Data-driven application configuration

David M. de Matos and Nuno J. Mamede

L^{2}F/INESC-ID/IST - Spoken Language Systems Laboratory
Rua Alves Redol 9, 1000-029 Lisboa, Portugal
david.matos@inesc-id.pt, nuno.mamede@inesc-id.pt

Abstract

Constructing modular applications from existing parts is difficult if there are mismatches due to input or output semantic differences during module interconnection. In order to minimize the effort of building such applications, and also as a guideline for designing modular applications from scratch, we propose an architecture in which modules are able to interface with each other without having to be reprogrammed. The architecture can be completely described using a small number of concepts. These factors allow rapid application building and reconfiguration with minimal manual intervention, potentiating module reuse and reducing the effort invested in building new applications.

1. Introduction

When building modular applications, it is possible to use parts that have been constructed by third parties, that solve part of the global problem. While this way of work is desirable because it promotes reuse, reducing the global development effort, it is all but straightforward: in fact, integrating foreign modules is almost never a simple task. The integration effort may become so expensive that it may seem better to build everything from scratch.

Managing these architectures is, thus, a challenging task and their complexity can be a serious hurdle when trying to bring together different components. Although not limited to the group, this problem also occurs when building natural language processing (NLP) applications and on various levels: from file-format handling or network-level communication to interaction between modules in a large application.

Here, we are concerned primarily with the latter aspect, even though the discussion could be applied to other levels, e.g. communication issues in a distributed application. We consider such lower-level aspects transport issues, though, that may be dealt with separately. Thus, CORBA (OMG, nda) and similar architectures are not an issue, since what we are concerned with is the way modules within an application exchange data and how to describe the way they do it.

We have two goals: to define a uniform way for modules to produce/consume data; and to define a uniform module interoperability model. We intend for these aspects to be realized complementarily: the latter will be a consequence of the former. In aiming at reaching these goals we are also promoting reuse and easy construction/configuration, since we provide a way for describing module interfaces for use with existing resources.

This document is organized as follows: the data model is defined in section 2.; a working example is presented in section 3.; and, finally, some concluding remarks and directions for evolution are presented.

2. Model

This section presents the architectural model. The first part presents structural aspects; the second part details the data model; the third part deals with semantics; and the fourth part details the implied application specification.

2.1. Structural aspects

We consider modular applications in which the modules exchange data through connections between their ports. These objects as well as their properties and relationships are presented here.

Definition 1 (portsets) *Let $\mathbb{M}$ be the set of all modules in an application. For a module m, the following portsets are defined: $\mathbb{O}^m$ (all output ports); $\mathbb{I}^m$ (all input ports); and $\mathbb{P}^m = \mathbb{I}^m \cup \mathbb{O}^m$ (all ports). In addition, $\mathbb{I}^m \cap \mathbb{O}^m = \varnothing$. We use p_i^m to denote the i-th port of m (i ranges over the corresponding portset).*

The definition for connection, while still a structural aspect, is better presented below (see def. 6).

2.2. Data model

Definition 2 (unrestricted grammar) *Unrestricted grammars (Lewis and Papadimitriou, 1981, def. 5.2.1.) are quadruples $G = (V, \Sigma, R, S)$, where V is an alphabet; Σ is the set of terminal symbols ($\Sigma \subseteq V$); $(V - \Sigma)$ is the set of nonterminal symbols; S is the start symbol; and R is the set of rules (finite subset of $(V^*(V - \Sigma)V^*) \times V^*$). Direct derivation (eq. 1), derivation (eq. 2), and generated language (eq. 3) are defined as follows:*

$$u \underset{G}{\Rightarrow} v \text{ iff } w_1, w_2 \in V^*, (u', v') \in R, \quad u = w_1 u' w_2 \wedge v = w_1 v' w_2 \tag{1}$$

$$w_0 \underset{G}{\Rightarrow} w_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} w_n \Leftrightarrow w_0 \overset{*}{\underset{G}{\Rightarrow}} w_n \tag{2}$$

$$L(G) = \{w \mid w \in \Sigma^* \wedge S \overset{*}{\underset{G}{\Rightarrow}} w\} \tag{3}$$

Σ is the union of three disjoint sets (eq. 4): Σ_k, the keyword set – the vocabulary for data description; Σ_d, used for writing data items; and Σ_i, used for writing intrinsic syntactic elements.

$$\Sigma = \Sigma_k \cup \Sigma_d \cup \Sigma_i \tag{4}$$

Definition 3 (data grammar; type grammar) *Consider port p and two grammars (as in def. 2): $\check{G}(p)$ – for*

writing data (the down-turned mark refers to data grammar entities); and $\hat{G}(p)$ – for writing datatypes (the upturned mark refers to datatype grammar entities).

These grammars must share the keyword set (denoted by $\Xi(p)$ (eq. 5)) and must be such that entities belonging to $L(\check{G}(p))$ describe the datatypes of the entities belonging to $L(\hat{G}(p))$. Each of the former entities works as a third grammar restricting $\hat{G}(p)$: it is used to validate data written according to the lowermost-level grammar.

$$\check{\Sigma}_k(p) = \hat{\Sigma}_k(p) = \Xi(p) \tag{5}$$

Definition 4 (data; datatype; correctness; validity) *Consider port p: $dat(p) \in L(\check{G}(p))$ denotes the data at p. We define port datatype, $typ(p) \in L(\hat{G}(p))$, as a data type specification according to $L(\check{G}(p))$ and $L(\hat{G}(p))$ (the third-level entities mentioned before). The following relation exists between a datastream and its associated datatype:*

$$typ(p) \rightsquigarrow dat(p) \tag{6}$$

Data is correct if it belongs to the language generated by the associated grammar: $dat(p) \in L(\check{G}(p))$, by definition; but it may happen that $dat(q) \notin L(\check{G}(p))$ (for some other port q) – in this case, $dat(q)$ would be incorrect according to $\check{G}(p)$.

Data is valid if $typ(p) \rightsquigarrow dat(p)$, i.e., the data stream follows the datatype definition (besides respecting the underlying grammar's rules).

The complete discussion of $typ(p)$ would only be complete taking into account the semantics of $L(\hat{G})$, but that is out of the scope of this document.

Taking into account the definitions in this section, we now give an example. Consider port p and a data representation containing the following XML (W3C, 2001a) fragment:

$dat(p) =$

```
[...]
<class name="nc">dog</class>
[...]
```

Then the terminal symbol sets would be (at least):

$$\check{\Sigma}_i(p) = \{<, >, =, /, "\}$$
$$\check{\Sigma}_k(p) = \Xi(p) = \{\texttt{class}, \texttt{name}\}$$
$$\check{\Sigma}_d(p) = \{w \mid w \notin \check{\Sigma}_i(p) \cup \Xi(p)\}$$

Consider a datatype description, for the data representation above, of which the following XML Document Type Definition (DTD) fragment is a part:

$typ(p) =$

```
[...]
<!ELEMENT class (#PCDATA)>
<!ATTLIST class name CDATA #REQUIRED>
[...]
```

Then the terminal symbol sets would be (at least):

$$\hat{\Sigma}_i(p) = \{<, >, !, \#, \texttt{ELEMENT}, \texttt{PCDATA}, \texttt{ATTLIST}, \texttt{CDATA}, \texttt{REQUIRED}\}$$
$$\hat{\Sigma}_k(p) = \Xi(p) = \{\texttt{class}, \texttt{name}\}$$
$$\hat{\Sigma}_d(p) = \{w \mid w \notin \hat{\Sigma}_i(p) \cup \Xi(p)\}$$

Thus verifying the grammar pair selection conditions (def. 3 and eq. 5).

2.3. Semantic aspects

This section deals with semantic aspects and restrictions that have to be observed when handling connections.

Each module has sole control over its internal semantics, in particular, in what concerns data semantics (defined by the receiver).

Definition 5 (semantics) *Consider port p and some interpretation function I (defined by the module's inner semantics): $sem(p)$ denotes the semantics required at p for normal processing behavior; $sem(dat(p))$ represents the data stream's semantics at p: computed by I (eq. 7). The data stream's semantics must subsume the port's semantics (eq. 8).*

$$sem(dat(p)) = I(dat(p)) \tag{7}$$
$$sem(p) \sqsubseteq sem(dat(p)) \tag{8}$$

Although we have no way of knowing how a module will interpret a piece of data, we can still write the following relations if we consider $\mathbb{D}$, the function denoting its argument's domain, and $\equiv$ the usual identity operator:

$$[\, typ(p) \rightsquigarrow dat(p) \,] \Leftrightarrow [\, I(typ(p)) \equiv \mathbb{D}(I(dat(p))) \,] \tag{9}$$

and thus (from 7, 9, and $\mathbb{D}$'s definition):

$$sem(dat(p)) \in I(typ(p)) \tag{10}$$

Definition 6 (connection) *Consider modules m and n and ports $p_i^m \in \mathbb{O}^m$ and $q_j^n \in \mathbb{I}^n$. Let predicate $con(p_i^m, q_j^n)$ be true if a connection exists between the pair. In the semantics domain, the output port's semantics must subsume the input's, i.e., condition 11 must be met.*

$$sem(q_j^n) \sqsubseteq sem(p_i^m) \tag{11}$$

Definition 7 (semantics mapping function) *When establishing a connection between two ports, p_i^m and q_j^n, if $typ(p_i^m) \neq typ(q_j^n)$, we need a semantics mapping function, $\theta_{i,j}^{m,n}$, for translating semantics across the connection (eq. 12, but also eq. 13). Furthermore, for the ports to be connectable, the receiving port's semantics must be subsumed by a transformation of the semantics of the previous module's output (cond. 14).*

$$\theta_{i,j}^{m,n} : typ(p_i^m) \rightarrow typ(q_j^n) \tag{12}$$
$$\theta_{i,j}^{m,n} : L(\check{G}(p_i^m)) \rightarrow L(\check{G}(q_j^n)) \tag{13}$$
$$sem(dat(q_j^n)) \sqsubseteq sem(\theta_{i,j}^{m,n}(dat(p_i^m))) \tag{14}$$

It is impossible, however, to guarantee a correct translation in the semantics domain, since, ultimately, input semantics is defined by the data consumer: we approach semantics conversion through datatype-directed data conversion. Since this conversion uses outside information about the ontologies of both sender and receiver, $\theta_{i,j}^{m,n}$ cannot be automatically generated solely from the information available at each end. Nevertheless, $\theta_{i,j}^{m,n}$ can be defined extensionally for each $typ(p_i^m)$.

We assume that it is always the receiver's responsibility to convert the data, since the data producer may be unable to determine how its results will be used. In the current discussion, we will also assume that condition 14 always holds, either because $\theta_{i,j}^{m,n}$ can satisfy it or, if that is not the case, because missing data parts can be supplemented by defaults when computing $sem(q_j^n)$.

2.4. Specifying the application

The model above gives rise to a data-oriented module interconnection architecture in which modules send/receive information to/from each other through typed channels that are uniquely defined by the datatypes at each end-point and by the corresponding translation function.

Since the architecture is not concerned with the modules' inner semantics, all that is needed to describe it completely are the collections of port datatypes and translation functions associated with connected ports. These collections are represented, respectively, by T, the datatype matrix, and by Θ, the translation matrix.

The datatype matrix is defined for all modules and their ports. Entries that do not correspond to actual ports are empty.

$$\underset{m_i \in \mathbb{M}}{T} = \begin{bmatrix} typ(p_1^{m_1}) & \cdots & typ(p_1^{m_\mathcal{M}}) \\ \vdots & & \vdots \\ typ(p_\mathcal{P}^{m_1}) & \cdots & typ(p_\mathcal{P}^{m_\mathcal{M}}) \end{bmatrix} \tag{15}$$

$$\begin{gathered} \mathcal{M} \equiv \#\mathbb{M} \qquad \mathcal{P} \equiv \max_{m \in \mathbb{M}}(\#\mathbb{P}^m) \\ \forall_{m \in \mathbb{M}} \forall_{1 \le v \le \mathcal{P}},\ p_v^m \notin \mathbb{P}^m \Rightarrow typ(p_v^m) = \varnothing \end{gathered} \tag{16}$$

The translation matrix is defined for all connected ports: one function for each connection. In all other cases, Θ is undefined.

$$\Theta = \begin{cases} \theta_{i,j}^{m,n} & con(p_i^m, q_j^n) \quad \text{(see def. 6)} \\ \textit{undefined} & \text{otherwise} \end{cases} \tag{17}$$

3. A small example

This example simplifies the model in important ways: all data flowing between ports is represented in XML and all datatypes can be specified either using DTDs or XML Schemas (XSD) (W3C, 2001d). Thus, in principle, all mismatches are due to variations in the XML data type definitions.

$$\forall_{m \in \mathbb{M}} \forall_{p,q \in \mathbb{P}^m}, \check{G}(p) \equiv \check{G}(q), \hat{G}(p) \equiv \hat{G}(q) \tag{18}$$

unless (only keywords are different)

$$\forall_{m \in \mathbb{M}} \forall_{p,q \in \mathbb{P}^m}, typ(p) \neq typ(q) \Rightarrow \Xi(p) \neq \Xi(q) \tag{19}$$

In our example, all datatypes have been described using DTDs and all necessary $\theta_{i,j}^{m,n}$ functions have been specified by Extensible Style Sheet (XSL) (W3C, 2001b) templates. By specifying all DTDs and XSL templates, the application becomes completely defined from the point of view of its data exchange paths.

The rest of this section will particularize further each of these aspects.

3.1. The application

The example application performs syntactic analysis of natural language sentences (fig. 1).

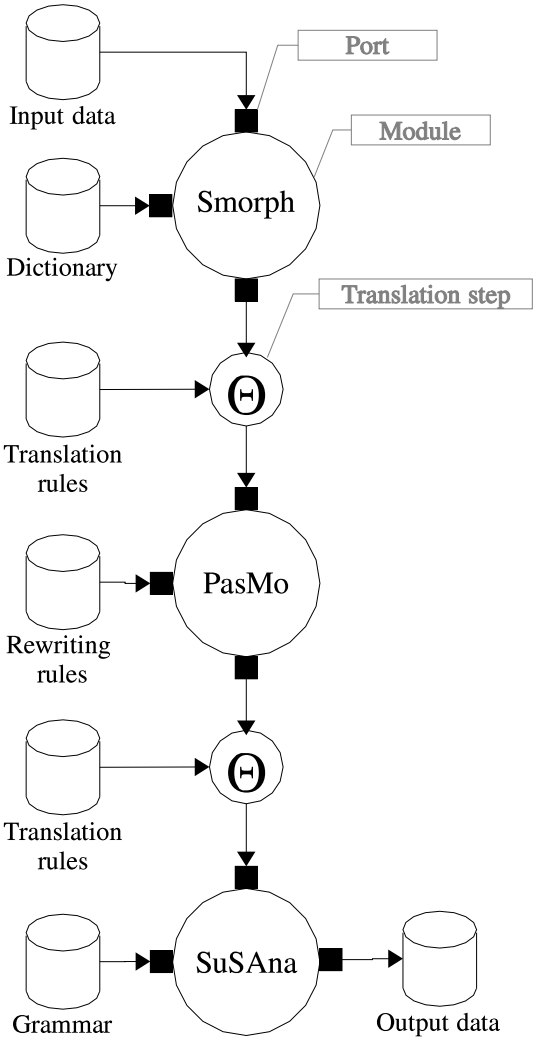

Figure 1: The example application.

The application consists of three modules: Smorph (Aït-Mokhtar, 1998) (morphological analyzer); PAsMo (Paulo, 2001) (rule-based rewriter); and SuSAna (Hagège, 2000; Batista, nd) (syntactic analyzer).

We consider only ports dealing with the data stream to be processed, thus ignoring those used for reading static data (such as dictionaries). Furthermore, in the following we will focus on the connection between Smorph and PAsMo, since the other relevant connection (that between PAsMo and SuSAna) is analogous.

3.2. The application ports

The relevant ports are Smorph's output (s) and PAsMo's input (p). To describe the data flowing through them, we need to specify just $typ(s)$ and $typ(p)$ (eq. 15 and figures 2 and 3). Smorph's output will be translated before being

```
<?xml version="1.0" encoding="iso-8859-15"?>
<!ELEMENT pasmo-in (word)*>
<!ELEMENT word     (class)*>
<!ATTLIST word  text CDATA #REQUIRED>
<!ELEMENT class    (flag)*>
<!ATTLIST class root CDATA #REQUIRED>
<!ELEMENT flag     EMPTY>
<!ATTLIST flag  name  NMTOKEN #REQUIRED>
<!ATTLIST flag  value CDATA   #REQUIRED>
```

Figure 2: DTD for PAsMo's input port, corresponding to $typ(p)$.

used by PAsMo. Note that Smorph's is a more expressive description (thus obeying condition 11), and that some information will be lost in the conversion (not a problem as long as condition 14 remains true).

```
<?xml version='1.0' encoding='iso-8859-1' ?>
<!ELEMENT smorph (item)*>
<!ELEMENT item (root)*>
<!ATTLIST item value CDATA #REQUIRED>
<!ELEMENT root (class)*>
<!ATTLIST root value CDATA #REQUIRED>
<!ELEMENT class (flags,flags)>
<!ATTLIST class type (0|mi) "0">
<!ELEMENT flags (flag)*>
<!ATTLIST flags level (1|2) #REQUIRED>
<!ELEMENT flag EMPTY>
<!ATTLIST flag name  NMTOKEN #REQUIRED>
<!ATTLIST flag value CDATA   #REQUIRED>
```

Figure 3: DTD for Smorph's output port, corresponding to $typ(s)$.

3.3. The translation step

The only relevant transformation, $\theta^{s,p}$, is the one mapping Smorph's output to PAsMo's input. It is implemented as a XSL transformation step and is completely specified by the set of XSL templates (figure 4) that map between data described according to Smorph's DTD and PAsMo's.

4. Related work

This work is related with several fields. The first is the field of data modeling, especially in what concerns very high-level modeling, such as the one done using UML (OMG, ndb). Specifications done in UML can be described using the XML Metadata Interchange (XMI) (OMG, 2002) specification that can then be used to specify the XSDs for the data being sent/received on a module's ports. This is useful because it allows us to describe graphically each module and its interconnections and, by extension, an entire application.

Since we plan on evolving in the direction of service specification(see sec. 5.), we have considered work in this area. One such is IBM's Web Services Flow Language (Leymann, 2001) which can be used for specifying

```
<?xml version='1.0' encoding='iso-8859-1' ?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
<xsl:output method="xml"
  encoding="iso-8859-15"
  doctype-system="pasmo-in.dtd"/>
<xsl:template match="/smorph">
  <pasmo-in>
    <xsl:apply-templates/>
  </pasmo-in>
</xsl:template>
<xsl:template match="item">
  <word text="{@value}">
    <xsl:apply-templates/>
  </word>
</xsl:template>
<xsl:template match="root">
  <class root="{@value}">
    <xsl:apply-templates
      select="class/flags/flag"/>
  </class>
</xsl:template>
<xsl:template match="flag">
  <flag name="{@name}" value="{@value}"/>
</xsl:template>
</xsl:stylesheet>
```

Figure 4: The translation specification in XSL, corresponding to $\theta^{s,p}$.

multiple aspects of web services. This language is also layered on top of others: Web Services Description Language (WSDL) (W3C, 2001c) and Web Services Endpoint Language (WSEL) (Leymann, 2001). Although this structure closely parallels what we intend in our work, it has a different focus and does not invalidate our proposal.

The third area is that of communication systems, which typically define module interconnection architectures. An example is CORBA. Another, of particular interest for NLP, is the Galaxy Communicator (MIT, 2001; DAR, nd). This architecture is a distributed, message-based, hub-and-spoke infrastructure optimized for constructing spoken dialogue systems. It uses a plug-and-play approach that enables the combination of commercial and research components. It supports the creation of speech-enabled interfaces that scale gracefully across modalities. In this context, our proposal enables easy specification of Galaxy applications. At a different level, our specifications can be gracefully translated into hub scripts and server interface definitions.

In the context of reference architectures, such as the ones proposed by the TIPSTER (TIP, nd) or RAGS (ITRI, nd) projects, our model may prove useful in facilitating integration of external modules into the frameworks defined by those architectures. Note that, unlike most software infrastructures for Language Enginnering research and development, e.g. GATE (Cunningham et al., 1996), our model does not say anything about any module's function or impose any restrictions on their interfaces and is, thus, application- and domain-independent. This is so because the model is exclusively concerned with the data streams flowing between modules and the relations between their semantics at each end and not with the way each stream is used, i.e., the model is not directly concerned with application-related issues. In this sense, the model could be used to describe a kind of "smart glue" for use with other

architectures, e.g. in integration efforts of existing modules into GATE's CREOLE sets, or in datatype management.

Other application-development or intercommunication infrastructures may benefit from using a high-level specification such as the one we propose here.

5. Conclusions and future directions

Our approach is useful for application development, since it focuses exclusively on the inputs and outputs of each module, without regard for module internals. This contributes to significant dependency reductions, for the modules can be almost anything and run almost anywhere, as long as a communications channel (according to our restrictions) can be established between them.

We envision various directions for future work.

The first is to provide higher-level service specifications on top of port descriptions. This would allow services to be defined using the descriptions of its inputs and outputs and, rather than exhaustively describing each port and its data, we would be able, at that higher abstraction level, to simply specify the name of the service. The rest would follow from lower-level descriptions.

Also, along the lines of higher-level abstractions and services, it would be interesting to try and specify automatic translation functions ($\theta_{i,j}^{m,n}$) based on service semantics. Of course, this would mean that semantics would have to be specified in some way as well.

Both these approaches would help to integrate user-developed modules and help integrators to develop transformation steps that cannot be wholly automatically generated.

Another direction worth considering is the construction of module and application servers: modules or pre-built applications would be presented, e.g. via a web browser, enabling users to specify custom applications.

6. References

S. Aït-Mokhtar. 1998. *L'analyse présyntaxique en une seule étape*. Thèse de doctorat, Université Blaise Pascal, GRIL, Clermont-Ferrand.

Fernando Batista. n.d. Análise sintáctica de superfície e coerência de regras. Master's thesis, Instituto Superior Técnico, UTL, Lisboa.

Hamish Cunningham, Yorick Wilks, and Robert J. Gaizauskas. 1996. Gate – a general architecture for text engineering. In *Proceedings of the 16th Conference on Computational Linguistics (COLING96)*, Copenhagen.

DARPA, n.d. *DARPA Communicator*. See: www.darpa.mil/ito/research/com/index.html.

Caroline Hagège. 2000. *Analyse syntaxique automatique du portugais*. Thèse de doctorat, Université Blaise Pascal, GRIL, Clermont-Ferrand.

ITRI, nd. *RAGS – A Reference Architecture for Generation Systems*. Information Technology Research Institute, University of Brighton. See: http://www.itri.brighton.ac.uk/projects/rags/.

Harry R. Lewis and Christos H. Papadimitriou. 1981. *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0-13-273426-5.

Frank Leymann, 2001. *Web Services Flow Language (WSFL 1.0)*. IBM Software Group, May. See also: xml.coverpages.org/wsfl.html.

MITRE Corporation, 2001. *DARPA Communicator*, May. See: http://fofoca.mitre.org/.

Object Management Group (OMG), 2002. *XML Metadata Interchange (XMI) Specification, v1.2*, January. See: www.omg.org/technology/documents/formal/xmi.htm.

Object Management Group (OMG), n.d.a. *Common Object Request Broker Architecture (CORBA)*. See: www.corba.org.

Object Management Group (OMG), n.d.b. *Unified Modelling Language*. See: www.uml.org.

Joana Lúcio Paulo. 2001. Aquisição automática de termos. Master's thesis, Instituto Superior Técnico, UTL, Lisboa.

NIST, nd. *TIPSTER Text Program*. See: http://www.itl.nist.gov/iaui/894.02/related_projects/tipster/.

World Wide Web Consortium (W3C), 2001a. *Extensible Markup Language*. See: www.w3c.org/XML.

World Wide Web Consortium (W3C), 2001b. *The Extensible Stylesheet Language*. See: www.w3.org/Style/XSL.

World Wide Web Consortium (W3C), 2001c. *Web Services Description Language (WSDL) 1.1*, March. See: www.w3.org/TR/wsdl.

World Wide Web Consortium (W3C), 2001d. *XML Schema*. See: www.w3c.org/XML/Schema and www.oasis-open.org/cover/schemas.html.