Empowering the User: a Data-oriented Application-building Framework

David M. de Matos, Alexandre Mateus, João Graça, 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, amfm@mega.ist.utl.pt,
javg@mega.ist.utl.pt, nuno.mamede@inesc-id.pt

Abstract. In order to minimize the effort of building modular applications, and also as a guideline for designing them from scratch, we propose a model that allows rapid application building and reconfiguration with minimal manual intervention, potentiating module reuse and improving application understanding. As a demonstration of the model's capabilities, we present a user interface which allows users to quickly and easily build modular applications.

1 Introduction

Although the use of third party modules to build modular applications is desirable – because it promotes reuse, thus 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.

We present a model for describing modular architectures that can be understood from the perspective of their data exchanges. It helps in application construction and module reuse and limits understanding requirements. The model provides a data interface specification for modules and defines a uniform interoperability model: the latter is a consequence of the former.

To illustrate the model's capabilities, we present a web-based user interface for building modular applications. The interface has proved to be quite useful in allowing non-specialists to build complex applications: the only requirement is understanding of the meaning of the data each module uses/produces – a requirement much less stringent than understanding the modules themselves.

This document is organized as follows: §2 presents the data model. The interface is then presented in light of the model: §3.1 introduces design issues and §3.2 discusses usage. A few remarks about related systems and architectures are presented as well as possible directions for evolution.

2 Model

We present the model in three parts: structural aspects (def. 1); the data model (defs. 2, 3, and 4); and data semantics (defs. 5 and 6). §2.1 discusses content restrictions and §2.2 defines a formal application specification.

Definition 1 (ports; portsets). *Applications consist of independent modules that exchange data through connections between input and output ports.* $\mathbb{M}$ *is the set of all modules in an application. The following sets are defined for module* m*:* $\mathbb{O}^m$ *(all output ports);* $\mathbb{I}^m$ *(all input ports); and* $\mathbb{P}^m = \mathbb{I}^m \cup \mathbb{O}^m$ *(all ports). In addition,* $\mathbb{I}^m \cap \mathbb{O}^m = \varnothing$. p_i^m *denotes the* i*-th port of* m.

Definition 2 (grammar). *An unrestricted grammar [3] is a quadruple* $G = (V, \Sigma, R, S)$*, 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* $(V^*(V-\Sigma)V^*) \times V^*$*). For our purposes,* $\Sigma = \Sigma_k \cup \Sigma_d \cup \Sigma_i$*:* 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. Direct derivation (eq. 1), derivation (eq. 2), and generated language (eq. 3) are defined as follows:

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

$$w_0 \underset{G}{\Rightarrow} w_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} w_n \Leftrightarrow w_0 \overset{*}{\underset{G}{\Rightarrow}} w_n \quad (2)$$

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

Definition 3 (data grammar; type grammar). *Consider port p and two grammars: $\check{G}(p)$ – for data (the down-turned mark refers to data grammar entities); and $\hat{G}(p)$ – for datatypes (the up-turned mark refers to datatype grammar entities).*

$\check{G}(p)$ and $\hat{G}(p)$ are such that $\check{\Sigma}_k(p) = \hat{\Sigma}_k(p)$. Moreover, entities from $L(\hat{G}(p))$ describe the datatypes of the entities from $L(\check{G}(p))$. Each of the former works as a third grammar further restricting $\check{G}(p)$: each validates data written according to the lowermost-level grammar.

Definition 4 (data; datatype; correctness; validity). *Consider port p: then $dat(p) \in L(\check{G}(p))$ denotes the data flowing through p; and $typ(p) \in L(\hat{G}(p))$ is a datatype specification according to $L(\check{G}(p))$ and $L(\hat{G}(p))$. $\mathcal{T} \rightsquigarrow \mathcal{D}$ holds between a datastream ($\mathcal{D}$) and its associated datatype ($\mathcal{T}$). Data is correct if $dat(p) \in L(\check{G}(p))$. Data is valid if $typ(p) \rightsquigarrow dat(p)$.*

Definition 5 (semantics). *Consider port p and an interpretation function **I** (defined by the module's inner semantics): $sem(p)$ denotes the semantics required at p for normal processing behavior; $sem(dat(p))$ is the datastream's semantics at p, as computed by **I**: $sem(dat(p)) = \mathbf{I}(dat(p))$. The datastream's semantics must subsume the port's semantics: $sem(p) \sqsubseteq sem(dat(p))$.*

Definition 6 (connection). *Consider modules m and n. Predicate $con(p_i^m, q_j^n)$ is true if a connection exists between $p_i^m \in \mathbb{O}^m$ and $q_j^n \in \mathbb{I}^n$. Condition 4 must hold:*

$$sem(q_j^n) \sqsubseteq sem(p_i^m) \quad (4)$$

Definition 7 (semantics mapping function). *A semantics mapping function, $\theta_{i,j}^{m,n}$ (eq. 5), may be needed when connecting $p_i^m \in \mathbb{O}^m$ and $q_j^n \in \mathbb{I}^n$, if $typ(p_i^m) \neq typ(q_j^n)$. Furthermore, for successful communication, condition 6 must hold.*

$$\theta_{i,j}^{m,n} : typ(p_i^m) \rightarrow typ(q_j^n) \qquad \text{but also} \qquad \theta_{i,j}^{m,n} : L(\check{G}(p_i^m)) \rightarrow L(\check{G}(q_j^n)) \quad (5)$$

$$sem(dat(q_j^n)) \sqsubseteq sem(\theta_{i,j}^{m,n}(dat(p_i^m))) \quad (6)$$

Since semantics is defined by the consumer's internals (outside the model), it is impossible to guarantee a correct semantic translation. Semantics conversion resorts to datatype-directed data conversion: this conversion uses outside information about the ontologies of both sender and receiver, implying that $\theta_{i,j}^{m,n}$ cannot be automatically generated within the model.

2.1 Maintaining information content levels

Before, we assumed that conditions 4 and 6 always held. Now, instead of assuming condition 6, we study the case of information deficits when interconnecting two modules and show how to integrate these cases into the previous model.

The first part of fig. 1 presents the initial scenario: modules A and C, connected via $p_u^A \in \mathbb{O}^A$ and $r_x^C \in \mathbb{I}^C$. We assume that p_u^A and r_x^C obey conditions 4 and 6.

The second part presents the insertion (1) of a module B, between A and C. We assume that conditions 4 and 6 are verified, i.e., between A and B, condition 7 holds.

$$q_v^B \in \mathbb{I}^B, p_u^A \in \mathbb{O}^A : \ sem(q_v^B) \sqsubseteq sem(p_u^A) \wedge sem(dat(q_v^B)) \sqsubseteq sem(\theta_{u,v}^{A,B}(dat(p_u^A))) \quad (7)$$

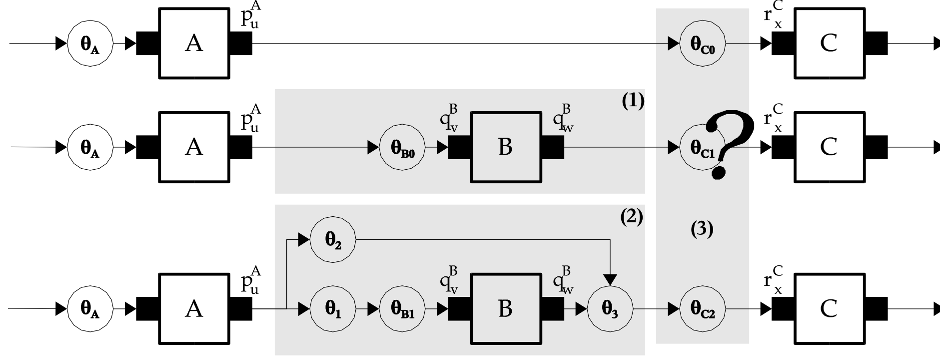

Fig. 1. Insertion of a module (B) that processes only a fragment of the input information.

If B's output causes information deficits, then $sem(r_x^C) \not\sqsubseteq sem(q_w^B)$: C now has less information than before and possibly below the level needed for correct behavior. Also, the lower information content makes it impossible to find a $\theta_{w,x}^{B,C}$ (θ_{C1} in the figure) satisfying condition 6:

$$sem(r_x^C) \not\sqsubseteq sem(q_w^B) \Rightarrow \forall_{\theta_{w,x}^{B,C}} sem(dat(r_x^C)) \not\sqsubseteq sem(\theta_{w,x}^{B,C}(dat(q_w^B))) \tag{8}$$

The scenario of eq. 8 may be circumvented if two aspects are considered: condition 7 holds (for the A-B link, in the third part of fig. 1: $\theta_{u,v}^{A,B} = \theta_{B1} \circ \theta_1$); and, information that does not now reach C, may be redirected without harming or being harmed by B. This is the scenario shown in (2), in the third part of fig. 1. The conditions for ensuring the correct behavior of this capsule depend on the functions used to perform data mappings.

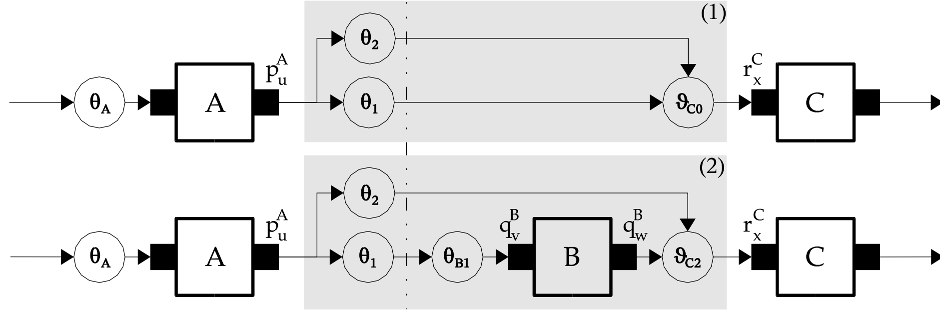

Fig. 2. Diagram of connection equivalence (first and third parts of fig. 1).

In fig. 2, the first and third parts of fig. 1 present parallel decompositions: functions θ_1 and θ_2 are as before; $\vartheta_{C0} = \theta_{C0} \circ \theta'_3$ (θ_{C0} is as in fig. 1 and θ'_3 – not shown – is a recombination for θ_1 and θ_2, in the absence of other transformations[1]); $\vartheta_{C2} = \theta_{C2} \circ \theta_3$ (θ_3 is as in fig. 1).

In both cases, θ_2 maps data from A to C without further processing. Thus, the constraints on B's output, and on ϑ_{C0} and ϑ_{C2}, stem from the other path: for the first case, we have condition 9, in which the second recombination input comes from A through θ_1 (we assumed that θ_1 and θ_2 were suitable for enabling a connection between A and C); for the second case, condition 10 must hold (in both cases, $\xi_2 = \theta_2(dat(p_u^A))$).

$$sem(dat(r_x^C)) \sqsubseteq sem(\vartheta_{C0}(\xi_2, \theta_1(dat(p_u^A)))) \tag{9}$$

$$sem(dat(r_x^C)) \sqsubseteq sem(\vartheta_{C2}(\xi_2, dat(q_w^B))) \tag{10}$$

[1] That is, for any suitable x: $\theta'_3(\theta_1(x), \theta_2(x)) = x$.

Since ϑ_{C0} and ϑ_{C2} are similar – they must differ only in the form and number of elements to recombine –, the remaining constraint must be imposed on $dat(q_w^B)$, i.e., as $\theta_1(dat(p_u^A))$, it must allow correct data interpretation at C.

A module such as B works as a partial filter: it may include additional information of its own on the data it produces, but this new information is irrelevant for later modules, when compared with a scenario in which the module does not exist, i.e., $dat(q_w^B)$ may suffer a maximum degradation, relative to $\theta_1(dat(p_u^A))$, after which condition 10 becomes false and the module useless.

2.2 Specifying the application

The model underlies a data-oriented module interconnection architecture in which modules exchange information with each other through channels that are described by the datatypes at each port and by the corresponding translation function. Collections of ports and functions completely describe the architecture and are represented by T, the datatype matrix (eq. 11), defined for all modules and their ports (entries that do not correspond to actual ports are empty); and by Θ, the translation matrix (eq. 12), defined for each connection (in all other cases, Θ is undefined).

$$\underset{m_i \in \mathbb{M}}{T} = \begin{bmatrix} typ(p_1^{m_1}) \cdots typ(p_1^{m_\mathcal{M}}) \\ \vdots \qquad\qquad \vdots \\ typ(p_\mathcal{P}^{m_1}) \cdots typ(p_\mathcal{P}^{m_\mathcal{M}}) \end{bmatrix} \qquad \begin{matrix} \mathcal{M} \equiv \#\mathbb{M} \qquad \mathcal{P} \equiv \underset{m \in \mathbb{M}}{max}(\#\mathbb{P}^m) \\ \forall_{m \in \mathbb{M}} \forall_{1 \leq v \leq \mathcal{P}},\ p_v^m \notin \mathbb{P}^m \Rightarrow typ(p_v^m) = \varnothing \end{matrix} \tag{11}$$

$$\Theta = \begin{cases} \theta_{i,j}^{m,n} & con(p_i^m, q_j^n) \qquad \text{(see def. 6)} \\ \textit{undefined} & \text{otherwise} \end{cases} \tag{12}$$

The existence of modules that cause information losses would cause changes to T, if we did not consider them as partial filters adapted to the modules that provide their input data. As such, they can be integrated in the model as part of virtual modules (as discussed in §2.1) that are, in what concerns T and Θ like any other normal module.

3 Web interface for Galaxy-based systems

The web interface, based on the model above, simplifies access to applications and libraries available from the Spoken Language Systems Lab: any user with a web browser is able to easily use and compose modules. In addition, the user can create, store, and load service chains (user-side service or program sequences). This allows users to easily test module interactions. The Galaxy Communicator architecture [5] was selected to provide inter-module communication support. Galaxy's is a distributed, message-based, hub-and-spoke infrastructure optimized for constructing spoken dialogue systems. Part (1) of fig. 3 shows how we integrated the system in our example.

3.1 The interface's components

There are two ways of porting a module into Galaxy: the first is to create the module anew, or alter its code so that it can be incorporated into the system; the second is to create a capsule for the existing module. This capsule then behaves as a Galaxy server.

Favoring the second option proved a wise choice, since almost no changes to existing modules were required. In truth, a few changes, mainly regarding input/output methods, had to be made, but these are much simpler than rebuilding a module from scratch.

Some changes were caused by the requirement that each module accept/produce XML [10] data in order to simplify the task of writing translations. Also, the use of XML as intermediate data representation language acts as a normalization measure and makes it easier for future users to understand a module's inputs and outputs, even if the same cannot be said about its internals.

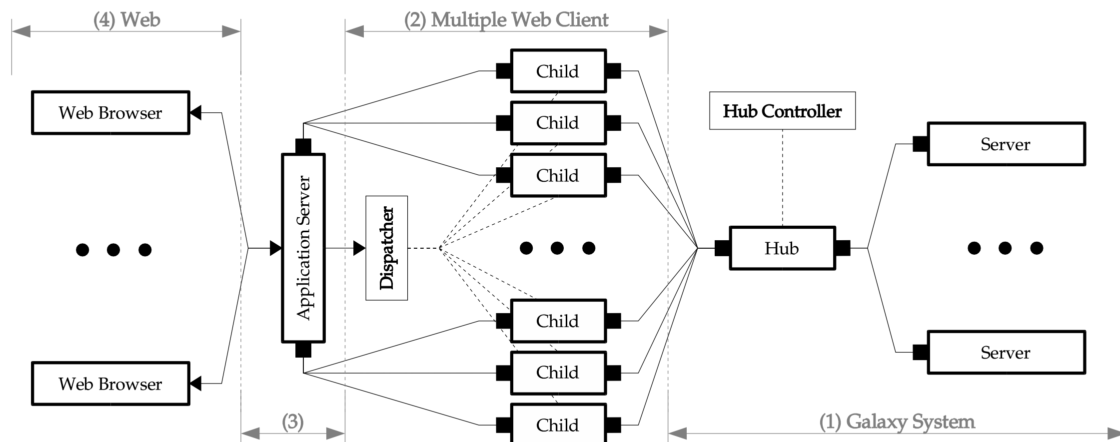

Fig. 3. The web interface's layered architecture.

MultipleWebClient

The MultipleWebClient is a gateway to the underlying Galaxy system, allowing multiplexed communication with the external clients (see part (2) of fig. 3): a dispatcher receives requests from the web interface and spawns children to handle them.

This layer exists to enable the system to serve more than one request at a time. Due to Galaxy design options, a dedicated connection would be needed and it must adhere to the system's event handling methodology, something that would not be to our advantage. The problem was solved by resorting to a few undocumented Galaxy features (support was kindly provided by the Galaxy team). The required functionality may become available in future Galaxy versions.

The application server

The application server is one of the interface's key components. It provides an environment that manages the execution of the various processes in the web application, maintains relevant data for each application user, guarantees security levels and manages access control, among other functionality. In our case, the application server is used as a bridge between the interface presentation layer (HTML, at the browser level) and the lower system levels (Galaxy system).

Web interface and service chains

The web interface proper consists of a set of Java classes, servlets, and server pages (JSPs). It is built on information about the location (host and port) of the MultipleWebClient that provides the bridge with the Galaxy system the user wants to contact; and on XML descriptions of the underlying Galaxy system (provided by the hub controller – see below).

Besides allowing the execution of server operations on user-specified data, the interface allows user-controlled building of service chains. These are service sequences provided by the Galaxy system servers: each service is invoked according to the user's sequence. The motivation for service chains arose from the need for a simple tool for allowing users to test sequences of module interactions without having to actually freeze those sequences or build an application. Hence, the interface allows not only inspection of the end results of a service chain, but also its intermediate results. Service chains may be stored locally, as XML documents, and loaded at will by the user.

The hub controller

The interface uses a special Galaxy server (see part (1) of fig. 3): the hub controller. This is an internal server that ensures correct system behavior: the first request from the interface is for a description of the system itself, allowing the interface to present the user with a list of servers and programs. The hub controller maintains and sends this description to the upper levels.

3.2 Using the web interface

The first step when accessing the web interface is to specify the location – host and port – of the back-end Galaxy system. Then, the interface presents the main page (fig. 4), divided into four areas: the top menu provides general commands, such as service chain manipulation (creation, load/save operations), help, and so on; this frame is always available. The left frame presents a system description and allows access to servers and programs at any time. Each of them in turn has additional subdivisions: each broadly corresponds to a module in the theoretical model. To the far right a column presents active service chains. The main frame (also the main input area) presents the current state of the system, or of its interaction with the user.

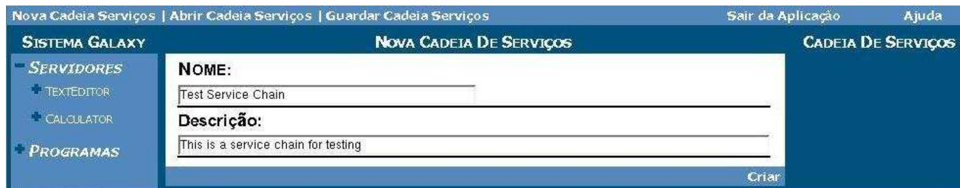

Fig. 4. Web interface: main screen, showing service chain creation.

Whenever a service is selected from the left frame, a description is presented to the user, stating the input and output ports, as well as a description of the service's actions. Also, service selection causes a list of possible operations on that module to be presented at the bottom of the central frame. In the example shown in fig. 5, it is possible to either remove the service from the current service chain or to update its data.

The right hand frame presents the currently active service chain, if any: the chain's name appears at the top followed by a collapsible free text description. The rest of the frame contains the list of modules in the service chain.

Each service in a chain can be in one of two states: complete, i.e., all input and output connections have been specified (these connections may be inline data, file data or a connection to another module); or incomplete (the default). The number to the left of the service's name also indicates this state: a light box indicates a completed service whereas a dark one indicates that at least one of the service's ports remains unconnected. Figure 5 shows the various aspects of service chain creation: the inset shows a service before completion. Collapsible boxes indicate the state of each service: in fig. 5, service 3 is shown to be executable (in Portuguese, "Serviço executável").

A chain becomes executable when all of its services are complete: a new box appears allowing execution of the service chain's actions. In fig. 5, "EXECUTAR" (Portuguese for "execute") appears as soon as the last service is completed. After execution of a service chain, the resulting data (both partial and final) may be viewed in the web browser or saved to a file.

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 [9]. Specifications done in

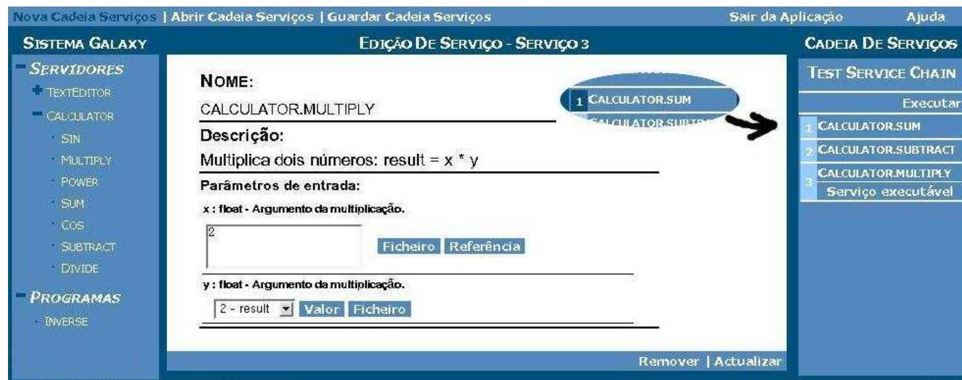

Fig. 5. Web interface: service chain ready to be executed. Inset: service 1 before completion, showing dark box.

UML can be described using the XML Metadata Interchange [7] specification that can then be used to specify the schemata for the data being sent/received through 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, we have considered work such as IBM's Web Services Flow Language [4] which can be used for specifying multiple aspects of web services. This language is also layered on top of others: Web Services Description Language [11] and Web Services Endpoint Language (WSEL) [4]. 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 [8]. Another, of particular interest for natural language applications, is the Galaxy system. In this context, as shown above, our proposal enables easy specification of Galaxy-based applications.

In the context of other architectures, such as the ones proposed in the fields of natural language processing and generation (of special interest to us), by the TIPSTER [6] or RAGS [2] 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 engineering research and development, e.g. GATE [1], 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 flowing between modules and their semantics relations and not with the way each stream is used, i.e., the model is not directly concerned with application-related issues. Thus, the model is useful to describe integration methods for use with other architectures and in datatype management.

Although some of the above architectures also provide user interfaces for building applications, we think our approach is still useful: we do not impose any special functionality to any module. Instead we provide a way of connecting any module that can be connected and so can be useful even in other contexts.

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 suitable communications channel can be established between them.

The model allows us to provide high-level service specifications on top of port descriptions. This allows services to be defined using the descriptions of their inputs and outputs and thus, rather

than exhaustively describing each port and its data, we are able, at that higher abstraction level, to simply specify the name of the service. The rest follows from the lower-level descriptions. The interface presented above is an example of a way of taking advantage of service specifications and, by implication, of the theoretical model.

The interface demonstrates that the model is useful for describing modular applications and for helping in bringing users to do so, since the level of understanding required for the process is greatly reduced. Furthermore, the interface, by hiding most of the complexities of the underlying system and of the modules attached to it, empowers non-expert users, such as students, to play with various scenarios to investigate possible differences, and more serious users, such as researchers or developers, to test and study various options in application and system construction, without going into irrelevant details, i.e., outside the application domain. Thus, the model is useful not only in helping in application construction, but also as a guide to thinking about modular applications. In this sense, it is most useful for designing applications and application components for non-expert users.

For the future, 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, such automation would mean that semantics would have to be specified in some way as well. Automatic translation generation, possibly partial, would help to integrate user-developed modules and help integrators to develop transformation steps that cannot be wholly automatically generated.

Regarding future developments in the interface: while the one described was aimed only at module users, we envision the development of another interface, this time aimed at helping module developers integrate their work for use in the system. This interface can be developed as an extension to the current one, or as a completely independent one.

A last remark: in this document, we used NLP applications and modules as examples simply because they were of special interest to us. The fact does not imply any specialization or dependencies in that direction (except regarding module production/customization): both the interface and the model are domain independent and can be used when- and wherever the model can be made to fit a particular application domain.

References

1. H. Cunningham, Y. Wilks, and R. J. Gaizauskas. Gate – a general architecture for text engineering. In *Proc. of the 16th Conf. on Computational Linguistics (COLING96)*, Copenhagen, 1996.
2. ITRI. *RAGS – A Reference Architecture for Generation Systems.* ITRI, University of Brighton, nd. See: http://www.itri.brighton.ac.uk/projects/rags/.
3. H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981. ISBN 0132734265.
4. Frank Leymann. *Web Services Flow Language (WSFL 1.0).* IBM Software Group, 2001. See also: xml.coverpages.org/wsfl.html.
5. MITRE Corporation. *DARPA Communicator*, 2001. See: http://fofoca.mitre.org/ and www.darpa.mil/ito/research/com/index.html (DARPA Communicator).
6. NIST. *TIPSTER Text Program*, nd. See: www.itl.nist.gov/iaui/894.02/related_projects/tipster/.
7. OMG. *XML Metadata Interchange (XMI) Specification*, 2002. See: www.omg.org/technology/documents/formal/xmi.htm.
8. OMG. *Common Object Request Broker Architecture (CORBA)*, n.d. See: www.corba.org.
9. OMG. *Unified Modelling Language*, n.d. See: www.uml.org.
10. World Wide Web Consortium (W3C). *Extensible Markup Language*, 2001. See: www.w3c.org/XML.
11. World Wide Web Consortium (W3C). *Web Services Description Language (WSDL) 1.1*, 2001. See: www.w3.org/TR/wsdl.