Web-based Architecture RES based on finite-state machines for distributed evaluation and development of speech synthesis systems

Matej Rojc
Faculty of Electrical Engineering and Computer Science
University of Maribor
Slovenia
matej.rojc@uni-mb.si

ABSTRACT: *This paper proposes flexible and multi-purpose web-based distributed architecture for a multilingual text-to-speech synthesis system (TTS). The proposed architecture is complex client/server architecture, composed of several modules implemented as finite-state engines and located by different users worldwide. Data exchange (text/audio) between modules is implemented through the use of protocol standards that have wide support throughout the speech and telecommunications area. The integration of modules into complex client/server architecture provides a flexible, reliable, easily re-configurable and maintainable architecture for the TTS system. In the proposed architecture all modules use the same data structure (finitestate engines), all input/output formats are standardized and compatible, the structure is modular, interchangeable, easily maintainable, and object oriented. The proposed architecture has the advantage that researchers can concentrate their efforts on a single module and test its performance within a complete TTS system, composed of modules from different researchers worldwide. The benefit is that such an approach enables the composition of a high- performance TTS system by using high performance modules from different institutions. This paper describes a flexible web-based distributed architecture of the TTS system and exposes its great flexibility regarding numerous simultaneous configuration possibilities.*

Key words: web-based distributed TTS system, Finite-state Machines, MRCP, RTP, UniMod framework, JavaCC

Received: 11 October 2010, Revised 19 November 2010, Accepted 1 December 2010

1. Introduction

Within the TC-STAR project[1], one of the activities was remote running of several text-to-speech systems located with partners. These tasks were accomplished by using the UIMA system from IBM, after some modifications to the framework were performed. UIMA stands for *Unstructured Information Management Architecture*. It is an open, industrial-strength, scaleable and extensible platform for creating, integrating and deploying unstructured information management solutions from combinations of semantic analysis and search components. Although UIMA originated at IBM, it has now moved on to be an Open Source project which is currently incubating at the Apache Software Foundation. UIMA's goal is to provide a common foundation for industry and academia to collaborate and accelerate the world-wide development of those technologies critical for discovering the vital knowledge present in the fastest growing sources of information today (UIMA Overview & SDK Setup, 2007). Nevertheless, UIMA system was not that easy to integrate with several external TTS modules, and partners had to do some modification on the framework before able to use it.Therefore, one of the objectives of the European Centre of Excellence for Speech Synthesis[2]

[1] www.tc-star.org

[2] www.ecess.eu

was to compose a novel modular, distributive, and more flexible TTS system. The ECESS ideas about splitting the TTS system into more specified modules, the implementation of complex client/server architecture through the finite-state machine formalism, simultaneously running various TTS systems' configurations by different users, and the simultaneous behaviour definition of the constituent system's modules open new possibilities in web-based distributed processing and in the related evaluation methodology of modules and systems.The proposed RES architecture also incorporates protocol standards that gain wide support within the speech and telecommunications areas. Recent synthesis systems are usually capable of processing some form of XML as input and use specialised XML vocabularies for internal communication between modules, as well as speech data annotation. The advantage of an XML-based interface is that existing libraries and software can be used for the generation, validation, and parsing of data, thus ensuring a fast and flexible interface implementation and re-definition. To facilitate a distributed development, all modules should be network applications, behaving as servers with respect to their predecessors and as clients with respect to their successors in the module hierarchy. The split of a general TTS structure into modules has the advantage that each institution can concentrate its effort on a single module and test its performance in a complete TTS system using missing modules from other researchers. In this way, high performance systems can be built using high performance modules from different institutions. One goal of using web-based distributed architecture is certainly to check the performance of those modules within the established common evaluation methodology. This methodology is based on the common use of those module-specific evaluation criteria and modulespecific language resources needed for training and testing the modules. Therefore, the developed architecture is called the "Remote Evaluation System (RES)". The other goal is that the architecture (RES system) is used for the construction of multilingual and multimodal TTS systems in several configurations. The TTS system constructed from the RES system's modules will be addressed as ECESS TTS system throughout the paper. RES system and ECESS approach does not demand any source code or data resources exchange. On the other hand it offers common specification development, evaluating, testing, and comparing different approaches and solutions between partners in flexible ways without laborious and tedious work. All users have possibility to run via RES up-to-date modules of other partners, doesn't matter what changes were made in between (since no proprietary code exchange is performed). They simply always have the same situation via the RES system. RES system is architecture-framework-skeleton for developers. Its functionality and structure is defined by any number of needed modules running TTS modules. RES system behaviour can be described by using XML mechanism. Further, you can integrate modules developed on different platforms. You can give the content and information you want or need from the RES system by selecting resources and modules all over the world. In this approach any desired or needed granulation of TTS processing steps can be defined, or any desired or needed logic processing task sequence can be performed. Finite-state machine based structure of all RES system's modules enable flexible description of RES module's behaviour in XML format. These descriptions can be easily added for emerging tasks to the RES system without need for constant re-compiling and updating RES modules of partners. The paper is organized as follows. Section 2 describes the functional scheme of the RES system. Section 3 addresses RES development issues. The design of ECESS TTS system using the web-based distributed RES system is described in section 4. A protocol implementation running a distributed TTS system is given inn section 5. Data format issue in the RES system is exposed in section 6. In section 7 all constituent RES system modules are presented in more detail. Section 8 demonstrates the construction of the web-based distributed TTS system, and at the end a conclusion is drawn.

2. Functional scheme of the RES system

The basic web-based distributed architecture of the RES system is presented in Figure 1. It consists of one or more RES clients, one main RES server (managing unit - MU) and one or more RES module servers. The RES system must take care of accessing various TTS modules without the need to install these components or resources locally. RES server (MU) is the core of the system that connects RES clients and RES module servers. RES clients and RES module servers can be located by different partners worldwide. All RES modules are connected to the internet using TCP/IP and UDP connections. The RES server is able to communicate with one or several partners' RES module servers in order to build partial or complete ECESS TTS system. The RES server can also communicate with one or several RES clients, simultaneously. Also the RES module servers can communicate with several RES clients simultaneously.

Obviously, such a scheme requires efficient and complex client/server architecture. The RES system is flexible, robust and supported by efficient monitoring and error recovery mechanisms. The RES system can be used by researchers to run TTS components or tools of other partners needed to evaluate and improve the performance of their own TTS components or tools. Researchers are allowed to perform different evaluation tasks of their modules using the necessary additional modules and/or language resources of other researchers that are installed locally at the other researchers' sites. The architecture enables the running of partial or complete web-based distributed TTS systems, composed of any selectable researchers' TTS components

available on the web. Firstly, researchers are able to integrate their own modules into the complete web-based distributed TTS system that uses the rest of the necessary modules of other partners. And secondly, by using the RES client the researchers are able to build up via web the complete webbased distributed TTS system in numerous configurations, even without using any of their own TTS components. The web-based distributed architecture of the RES system from the protocol point of view is shown in Figure 2. In the figure it can be seen that each user needs RES client in order to run the RES system. Since users can use the RES system in different ways, three different kinds of users are located in the figure. **User I** (e.g. evaluation institution) has only RES client module. This user is able to run any other user's TTS component available on the web, or their combination, or complete TTS system. **User II** has RES client module and RES module server. All users who would like to integrate their TTS component into the web-based distributed RES system, also needs a RES module server that takes care of running their TTS component and transferring I/O data between the RES server and corresponding TTS component. These users are able to integrate even more TTS component into the RES system, not just one. But one RES module server must be used per each TTS component. **User III** only has a RES module server. These users are unable to run the RES system in any possible configuration, but they integrate their TTS component into the RES system and make it available to other users. In this case the users would just like their TTS component to be evaluated. Each RES system module uses two XML scripts. One for module configuration (XML configuration file), and one for its behaviour definition (XML protocol scenario file). Since this is complex web-based distributed architecture, all modules must communicate with each other in meaningful ways following used protocol standards within the RES system.

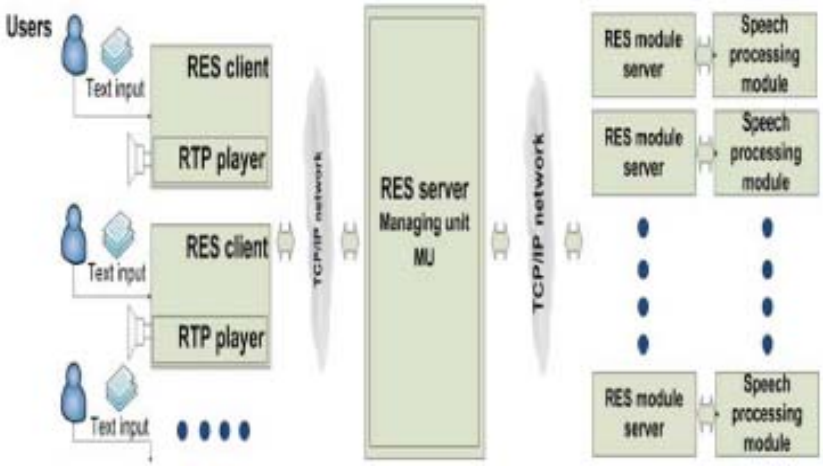

Figure 1. Basic web-based distributed architecture of the RES system

Users running RES clients actually define how and in what configuration the RES system will be used. Different users can even simultaneously run different RES system configurations. Therefore, all XML scripts that define the behaviour of the main RES system and included RES module servers in a way as the corresponding user would like, are transferred by corresponding users' RES client module. This proposed solution in the web-based distributed architecture enables great flexibility and numerous, even simultaneous, re-configuration possibilities of the RES system. Data exchange (text/audio) between modules is implemented today through the use of protocol standards that gain wide support in the speech and telecommunications areas. As it can be seen, the RES clients and the main RES server (MU) use the MRCP protocol (Media Resource Control Protocol) for ascii data exchange. This protocol is used as a control for speech synthesizers to stream audio from a common location to the user. This protocol is implemented inside RTSP sessions using RTSP protocol. RTSP protocol is based on TCP, a secure, connection-oriented protocol. This way there is no need for the RES client or RES server to implement any additional error-correction mechanisms. The RES server (MU) and the RES module servers use proprietary XML-based protocol, taking care of data exchange in both ways. Some RES module servers run acoustic TTS components that generate audio data as output. Audio data are transferred from the corresponding user's site through the RES server to the dedicated RES client, by using the RTP protocol (Burke, 2007). How is the RES system used? Each user is able to put on the web locally, one or more TTS components (by using one RES module server per each TTS component), and/or each user just accesses various TTS components or some combination of them (by using RES client) that are made available by other users on the web. By using the RES client, users are able to select any RES module server within the RES system, running specific TTS component. They may want to evaluate it or just use it within a complete TTS system. Users have to define the appropriate input. The input is then sent through the RES server (MU) to the selected RES module server/TTS component, where it is stored in some predefined file. Then the selected TTS component

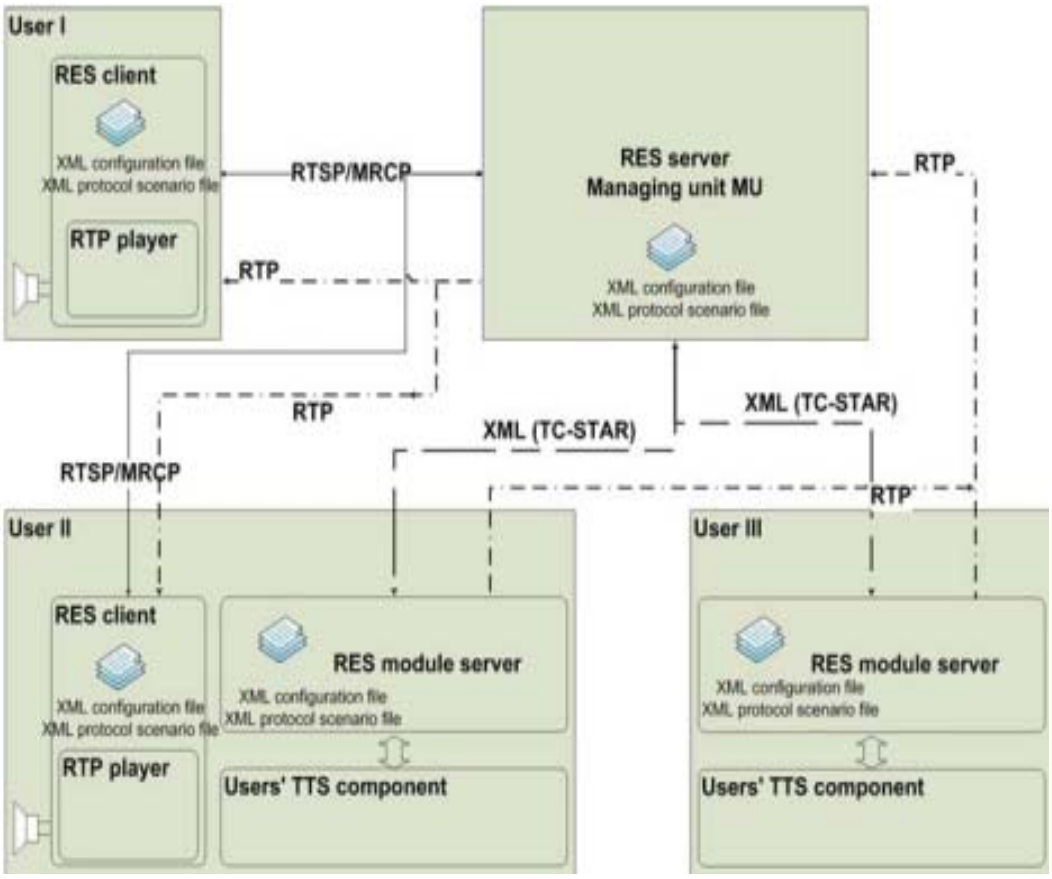

Figure 2. Web-based distributed architecture of the RES system from the protocol point of view

is run by the RES module server, and output data are generated. These data are stored into a predefined output file, which is then sent back through the RES server (MU) to the RES client. In this way, no manual effort is necessary for the developers of a TTS component to take part in the evaluation. Integration of additional TTS components into the proposed RES system is very simple. In order to construct the complete TTS system in any configuration, users just have to select three RES module servers (running TTS components for text processing, prosody processing and acoustic processing) from any users who make these TTS components available via the web. This last aspect of using the RES system is mainly addressed in this paper.

3. Web-based distributed architecture – RES system

The proposed RES system is platform independent, able to run users' TTS components on Linux and Windows platforms. The user is able to select among available TTS components on the web and is able to select any desired combination of them and construct desired configuration of TTS system. I/O data has to be compatible between any two subsequent TTS modules based on TCSTAR data format (Bonafonte et al., 2006; Perez, 2006). The user is informed, which are available TTS components on the web through the RES client GUI. The RES server is able to simultaneously run various behaviour descriptions, call various RES module servers simultaneously etc.

3.1 RES client

The RES client functional architecture is shown in Figure 3. At the user level, GUI enables users to select TTS components (users' RES module server selection), specify input data to the RES system, process output data generated by the RES system, specify RES system behaviour and task specification, and monitoring FSM engines within RES system. The RES client module is basically an FSM engine (Mohri, 1995). It is described in the form of XML scripts (XML protocol scenario files), already written off-line. XML script is at run-time first loaded, compiled into the FSM engine, validated, and finally run.

Therefore, using these XML scripts the RES client behaviour is actually specified and contains a description of the desired task/

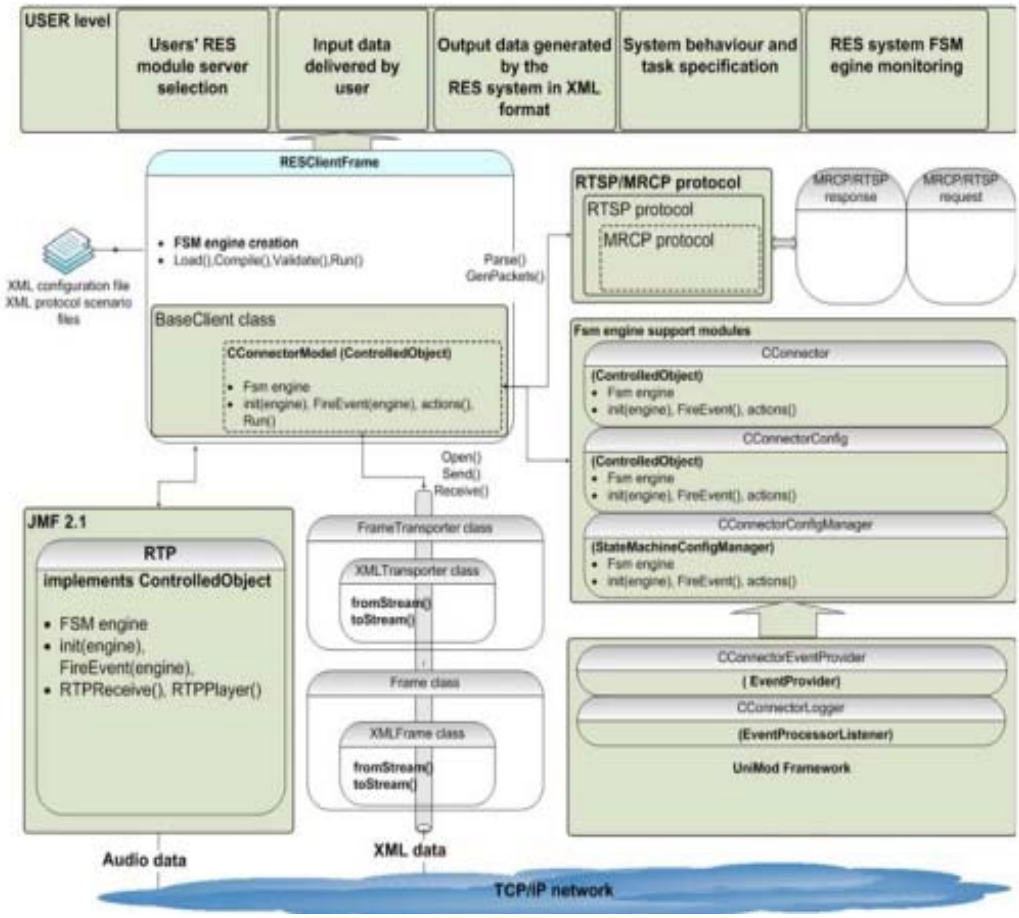

Figure 3. The RES client functional architecture

role within the RES system. Since users are able to select any RES module servers and use the RES system in many different ways, all XML scripts (also those specifying the desired behaviour of the RES server and included RES module servers) are located at the RES client side. They are sent on the RES server and included RES module servers, after the user specifies the RES system behaviour. Therefore, by using the RES client, users are able to 'define' the behaviour of the whole RES system according to their needs, and the nature of the task. The RES client's finite-state machine engine consists of several "Controlled Object", "EventProvider", "EventProcessorListener" and "StateMachineConfigManager" objects, as seen in Figure 3. All these objects are implemented versions of corresponding interfaces specified within the UniMOD framework (Weyns, 2007). All RES client objects (the actions of which have to be run by an FSM engine) are also controlled objects. Controlled objects are also able to "fire" events using the "FireEvent" method. These events actually trigger the transitions of the finite-state machine engine. Additional event processing objects are used for the definition and detection of events that are, in general, randomly triggered within the RES client. As a whole, they constitute a flexible FSM engine. Such FSM engines are also used in RES server and RES module server. Audio data can be received via UDP port and processed by RTP module developed within the JMF 2.1 framework (Alejandro, 2002). Between the RES client and RES server the RTSP/MRCP XML data packets are exchanged. Received packets must be parsed and sent packets must be generated. For this purpose the RTSP/MRCP module is included in the RES client.

3.2 RES server

The RES server's functional architecture is shown in Figure 4. It is a multi-threaded module, where the specific number of threads is defined in the so-called pool structure (PooledThread). Each thread is dedicated for serving a specific RES client and running specified RES module servers. When the RES server accepts the request from the RES client, the already initialised thread is picked up from the pool of threads and used for running a new RTSP/MRCP session started with the RES client. When no more

threads are available, the RES server rejects any further RES clients. The default number of threads in the pool is currently 200, but can be changed in the XML configuration file. At the user level, GUI enables monitoring traffic between the RES client and RES server, and between the RES server and RES module servers. Additionally, the RES server status (used/available threads, established connections with the RES clients and RES module servers etc.), and FSM engine traversal can also be monitored. The RES server does not know what should be done after connection with the specific RES client is established, since the servers' behaviour is still unspecified, and is unknown. The RES servers' behaviour and actions to be performed are defined by the corresponding XML script, send by the RES client. This is a very flexible solution as it is possible that e.g. 200 RES clients run different configurations of the TTS system simultaneously, each one in its own thread. Next, the RES server establishes a connection with the included RES module servers, and transfers input data from the RES client to the first RES module server in the sequence. The connection between the RES server and RES module server is again a client/server connection. ASCII data sent between RES server and RES module server are in TCSTAR data format. When results from the RES module server are received by the RES server, the RES server just forwards these data back to the RES client and closes the established connections. When running acoustic processing TTS components (as in the case of the complete TTS system), the RES server additionally acts as a RTP receiver and RTP transmitter (RTP module), forwarding audio data obtained from the RES module server to the RES client. Between the RES client and RES server, RTSP/MRCP XML data packets are exchanged, and between the RES server and RES module servers, proprietary XML data packets are exchanged. The received packets must be parsed and the sent packets generated. For this purpose, the RTSP/MRCP module is included in the RES server for processing packets between the RES client and RES server, and additional the XML module for processing packets between the RES server and RES module servers.

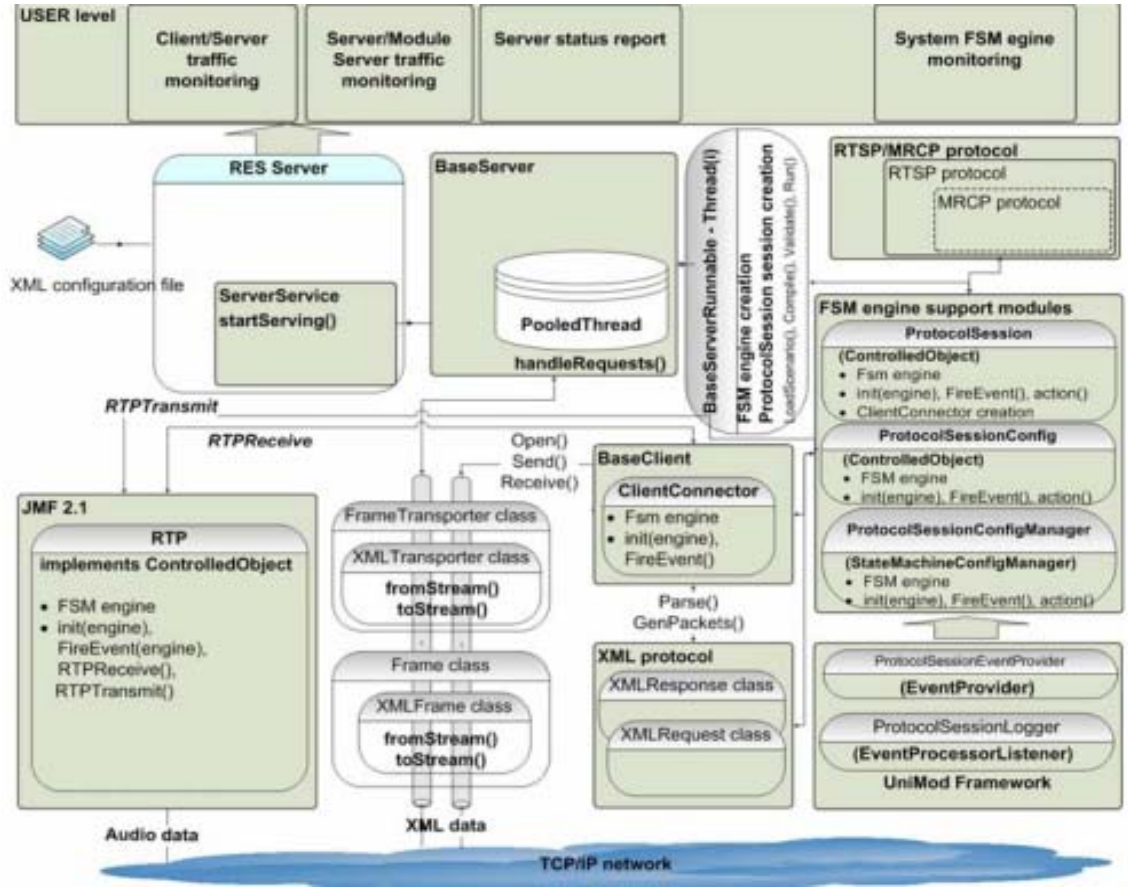

Figure 4. The RES server functional architecture

3.3 RES module server

In order to "integrate" users' TTS components into the RES system, the RES module server was developed with the following functional architecture, as shown in Figure 5. The RES module server's architecture is quite similar to RES server architecture. Namely, it also contains a pool of threads (PooledThread) and dedicates new thread for each new call coming from the RES

server. Default number of threads is 200, but this can be changed in the XML configuration file. Therefore, the RES module server can serve more RES clients simultaneously. But since the RES module server has to run users' TTS component, it contains an additional class named "CommandExecution". This class takes care for running TTS components, commands or even scripts. The RES module server and users' TTS components run as separate programs. In this way the RES system is insensitive to users' TTS components' crashes, which could happen for any given input. The error handling mechanism integrated into the RES system, only reports about an error to the RES server and to the RES client. At the user level, GUI enables the monitoring of traffic between the RES server and RES module server, the RES module server status (used/available threads, established connections with the RES server etc.) and FSM engine traversal can also be monitored. The RES module server does not know what should be done after connection with the RES server is established, since the module server's behaviour is still unspecified, and unknown. The RES module server's behaviour and actions to be performed are defined by the corresponding XML script sent by the RES client. This is a very flexible solution as it is possible that e.g. 200 RES clients run different configurations of corresponding RES module server simultaneously, each one in its own thread. The user has to configure the RES module server regarding IP/port settings and specify in the XML configuration file to the command that can run their TTS component. In this way, the integration of running any TTS component by the RES module server is done by simply including the name of a program or a script to be called, e.g. from the command prompt, in the XML configuration file. The input which is sent by the RES client through the RES server (MU) is stored by the RES module server in a predefined file. Then the users' TTS component or a script is started automatically by the RES module server (CommandExecution class). The RES module server waits until execution of the users' TTS component finishes and stores the output results in a predefined file sent through the RES server (MU) back to the RES client. When running acoustic processing TTS component using the RES module server, the RTP transmission module takes care of transferring the generated audio through the RES server back to the RES client, using RTP protocol. Proprietary XML data packets are exchanged between the RES server and RES module server. The received packets must be parsed and the sent packets must be generated. For this purpose an XML protocol module is included in the RES module server for processing packets between the RES server and RES module server.

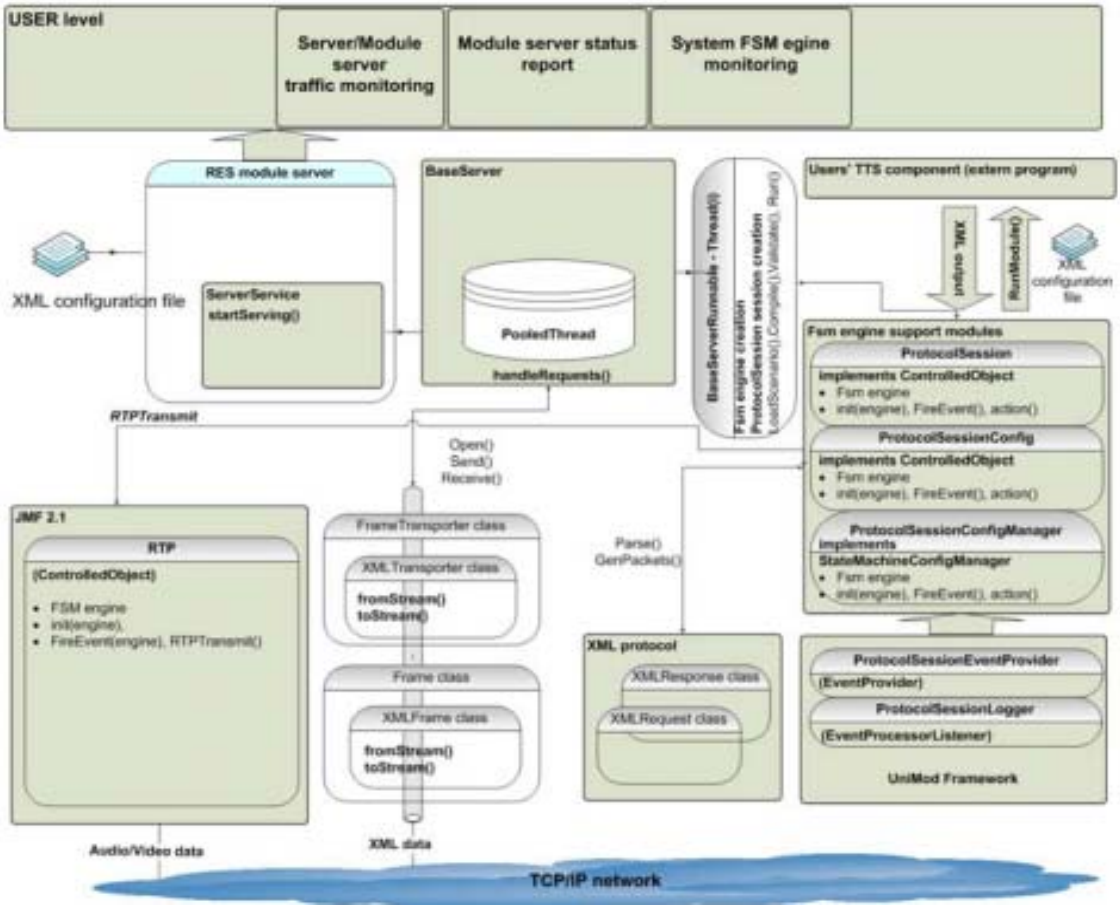

Figure 5. The RES module server functional architecture

4. RES system modules' behaviour implementation

RES system modules must co-operate in order to perform certain dedicated task for the whole system. These tasks can be different e.g. evaluation of text processing, prosody processing tools from different users or running the complete web-based distributed TTS system. It is clear that such co-operation demands specific data packet exchanges in a time synchronous/ asynchronous manner, and different actions and events within different RES modules, following RTSP/MRCP, XML, RTP protocol specifications. Therefore, each RES module's behaviour has to be defined, described and presented in a flexible and efficient manner for each specific task performed by the RES system (e.g. constructing and running a common TTS system). Implementation of the RES module's behaviour is proposed to be performed by using the UniMOD framework (Shalyto, 2001, Weyns, 2007). The communication protocols that have to be followed by the RES client, RES server, and RES module servers, regardless of the task within the RES system, can be described by states and actions. Within each time frame, each RES module (as shown in Figure 6) processes specific events and performs specific actions that have to be co-ordinated with actions and the events of all other RES modules within the RES system. Therefore, these time frames can be identified by states of the finite-state machine. In each state the RES client, RES server, and RES module servers perform certain actions or have to wait for specific events in order to proceed into the following states. All this information can be described and presented by flexible and efficient formalism – finite-state machine formalism (Mohri, 1995). Since the RES system modules have a specific role within the system, they have different finite-state machine descriptions, but the general idea and the way these finite-state machines are built, are the same. As already mentioned, RES modules are finite-state engines able to run finite-state machine descriptions, as long as they are written in standardized UniMod data format. They are XML scripts, describing the desired behaviour of a specific RES module in the form of graphs. Using finite-state machines makes the RES system very flexible and adaptable for running many different tasks including running a common distributed TTS system. XML scripts that specify the behavior of each RES module are firstly written from the graph as shown in Figure 6. This graph illustrates how the behaviour of the RES client can be fully defined for certain task within the RES system. It is composed of state nodes that are interconnected by one or more joined transition arcs. Each transition is described by unique name, guard function (specifying conditional transitions), event name that triggers corresponding transition, transition source state, transition target state, and transition output actions. Events are generated automatically within the RES system or by the users on the RES client GUI side. Nevertheless, most events are "fired" when packets are received or transmitted, certain conditions are met, or some action "fires" the event. Therefore, all RES modules are event-driven finite-state engines, able to perform any specified sequence of actions, as specified within the RES system.

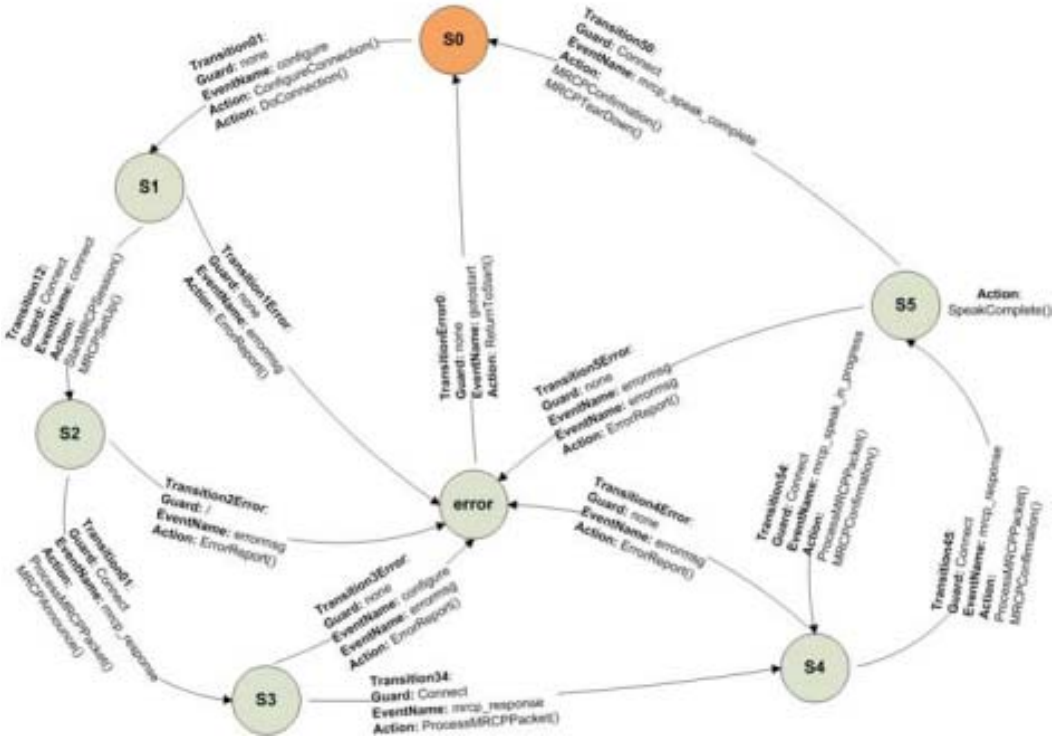

Figure 6. Graph describing the behaviour of the RES client

5. Flexible data format conversion

A very important issue is that TTS components of different users use different input/output data formats. Usually, very different than the format that is used within the RES system. The adaptation of proprietary formats to this data format can take a lot of effort and can be quite time consuming. The crucial necessity is to make this step as flexible and as easy as possible for the users, interested in running their TTS components within the RES system. The *Unforma* tool has been developed to solve this problem. It is Java-based tool that integrates Java and JavaCC compilers. The tool enables the writing of Java parser classes for converting proprietary data formats into the RES system TC-STAR data format, and vice versa. Two Java parser grammars have to be written for each user TTS component (e.g. XFormat_2_TCSTAR.jj and TC-STAR_2_XFormat.jj), by using JavaCC syntax. The generated Java parser class files are then loaded within the *UnForma* tool and verified on a certain file given in the proprietary data format. In this way, we are able to iteratively test/correct parser again and again until the expected conversion is obtained. Finally, developed parsers are simply added to the RES module servers. By using this proposed approach, partners don't have to change their TTS modules before integrating them into the RES system.

6. ECESS TTS system

Figure 7 shows the architecture of the web-based distributed ECESS TTS system implemented by the RES system modules described above. The RES system depicted in this figure performs a specific set of actions, and should process a specific set of events. These sets of actions and events are defined by the used protocols (RTSP, MRCP, RTP, XML) and by ECESS specifications of the ECESS TTS system architecture. ECESS TTS system architecture is composed of the following RES modules: the RES client, RES server and three RES module servers. All three RES module servers are used for running the following users' TTS components: text processing, prosody generation, and acoustic processing TTS component (marked as partners A, B, C). In order to use the RES system as a web-based distributed ECESS TTS system, the behaviour of the RES system modules has to be specified in the form of XML scripts. In these scripts, for each RES module, interaction mechanisms between RES modules in order to function as a complete distributed ECESS TTS system are described and specified (when and where to connect, when and what data has to be exchanged between modules etc.). Therefore, XML scripts can also be called RES modules' scenarios (written off-line). The proposed approach makes the RES system very flexible, since by writing new XML scripts, the behaviour of RES system modules can be arbitrarily changed, enabling numerous configuration possibilities for the RES system and distributed ECESS TTS system. Each of the RES modules (RES client, RES server and RES module servers) uses specific XML script, because of their specific role in the ECESS TTS system. The RES modules perform actions and events in specific sequences and combinations (time synchronously or asynchronously), which actually defines the behaviour of each RES module within ECESS TTS system. Communication between the RES modules within the ECESS TTS system can be split into three phases, as presented in figures 8-10.

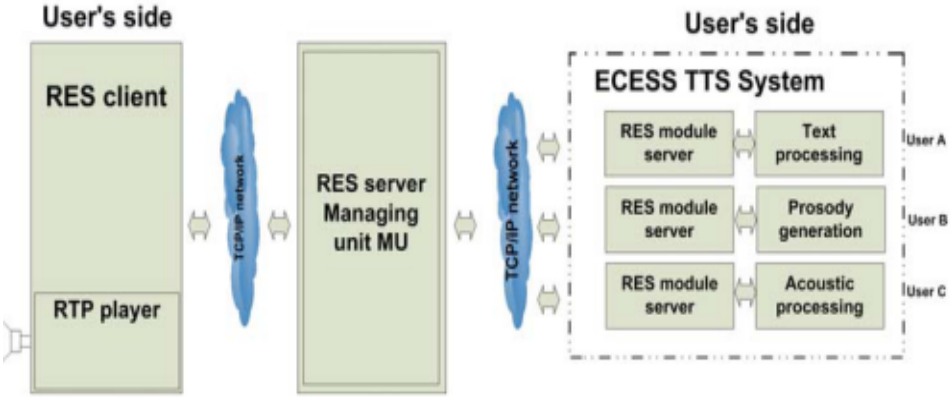

Figure 7. ECESS TTS system architecture

Figure 8 illustrates phase 1, which describes data packet exchange between the RES client, RES mserver (RTSP/MRCP protocol is used), and the first RES module server for text processing. After the user feeds in the GUI input text, the RES client automatically opens a connection to the RES server, sending a GET-PARAMS request packet in order to obtain information about the available RES module servers (runs users' TTS components) on the web from the RES server. The RES server responds with the RTSP confirmation packet and returns required information. Then the user has to select RES module servers for all three TTS

Figure 8. Running ECESS TTS system – phase 1

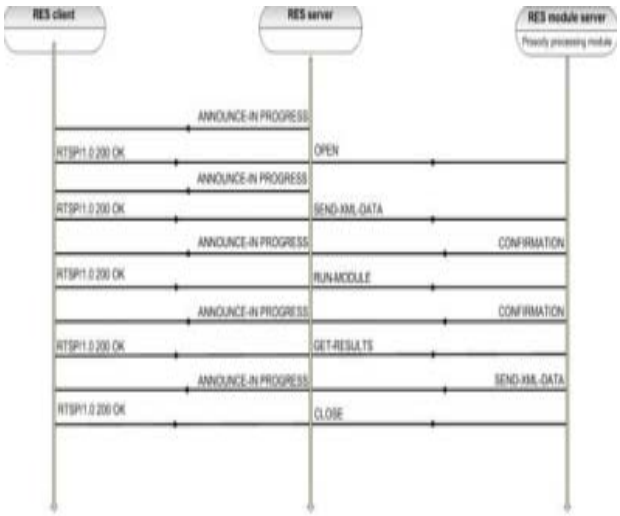

Figure 9. Running ECESS TTS system – phase 2

components (text processing, prosody generation and acoustic processing) from those available on the web. Next, the RES client sends a SET-PARAMS request packet with the XML scripts describing the RES server and included RES module servers' behaviour (as defined within the ECESS TTS system), to the RES server. After confirmation from the RES server, the RES client finite-state machine engine can start the ECESS TTS system session by sending the SETUP request packet. After confirmation from the RES server is received, it continues with ANNOUNCE request packet containing given input text data. As can be seen in Figure 8, the RES server (by now it already knows its behaviour from the received XML script from the RES client) first connects to the RES module server running the text processing TTS component selected by the user. After establishing a client/server connection between the RES server and RES module server for text processing, the corresponding RES module server runs the users' TTS component for text processing, waits for the results, and obtains them using the GETRESULTS method. At the same time the RES server constantly sends ANNOUNCE-IN PROGRESS messages to the RES client. In this way the RES client is able to monitor what is going on within the ECESS TTS system, and is able to automatically recover in case of errors. When the RES server obtains results from the RES module server performing the text processing task, it closes the connection and connects to the next RES module server, responsible for running the prosody generation TTS component as selected by the user, as can be seen in Figure 9. This figure describes data packet exchange between the RES server and the second RES module server responsible for running the user's prosody generation TTS component. As can be seen, the RES server keeps informing the RES client that the TTS task is still in progress by ANNOUNCE-IN PROGRESS packets and sends output from the first RES module server running text processing tool to the next RES module server, running the prosody generation TTS component. Communication between the RES server and the RES module server for prosody generation is very much the same as in phase 1, only input/output data are different. The RES server keeps sending ANNOUNCE-IN PROGRESS messages to the RES client, until the RES module server for prosody generation responds with generated output to the RES server (after GET-RESULTS packet). Then the RES server closes the connection with the RES module server for prosody generation, and initiates the connection with the last RES module server responsible for running acoustic processing TTS component.

This step is shown in Figure 10 (phase 3) and describes data packet exchange between the RES client, RES server (RTSP/MRCP protocol is used), and the last RES module server for acoustic processing. The RES server keeps informing the RES client that the TTS task is still in progress by ANNOUNCE-IN PROGRESS packets and sends the obtained output from the prosody generation TTS component to the RES module server used for running the acoustic processing TTS component. The RES server keeps sending ANNOUNCE-IN PROGRESS messages, until the RES module server for running the acoustic processing TTS component responds by obtained audio output to the RES server, and closes the connection with the RES server. At the end, the RES server sends an ANNOUNCE-SPEAK COMPLETE message to the RES client in order to notify the RES client that the TTS task has been finished and that generated audio data has been transferred to the RES client. The RES client responds with confirmation and TEARDOWN message, as specified by MRCP protocol. After final confirmation from the RES server, the RES client closes the connection with the RES server. Additionally, the RTP transmission of generated audio data from the last RES

module server to the RES server and then further from the RES server to the RES client is also performed at the end and played out to the user. The RES client has the role of an RTP player. RTP players and *processors* provide the presentation, capture, and data conversion mechanisms for incoming RTP streams.

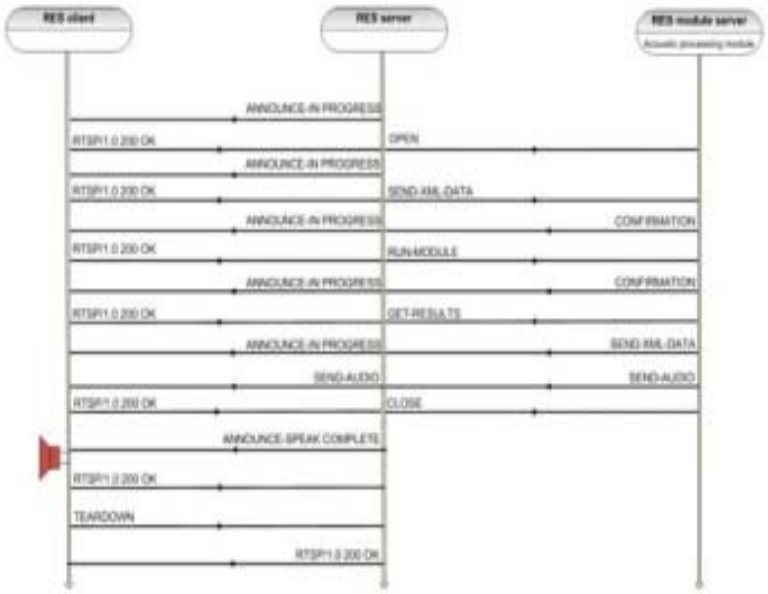

Figure 10. Running ECESS TTS system – phase 3

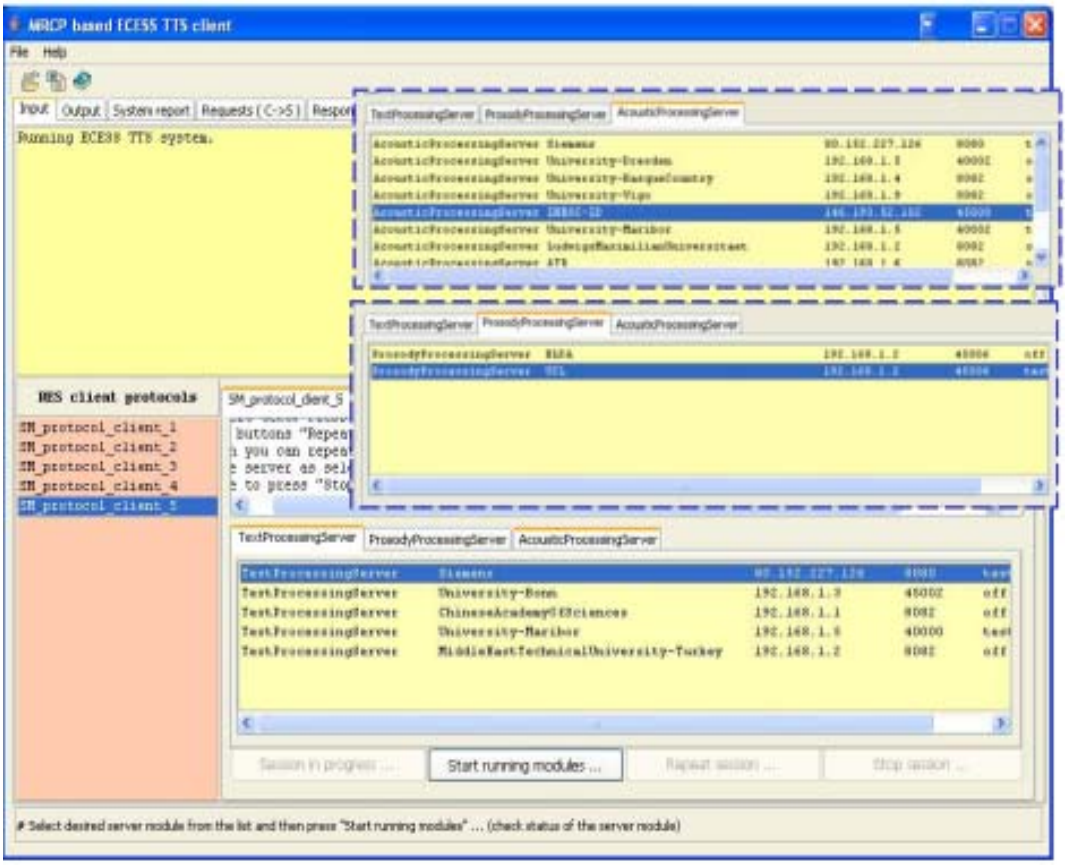

Figure 11. One of several ECESS TTS system's configuration

In order to run the ECESS TTS system in any configuration, the user has to install the RES client and run it as shown in Figure 12. As it can be seen from the figure, the user is currently able to select between five list items (*SM_protocol_client_5*). These items define those different RES system architectures that are currently used for different tasks performed by the RES system. Each architecture further uses a specific set of XML scripts for RES modules behaviour specification. The RES system architecture named *SM_protocol_client_5* can be used for running the RES system as ECESS TTS system. After selection has been made, the RES client starts using XML scripts dedicated for ECESS TTS system task, and enables panels with those RES module servers that run the three needed TTS components (text processing, prosody generation and acoustic processing). All available users' TTS components are shown to the user. The user has then just to select the desired RES module server from each panel (one from each group), as can also be seen in Figure 11. After the selection of all three RES module servers has been made, the user can run the constructed ECESS TTS system with this configuration, giving text input and listening to incoming generated audio data. It is obvious that it is very easy for the user to change the selected configuration into some other combination of available RES module servers.

7. Conclusion

This paper presents RES system, the web-based distributed architecture, for the constructing and evaluating of ECESS TTS systems. The architecture is used for online evaluation and the demonstration of various ECESS TTS modules running at different institutes and universities worldwide. The users can put their modules on the web locally using the RES module server, or just access different modules via the web using the RES client. The modules in the architecture are accessible via TCP/IP. In the paper, protocol standards are suggested that gain wide support in the speech and telecommunications areas today for use in data exchange. It is suggested that the proposed architecture, the RES client, RES server and three RES module servers, are used for the ECESS TTS system. Each RES module server performs specific action sequences, which are defined by the XML script describing the finite-state machine. The TC-STAR XML format is suggested for use in data exchange. Different partners use different formats in their TTS modules. In order to speed up the integration process, the *Unforma* tool, integrating the JavaCC framework, has been developed. By using this tool, it is possible to write Java parsers for I/O data conversion processes, which are then compiled into Java classes, and included in the RES system using a configuration file.

References

[1] Alejandro Terrazas, John Ostuni, Michael Barlow. (2002). Java Media APIs: Cross-Platform Imaging, Media and Visualization, Sams publishing.

[2] Bonafonte, A., Höge, H., Kiss, I., Moreno, A., Ziegenhain, U., Van den Heuvel, H., Hain, H-U., Wang, X. S., Garcia, M. N. (2006). TC-STAR: Specifications of Language Resources and Evaluation for Speech Synthesis, Proc. LREC.

[3] Burke, Dave. (2007). Speech Processing for IP networks / Media resource control protocol (MRCP). John Wiley & Sons, Ltd. Copeland, Tom. Generating Parsers with JavaCC, Centennial Books, Alexandria.

[4] Mohri, M. (1995). On Some Applications of Finite-State Automata Theory to Natural Language Processing. Natural Language Engineering 1. Perez, J., Bonafonte, A., Hain, H-U., Eric Keller, Breuer, S., Tian, J. (2006). ECESS Inter-Module Interface Specification for Speech Synthesis, Proc. LREC.

[5] Shalyto, A. A. (2001). Logic Control and "Reactive" Systems: Algorithmization and Programming, *Automation and Remote Control*, 62 (1) 1-29. Translated from Avtomatika i Telemekhanika, No. 1, p. 3-39.

[6] UIMA Overview & SDK Setup. (2007). Written and maintained by the Apache UIMA Development Community (http://incubator.apache.org/uima/downloads/releaseDocs/2.2.0incubating/docs/html/overview_and_setup/overview_and_setup.html).

[7] Weyns D., Boucke N., Holvoet T., Demarsin B. (2007). DynCNET: A protocol for flexible transport assignment in AGV transportation systems. Katholieke Universiteit Leuven, Report CW 478.