksen. *Open-Source ESBs in Action*. Manning, 2009.
25. Poornachandra Sarang, Matjaž Juric, and Benny Mathew. *Business Process Execution Language for Web Services BPEL and BPEL4WS*. Packt, 2006.
26. Thorsten Scheibler and Frank Leymann. Realizing enterprise integration patterns in WebSphere. Technical report, University of Stuttgart, 2005.
27. Thorsten Scheibler, Ralph Mietzner, and Frank Leymann. EAI as a service: Combining the power of executable EAI patterns and SaaS. In *Int. IEEE Enterprise Distributed Object Computing Conference*, pages 107–116, 2008.
28. Mostafa H. Sherif. *Handbook of Enterprise Integration*. Auerbach, 2009.
29. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2008.
30. W3C. WS-Choreography specification, 2004.
31. Jeff Weiss. Aligning relationships: Optimizing the value of strategic outsourcing. Technical report, IBM, 2005.
32. Edward Willink. MDT/OCLinEcore. http://wiki.eclipse.org/MDT/OCLinEcore.
33. Xin Yuan. Prototype for executable EAI patterns. Technical report, University of Stuttgart, 2008.