# On the Formalisation of an Application Integration Language Using Z Notation

Mauri J. Klein, Sandro Sawicki, Fabricia Roos-Frantz and Rafael Z. Frantz
*UNIJUÍ University, Department of Exact Sciences and Engineering, Bairro Universitário, Ijuí, Brazil*

Keywords: Enterprise Application Integration, Domain-Specific Language, Formalisation, Z Notation.

Abstract: Companies rely on applications in their software ecosystem to provide IT support for their business processes. It is common that these applications were not designed taking integration into account, which makes hard their reuse. Enterprise Application Integration (EAI) focuses on the design and implementation of integration solutions. The demand for integration has motivated the rapid growing of tools to support the construction of EAI solutions. Guaraná is a proposal that can be used to design and implement EAI solutions, and different from other proposals includes a monitoring system that can be configured using a rule-based language to endow solutions with fault-tolerance. Although Guaraná is available, it has not been formalised yet. This is a limitation since it is not possible to validate the rules written by software engineers, using the rule-based language, to ensure that all possibilities of failure in a given EAI solution are covered. Besides, it is not possible to generate automatically these rules based on the semantics of the EAI solution. In this paper we provide a formal specification of the language provided by Guaraná to design EAI solutions, using Z notation.

## 1 INTRODUCTION

Business processes frequently demand support from applications. Companies count on software ecosystems (Messerschmitt and Szyperski, 2003) that are mostly composed of applications that are legacy systems, packages purchased from third parties, or developed at home to solve a particular problem. As a result, these heterogeneous software ecosystems contain applications that were not designed taking integration into account. Integration is necessary, chiefly because it allows to reuse two or more applications to support new business processes, or because the current business processes have to be optimised by interacting with other applications within the software ecosystem.

Enterprise Application Integration (EAI) provides methodologies and tools to design and implement EAI solutions. The goal of an EAI solution is to keep a number of applications' data in synchrony or to develop new functionality on top of them, so that applications do not have to be changed and are not disturbed by the EAI solution (Hohpe and Woolf, 2003). Figure 1 illustrates a typical EAI solution, composed of two processes that contain the integration logic and five communication ports that allow for the interaction with applications and between processes.

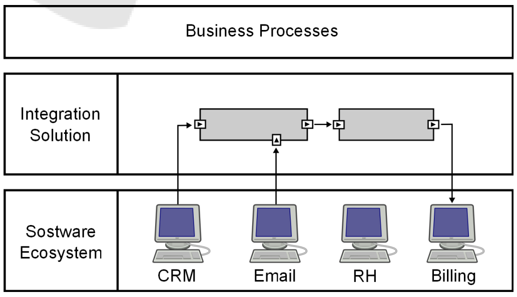

Figure 1: Typical integration solution.

In the last few years, the demand for integration has motivated the rapid growing of tools to support the design and the implementation of EAI solutions, such as Camel (Ibsen and Anstey, 2010), Spring Integration (Fisher et al., 2010), Mule (Dossot and D'Emic, 2009), and Guaraná (Frantz et al., 2011). As applications, an EAI solution can fail during its execution. A desirable feature on integration tools is fault-tolerance, so that an EAI solution can keep running despite of the occurrence of failures. Error monitoring is an important activity in fault-tolerance, since it enables the detection of errors. Camel, Spring Integration, and Mule provide an error detection mechanism mostly based on the widely-used try-catch (Goodenough, 1975). To the best of our knowledge, Guaraná is the single proposal

J. Klein M., Sawicki S., Roos-Frantz F. and Z. Frantz R. (2014).
On the Formalisation of an Application Integration Language Using Z Notation.
In *Proceedings of the 16th International Conference on Enterprise Information Systems*, pages 314-319
DOI: 10.5220/0004967003140319

that provides an error detection mechanism based on a monitoring system that can be configured using a rule-based language to help in the detection of errors (Frantz et al., 2012). Guaraná is composed of a domain-specific language (DSL) to design EAI solutions and a framework plus a runtime system that allows for the implementation and execution of EAI solutions. By means of the rule-based language software engineers can write rules to express the expected behaviour and then the correct execution of an EAI solution designed with Guaraná DSL.

The construction of a model that precisely describes an EAI solution and allows for an analysable model depends on formality. Although Guaraná DSL is available, it has not been formalised yet. Its metamodel is expressed using the Unified Modeling Language (UML) notation (OMG, 2011) and constrained using the Object Constraint Language (OCL) (OMG, 2012). Thus, it is not possible to reason about EAI solutions designed using Guaraná DSL. Reasoning would allow us not only to validate whether the rules written by software engineers to detect errors cover all possibilities of failure in a given EAI solution, but also build the foundations to generate in an automated fashion these rules based on the semantics of the EAI solution.

According to van Lamsweerde (2000), a formal specification must be expressed in a language made of three components: syntax, semantics, and rules for reasoning on the specification. The latter component enable analysis on the specification in an automated fashion. Therefore, in this paper, we start the formalisation of Guaraná DSL by providing a formal specification of its abstract syntax using *Z notation* (Spivey, 1992).

Formal Specification has been studied since of the principles of Computer Science (Floyd, 1967; Dijkstra, 1975; Wing et al., 1999). The interest on formality, mainly concerning language definition, has been growing continually since that point (Bowen and Hinchey, 2006; Harel and Rumpe, 2004; Shroff and France, 1997). In Harel and Rumpe (2004), the authors argue that any language, no matter how it is, whether textual or visual, for programming or design, it must be complete, with clear syntactic rules and rigid description of their meaning. A sound language definition must rigorously define the mapping of each syntactic element to its meaning. Often, language designers make such mapping informally. According to the same authors, providing a formal semantics for the languages, together with tools for analysing and executing their behaviour and for consistency checking, is crucial (Harel and Rumpe, 2004).

In the literature, there are a number of works providing formal specifications for modelling languages, likewise we propose in this paper. In Mostafa et al. (2007); Shroff and France (1997); Kim and Carrington (1999), the authors formalise UML diagrams using Z notation. Hong and Mannino (1995) provide a denotational semantics for UML. Kaliappan and König (2012) give formal semantics to UML activity diagrams using the compositional Temporal Logic of Actions. In another work, Vidal et al. (2012) formalise the semantics of OWL-S services choreography using Petri nets. As argued by van Lamsweerde (2000), the major contribution of a formal specification is that it can be automatically manipulated by tools for a wide variety of purposes, such as animations of the specification in order to check its adequacy, generation of test cases and counterexamples.

The rest of this paper is organised as follows: Section 2 gives an overview of the metamodel of Guaraná DSL; Section 3 presents the formal specification of the abstract syntax of Guaraná DSL; Section 4 discusses challenges and future work; and, finally, Section 5 presents our conclusions.

## 2 Guaraná DSL

In this section, we provide an overview of the abstract syntax of Guaraná DSL. Figure 2 depicts the UML-based metamodel as introduced by authors in Frantz et al. (2011).

In this metamodel, *Solution* is the root class 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 *Link*s. Class *Process* contains the integration logic necessary to interact with applications within the software ecosystem and to process data that flows in the EAI solution. A *Process* is composed of at least one *EntryPort*, at least one *ExitPort*, at least one *Task*, and at least two *Slot*s. *Port*s are composed of tasks and slots, that get connected by *Link*s; these, in turn, can be either *ApplicationLink*s, which connect *Application*s to *Port*s, or *IntegrationLink*s, which connect *EntryPort*s to *ExitPort*s. 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 atomic activity that must be carried out by the task. Inside the execution body, a software engineer may reference the messages held in the inputs and outputs.

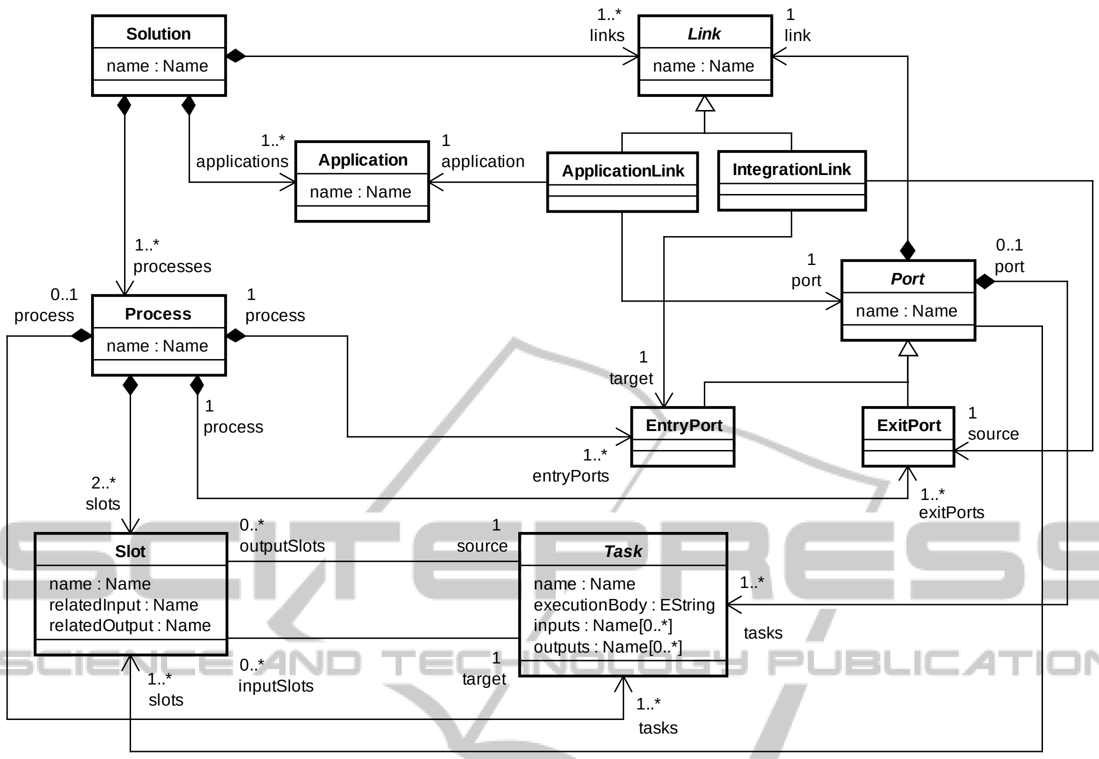

Figure 2: Guaraná DSL Metamodel from Frantz et al. (2011).

## 3 FORMALISATION

In this section we formalise the abstract syntax of Guaraná DSL using Z schemas, describing its classes and relationships.

### 3.1 Types and Constraints

In an EAI solution designed with Guaraná DSL, several building blocks are identified by names. For this purpose, we introduce the set of all names as basic type of the specification. Similarly, we introduce a basic type for the set of all scripts, which are a piece of source code that implements the business logic of tasks, cf. expression (1).

$$[Name, Script] \tag{1}$$

The free type defined in expression (2) defines *Type* to be a set containing the main building blocks of Guaraná DSL.

$$Type ::= Task|Slot|Process|Application|Link \tag{2}$$

In order to ensure that all building blocks have a unique name, we write out explicitly a constraint denoted by expression (3). Two variables *type1* and *type2* should not have the same name; therefore, if *type1.name* equals *type2.name*, then *type1* equals *type2*.

$$\forall\, type1, type2 : Type \bullet \quad type1.name = type2.name \Leftrightarrow type1 = type2 \tag{3}$$

We defined the remaining sets using Z schemas, which are composed of two parts: declarative and predicative. In order to keep the formalisation comprehensive and facilitate initial understanding of our proposal, only *Process*, *Port* and *Solution* schemas are composed of predicative parts.

### 3.2 Schemas Definition

In Z notation, a state schema that formalises the static aspect of a class is called *Class Schema*. It consists of a declarative part in which variables are declared and a predicative part that contains a predicate (expressed as a list of conjuncts) constraining variable values. The variables in a class schema represent attributes and object identifiers of a class.

The *Task* schema describes the building block *Task*, in which four variables are declared: name, inputs, outputs, and executionBody. We define the declarative part of the *Task* schema using variables without any relationship between others Z schemas. The sets *Name* and *Script* were declared in expression (1), and variable *name* is restricted by expression (3).

*Task*

$name\colon Name$
$inputs\colon \mathbb{F}\ Name$
$outputs\colon \mathbb{F}\ Name$
$executionBody\colon \mathbb{F}\ Script$

We identified sets *EntryPort* and *ExitPort* with a *name* and a *type* of *Task*. The set *Port* is composed of the union of *EntryPort* and *ExitPort*. The intersection of *EntryPort* and *ExitPort* forms an empty set. This can be observed at *Port* schema definition.

*EntryPort*

$name\colon Name$
$comunicator, inComunicator\colon \mathbb{F}\ Task$

*ExitPort*

$name\colon Name$
$comunicator, outComunicator\colon \mathbb{F}\ Task$

*Port*

$entryPorts\colon \mathbb{F}\ EntryPort$
$exitPorts\colon \mathbb{F}\ ExitPort$

$entryPorts \cap exitPorts = \emptyset$

In Guaraná DSL, a Slot can connect two tasks or a task with a port. Thus, in the specification, the *Slot* schema describes the building block Slot, in which three variables are declared: *name*, *slots*, and *interslots*. The Slot that connects two tasks is modelled as a set of binary relations between the sets *Task* and *Task*, while connections between Task and Port is modelled as a set of binary relations between the sets *Task* and *Port*.

*Slot*

$name\colon Name$
$slots\colon Task \leftrightarrow Task$
$interslots\colon Task \leftrightarrow Port$

A process of a Guaraná solution is identified by a name and composed of slots, tasks, and ports (entryPorts and exitPorts). A process is specified as a schema, where the main predicates (constraints) and declarations of subsets of its composition are defined. In its predicative part, the slot definition has relation with two distinct tasks. In other words, for all *s1* from set *Slot*, where *t1*, *t2* belonging to set *Task*, the term *s1.slots* relates task *t1* with task *t2*. The definition interslot relates bilaterally *Task* with *Port*. Thus, for all *s1* from set *Slot*, where *t1* belongs to set *Task* and *pt1* belongs to set *Port*, the term *s1.interslot* relates task *t1* with port *pt1*.

*Process*

$name\colon Name$
$slots\colon \mathbb{P}\ Slot$
$tasks\colon \mathbb{P}\ Task$
$ports\colon \mathbb{P}\ Port$

$\forall\ s1\colon slots;\ t1, t2\colon \mathbb{P}\ tasks \bullet s1.slots = t1 \times t2$
$\forall\ s1\colon slots;\ t1\colon \mathbb{P}\ tasks;\ pt1\colon \mathbb{P}\ ports \bullet s1.interslots = t1 \times pt1 \wedge \#s1.interslots =$
$\#EntryPort + \#ExitPort$
$allTasks = tasks \cup EntryPort.tasks \cup ExitPort.tasks$
$\#slots \geq 2$
$\#tasks \geq 1$
$\#ports \geq 2$

After the declarations of sub-sets of a process and the definition of some predicates, we declare the other sets that compose an integration solution. Therefore, we describe at this stage, the Application and Link schemas. Application is composed of a name, which uniquely identifies the application and two sets of ports: *entryPorts* and *exitPorts*. Link is composed of a name which uniquely identifies the Link, and two other sets of relationships: *integrationLink* and *applicationLink*. The former contains relationships between *EntryPorts* and *ExitPorts*, and the latter relationships between *Application* and *Port*. With the declaration of these two schemas, we can represent the general formalisation of the EAI solution, which is composed of *Process*, *Application*, *Link* and *Port*.

*Application*

$name\colon Name$
$entryPorts\colon \mathbb{P}\ EntryPort$
$exitPorts\colon \mathbb{P}\ ExitPort$

*Link*

$name\colon Name$
$integrationLink\colon \mathbb{P}\ (EntryPort \leftrightarrow ExitPort)$
$applicationLink\colon \mathbb{P}\ (Application \leftrightarrow Port)$

Finally, we specify an EAI solution. To this end, we define the *Solution* schema, which has two constraints. The first constraint introduces integrationLink as the relation between *EntryPort* of process *p1* and *ExitPort* of process *p2*, where *p1* must be different from *p2*. For all *l1* in set *Link* and *p1*, *p2* in set Process, where *l1.integrationLink* relates *EntryPort* from process *p1*, with *ExitPort* from process *p2*, such as *p1* and *p2* must be different. The second constraint represents the applicationLink as the relationship between an *Application* and a *Port* ensuring that two different Applications will never be connected to

```
Solution
 name: Name
 processes: ℙ Process
 applications: ℙ Application
 links: ℙ Link
 ports: ℙ Port
─────────────
 ∀ l1: links; p1, p2: processes
     • l1.integrationLink = p1.entryPorts ↔ p2.exitPorts ∧ p1 ≠ p2
 ∀ l1, l2: links; app1, app2: ℙ applications; pt1: ℙ ports
     • l1.applicationLink = app1 ↔ pt1 ∧ l2.applicationLink = app2 ↔ pt1
       ⇔ app1 = app2
 # links ⩾ 1
 # processes ⩾ 1
 # applications ⩾ 1
```

the same port. Thus, for all *l1*, *l2* from set Link, and *app1*, *app2* from set Application, and *pt1* from set *Port*, where *l1.applicationLink* relates *app1* with *pt1* and *l2.applicationLink* relates *app2* with *pt1*, if and only if, application *app1* is equals to *app2*.

## 4 CHALLENGES AND FUTURE WORK

There is a long way to go before Guaraná DSL can have a well-defined semantics. To the best of our knowledge, this is the first proposal addressing formalisation of this language. Amongst the challenges raised, we are aware of the limitations of tool support for Z specifications. Although Z notation has been widely used to provide formal specification, there are few tool support to assist in the detection of semantic errors, which makes difficult the validation of the specification. We have to investigate which technique can be combined with Z to cope with this limitation.

The main goals in our ongoing research are: *i)* formalise the semantics and the rules for reasoning on the domain-specific language of Guaraná proposal; *ii)* validate the rules for error detection, so that we can ensure they cover every possibility of failure in an EAI solution designed with Guaraná DSL; *iii* automatically generate these rules based on the semantics of an EAI solution; and, *iv)* generate test cases from the EAI solution specification.

## 5 CONCLUSIONS

Currently, the lack of formal semantics on Guaraná DSL prevents the automated reasoning of the EAI solutions designed with this language. In this paper, we presented a formal specification of the abstract syntax of Guaraná DSL using Z notation. This is a first step towards the formalisation of the language.

We seek, with the use of formalism, to eliminate ambiguities and inconsistencies that would only be detected during development and testing phases. Furthermore, the correction of the specification in a preliminary stage of the development process avoids errors that can occur later, resulting in more onerous problems. In this context, we know that a key challenge for software engineers is to ensure the reliability of designed EAI solutions. For this reason, the current paper addressed the formal specification of the abstract syntax of Guaraná DSL, an initial step towards the formalisation of the whole language, with that we want to ensure that an EAI solution is in accordance with what was specified.

Using the mathematical basis of a formal specification technique, such as Z notation, it became possible to explore the abstract syntax of Guaraná DSL with precision and rigour. For that, we used Z schemas to describe the Unified Modeling Language classes and the Object Constraint Language constraints of the Guaraná DSL metamodel. Thus, the formalisation of Guaraná DSL would contribute to the validation of an EAI solution, allowing to check that its specification meets the integration solution requirements. Besides, we believe that our work on the formalisation of Guaraná DSL shall contribute to the improvement and adoption of this proposal.

## ACKNOWLEDGEMENTS

The research work on which we report in this paper was supported by CAPES, FAPERGS, and the internal Research Programme 2013/14 at UNIJUI University.

## REFERENCES

Bowen, J. P. and Hinchey, M. G. (2006). Ten Commandments of Formal Methods ...Ten Years Later. *IEEE Computer*, 39(1):40–48.

Dijkstra, E. W. (1975). Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457.

Dossot, D. and D'Emic, J. (2009). *Mule in Action*. Manning.

Fisher, M., Partner, J., Bogoevici, M., and Fuld, I. (2010). *Spring Integration in Action*. Manning.

Floyd, R. W. (1967). Assigning meanings to programs. In *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31.

Frantz, R. Z., Corchuelo, R., and Molina-Jiménez, C. (2012). A proposal to detect errors in Enterprise Application Integration solutions. *Journal of Systems and Software*, 85(3):480–497.

Frantz, R. Z., Quintero, A. M. R., and Corchuelo, R. (2011). A Domain-Specific Language to Design Enterprise Application Integration Solutions. *Int. J. Cooperative Inf. Syst.*, 20(2):143–176.

Goodenough, J. B. (1975). Exception handling: Issues and proposed notation. *Communications of the ACM*, 18(12):683–696.

Harel, D. and Rumpe, B. (2004). Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72.

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

Hong, S. N. and Mannino, M. V. (1995). Formal semantics of the unified modeling language {LU} . *Decision Support Systems*, 13(3-4):263–293.

Ibsen, C. and Anstey, J. (2010). *Camel in Action*. Manning.

Kaliappan, P. S. and König, H. (2012). On the Formalization of UML Activities for Component-based Protocol Design Specifications. In *Proceedings of the 38th International Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM'12, pages 479–491.

Kim, S.-K. and Carrington, D. (1999). Formalizing the UML Class Diagram Using object-Z. In *Proceedings of the 2Nd International Conference on The Unified Modeling Language: Beyond the Standard*, UML'99, pages 83–98.

Messerschmitt, D. and Szyperski, C. A. (2003). *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press.

Mostafa, A. M., Ismail, M. A., Bolok, H. E., and Saad, E. M. (2007). Toward a Formalization of UML2.0 Metamodel using Z Specifications. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, volume 1, pages 694–701.

OMG (2011). UML 2.4 Superstructure Specification.

OMG (2012). OCL 2.3.1 Object Constraint Language.

Shroff, M. and France, R. B. (1997). Towards a formalization of UML class structures in Z. In *COMPSAC*, pages 646–651. IEEE Computer Society.

Spivey, J. M. (1992). *The Z Notation: A Reference Manual*. Prentice Hall.

van Lamsweerde, A. (2000). Formal specification: a roadmap. In *ICSE - Future of SE Track*, pages 147–159.

Vidal, J. C., Lama, M., and BugarÃn, A. (2012). Toward the use of Petri nets for the formalization of OWL-S choreographies. *Knowledge and Information Systems*, 32(3):629–665.

Wing, J., Woodcock, J., and Davies, J. (1999). In *FM'99 - Formal Methods: World Congress on Formal Methods in the Development of Computing Systems*, volume 1708–1709 of *LNCS*. Springer-Verlag.