Managing Linguistic Resources and Tools

David M. de Matos, Joana L. Paulo, and Nuno J. Mamede

L^2F – Spoken Language Systems Laboratory
INESC-ID Lisboa/IST, Rua Alves Redol 9, 1000-029 Lisboa, Portugal
{david.matos,joana.paulo,nuno.mamede}@inesc-id.pt
http://www.l2f.inesc-id.pt/

Abstract. We present Galinha, a system that integrates multiple linguistic resources and tools. Galinha enables easy module integration and testing of prototypical configurations, thereby reducing the effort and backtracking usual in the construction of modular applications. Moreover, it has a soft learning curve, enabling new users and developers to use it successfully.

1 Introduction

Large R&D groups are often presented with the problem of reusing existing resources and tools. These may have been produced in-house or they may be third-party modules. In either case, the task of managing them is not simple: for instance, some tool may be available but may be deemed to hard to reuse for a particular task, causing the redevelopment of a similar tool. This makes application construction more difficult and diverts productive effort to tasks that have already been carried out.

If reuse is a problem, the contact between old tools and new users is also a critical issue. The problem here is often in terms of the time required to acquire the necessary expertise to fully and productively use some resource.

To address the above issues, we present a web-based user interface for building modular applications. The interface has proved to be quite useful in allowing new users and non-specialists to assemble and test complex prototypes: the only requirement is a clear understanding of the meaning of the data used by each module – a requirement much less stringent than understanding the modules themselves.

This document is organized as follows: Sect. 2 briefly presents the underlying support system. The interface itself is presented in Sect. 3; design and usage issues are also covered here. Section 4 discusses the development and deployment of new modules, as well as the integration of existing ones, in our framework. Finally, a few remarks about related systems and architectures are presented (Sect. 5) as well as directions for future work (Sect. 6).

2 Infrastructure

The infrastructure used to support the interface is a partial implementation[1] of the theoretical interconnection model proposed in [4]. The Galaxy Communicator system [7]

[1] Currently, the main capabilities have been implemented, but things like message type checking is still in infancy.

N.J. Mamede et al. (Eds.): PROPOR 2003, LNAI 2721, pp. 135–142, 2003.

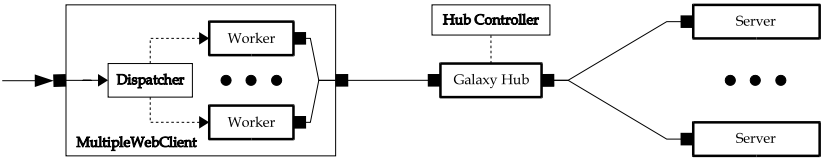

Fig. 1. The support infrastructure: Galaxy system parts and dedicated servers

was selected to provide messaging support for the infrastructure's message exchanges. Galaxy has a distributed hub-and-spoke message-based architecture optimized for constructing spoken dialogue systems. It was selected because it was already being used by us for other purposes and because the new purpose did not in any way affect existing uses. This conjunction of factors allowed us to easily migrate/integrate existing modules to the new framework with minimal repercussions.

Figure 1 shows the interconnection infrastructure along with two custom control servers: the *MultipleWebClient* and the *hub controller*. Figure 2 shows the infrastructure within the interfacing system.

The MultipleWebClient is a gateway that routes interface calls to the underlying system, allowing multiplexed communication with external clients: a dispatcher receives requests from the web interface and spawns workers to handle them. This layer exists to enable the infrastructure to serve more than one request at a time. Due to Galaxy design options, a dedicated connection would otherwise be needed and it would have to adhere to the system's event handling methodology, something that would not be to our advantage. The problem was solved by resorting to undocumented Galaxy features (support was kindly provided by the Galaxy team). The required functionality may become available in future Galaxy versions.

The hub controller server, a meta-information manager, ensures correct system behavior: the first request from the user interface is for a description of the infrastructure itself. The hub controller sends this description to the upper levels, allowing the interface to present a list of servers and programs.

3 The Interface

Galinha (Galaxy Interface Handler) simplifies access to modules, applications, and library interfaces: it enables users to access and compose modules using a web browser.

The application server is one of the interface's key components. It provides the runtime environment for the execution of the various processes within the web application. Moreover, it maintains relevant data for each user, guarantees security levels and manages access control. The interface also uses the application server as a bridge between the interface's presentation layer (HTML [15]/JavaScript [5], at the browser level) and the infrastructure.

The presentation layer consists of a set of Java classes, servlets, and server pages (JSPs). It is built from information about the location (host and port) of the MultipleWebClient that provides the bridge with the Galaxy system the user wants to contact; and

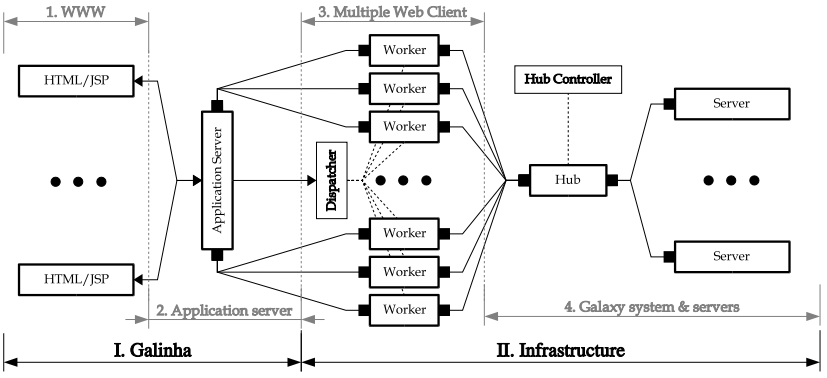

Fig. 2. The Galaxy infrastructure, control servers, and user-side levels

from XML [14] descriptions of the underlying Galaxy system (provided by the hub controller – see Sect. 2).

Besides allowing execution of services on user-specified data, the interface allows users to create, store, and load service chains. Service chains are user-side service- or program sequences provided by the servers connected to the infrastructure: each service is invoked according to the user-specified sequence. Service chains provide a simple way for users to test sequences of module interactions without having to actually freeze those sequences or build an application. The interface allows not only inspection of the end results of a service chain, but also of its intermediate results. Service chains may be stored locally, as XML documents, and may be loaded at any time by the user. Even though, from the infrastructure's point of view, service chains simply do not exist, selected service chains[2] may be frozen into system-side programs and become available for general use.

Using the Web Interface

To use the interface, users must first specify the location – host and port – of the back-end system. Then, the interface presents the main view, divided into four areas (see Fig. 3): the top one provides a general control menu; this frame is always available. The left frame presents the back-end system's description and allows access to servers and programs at any time, each of which in turn has additional subdivisions. The frame to the right presents the service chain currently active, if any. The main frame (also the main input area) presents various states of the system or of its interaction with the user.

When a service is selected, its description is presented to the user, stating the input and output ports, as well as a description of its actions. Service selection also presents the user with the list of possible operations on that module.

On the right hand frame, the active service 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, as well as the state of their interconnections.

[2] Tested and approved by the infrastructure's administration, according to relevant criteria.

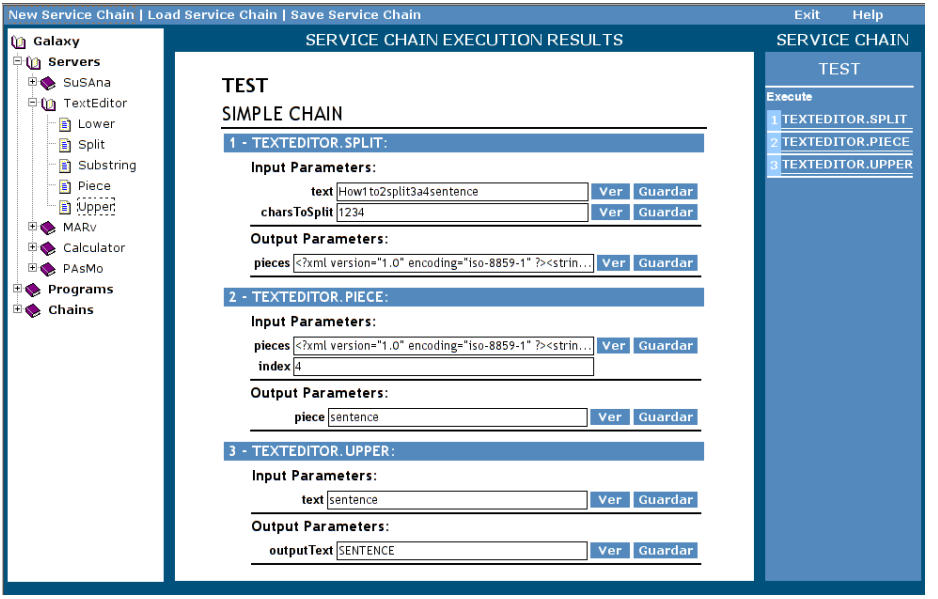

Fig. 3. Execution of a simple service chain. Main window shows all partial results

Each service in a chain can be in one of two states: complete, i.e., all input and output connections have been specified; 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. A chain becomes executable when all of its services are complete. After execution of a service chain, resulting data (both partial and final) may be viewed in the web browser or saved to a file.

4 Module Definition

Modules are included within the infrastructure in two ways: the first is to create the module anew or to adapt it 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 normal Galaxy server would.

Whenever possible or practical, we chose the second path. Favoring the second option proved a wise choice, since almost no changes to existing programs 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.

Mainly, changes were caused by the requirement that some of the modules accept/produce XML data in order to simplify the task of writing translations. This is not a negative aspect, since the use of XML as intermediate data representation language also acts as a normalization measure: it actually makes it easier for future users to understand modules' inputs and outputs.

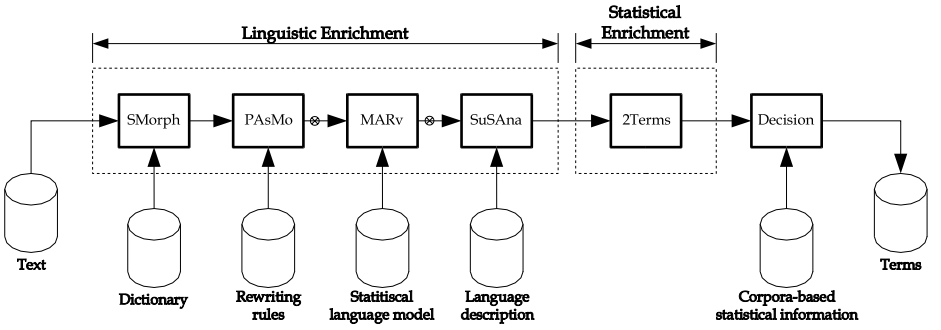

Fig. 4. The ATA system: once defined, the chain implementing linguistic enrichment may be reused by other, possibly unrelated, applications. The symbol ⊗ marks connections performing data format translations

The first modules included in the system were simply for test purposes and were not, in any case, very complex. The first practical test took place when production modules had to be incorporated into the system. Several modules, namely SMorph [1] (morphological analysis), PAsMO [11] and MARv [13] (morphological processing), and SuSAna [2], based on AF [6] (syntactic analysis), were added. Writing general adapter modules (such as data format converters) was also required. In both cases, the work to be done proved to be simple (only data format manipulations were required).

As an example, one of our co-workers, with no previous contact with the system, was able to successfully have new modules incorporated into the infrastructure. In addition to existing servers (the ones mentioned above), new servers were used to build the ATA system [12], an automatic term extractor that uses linguistic and statistic information (Fig. 4). The first of the new modules enriches the text with statistical information about words and noun-phrases. The second decides whether each candidate term is in fact a term (taking into account corpora-based statistical information).

Existing modules for morphological analysis and processing were reused, but, since they accept/produce different data formats, two other modules were added to provide data format conversion through XSLT [16] templates.

All that was needed to integrate an existing XSLT processor into the system was the coding of a wrapper to call the external application. The wrapper was simple enough that we were able to generalize it so that other similar applications could be incorporated in this way into the infrastructure. This line of work simplifies the overall integration process and enables users to add new modules knowing only how they are activated.

Building the ATA system had a beneficial side-effect: the creation of a reusable chain. This chain may now be used for other purposes (in parallel with ATA), by other applications running on top of the same infrastructure. This freedom when making new connections allows users to explore new applications for “old” chains.

5 Related Work

Although our work is not directly related to the field of data modeling, we can take advantage of data and metadata descriptions, such as UML [9] specifications. These specifications can be represented using the XML Metadata Interchange [10] format and, thus, easily processed. Also, they can be used to specify the schemata for the data being sent/received by infrastructure modules. UML, thus, allows for graphical module and module interconnection descriptions and, by extension, the description of complete applications.

The work presented here, using the Galaxy infrastructure, is a simple way of delivering messages from on module to another. Others, such as CORBA-based [8] communication systems, could be used as long as the basic interchange model is respected, much as Galaxy does (it is not the reference implementation).

Note that, unlike most software infrastructures for language engineering research and development, e.g. GATE [3], the model underlying the infrastructure does not say anything about any module's function and does not impose any restrictions on their interfaces. Thus, the entire framework is application- and domain-independent.

6 Conclusions and Future Directions

The interface is useful for application development, since it focuses exclusively on the data flowing to/from of each module, without regard for module internals, including the implementation language or internal data representation. Significant dependency reductions can be achieved and module reuse boosted. Another consequence is that modules can be almost anything and run almost anywhere, as long as a communication channel can be established between them. Also, the use of text frames as a communication media allows for flexible module deployment.

As shown, for the ATA system, only two of the six processing modules had to be included in the infrastructure (the other were reused). In addition, the infrastructure is now richer, since the two new modules become available for use in other contexts. The problems encountered concerned data format translation and were easily solved through the inclusion of general mapping steps, using XSLT (as described in Sect. 4). Note that the whole ATA system was built by someone who had no previous experience with the infrastructure. The learning curve was both fast and comfortable (the whole integration task took approximately three days).

The interface, by hiding most of the complexities of the underlying system and of the modules attached to it, empowers non-expert users to play with various scenarios and to investigate possible differences. This is particularly important in environments such as schools, in which students have to become acquainted with the tools relevant for a particular field; and whenever it is desirable to have a fast learning curve, e.g. when new people integrate a project team.

The advantages for expert users lie in simplified prototype construction as well as in application configurations more amenable to change. Also, most irrelevant aspects (those outside the application's domain) may me safely ignored, i.e., the modules may be treated as real black boxes.

The underlying model is useful not only in helping in application construction, but also as a guide to thinking about modular applications: the interface materializes this aspect, since it allows non-expert users to successfully design applications and application components.

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. Another development is to allow multiple chains on the client side. This would make the system more flexible in reusing chains without these having to be transformed in infrastructure-resident programs (reusable, but less amenable to changes).

As a means of efficiently reuse the result sets of previous computations, we intend to include caching capabilities on the client side of the interface handler, thus avoiding unnecessary server/infrastructure calls and improving bandwidth usage.

Acknowledgements. We would like to acknowledge the work by João Graça and Alexandre Mateus on the web interface.

References

1. S. Aït-Mokhtar. *L'analyse présyntaxique en une seule étape*. Thèse de doctorat, Université Blaise Pascal, GRIL, Clermont-Ferrand, 1998.
2. F. Batista. Análise Sintáctica de Superfície e Coerência de Regras. MSc thesis, Instituto Superior Técnico, UTL, Lisboa, 2003.
3. 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.
4. D.M. de Matos, A. Mateus, J. Graça, and N.J. Mamede. Empowering the user: a data-oriented application-building framework. In *Adj. Proc. of the 7th ERCIM Workshop "User Interfaces for All"*, pages 37–44, Chantilly, France, October 2002. European Research Consortium for Informatics and Mathematics.
5. ECMA International, Geneva, Switzerland. *Standard ECMA-262 – ECMAScript Language Specification*, 3rd edition, December 1999. See also: `http://www.ecma.ch/`.
6. C. Hagège. *Analyse syntaxique automatique du portugais*. Thèse de doctorat, Université Blaise Pascal, GRIL, Clermont-Ferrand, 2000.
7. Massachusetts Institute of Technology (MIT), The MITRE Corporation. *Galaxy Communicator (DARPA Communicator)*. See: `http://communicator.sf.net/`.
8. Object Management Group (OMG). *Common Object Request Broker Architecture (CORBA)*. See: `www.corba.org`.
9. Object Management Group (OMG). *Unified Modelling Language*. See: `www.uml.org`.
10. Object Management Group (OMG). *XML Metadata Interchange (XMI) Specification*, 2002. See: `www.omg.org/technology/documents/formal/xmi.htm`.
11. J.L. Paulo. PAsMo – Pós-Análise Morfológica. Technical report, L^2F – INESC-ID, Lisboa, 2001.
12. J.L. Paulo, M. Correia, N.J. Mamede, and C. Hagège. Using Morphological, Syntactical, and Statistical Information for Automatic Term Acquisition. In E. Ranchhod and N. Mamede, editors, *Advances in Natural Language Processing, 3rd Intl. Conf., Portugal for Natural Language Processing (PorTAL)*, pages 219–227, Faro, Portugal, 2002. Springer-Verlag, LNAI 2389.

13. R. Ribeiro, L. Oliveira, and I. Trancoso. Morphossyntactic Disambiguation for TTS Systems. In *Proc. of the 3rd Intl. Conf. on Language Resources and Evaluation*, volume V, pages 1427–1431. ELRA, 2002. ISBN 2951740808.
14. World Wide Web Consortium (W3C). *Extensible Markup Language (XML)*. See: `www.w3.org/XML`.
15. World Wide Web Consortium (W3C). *HyperText Markup Language (HTML)*. See: `www.w3.org/MarkUp`.
16. World Wide Web Consortium (W3C). *The Extensible Stylesheet Language (XSL)*. See: `www.w3.org/Style/XSL`.