Deliverable 5.1

Design of components for understanding, dialogue management and feedback to the user

Authors: Roberto Manione (NewAmuser),
Fiorenza Arisio (NewAmuser),
Elisabetta Gerbino (NewAmuser),
Claudio Giuliano (FBK),
Marco Matassoni (FBK)

Date: March 15th, 2013

Document Type: R

Status/Version: Final 1.0

Dissemination Level: PU
D5.1 - Design of components for understanding, dialogue management and feedback to the user

Project Reference | FP7-ICT-2011-7 - 288121
--- | ---
Project Acronym | DIRHA
Project Full Title | Distant-speech Interaction for Robust Home Applications
Dissemination Level | PU
Contractual Date of Delivery | December 31, 2012
Actual Date of Delivery | March 15, 2013
Document Number | DIRHA_D5.1_20130315
Type | R
Status & Version | Final 1.0
Number of Pages | 3+83
WP Contributing to the Deliverable | WP5
WP Task responsible | Roberto Manione (NewAmuser)
Authors (Affiliation) | Roberto Manione, Fiorenza Arisio, Elisabetta Gerbino (NewAmuser), Claudio Giuliano, Marco Matassoni (FBK)
Other Contributors | 
Reviewer | 
EC Project Officer | Pierre Paul Sondag

Keywords: understanding, dialog management, voice user interface, voice recognition, state charts, state machines,

Abstract: This document describes the achievements obtained during Year 1 in the DIRHA project towards the developments of components aimed to handle and fulfil user requests expressed through spoken utterances. The Dialog Manager, the Speech Understanding, the Prompt Producer, and the House State Keeper are devoted to figuring out users’ requests from their utterances and fulfilling them, issuing the proper commands to the House Automation system; to accomplish this task they gather input from the rest of the DIRHA system, in particular Source Localization, Speech and Speaker Recognition, dealt with by other WPs of the project.
D5.1 - Design of components for understanding, dialogue management and feedback to the user

Contents

1. Introduction ...................................................................................................................... 1
2. DIRHA approach to Concurrent Dialogue Management .............................................. 3
   2.1 Introduction to Dialog Management ......................................................................... 3
   2.2 Design choices in the DIRHA Dialog Management ............................................... 5
3. Natural Language Understanding .................................................................................... 10
   3.1 Grammar-based approach ....................................................................................... 10
   3.2 Data-driven approach ............................................................................................. 14
4. Design of the State Based Concurrent Dialog Manager ............................................. 24
   4.1 Introduction to the MIA-XML language .................................................................. 24
   4.2 The Core Constructs of the MIA-XML language .................................................... 30
   4.3 Executable Content in the MIA-XML language ..................................................... 31
   4.4 Data modelling in the MIA-XML language ............................................................. 32
   4.5 External Communication in the MIA-XML language ............................................ 35
   4.6 Release 1.0 of the MIA-XML executor .................................................................. 37
5. House+User Profile and House+User State ................................................................. 40
   5.1 The House+User Profile .......................................................................................... 40
   5.2 The House+User State ............................................................................................ 41
   5.3 Synchronization among physical resources and House State ................................ 44
6. Integration of the CDM within the DIRHA environment ............................................. 45
7. Design of the User Interface Dialogue Flow ................................................................ 50
   7.1 Introduction to the Dialog Flow .............................................................................. 51
   7.2 Discussion of the dialogue flow .............................................................................. 54
8. Conclusions ..................................................................................................................... 62

Appendix 1. The MIA-XML Reference Manual ................................................................ 67
   A1.1 Core Constructs of the MIA-XML language ........................................................ 67
   A1.2 Executable Content in the MIA-XML language ................................................. 72
   A1.3 The Data Model in the MIA-XML language ....................................................... 76
   A1.4 External Communication in the MIA-XML language ......................................... 78
   A1.5 The DTD of the MIA-XML language as of Release1.0 ...................................... 82
1. Introduction

This document describes the activities and achievements in Year 1 of the DIRHA project in understanding and dialog management, that is the part of the project dedicated to understand and negotiate users’ requests from their utterances (gathering input from the rest of the DIRHA system, in particular the speech/speaker recognition and the source localization components, described in other deliverables) to implement them through the House Automation system.

While in a few cases this job will simply be to execute the requested command and play a confirmation prompt back, in most cases the Dialog Manager shall guide the user to complete the needed formation through additional prompts and possibly confirmation requests for the most important functions.

Moreover, the Dialog Manager shall be able to handle more than one dialogue at the same time, in case more users interact with the system from distinct positions in the house (i.e. rooms) at the same time on separated issues, keeping for each of them its state; such a feature, taking advantage of the microphone network and the source localization resources built in the DIRHA system, will allow handling one different dialog in each room of the house; such dialogues will be handled by one single Concurrent Dialog Manager, handling the House State as well in a coordinated way; this would potentially allow the implementation of behaviours triggered by the joint state of the various dialogues going on at the same time.

The document discusses the following components:

The Dialog Manager is in charge of conducting the interaction with the user coordinating the other modules towards the goal of gathering from the user the information needed to accomplish the requested task; multiple turns of dialog could be needed in case the user is not providing enough information in its first utterance, or it is not fully recognized; in such a case dedicated questions are asked, generating the proper prompts to obtain by the user the needed information to operate the house automated devices. An introduction to Dialog Management and a discussion of the approach followed within the DIRHA project can be found in Section 2. the reusable concurrent dialog manager engine developed in the project is described in Section 4.

The Speech Understanding module is dedicated to parse the users’ utterances as returned by the ASR at the purpose of extracting the information needed by the Dialog Manager; typically two approaches can be followed to accomplish this task. The most established solution is based on knowledge: defining suitable grammars allows parsing the utterance and extracting the meaningful components. A grammar specifies the patterns of words accepted by the recognizer: these constraints simplify the related semantic interpretation but require specific knowledge in the design process; moreover, recognition can fail in case of unexpected input. Hence, an alternative data-driven approach is being investigated: a statistical learning procedure can be in principle more flexible and able to manage also unlikely requests. A discussion of the Speech Understanding approach followed within the DIRHA project can be found in Section 3.

The House State Keeper module is in charge of holding the Configuration of the specific house to be handled (i.e. the House Profile) and the specific State of each and every item contained in the House Profile (i.e. the House State), interacting with the House Automation...
system. A discussion of the House+User Keeping approach followed within the DIRHA project can be found in Section 5.

The **Prompt Generator** module is in charge of playing audio messages to the user; depending on the need they could be pre-recorded voice messages, voice messages generated by Text to Speech (TTS) system, or simple chimes. A brief discussion of the Prompt Generation approach followed within the DIRHA project can be found in Section 6; however, for the choice of each and every prompt/chime refer to Section 7, where one complete dialogue example is discussed.
2. DIRHA approach to Concurrent Dialogue Management

A dialogue management system is used to simulate the process of a dialogue. Dialogue modelling is necessary for any type of dialogue, be it text-based, speech input or using other modalities. In the age of user friendly interfaces, pleasant and easy interaction is an essential aspect of the design of any system. Dialogue systems have specific requirements for this, including adequate recovery from error. The Dialogue Manager should be able to identify errors and adopt a strategy which recovers the dialogue.

2.1 Introduction to Dialog Management

Dialogue management techniques are particularly beneficial to systems using speech recognition; spoken language dialogues require sophisticated modelling strategies, but these in turn can provide a level of constraint that can mitigate the shortcomings of speech recognition technology.

The Dialogue Manager is the heart of a spoken language system, as its main purpose is to guide the user to provide the needed information. In order to reach this goal it coordinates the activity of several components, controls the dialog flow, and communicates with external subsystems. The Dialogue Manager may exploit many techniques which include discourse analysis, knowledge database query, and system action prediction based on the discourse context.

Main roles of Dialog Management

In general, the DM accepts as input the best estimate of the user's request, represented as a semantic (multislot) frame produced by the Speech Recognition and Understanding modules, and outputs system responses together with operative commands to the executive subsystems. The system responses have to reflect the discourse context by maintaining the dialogue history. Although the roles of the DM may depend on the type of task where it is involved, its main roles include:

• Searching and providing query results by connecting to an external knowledge database, based on the current input and the discourse context
• Asking further slots of information to submit an appropriate query
• Requesting to confirm unclear information slots and/or to rephrase if the user's input is out-of-coverage.

Degrees of Initiative

A dialog consists of a sequence of user and system turns which usually depend on the discourse context. The process of dialog can be viewed as an exchange of information in which the initiative may shift between the user and the system. The term initiative is related to who directs the progression of the dialog. In general, the degrees of initiative in the spoken dialog system fall into one of the following strategies:

• System-initiative: The system has the initiative to guide the dialog at each step.
• User-initiative: The user takes a control of the dialogs, and the system responds to whatever the user directs.
Mixed-initiative: The system has overall control of the dialogs. However, the users can barge in and change the dialog direction.

In a system-initiative dialog, the system usually asks one or more questions to extract some slots from the user step by step. After enough slots are filled, it can submit an appropriate query to the external knowledge database.

In a user-initiative dialog, the user takes control of the dialog although the system may sometimes ask confirmation questions if some slots are unclear.

In a mixed-initiative dialog, the system is supposed to control dialog, but the user can have some flexibility at times to provide more information or to change the task.

A number of different approaches to the Dialogue management problem have been developed to date in the community. They can be classified into two main categories; knowledge-based dialog management, and data-driven dialog management.

Knowledge-based Dialog Management

Early dialog systems such as SUNDIAL [1] and ARISE [2] were designed by application developers who have domain-specific knowledge. These systems are usually confined to both highly structured tasks and system initiative dialogs, where a restricted and regularized language set can be expected. In this knowledge-based approach the state behaviour of the whole system is usually abstracted into some kind of “control flow diagram”, i.e. a “flow chart” (for the state-implicit approach, such as the VoiceXML [3]), or a “state chart” (for the state-explicit approach) or other formalisms suitable to express imperative behaviour.

A reusable, domain-independent dialog engine manages the conversation by executing the given dialog task specification, as in the case of the DIRHA Concurrent Dialog Manager. The Dialog Manager knows and updates the “state” of the dialogue according to input utterances, playing prompts and setting recognition contexts (e.g. grammars or language models) according to the current state. In this way, for each state the needed prompts and recognition contexts can be specified: the advantage is that the Speech Recognition and Understanding modules are requested to recognize against smaller vocabularies (e.g. containing the set of utterances that are allowed in that state only) and this potentially brings high accuracy; conversely, the disadvantage is that the dialog can become too constrained if the dialog flow does not allow users to easily “jump” to different parts of the dialog; this disadvantage can be overcome making the dialog definition more sophisticated, hence more complex.

This approach is often used for rapid prototyping of dialog systems for strong-typed interactions with clearly-defined structures and goals [4]. This approach has also been deployed in many practical applications because of its simplicity.

Data-Driven Dialog Management

More recently, the research community for Dialogue management has exploited the benefits of data driven approaches to Speech Recognition and Natural Language Understanding. Although a data-driven approach requires time consuming data annotation, the training is done automatically and requires little human supervision. In addition, new systems can be developed at only the cost of collecting new data for moving to a new domain; this requires less time and effort than the knowledge-based approach.
The behaviour of such systems is abstracted by a “data flow” or other formalism suitable to express declarative behaviour: a production system (either forward or backward chaining) is usually derived from the data flow network able to plan and execute the task of gathering all the needed information.

The advantage of such an approach is that the flow of dialog needs not to be designed in advance, being it automatically determined by the production system; the disadvantages are that the ASR must recognize larger vocabularies, being it exposed to the whole set of information for every turn of dialog, yielding potentially lower accuracies; moreover, the data flow network and the consequent production systems can become complex when the task to be accomplished becomes more articulated.

Practical deployment of Data-Driven based dialog systems has encountered several obstacles [5]. For example, the optimized policy may remove control from application developers and the refinement of the dialog control is difficult. These are serious problems because the developers should have the opportunity to easily control the dialog flow in practical systems. For example, there are some studies on how to mix traditional knowledge-based Dialogue Manager design with Data-Driven based DM to reflect domain dependent business rules and to reduce the large policy space [6] [7]. However, this approach still needs improvement before it can be applied to developing practical dialog systems.

Support for Multimodality

In more recent years, as voice only systems have been replaced by multimodal ones (e.g. systems able to react to both voice and other “modes”, such as keyboards, remote controls, gestures, …) the State Based approach has been more widely adopted for its versatility; this is the case of SALT by Microsoft, CIMA (adopted in DICIT) [28], etc.

Among such approach the maximal versatility is given by the “state explicit” approach. This is the approach adopted in DIRHA.

2.2 Design choices in the DIRHA Dialog Management

The DIRHA Dialog Manager will be state based and, precisely, state-explicit; as briefly discussed in the previous section this choice can potentially yield the highest accuracy in the voice-related processing, as it allows directing the ASR and possibly other DIRHA subsystems according to the particular state assumed the dialog over time.

Such a flexibility will play a crucial role in DIRHA, as more than one dialog could take place in the same house (i.e. one per each room) and potentially dialogs could interact one with the other.

As anticipated in the previous discussion, together with the advantage of versatility and accuracy, the state based approach has the disadvantage of potentially needing a complex state design to provide enough flexibility in the user experience: in order to help cope with the dialog complexity, a powerful formalism has been chosen to design the dialog State Machines: the Harel’s State Charts [29] defined in the late 70s: such a formalism has been adopted in the State Charts “graphic language” of OMT and UML and has been taken by W3C as the basis for the design of the SCXML language [30].

---

SCXML and MIA-XML based State Charts

The main advantages of State Charts over more basic state machines approaches are the following:

1. Allow concurrency (i.e. more threads of flow within a single state machine), with possibility of synchronization.
2. Include data modelling (i.e. allowing to completely define the behaviour of a system, not limited to the control part) and encapsulation of data, for modularity.
3. Allow state nesting, to improve the expressivity of the language (i.e. making the dialogue definitions more compact).

At the time of writing the SCXML specification is still in the state of Last Call Working Draft; however the last issue (Dec the 6th, 2012) will hopefully be finalized soon.

Due to the relative instability of the specification (the DIRHA implementation started around M1, that is the beginning of 2012), and also to the planned kind of usage (somehow different from the one envisioned by W3C), the DIRHA implementation does not fully cover the specification, being somehow lightweight in some aspects and adding some original features; however it covers the majority of the SCXML definition; for this reason we named it “MIA-XML”

Section 4 of this document discusses in detail the MIA-XML formalism and Appendix 1 will contain the reference manual; for the rest of the discussion here it’s enough to notice that the one dialogue specification, expressed in MIA-XML is parsed and interpreted by the MIA-XML executor which implements (i.e. runs) the dialogue state machine interacting with the rest of the system (i.e. its “ecosystem”).

Unlike other approaches, the MIA-XML executor is not tied to any particular voice of other technologies; this yields the maximal versatility in the kind of dialogues that can be implemented (i.e. multimodal) and in the run-time environments where it will be deployed; actually it is implemented as a C++ program, hence portable to virtually any computing platform: from Linux to Windows, to MAC OS, iOS, Android, etc.

The technology-agnosticism of the MIA-XML executor is compensated by the adoption of the most spread communication technology: TCP-IP and in particular HTTP; the state machine being executed can exchange (i.e. receive and transmit) “events” with attached data in form of POST payloads of HTTP to and from any node in the available IP network, including other instances of the MIA-XML executor running other dialogues.

The “events” are sent in asynchronous way, that is: a POST payload sent from node A to node B is acknowledged by B as soon as it is received; however this is not a feedback to the requested action; when a reply to the action is to be sent from B, a new event is sent by B to A in the same way; such a reply event is in turn acknowledged by A.

Acknowledges can be positive, with Error Code = 200 OK or negative, with some variants of 40x code.

Such a choice provides a great flexibility in the implementation of whatever dialogue: the only requirement is that the chosen technological engines are able to send or receive the “MIA-XML events” through a TCP-IP network; the choice is also fully coherent with the expected reaction times for a dialogue system: typical latencies within local TCP-IP networks are in the range of milliseconds; in the next phases of the project, some experiments could also be carried out geographically displacing some nodes of the system.
Figure 2-1: shows a generic example of “ecosystem” for a dialog system based on the MIA-XML executor; in the general case each and every module must implement both the http server and client roles, as it needs to both send and receive “events” (i.e. POST payloads); however the number of open channels shall be minimal.

For voice based systems one of the “event producers” will be the ASR+SU, while one of the “event consumers” will be the TTS or a pre-recorded prompt player.

**Possible Structure of Dialogue State Machines in DIRHA**

The MIA-XML executor will be interfaced at least to the ASR (encapsulating most of the rest of the DIRHA system, such as the Speech Understanding, Source Localization, Speaker Recognition, ….), the Prompt Generator (i.e. TTS of prompt player), Home Automation system; in the following, those subsystems will be referred to as the dialogue “ecosystem”.

The anatomy of the DIRHA dialogue running on the MIA-XML executor is one State Chart with several parallels threads of execution sharing the house configuration and actual state; in the following a possible set of parallel threads is reported:

1. one parallel thread of dialogue for each area of the house (i.e. a room, or a different space partitioning) where a separate dialogue can take place: the space partitioning will depend upon the Speech Localization algorithm being provided in DIRHA: speech input and output coming and going to/from the MIA-XML executor and its ecosystem will be tagged as belonging to a specific area (exceptions will be: unknown-source and all-destinations)

2. one coordination thread handling speech input with unknown source and other house-wide commands.

3. one thread updating the House state among the dialog and the real house: such a thread will update the dialog internal state of the house upon reception of update messages sent by the house automation system in case the state was changed outside the DIRHA
dialogue and will issue commands to the House automation system upon receiving a command through the dialogue.

Other attribution of parallel threads to physical entities are possible, such as attributing one parallel thread of dialogue to each user of the house, but this could be more difficult to achieve; the actual decision to be implemented will be taken when finalizing the design of the final DIRHA prototype.

Relationship among Dialog Manager, ASR and Speech Understanding

In the interaction between the (State Machine interpreted by the) Dialog Manager and the ASR which encapsulates the SU module, there is a master-slave relationship: the former keeps the current state of the dialogue and directs the latter towards a specific goal (sending a specific activation event) while the latter, once activated upon a specific goal, waits for a user utterance and aims to the requested goal. Once reached the requested goal, the ASR+SU returns a number of (predefined) semantic slots to the master, sending a specific recognition event.

The above sketched approach exploits the ability of the Dialog Manager to hold the current state of the dialog, hence activating the ASR+SU over a specific subset of the whole language domain (as an extreme example, when the dialog needs a simple “Yes/No” answer it would be useless-and harmful to accept a vocabulary larger than “Yes”, “No” and a few tens of synonyms.

The ability of the ASR+SU to take advantage of such an “advice” can lead to better accuracy in the recognition; with this respect, a short introduction to the difference among the two approaches to SU to be evaluated and compared in the first project phase are reported here; such techniques will be discussed in deeper detail in Section 3.

Grammar-based: the utterance is recognized and semantically parsed by the ASR+SU using one or more grammars according to the requested goal. Semantic slots could be defined so that parsing rules could fill them when the rule is applied. This approach has the disadvantage that each grammar must be designed by hand: furthermore, grammar based Understanding would never be able to recognize an utterance if it was not handled by a grammar; the advantage is that it fits well into the state-based approach as one set of grammars could be passed by Dialog Manager to the ASR+SU for any specific turn of dialog. Furthermore, a localization of the whole system for different languages would require the translation of the grammars from one language to the other, a relatively easy task.

In the following, a simple example of Yes/No grammar is reported.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<grammar version="1.0" xml:lang="en-GB" root="command">
  <rule id="command" scope="public">
    <one-of>
      <item> <ruleref uri="#yes"/> <tag>:var</tag> </item>
      <item> <ruleref uri="#no"/> <tag>:var</tag> </item>
    </one-of>
  </rule>
</grammar>
```
<item>correct</item>
<item>right</item>
<item>ok</item>
<item>confirm</item>
</one-of>
<tag>return("true")</tag>
</rule>
<rule id="no">
<one-of>
<item>no</item>
<item>incorrect</item>
<item>wrong</item>
<item>not ok</item>
</one-of>
<tag>return("false")</tag>
</rule>
</grammar>

Notice the semantic slot named “command” filled with the only two possible values (i.e. true and false) as side effect of parsing rules.

Data-driven: the utterance is recognized by the ASR using a large statistical Language Model. The recognized string is then interpreted by the Speech Understanding component that maps the word sequence into a frame-semantic representation of its meaning. The main advantage of this technique is that an utterance can be interpreted even if it was not originally contained in the training set or specifically covered by the handcrafted grammars. The disadvantage of this approach is that it requires a large amount of semantically annotated data; the localization of the system for another language would require to acquire and annotate a complete training set for the new language.

As the two approaches have both advantages and disadvantages the project plan to investigate both: for example, where the expected utterances will range over a narrow vocabulary (i.e. the yes/no case described above) the grammar approach will be used; when the vocabularies are larger (i.e. the initial menu, or the selection of a music song to be played) wider statistical Language Models could be chosen. From the general Dialog Management architecture and even design of the State Machine point of view, such a choice would be completely transparent.
3. Natural Language Understanding

The speech understanding component aims at extract the "meaning" from the recognized utterance. A well-studied approach is based on the use of semantic frames: it has been adopted in many spoken language processing tasks where various pieces of information need to be collected from the user; as such, a frame-based system is limited to a restricted domain and has a relatively small semantic space. This structure is usually modelled by templates represented by semantic frames, whose frame elements (or slots) identify the requested variables. The goal of a frame-based understanding system is to select the proper semantic frame for the incoming utterance and fill from the sequence of recognized words its slots with the actual values.

A popular solution is based on knowledge: defining suitable grammars allows parsing the utterance and extracting the meaningful components. This authoring process requires expertise and is usually expensive so alternative data-driven approaches are currently studied, where a statistical learning procedure can in principle provide a more flexible and powerful understanding. Indeed, the related field of natural language understanding mainly focus on understanding general domain written texts and the corresponding semantics is defined in a broader sense (e.g. using thematic roles). Hence, many constraints related to the applicative domain should be introduced; although this in principle simplifies the problem, when dealing with speech other variables should be taken into account such as recognitions errors, spontaneous speech phenomena (disfluencies and not well-formed expressions) and out-of-domain utterances. So, robustness is a major issue in speech understanding since the system should handle any input, isolating in the recognized string accepted by the grammar or the parser only the current concepts important for the given domain. At the same time, this generalization feature may introduce ambiguities and reduce the accuracy.

In the DIRHA project both approaches are pursued with the intention to compare the safer but limited use of grammars with the more generalized paradigm based on statistics. As better explained in the dedicated section, the work on the data-driven approach is considered as medium-term research activity and the resulting component will not be directly integrated in the intermediate prototype.

3.1 Grammar-based approach

The knowledge-based approach adopting semantic grammars requires the exact matching of input sentences to the designed rules: natural expressions or uncommon formulation cannot be properly handled by the recognizer, leading to errors that often change the semantic content of the hypothesis. As such, it is important to carefully design the grammars in order to deal with these possible phenomena: specific fillers are able to model these less predictable portions of the sentences, assuring the correct identification and classification of the relevant semantic content. For example it is possible to take into account courtesy forms (e.g. please) or synonyms (e.g. light, lamp, abat-jour). On the other hand, the resulting grammars cannot be extremely large because in this case recognition performance usually drops. A mixed solution could represent a feasible option. Indeed, an interesting feature of the FBK ASR system is the capability to handle recursive transition networks: the arcs in the network can be labelled not only by the words but also the names of other networks. These networks can
be compiled by mixing in an arbitrary manner grammars based on regular expressions and statistical language models. This feature allows the designer to model directly common expressions like “yes/no” as well as to handle more complex utterances thanks to statistical language models. At the same time, the items belonging to a class (e. g. a device) can be dynamically changed without a complete reload of the nets in the recognizer. Hence this flexibility is also linked to a high efficiency in terms of computational load. As a result the recognizer in the search path produces also the semantic parsing of the (best) string, identifying the slots and their content.

Hence the adoption of the FBK recognizer for the DIRHA prototypes will make easier the integration of the understanding component since the semantic slots are embedded as enriched representation in the output. A parser is then required to properly compose and fill the form for the Dialogue Manager, according to the exchange protocol under definition.

The preliminary evidence coming from the WOZ experiment indicates that the users tend to prefer very short commands with well-structured grammars: this suggests that it is likely that the grammar-based approach can provide a reliable framework for the considered domain. Moreover, due to the daily usage of the DIRHA system, users will probably interact with a restricted vocabulary, because of habits and learning effect.

The impact of recognition errors is currently under investigation; a specific set of grammars, according to the dialogue designed and described in the next sections, will be used to recognize and semantically parse the WOZ signals. A more general LM based on n-grams and trained on similar text material is considered as reference.

Integration between ASR and the Dialog Manager

A dialogue description can be directly associated to a set of recognition grammars which will be used by the speech recognizer, and which will contain some semantic labels that allow to establish a direct relation between the dialogue concepts and the sub-grammars activated during the recognition. From the speech recognition point of view, each concept is associated both to a sub-grammar and to a semantic label. Every sub-grammar can be recursively combined to form a bigger language model which will be used to recognize a complete sentence. The output of the recognizer is not only a sequence of words, but also includes information (basically the semantic labels) about the sub-grammars crossed during the decoding step, and thus is a parse tree of the sentence. The resulting association between the grammar identifier and a semantic concept provides directly an interpretation of the sentence, at least in the restricted domain under analysis.

For instance, if we consider a sample sentence for controlling the light: “please switch on the light in the living room” a properly combination of three sub-grammars (ACTION, CLASS, ATTRIBUTE) may allow the recognition of:

```
please (ACTION( switch on )ACTION) the (CLASS ( light ) CLASS) in the (LOCATION ( living room )LOCATION)
```

where the semantically relevant words can be extracted easily from the recognized string and immediately referred to the required action associated to the label ACTION in a specific room (associated to the label LOCATION).

In this way there is no need for a subsequent parser, because the relevant information is already labelled, and only some text processing is required to properly edit the output in the format expected by the DM.
The application designer can easily build a system-initiative dialogue, simply by associating to each concept a very strict grammar (e.g. a list of devices). On the other hand, a mixed initiative dialogue can be designed by defining recognition grammars (possibly just one) able to handle all of the concepts of the application. In case of multiple semantically relevant data into a sentence, the corresponding concepts can be activated simultaneously.

**Grammar design**

Although a speech recognizer is able to manage very large vocabularies, the noisy environment makes the recognition task extremely complex and can lead to very low performance. Hand-crafted grammars introduce strong constrains in the hypotheses generated by the recognizer, exploiting the designer knowledge of the domain.

In the Table 1 a minimal set of possible grammars (see §7 for details) is shown:

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Semantic Concept</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyword</td>
<td>dirha, aladdin, dirha system, activate</td>
<td></td>
</tr>
<tr>
<td>complete</td>
<td>class, attribute, action, location</td>
<td>open the door, close the window, switch on the light, close the door of the kitchen, open the small window, switch off the light in the living-room</td>
</tr>
<tr>
<td>cancel</td>
<td>confirm</td>
<td>cancel, delete</td>
</tr>
<tr>
<td>location</td>
<td>location</td>
<td>the door of the bathroom, the kitchen door</td>
</tr>
<tr>
<td>yesno</td>
<td>confirm</td>
<td>yes, no, that’s right</td>
</tr>
<tr>
<td>attribute</td>
<td>attribute</td>
<td>the red lamp, the small one</td>
</tr>
</tbody>
</table>

According to the dialog design, the required grammars should cover utterances comprising the relevant semantic concepts. In the following the main grammars are described and discussed using simple examples.

- **Activation keyword**: the grammar accepts the selected keyword (e.g. “Dirha listen to me”), triggering a dialogue session; the grammar rejects any other user sentence (KWS task). This modality represents the equivalent “push-to-talk” button in a typical spoken dialog system: the system is listening but not reacting to generic speech, only a predefined keyword actives a dialog session. The semantic concept is associated to the grammar identifiers KW and REJECT. REJECT is modeled either by a very general statistical language model or by an ad-hoc grammar (e.g. phone-loop).

| please listen to me | (REJECT( @rj )REJECT) |
**Location**: the grammar accepts a location in order to identify the object to be controlled; again, a filler can model parts of speech that are not semantically relevant. This modality is used to select the desired object in case the implicit strategy (the default item in the room specified by the user) cannot be applied. The semantic concept is associated to the content of the grammar LOCATION.

<table>
<thead>
<tr>
<th>Sentence</th>
<th>Grammar Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>the one of the kitchen</td>
<td>(FILLER( @rj )FILLER) (LOCATION( the kitchen ) LOCATION)</td>
</tr>
<tr>
<td>in the living-room</td>
<td>in (LOCATION( the living-room ) LOCATION)</td>
</tr>
</tbody>
</table>

**Confirmation**: the grammar handles confirmations and rejections. The dialog asks for an explicit confirmation in case of critical operation (e.g. opening of the main door) or low confidence (e.g. uncertainties in the acoustic front-end). The user can also decide to abandon the sessions or cancel the requested operation. In the example the grammar CONFIRM comprises the two sub-grammars YESNO and CANCEL.

<table>
<thead>
<tr>
<th>Sentence</th>
<th>Grammar Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>(CONFIRM( (YESNO( yes )YESNO )CONFIRM)</td>
</tr>
<tr>
<td>cancel</td>
<td>(CONFIRM( (CANCEL( cancel )CANCEL) )CONFIRM)</td>
</tr>
</tbody>
</table>

**Attribute**: this grammar is used to specify or identify an object. In some cases it is required to identify the device to be controlled again in case of ambiguous requests. Here the relevant content is associated to the grammar ATTRIBUTE. FILLER represents an auxiliary grammar able to match other parts of the sentence.

<table>
<thead>
<tr>
<th>Sentence</th>
<th>Grammar Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>the small window</td>
<td>(FILLER( the )FILLER) (ATTRIBUTE( small )ATTRIBUTE) (FILLER( window )FILLER)</td>
</tr>
<tr>
<td>the red one</td>
<td>(FILLER( the )FILLER) (ATTRIBUTE( red )ATTRIBUTE) (FILLER( one )FILLER)</td>
</tr>
</tbody>
</table>

**Complete command**: the grammar accepts a complete command sentence in which the semantic concepts (class, action, location, attribute) are directly modeled by specific sub-grammars. The general request contains all these relevant semantic concepts, encapsulated by the grammars ACTION, CLASS, LOCATION. The optional value of ATTRIBUTE can help the correct identification of the device.

<table>
<thead>
<tr>
<th>Sentence</th>
<th>Grammar Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>open the door</td>
<td>(ACTION( open )ACTION) (CLASS( the door )CLASS)</td>
</tr>
<tr>
<td>switch on the red light in the kitchen</td>
<td>(ACTION( switch on )ACTION) the (ATTRIBUTE( red )ATTRIBUTE) (CLASS( light )CLASS) in (LOCATION( the kitchen )LOCATION)</td>
</tr>
</tbody>
</table>
These basic grammars can then also combined with other more general expressions or parallel statistical language models in order to handle more natural expressions. The final design will take into account the actual setup of the automatized home.

**Evaluation**

Usually performance is measured as Word Error Rate but a dialog system that uses grammars to determining the requested actions the possible ASR errors have different impact: words belonging to the embedded grammars are more relevant. From the dialog point of view a decoded sentence is correct if the corresponding interpretation matches the actual intention of the user. Therefore, beside WER, it is important to evaluate grammars using precision and recall. As a result, the reference set has to be labelled with semantic tags in order to measure the capability of the grammars to correctly (recognize and) classify the meaningful information of the sentence. The transcription and semantic annotation of the WOZ data is the basis for a first evaluation of the designed grammars and the comparison benchmark for the investigated approaches.

**3.2 Data-driven approach**

This alternative component takes as input an utterance and returns as output a structured interpretation that allow the Dialog Manager to fulfill the user request, or an exception if the request cannot be interpreted.

To this aim, a solution based on semantic parsing of users’ requests has been designed and implemented: a preliminary comparison with a traditional solution based on contextual grammars has also been carried out.

A semantic parser maps a natural-language sentence into a formal representation of its meaning. We use semantic role labeling (SRL), a widely used form of semantic representation which identifies roles such as agent, patient, source, and destination. Specifically, the implemented semantic parser transforms an Italian sentence into a frame-semantic representation based on FrameNet. FrameNet is a lexical resource that groups predicates in a hierarchy of structured concepts, known as “frames.” Each frame in the lexicon in turn defines several named “roles” corresponding to aspects of that concept, e.g., participants in an event. Our parser extends Semafor\(^2\), an open source tool for automatic analysis of the frame-semantic structure of English text developed at CMU. Semafor uses WordNet\(^3\) and FrameNet\(^4\) as lexical and semantic resources.

During the first year of the project, we have focused on the implementation of the semantic parser. Specifically, the following activities have been carried out. First, the tool and models made available by the Semafor developers have been tested on English and the declared performance has been confirmed on some dataset available from the SemEval 2007 and 2010 evaluation campaigns. Second, all the language-dependent parts present in the original source code have been replaced with calls to MultiWordNet\(^5\), a resource that includes the Italian WordNet, TextPro\(^6\), a suite of modular Natural Language Processing (NLP) tools for analysis.

---

\(^2\) [http://www.ark.cs.cmu/Semafor](http://www.ark.cs.cmu/Semafor)
\(^3\) [http://wordnet.princeton.edu/](http://wordnet.princeton.edu/)
\(^4\) [http://framenet.icsi.berkeley.edu](http://framenet.icsi.berkeley.edu)
\(^5\) [http://multiwordnet.fbk.eu](http://multiwordnet.fbk.eu)
\(^6\) [http://texpro.fbk.eu](http://texpro.fbk.eu)
of Italian and English texts, and the MaltParser, a dependency parser trained on the CCG-TUT, a treebank for Italian based on Combinatory Categorial Grammar. Third, we used the Evalita 2011 dataset to train and test Semafor on Italian data.

In addition, we performed an evaluation on sentences processed by the FBK ASR system in order to be able to handle input with word errors and sentence segmentation errors. Finally, we are extending the available training data to include sentences used in the DIRHA use cases; to this purpose we have developed a specific annotation platform. In the remainder of the section, we first describe the FrameNet and Semafor, then the porting to Italian and its evaluation. Finally, we present the related work and the open issues.

FrameNet

FrameNet [8, 16] is a lexical resource for English, based on frame semantics [15], that is being created in the context of the Berkeley FrameNet project. Its aim is to collect the range of semantic and syntactic combinatorial possibilities of each word in each of its senses through the annotation of example sentences. The conceptual model is based on three main elements:

- **Semantic frames**: Cognitive schemata or scenarios necessary to understand the meaning of words. They describe situations, objects and events and the participants involved in them (in our case the DIRHA system is the implicit agent).
- **Lexical units (LUs)**: Words, multiwords, idiomatic expressions evoking a frame.
- **Frame elements (FEs)**: Semantic roles involved in the situation or event expressed by a frame. They apply to all LUs in the same frame.

FrameNet 1.3, released in 2006, is comprised of more than 10,000 lexical units, 6,000 of which are fully annotated, and nearly 800 semantic frames with hierarchical relations. An essential element of the FrameNet database is the corpus-based evidence, i.e., every lexical has to be instantiated by at least one example sentence. In FrameNet 1.3, more than 135,000 sentences have been manually annotated with frame information.

As an example, we report in Table 3.2 the FrameNet entry for the WEARING frame.

<table>
<thead>
<tr>
<th>Frame: WEARING</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Def.</strong> The words in this frame refer to what CLOTHING a WEARER (or a specific BODY_PART of the WEARER) has on.</td>
</tr>
<tr>
<td><strong>FEs.</strong></td>
</tr>
<tr>
<td>BODY_PART The body part of the WEARER which is covered by the CLOTHING.</td>
</tr>
<tr>
<td>WEARER The person whose clothing is under discussion.</td>
</tr>
<tr>
<td><strong>LUs.</strong> attired.a, bare-armed.a, bare-breasted.a, bare.a, braless.a, clothed.a, courtless.a, costumed.a, decked out.a, dressed.a, have got on.v, sport.v, swaddled.a, swathed.a, wear.v [...]</td>
</tr>
<tr>
<td><strong>Ex.</strong> [The leader] wore [a golden helmet] costing.</td>
</tr>
<tr>
<td>She saw that [her] bare [left hand] lost its covering was bare.</td>
</tr>
<tr>
<td>[She] had [an apron] costing on.</td>
</tr>
</tbody>
</table>

Table 3.2 Example of frame “wearing”

In the first row, the frame definition in natural language is reported, while the second includes the list of the core frame elements. The third row contains part of the LU list.
including all frame-evoking predicates, while in the fourth a few example sentences are reported. All LUs are printed in bold, while the phrases bearing a FE label are reported between square brackets, followed by the role label.

In the remainder of this article, we call frame semantic annotation the annotation of sentences with both frame and FE (or role) information, as performed by frame-semantic parsers (e.g. [10] and [12]). The sub-task of assigning a frame label to a lexical unit in a sentence is called frame identification. This concerns both lexical units that are listed in FrameNet, the so-called seen LUs, and those that are not present in the resource, the unseen LUs. When frame identification is applied to unseen LUs, and leads to the acquisition of new LUs, it is also known as LU induction [23].

The second resource we take into account in this work is Wikipedia, the largest online repository of encyclopedic knowledge. At the moment of writing, there are 20 million articles in 282 languages (over 3.82 million in English alone) written collaboratively by approximately 100,000 regularly active contributors around the world. This makes Wikipedia a reliable source of knowledge both for Internet users and researchers.

Semafor

Semafor [13, 14] is a state-of-the-art open source Java application developed at CMU that transforms an English sentence into a frame-semantic representation in a three-step process. First, Semafor identifies words that evoke FrameNet frames, second, selects frames for them, and, finally, locates the arguments for each frame. The frame-semantic parsing is cast as a structure prediction problem. The system uses two feature-based, discriminative probabilistic (log-linear) models, one with latent variables to permit disambiguation of new predicate words. The parser is demonstrated to significantly outperform previously published results and is released for public use under the GPL license.

Semafor preprocesses sentences with a standard set of annotations: POS tags from MXPOST [24] and dependency parses from the MST parser [22] since manual syntactic parses are not available for most of the FrameNet-annotated documents.

Semafor used WordNet for lemmatization and labeled each verb in the data as having ACTIVE or PASSIVE voice, using code from the SRL system described by Johansson and Nugues [19].

The probabilistic models have been trained and tested on SemEval’07 data. The system improves the state of the art at each stage of processing, e.g., frame prediction, boundary identification, and argument classification.

Our investigation started replacing the MST parser with the Stanford Parser [15], this drastically reduced the memory footprint of the system without significantly changing the performance on English annotation. This is the only change we made to the English version of the tool, from now on, we focus on the porting to Italian.

Semafor for Italian

Porting the system to Italian is a challenging task due to several reasons. First, some of the resources employed do not have a counterpart in Italian, or they are not as rich as in English. For example, the Evalita dataset is the only available training set for Italian. This training set is quite limited if compared with the amount of annotated sentences available on FrameNet.
Second, we had to replace the whole preprocessing pipeline. This point has a strong impact as Semafor uses a set of heuristics based on the English grammar and the output of the English preprocessing. These heuristics have been partially rewritten according to the Italian grammar and the output of the Italian preprocessing.

Finally, the available software is monolithic, difficult to understand, and expensive to modify. For example, many part of the code are duplicates, changing just few details. A software reengineering process would be required, however for the moment we limited our action to modify isolated part of the code.

**TextPro**

The Italian preprocessing is performed using TextPro, a state-of-the-art suite of modular NLP tools for analysis of Italian and English texts. All tools have been designed so as to integrate and reuse state of the art NLP components developed by researchers at FBK. TextPro is a pipeline of processors wherein each stage accepts data from an initial input or from an output of a previous stage, executes a specific task, and sends the resulting data to the next stage, or to the output of the pipeline. The current version of the tool suite provides functions ranging from tokenization to chunking and Named Entity Recognition. Specifically, we use tokenization, lemmatization, and part-of-speech tagging.

In addition, we created some heuristics rules, for example, to detect active and passive verbs by considering a list of transitive/intransitive verbs. If the auxiliary verb is to be and the verb is intransitive then we classify the verb as passive; otherwise active.

**Malt Parser**

The parser used is the MaltParser\(^7\), a tool for data-driven dependency parsing that can be used to induce a parsing model from treebank data and to parse new data using the induced model. Malt-Parser was one of the top performing systems in the multilingual track of the CoNLL shared tasks on dependency parsing in 2006 and 2007. In this project, we used the parser trained on the CCG-TUT\(^8\), a treebank for Italian based on Combinatory Categorial Grammar [20].

**WordNet**

Semafor uses JWNL (Java WordNet Library) to access the English WordNet\(^9\). In our porting, we use MultiWordNet, a multilingual lexical database developed by researchers at FBK in which the Italian WordNet is strictly aligned with Princeton WordNet 1.6.

The Italian synsets are created in correspondence with the Princeton WordNet synsets, whenever possible, and semantic relations are imported from the corresponding English synsets. However, MultiWordNet is made available as MySql dump, this required the conversion of the dump in a format compatible with the input format of JWNL. The conversion scripts are made available in the MultiWordNet distribution, MultiWordNet is licensed under a Creative Commons Attribution 3.0 Unported License.

\(^7\)[http://www.maltparser.org/](http://www.maltparser.org/)
\(^8\)[http://www.di.unito.it/~tutreeb/](http://www.di.unito.it/~tutreeb/)
Evaluation

The evaluation has been performed on the FLaIT dataset, used at the Evalita 2011 Frame Labeling over Italian Texts Task [9]. The training data made available by the task organizers consists in the merging of two independently created datasets. The first has been annotated by Tonelli at FBK [26]. It includes the annotation of 605 sentences (605 predicates and 1074 roles) at the syntactic and semantic level under the XML Tiger format also used by the Salsa project, where the reference syntactic formalism of the annotation is derived by a constituency-based parser.

The second dataset has been developed at the ILC in Pisa by Lenci and his colleagues[21]. It consists of the ISST-TANL Corpus, a dependency-annotated corpus originating as a revision of a subset of the Italian Syntactic-Semantic Treebank or ISST, enriched with Semantic Frames under the XML Tiger format also used by the Salsa project. The whole corpus contains 650 sentences with 1763 roles. The resulting training set thus includes 1255 sentences for about 38 frames. The total amount of roles completely annotated correspond to 2837 arguments. The test set has been obtained through the exploitation of the aligned English-Italian Europarl section [10]. It consists of 318 sentences, again focusing on 36 of the training set frames, for a total of 318 targets and 560 other arguments.

The evaluation is split in subtasks. Frame Detection (FD) aims at verifying the ability in recognizing the true frame of an occurring predicate word, and to select it even against possibly ambiguous lexical units. Boundary Detection (BD) and Argument Classification (AC) require to locate and annotate all the semantic arguments of a frame, which are explicitly realized in a sentence, given the marked lexical unit.

Tables 3, 4, and 5 show the results obtained on the FP, BD, and AC, respectively, by our system (Semafor IT) and the 2 participants at the shared task (CELI and University of Roma, Tor Vergata).

<table>
<thead>
<tr>
<th>Systems</th>
<th>Semafor.IT</th>
<th>CELI_NT</th>
<th>CELI_WT</th>
<th>TV_SVM-SPTK</th>
<th>TV_SVM-HMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Precision</td>
<td>69.50%</td>
<td>73.93%</td>
<td>73.93%</td>
<td>80.82%</td>
<td>78.62%</td>
</tr>
<tr>
<td>Frame Recall</td>
<td>69.50%</td>
<td>65.09%</td>
<td>65.09%</td>
<td>80.82%</td>
<td>78.62%</td>
</tr>
<tr>
<td>Frame F1</td>
<td>69.50%</td>
<td>69.23%</td>
<td>69.23%</td>
<td>80.82%</td>
<td>78.62%</td>
</tr>
</tbody>
</table>

Table 3.3 Results of the Frame Detection task

<table>
<thead>
<tr>
<th>Systems</th>
<th>Semafor.IT</th>
<th>CELI_NT</th>
<th>CELI_WT</th>
<th>TV_SVM-SPTK</th>
<th>TV_SVM-HMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>BD Prec.</td>
<td>57.27%</td>
<td>45.88%</td>
<td>40.66%</td>
<td>66.67%</td>
<td>50.70%</td>
</tr>
<tr>
<td>BD Rec.</td>
<td>34.46%</td>
<td>28.71%</td>
<td>30.27%</td>
<td>69.46%</td>
<td>51.06%</td>
</tr>
<tr>
<td>BD F1</td>
<td>43.03%</td>
<td>20.89%</td>
<td>24.11%</td>
<td>72.50%</td>
<td>51.43%</td>
</tr>
<tr>
<td>BD Token Prec.</td>
<td>79.46%</td>
<td>81.12%</td>
<td>78.67%</td>
<td>81.99%</td>
<td>68.02%</td>
</tr>
<tr>
<td>BD Token Rec</td>
<td>54.61%</td>
<td>27.06%</td>
<td>33.28%</td>
<td>84.34%</td>
<td>77.18%</td>
</tr>
<tr>
<td>BD Token F1</td>
<td>64.73%</td>
<td>40.58%</td>
<td>46.77%</td>
<td>83.15%</td>
<td>72.31%</td>
</tr>
</tbody>
</table>

Table 3.4 Results of the Boundary Detection (BD) task
BD token and AC token results account for the number of individual tokens correctly classified instead of the number of exact arguments.

These results show that the Semafor can be ported to Italian and the results are comparable with the state of the art. However, the accuracy still needs to be improved.

We also extended the evaluation to be nearer to the DIRHA scenario, in which the sentences to parse can present errors due to ASR. To this aim, we recorded some volunteers reading a subset of the Evalita dataset.

The result consists in 189 sentences out of 318: the read speech has been then recognized using a generic ASR trained on Parliament speech (see [27,28] for details) in order to introduce typical recognition errors and evaluate the robustness of the forthcoming parser. The resulting WER on this set is about 30%: on purpose the ASR system has not been tuned in order to generate a large number of errors.

Table 3.5 Results of the Argument Classification (AC) task

<table>
<thead>
<tr>
<th>Systems</th>
<th>Semafor.IT</th>
<th>CELI NT</th>
<th>CELI WT</th>
<th>TV.SVM-SPTK</th>
<th>TV.SVM-HMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC Prec.</td>
<td>38.87%</td>
<td>32.55%</td>
<td>27.41%</td>
<td>48.44%</td>
<td>33.10%</td>
</tr>
<tr>
<td>AC Rec.</td>
<td>23.39%</td>
<td>14.82%</td>
<td>16.25%</td>
<td>52.68%</td>
<td>33.57%</td>
</tr>
<tr>
<td>AC F1</td>
<td>29.21%</td>
<td>20.37%</td>
<td>20.40%</td>
<td>50.47%</td>
<td>33.33%</td>
</tr>
<tr>
<td>AC Token Prec.</td>
<td>54.67%</td>
<td>47.90%</td>
<td>49.49%</td>
<td>62.58%</td>
<td>46.77%</td>
</tr>
<tr>
<td>AC Token Rec</td>
<td>37.57%</td>
<td>15.98%</td>
<td>20.93%</td>
<td>64.38%</td>
<td>53.06%</td>
</tr>
<tr>
<td>AC Token F1</td>
<td>44.53%</td>
<td>23.96%</td>
<td>29.42%</td>
<td>63.47%</td>
<td>49.72%</td>
</tr>
</tbody>
</table>

Table 3.6 Results of the Frame Detection task working on the output of ASR

<table>
<thead>
<tr>
<th>System</th>
<th>ASR+Semafor.IT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Precision</td>
<td>53.96%</td>
</tr>
<tr>
<td>Frame Recall</td>
<td>53.96%</td>
</tr>
<tr>
<td>Frame F1</td>
<td>53.96%</td>
</tr>
</tbody>
</table>

Table 3.7 Results of the Argument Classification task working on the output of ASR

This evaluation cannot be performed using the official scorer as it is based on the assumption that gold and system answers are aligned at token level, assumption clearly violated in the output of a ASR system due to errors and missing punctuation marks. Consequently, we had to rewrite a scorer that could keep into account all these problems. The new scorer does not evaluate BD as text is not aligned at level of token, so we can only evaluate FD and AC. These results are not available for the CELI and Tor Vergata systems. Table 6 and 7 show the FD and AC results, respectively. These results correspond to the exact match, it is difficult to obtain the token AC results due to the errors and misalignment issues introduced by the ASR.
Details on the Speech Understanding Showcase

The first prototype is a Java application that integrates what described above. The application can be accessed through a RESTful API and a command line interface. The input/output format can be either XML or JSON. The main method takes as input a sentence and returns as output the semantic representation of the sentence, namely, the recognized frames, the evoking terms, and the roles involved.

![Semantic parsing for an Italian DIRHA-like sentence](image)

**Figure 3-1**: Semantic parsing for an Italian DIRHA-like sentence

Figure 3-1 shows an example for the input sentence “accendi il forno per 10 minuti a 200 gradi” (turn the oven on at 200 degrees for 10 minutes). In the example, the system recognizes that the term “accendi” (turn on) evokes the frame Change operational state, that requires an Agent, the entity who changes the operational state of a Device (the oven) and the Place (not found) in which the device is put into or out of operation. In the given command, the Agent (usually the user) and Place (the kitchen) are implicit and can be recognized by other modules of the architecture, or with simple reasoning based on the output of the other roles and using a description of the environment. For instance, knowing that the oven is placed in the kitchen, we can recognize the role Place. In this case, Non-core roles Degree (200 degrees) and Time (10 minutes) are necessary to execute the command. The system returns only the roles for which it has high confidence, if a compulsory role to execute the command is not found in the input sentence the system could explicit ask the user to specify it.
FrameNet Annotation for Italian

chiama la cucina. [Abandonment] [Selezione] ✤ Annota

accendi il clima. [Abounding_with] [Selezione] ✤ Annota

fai partire l'aria condizionata a 16 gradi. [Abandonment] [Selezione] ✤ Annota

attiva il video del citofono. [Selezione] ✤ Annota

ci sono messaggi. [Selezione] ✤ Annota

accendi il lampadario. [Selezione] ✤ Annota

Figure 3-2: The annotation interface that allows users to annotate frames

The Annotation tool

In order to annotate utterances collected within the Dirha use cases we developed a specific tool.

The annotation tool is a PHP application that allows users to annotate new sentences according to FrameNet 1.5. Sentences are first loaded into a database (MySql) and then made available for annotation through a Web-based interface. The user must follow a two-step annotation process. First, they are asked to select the frame evoked in the considered sentence. Second, they have to specify the lexical unit that evoked the frame selected in the first step and to annotate all the explicit semantic roles by selecting the token(s) and assigning the appropriate frame’s roles. For example, Figure 3-2 shows a set of sentences, from a combo box the user can select the frame. In the example, the first 3 sentences have been already annotated; the assigned frame label is shown after the sentence between squared parentheses. The lexical unit is underlined and the different frame’s roles are highlighted with different colours. Figure 3-3 shows a sentence annotated with the lexical unit and semantic roles. Words to be annotated are selected through a check button shown below the words, and the role is selected by means of a combo box shown below the words that lists all possible frame roles. If the annotation must be extended to the following word the users can simply click the arrow button and the previous annotation is extended. The annotations are saved into the database and can be converted to different annotation format (e.g., XML Tiger format).
Related Work

First, we consider the 2 participants to the Evalita task. CELI parses the input sentences with a legacy parser [25] that uses a combination of dependency based rules (e.g., subcategorization patterns) and machine learning techniques, based on Markov Logic Networks. Two systems are presented. CELI WT uses a set of hand coded rules for SRL, while CELI NT only relies on learned rules. University of Roma, Tor Vergata proposed two kernel-based systems that use SVM as learning algorithm. Specifically, TV SVM-SPTK extends the standard tree kernels formulation by embedding a corpus-driven lexical similarity metrics between terminal nodes (i.e. words in the leaves). TV SVMHMM combines discriminative and generative models. It cast BD and AC in a labelling task, without counting on any information about grammatical dependencies and the parse tree. SRL has been also used in the context of Spoken Dialog Systems within the project Luna, in which a machine learning approach based on frame semantics obtained successful results [11]. Here, a FrameNet-based parser both for English written texts and for Italian dialog utterances has been designed and evaluated. The results show that errors on dialog data do not severely hurt performance.

Also, a small set of FrameNet-like manual annotations is enough for realizing accurate Semantic Role Labelling on the target domains of typical Dialog Systems. [16] present an extension of the standard evaluation metrics for SRL in order to be able to handle speech recognition output with word errors and sentence segmentation errors. They propose metrics based on word alignments and bags of relations, and compare their results on the output of several SRL systems on broadcast news and conversations of the OntoNotes corpus. They evaluate the relation between the results on the subtasks that lead to SRL, including ASR, part-of-speech tagging or sentence segmentation. The analysis of the performance of retrained systems shows that the errors of at different levels (i.e., part-of-speech tagging, dependency parsing and SRL) are strongly correlated. They conclude that errors are due to the fact that systems are trained on reference data and suggest that one possible solution for improving SRL on speech could be to retrain systems on ASR output or modify them to process word lattices.
Discussion on preliminary results

Besides the standard approach based on hand-crafted grammars, we have developed a Semantic Role Labelling system for Italian starting from a pool of existing resources and tools for syntactic and semantic analysis of English and Italian. In spite of the complexity of the task, we have implemented a system that can be compared with the state of the art: specifically, our system if compared with the participants at Evalita 2011 scores second. We have also released a spoken version of the Evalita dataset and a tool for SRL annotation. However, several issues have to be addressed to reach the results obtained by the best system.

We think that the main weakness is the performance of the parser that strongly influences the boundary detection and consequently propagates the errors up to the argument classification phase. Typically, these issues are alleviated by long list of heuristics. We limit these processes to supply information not returned by the parser, not to fix frequent mistakes. All these problems are further amplified when we work on the output of the ASR system. Indeed we also have some errors at word level and, hence, it is more difficult to understand the causes of the misclassifications. In order to improve the performance we need to run an error analysis on the system with and without ASR. However, since the DIRHA domain is more limited, we are confident to have a good coverage in terms of language model (either with handcrafted grammars or statistical) so it is expected to reduce the parser errors.

Finally, in the targeted scenario the end-user is exposed continuously to the system and probably will tend to adapt his/her interaction, progressively selecting only a few ways to reach the goal in a minimal number of turns.

Furthermore, since houses are different from each other (see Section 5 for the discussion on House+User profile), there is the need to adapt the recognition/understanding domain accordingly. To this purpose, the grammar based approach seems able to accommodate this requirement in an easier way, due to its higher modularity.

On the other hand the alternative approach based on SRL generalizes better and in the long-term can guarantee a good coverage of the current list of services without requiring an explicit profiling procedure. A comparison carried out on real interactions in a sufficiently long period will clarify the reasonable trade-off between performance and efforts in the user profiling/customization.
4. Design of the State Based Concurrent Dialog Manager

The Concurrent Dialog Manager implemented within the DIRHA project is an interpreter of concurrent State Charts expressed with the MIA-XML language, a specialization of XML. Such an interpreter is being developed as a C++ program (Version 1.0 has been released at M12 and is being used in the Dialog Showcase) initially running on the Windows OS; however it is easily portable to other OS, such as Linux, the chosen integration OS for DIRHA.

The interpreter, in the following referred to as the MIA-XML executor works in 3 passes:

1. parsing of the input MIA-XML specification: during this pass the language syntax is checked and the internal data structures are loaded.
2. semantic (post-parse) analysis: during this pass the internal data structures are cross-checked in order to verify the semantic consistence of the State Chart specification; as an example, for every transition, the “target” state is searched for; in case it is not defined, a semantic error is flagged; same occurs for expressions: each and every used variable must be defined in the data model; one important by product of this analysis is that each and every element contained in the specification (i.e. states, transitions, variables, events, …) are directly indexed by the element that uses it; in this way the execution of the state machine will be very efficient, as if the state machine was compiled, instead of interpreted.
3. execution; this pass is iterated over and over again until the program is stopped; at each iteration the actual Configuration Set (i.e. the set of active states) is evaluated and the future Configuration Set is calculated, to prepare the next iteration. A major figure of merit for this kind of interpreters is the iteration time. i.e. the time taken to evaluate the Configuration Set; thanks to the heavy indexing performed in pass 2 the DIRHA MIA-XML executor is capable of running 1k to 10K iterations per second on complex to medium complexity state charts on laptop-desktop PCs running Windows; furthermore, such a figure grows linearly with the number of states in the Configuration Set, with no dependency upon the total number of steps or other sizes in the State Machine.

4.1 Introduction to the MIA-XML language

With the term MIA-XML we denote the subset of the SCXML language supported by the Concurrent Dialog Manager implemented within the DIRHA project.

The SCXML language is being defined by the W3C Consortium [30]; as of end of 2012 it is now in its Dorking Draft stage and the Last Call for changes has just deadlined in Jan the 13th 2013.

The MIA-XML language is mainly obtained by subtraction of a few elements from the SCXML language; however some additions have also been done; such changes have been introduced at the purpose of needing a lighter weighted executor, suitable to be ported to lower power devices.

In the following, the MIA-XML language is introduced; part of the concepts come from the SCXML material, released by W3C (see [30]); however, since differences have been
introduced in several places, a full description of the MIA-XML language is reported, for sake of completeness.

The most basic state machine concepts are State, Transition and Event. Each state contains a set of transitions that define how it reacts to events. Events can be generated by the state machine itself or by external entities. In a traditional state machine, the machine is always in a single state. This state is called the active state. When an event occurs, the state machine checks the transitions that are defined in the active state. If it finds one that matches the event, it moves from the active state to the state specified by the transition (called the "target" of the transition.) Thus the target state becomes the new active state.

The Harel [5] state notation defines several extensions to these basic notions. First of all, the state machine may take actions (Executable Contents) while taking transitions. Specifically, each state may contain both ‘onentry’ and ‘onexit’ actions.

Transitions may also contain actions. If a state machine takes transition T from state s1 to state s2, it first performs the onexit actions in s1, then the actions in T, then the onentry actions in s2. Secondly, in addition to the ‘event’ attribute that specifies the event(s) that can trigger it, transitions also have a ‘cond’ attribute. If a transition has both ‘event’ and ‘cond’ attributes, it will be selected only if an event is raised whose name matches the ‘event’ attribute and the ‘cond’ condition evaluates to true. If the ‘event’ attribute is missing, the transition is taken whenever the ‘cond’ evaluates to true. If more than one transition matches, the first one in document order will be taken. Thus, in the following example, the system will transition to st1 when event evt1 occurs if x is equal to 1, but will transition to s2 if event e occurs and x is not equal to 1; finally it will go to s3 if any other event occurs.

```xml
<state id="st0">
  <transition event="evt1" cond="x==1" target="st1"/>
  <transition event="evt1" target="st2"/>
  <transition event="*" target="st3"/>
</state>
```

**Compound States**

One of the most powerful concepts in Harel notation is the idea that states may have their internal structure. In particular, a <state> element may contain nested <state> elements. Such a state is called a compound state (called the parent state), while the nested elements are child states. Children states may in turn have nested children and the nesting may proceed to any depth. At the end of this nesting structure we will reach a state which does not contain any child states: such a state is called an atomic state. When a compound state is active, one and only one of its child states is active. Conversely, when a child state is active, all its parent states must be active too. Thus at any point we have a set of active states, containing an atomic state and all of its ancestors. (We will see in the “Parallel States” section that multiple atomic states can be active at the same time).

Compound states also affect how transitions are selected. When looking for transitions, the state machine first looks in the most deeply nested active state(s), i.e., in the atomic state(s). If no transitions match in the atomic state, the state machine will look in its parent state, then in the parent's parent, etc. Thus transitions in ancestor states serve as defaults that will be taken if no transition matches in a descendant state. If no transition matches in any state, the event is discarded.
Parallel States

The `<parallel>` element represents a state whose children execute in parallel. Like `<state>`, the `<parallel>` element contains `<onentry>`, `<onexit>`, `<transition>`, and `<state>` or `<parallel>` children. However, the semantics of `<parallel>` is different. When a `<state>` is active, exactly one of its children is active. When a `<parallel>` element is active, all of its children are active at the same time. Specifically, when the state machine enters the parent `<parallel>` state, it also enters each child state. The children states execute in parallel in the sense that any event that is processed is processed in each child state independently, and each child state may take a different transition in response to the event, including ignoring it.

Transitions within each individual child element operate normally. However whenever a transition is taken with a target outside the `<parallel>` element, the `<parallel>` element and all of its child elements are exited and the corresponding `<onexit>` handlers are executed. The handlers for the child elements execute first, in document order, followed by those of the parent `<parallel>` element, followed by an action expression in the `<transition>` element, and then the `<onentry>` handlers in the "target" state.

On the other hand a `<parallel>` element is exited if and only if all of its child states are in a final state, as illustrated in the following example: parallel state 'Par' has two children S1 and S2. Suppose a transition takes S1's child S12 as a target. Upon this transition, the state machine, in addition to entering S1 and S12, will also enter S1 parallel sibling S2 and its initial state S21. Once the transition has been taken, Par, S1, S2, S12, and S2Ini will all be active. If event 'e1' occurs, it will cause S12 to transition to S1Final, and S2Ini to transition to S22. At this point, S1 is in a final state, but S2 is still active. Now suppose event 'e2' occurs. This will cause S22 to transition to S2Fin. Now, since all of Par children are now in final states the entire Par region is exited.

```xml
<parallel id="Par">
  <transition event="evtent1" target="someOtherState"/>
  <state id="S1" initial="S1Ini">
    <state id="S1Ini">
      <transition event="e4" target="S12"/>
    </state>
    <state id="S12">
      <transition event="e1" target="S1Final"/>
    </state>
  </state>
  <final id="S1Fin"/>
</state>
  <state id="S2" initial="S2Ini">
    <state id="S2Ini">
      <transition event="e1" target="S22"/>
    </state>
    <state id="S22">
      <transition event="e2" target="S2Fin"/>
    </state>
  </state>
  <final id="S2Fin"/>
</parallel>
```

Note that the semantics of the `<parallel>` does mean that it must be implemented via multiple threads or truly concurrent processing: the children of `<parallel>` execute in parallel in the
sense that they are all simultaneously active and each one independently selects transitions for any event that is received. Hence, the parallel children process the event in a defined, serial (i.e. document) order, so no conflicts or race conditions can occur.

**Initial States**

In the presence of compound states, transitions no longer simply move from the current active state to a new active state, but from one set of active states to another. If the target of a transition is an atomic state, the state machine will enter not only the atomic state, but also any of its ancestor states that are not already active. Conversely, a transition may take a compound state as its target; in such a case one of the compound state's children must also become active, but the transition does not specify which one. In this case we look at the target state's `<initial>` child that specifies the state's default initial state, which is, the child state to enter if the transition does not specify one. (If the default initial state is itself compound, the state machine will also enter its default initial state, and so on recursively until it reaches an atomic state). The presence of default initial states provides a form of encapsulation, since a transition may select a compound state as its target without knowing its internal substate structure.

The default initial state of a compound state may also be specified via the 'initial' attribute. The only difference between the `<initial>` element and the 'initial' attribute is that the `<initial>` element contains a `<transition>` element which may in turn contain executable content which will be executed before the default state is entered. If the 'initial' attribute is specified instead, the specified state will be entered, but no executable content will be executed. (If neither the `<initial>` child nor the 'initial' element is specified, the default initial state is the first child state in document order). As an example, suppose that parent state S contains child states S1 and S2 in that order. If S specifies S1 as its default initial state via the 'initial' attribute (or fails to specify any initial state), then any transition that specifies S as its target will result in the state machine entering S1 as well as S. In this case, the result is exactly the same as if the transition had taken S1 as its target. If, on the other hand, S specifies S1 as its default initial state via an `<initial>` element containing a `<transition>` with S1 as its target, the `<transition>` can contain executable content which will execute before the default entry into S1. In this case, a difference between a transition that takes S as its target and one that takes S1 as its target. In the former case, but not in the latter, the executable content inside the `<initial>` transition will be executed.

**History States**

A compound state may also have history states as children: `<history>` allows achieving pause and resume semantics in compound states: before the state machine exits a compound state, it records the state's active descendants of the state. If the 'type' attribute of the `<history>` state is set to "deep", the state machine saves the state's full active descendant configuration, down to the atomic descendant(s). If 'type' is set to "shallow", the state machine remembers only which immediate child was active. After that, if a transition takes a `<history>` child of the state as its target, the state machine re-enters not only the parent compound state but also the state(s) in the saved configuration. Thus a transition with a deep history state as its target returns to exactly where the state was when it was last exited, while a transition with a shallow history state as a target re-enters the previously active child state, but will enter the default initial state of the child (if the child is itself compound.)
Transitions

As anticipated in the previous sections, transitions allow to leave one state (including compound ones) and enter another one: in case of a transition located in a compound state, the 'type' attribute is significant.

The behaviour of a transition with 'type' of "external" (the default) is defined in terms of the transition's source state (i.e. the state containing the transition), the transition's target state(s), and the Least Common Compound Ancestor (LCCA) of the source and target states (the LCCA of 2 states is the closest compound state that is an ancestor of both the states, including top level <mia_xml> tag in case no common ancestor is found).

When a transition is taken, the state machine will exit all active states that are proper descendants of the LCCA, starting with the innermost one(s) and working up to the immediate descendant(s) of the LCCA. Then the state machine enters the target state(s), plus any states that are between it and the LCCA, starting with the outermost one (i.e., the immediate descendant of the LCCA) and working down to the target state(s).

As states are exited, their <onexit> handlers are executed. Then the executable content in the transition is executed, followed by the <onentry> handlers of the states that are entered. If the target state(s) of the transition is not atomic, the state machine will enter their default initial states recursively until it reaches an atomic state(s).

In the example below, assume that state s11 is active when event 'e' occurs. The source of the transition is state s1, its target is state s21, and the LCCA is state S. When the transition is taken, first state S11 is exited, then state s1, then state s2 is entered, then state s21. Note that the LCCA S is neither entered nor exited.

```
<state id="S" initial="s1">
  <state id="s1" initial="s11">
    <onexit>
      <log expr="'leaving s1'"/>
    </onexit>
    <state id="s11">
      <onexit>
        <log expr="'leaving s11'"/>
      </onexit>
    </state>
  </state>
  <transition event="e" target="s21">
    <log expr="'executing transition'"/>
  </transition>
</state>
<state id="s2" initial="s21">
  <state id="s21">
    <onentry>
      <log expr="'entering s21'"/>
    </onentry>
  </state>
  <onentry>
    <log expr="'entering s2'"/>
  </onentry>
</state>
<onentry>
  <log expr="'entering S'"/>
<onentry>
```
<onexit>
  <log expr="'leaving S'"/>
</onexit>
</state>

The sequence of execution of executable contents will be:
leaving s11; leaving s1; executing transition; entering s2; entering s21

The behaviour of transitions with 'type' of "internal" is identical, except in the case of a transition whose source state is a compound state and whose target(s) is a descendant of the source. In such a case, an internal transition will not exit and re-enter its source state, while an external one will, as shown in the example below.

<state id="S" initial="s1">
  <state id="s1" initial="s11">
    <onentry>
      <log expr="entering S1"/>
    </onentry>
    <onexit>
      <log expr="'leaving s1'"/>
    </onexit>
    <state id="s11">
      <onentry>
        <log expr="entering s11"/>
      </onentry>
      <onexit>
        <log expr="'leaving s11'"/>
      </onexit>
      <transition event="e" target="s11" type="internal">
        <log expr="'executing transition'"/>
      </transition>
    </state>
    <transition event="e" target="s1"/>
  </state>
</state>

The sequence of execution of executable contents will be:
leaving s11; executing transition; entering s11

If transition type was "external (default)" the sequence of execution of was:
leaving s11; leaving s1; executing transition; entering s1; entering s11

If the 'target' on a <transition> is omitted, then the value of 'type' does not have any effect and taking the transition does not change the state configuration but does invoke the executable content that is included in the transition. Note that this is different from a <transition> whose 'target' is its own state. In such a case, the state is exited and reentered, triggering execution of its <onentry> and <onexit> executable content according to the type of the transition.

As data can be sent along with events, Transition can store such data into the datamodel, through the <param> tag; for each item of data received along with the event, all the ones dealt with in the <param> tags are stored in the relative data tags; received data not
mentioned in any `<param>` tag are just discarded; likewise, any `<param>` tag mentioned data not received in the event is just neglected.

### 4.2 The Core Constructs of the MIA-XML language

In the following, a brief introduction the tags of the MIA-XML language is reported; see Appendix 1 for an exhaustive definition of each and every tag.

**<mia_xml>**

The top-level wrapper element, carrying version information. The actual state machine consists of its children. Note that only one of the children is active at any one time.

**<state>**

Holds the representation of a state.

**<parallel>**

The `<parallel>` element encapsulates a set of child states which are simultaneously active when the parent element is active.

**<transition>**

Transitions between states are triggered by events and conditioned via guard conditions. They may contain executable content, which is executed when the transition is taken.

**<initial>**

This element represents the default initial state for a complex `<state>` element (i.e. one one containing child `<state>` or `<parallel>` elements.

**<final>**

This element represents one final state for a complex `<state>` element (i.e. one one containing child `<state>` or `<parallel>` elements. When the state machine reaches one `<final>` child of an `<mia_xml>` element, it terminates execution.

**<onentry>**

A wrapper element containing executable content to be executed when the state is entered.

**<onexit>**

A wrapper element containing executable content to be executed when the state is exited.

**<history>**

The `<history>` pseudo-state allows a state machine to remember its state configuration. A `<transition>` taking the `<history>` state as its target will revert the state machine to this recorded configuration.
4.3 Executable Content in the MIA-XML language

Executable content allows the State Machine to do things, such as modify its data model and/or interact with external entities. Executable content consists of actions that are performed as part of taking transitions. In particular, executable content occurs inside <onentry> and <onexit> elements as well as inside transitions. Notice that targetless transitions may contain executable content as well; this would allow one state to do things simply because it is active. When the state machine takes a transition, it executes the <onexit> executable content in the states it is leaving, followed by the content in the transition, followed by the <onentry> content in the states it is entering.

Wherever executable content is permitted, an arbitrary number of elements may occur. Such a sequence of elements of executable content is called a block. For example, if transition t takes the state machine from atomic state S1 to atomic state S2, there are three blocks of executable content executed: the one in the <onexit> handler of S1, the one inside T, and the one inside the <onentry> handler of S2. The MIA-XML executor executes the elements of a block in document order. If the processing of an element causes an error to be raised, the executor stops processing the remaining elements of the block, while the execution of other blocks of executable content is not affected.

$log$

$log$ allows an application to generate a logging or debug message useful in the development of the State Machine at applicative level.

$assign$

The $assign$ element is used to modify the data model.

$raise$

The $raise$ element raises an event in the current MIA-XML session. Note that the event will not be processed until the current block of executable content has completed and all events that are already in the internal event queue have been processed. For example, suppose the $raise$ element occurs first in the <onentry> handler of state S followed by executable content elements ec1 and ec2. If event e1 is already in the internal event queue when S is entered, the event generated by $raise$ will not be processed until ec1 and ec2 have finished execution and e1 has been processed.

$if$  $elseif$  $else$

$if$ is a container for conditionally executed elements.
$elseif$ is an empty element that partitions the content of an $if$, and provides a condition that determines whether the partition is executed.
$else$ is an empty element that partitions the content of an $if$. It is equivalent to an $elseif$ with a "cond" that always evaluates to true.

The following example shows the usage of the if, elseif and else tags:
The `<foreach>` element allows a State Chart application to iterate through a collection in the data model and to execute the actions contained within it for each item in the collection.

The MIA-XML executor acts as if it has made a shallow copy of the collection produced by the evaluation of 'array' (modifications to the collection during the execution of `<foreach>` shall affect the iteration behaviour). The executor starts with the first item in the collection and proceeds to the last item in the iteration order that is defined for the collection. For each item in turn, the processor assigns it to the item variable. After making the assignment, the executor evaluates its child executable content. It then proceeds to the next item in iteration order. If the evaluation of any child element causes an error, the processor ceases execution of the `<foreach>` element and the block that contains it. Note that there is no break functionality to interrupt `<foreach>`, however, using targetless and/or eventless transitions sophisticated iterative behavior can be achieved.

### 4.4 Data modelling in the MIA-XML language

The Data Model offers the capability of storing, reading, and modifying a set of data that is internal to the state machine. In addition to the underlying data structure, the data model defines a set of expressions. These expressions are used to refer to specific locations in the data model, to compute values to assign to those locations, and to evaluate Boolean conditions. Finally, the data model includes a set of system variables which are automatically maintained by the MIA-XML executor.

The data model is defined via the `<datamodel>` element, which contains zero or more `<data>` elements, each of which defines a single data element and its type (i.e. number, the default, or string) and optionally assigns an initial value to it. Values can then be updated via the `<assign>` element. The `<content:> and `<param>` elements can be used to incorporate data into communications with external entities.

### Data binding and scoping

There is a globally visible data model for the entire state machine. Specifically, the MIA-XML executor allows any data element of that datamodel to be accessed from any state. On the other hand, each and every `<state>` or `<parallel>` can have its own `<datamodel>` to encapsulate data for its own and its substates; such private datamodels are initialized with default values (if any) every time their owning `<state>` or `<parallel>` is entered.

The initial value specified by 'expr' is assigned to the data element even if the element already has a non-null value every time the owning state is entered.
Ordering dependencies <data> elements are possible and follow the document order. Suppose, for example, that the declaration of element "a" precedes the declaration of element "b" in a document. It can be assumed that "a" will be instantiated and have a value when the declaration of "b" is executed. Therefore the "expr" in "b" can safely reference the value of "a". Note that the data model can only modified by <assign>, <param> and <finalize>. In particular, no means is defined for external entities to modify the data model. In this sense the data model is local to the MIA-XML execution.

<data>

The <data> element is used to declare and populate portions of the datamodel.

<content>

A container element holding data to be passed to an external service: by means of <send> or <invoke> tags. When evaluating the <content> element, if the 'expr' value expression is present it is evaluated first and the result is taken as the <content> element. If the evaluation of 'expr' produces an error, the empty string is used as the value of the <content> element.

<param>

The <param> tag provides a general way of identifying a key and a dynamically calculated value which can be passed to an external service or included in an event. It can only used to assign data received from an external service to data in the datamodel.

Expressions

Expressions are used to produce values to be used in assigning variables, i.e. data items. As data items are either numerical or string typed, so are expressions.

For numerical expressions, the usual operators, i.e. +-*%() have their usual meaning and any arbitrary expression can be built; the following built-in constants and function are available: Pi, e, exp(x), log(x), log10(x), sqrt(x), floor(x), abs(x), rand(), rand100(), fac(x), int(x), dec(x), sin(x), cos(x), tan(x), aSin(x), aCos(x), aTan(x); the Active(x) function is also provided, which takes a state ID as its argument and returns the time by which the state is active since its last activation, if its argument is omitted, the current state is considered. Such a function is useful to implement timers.

For string expressions, operator + is available, with its usual meaning of “string concatenation”; the builtin function substr(str,pos,len) is available to extract a substring from a string starting after the pos th character of the string, for a length of len characters, or less if the end of string comes first.

The type of the expression is provided by the type of the variable to assign; in case variables (or constants) of the opposite type are used, type conversion is applied just after accessing it, so:
1. if a string typed variable is accessed in a number expression its length in characters is used instead
2. if a number variable is accessed in a string variable, its character representation is used instead.

In any case evaluation of expressions do not produce side effects.

**Conditions**

Conditions are used inside the 'cond' attribute of <transition>, <if> and <elseif>. They are in the form of Boolean expressions combining the basic Boolean term [notOp]{lhs pred rhs}, with optional notOp is ~ and lhs and rhs expressions and pred belonging to { ==, !=, >, <, >=, <= } (actually: ==, !=, <, >, >=, <=) combined at any complexity through the &&, ||, ^^ operators, (actually &&, ||, ^^) and braces, used to specify precedence at the boolean level, in order to distinguish them from the ordinary braces, used with arithmetic meaning inside lhs and rhs expressions.

By default the 6 predicates operate with numeric behaviour, i.e. a > b evaluates to true if a is greater than b; however they can also be explicitly requested to work with string behaviour adding a $ (dollar sign) on the side that has to take the string behaviour, i.e. a $>$ b is true if a, taken as string comes after b in alphabetical ordering, with b taken as a string; in case a or b variables were not strings they are converted using the rules listed in the previous section, about Expressions.

If the evaluation of a condition causes an error, the false value is returned. The 'In(x)' predicate is supported, which takes a state ID as its argument and returns true if the state machine is in that state. This predicate allows coordination among parallel regions.

Conditional expressions do not produce side effects.

**Location Expressions**

Location expressions are used to refer to a variable within the datamodel. Such expressions are of the type: a[sel]/……../b[sel].c[sel] where c is an attribute of b and b is a data item under a and [sel] is a selection condition, i.e. id="foo". Every occurrence of Location Expressions with exception to the “foreach” is evaluated at post-parse time to speed up execution time; for this reason it must evaluate to one and only one item in one of the active datamodels, starting from the innermost one. It is error if it evaluates to 0 or > 1 elements.

In case of “foreach” the Location Expression will be evaluated at run time and can contain any number of elements, including 0 elements.

**Errors in Expressions**

Syntactic errors are are captured at compile time and flagged; such errors stop the further execution of the state machine.

In case a run time error arises the result is the empty string for string typed expressions and 0-False for number type expressions.

**System Variables**
A protected portion of the data model is kept, to store information that can be useful to applications. We refer to the items in this special part of the data model as 'system variables'. Variable names beginning with '_' are reserved for system use. No ids beginning with '_' in the <data> element are allowed. The following variables are supported in the root datamodel:

- _sessionid. This is a system-generated id for the current MIA-XML session; such a variable is valid and constant until the session terminates.
- _name. This variable is bound at load time to the value of the 'name' attribute of the <mia_xml> element. such a variable is valid and constant until the session terminates.

4.5 External Communication in the MIA-XML language

The External Communications capability allows an MIA-XML session to send and receive events from external entities, and to invoke external services.

1. The <send> tag provides the capability to deliver events and data to any destination, including other MIA-XML sessions; the 'delay' attribute allows for deferred event delivery and can be used to implement a timer. The available transport is HTTP over TCP/IP. Events are sent asynchronously, without the state machine wait for response; however a confirmation is waited for by the executor, and if such a wait times out a warning message is generated.

2. The <invoke> tag offers a more tightly coupled form of communication, specifically the ability to trigger an external service, pass data to it and receive data from it through its child <finalize>. The semantics of the <invoke> is compatible to an HTTP request where the invoking party waits synchronously for the reception of the result from the invoked party. The <invoke> element is executed after the state's <onentry> element and causes an instance of the external service to be created. The <param> and <content> elements can be used to pass data to the service. The <finalize> code is used to normalize the form of the returned data and to update the data model before the transitions' "event" and "cond" clauses are evaluated. When parallel states invoke the same external service concurrently, separate instances of the external service will be started. They can be distinguished by ids which are associated with them. Similarly, the ids contained in the events returned from the external services can be used to determine which events are responses to which invocation.

<send>

<send> is used to send events and data to external systems, including external MIA-XML executors, or to raise events in the current MIA-XML session.

The target of the <send> operation specifies the destination of the event. The target is defined by either the 'target' or the 'targetexpr' attribute.

The type of the <send> operation specifies the method that the MIA-XML executor uses to deliver the message to its target. Either the 'type' or the 'typeexpr' attribute to define the type (not both at the same time). At the moment the only supported method is HTTP-POST; the default type/typeexpr value is “xml”; in the future “json” will also be allowed.

Message Content
The sending MIA-XML executor does not alter the content of the <send> and includes it in the message that it sends to the destination specified in the target attribute of <send>.

In the following, an example POST payload generated and sent out of the following MIA-XML fragment:

```
<send id="recogniseRequest" target="http://receiver.newamuser.it:8082/asr"
type="xml" event="recogniseRequest">
  <param name="grammar" expr="'grammar_complete.xml'" type="string"/>
  <param name="grammar" expr="'grammar_cancel.xml'" type="string"/>
</send>
```

POST payload Received at receiver.newamuser.it site on port 8082:

```
<event name="stopRecognition" sendId="stopRecognition">
  <payload>
    <parameter name="grammar" expr="'grammar_complete.xml'" type="string"/>
    <parameter name="grammar" expr="'grammar_cancel.xml'" type="string"/>
  </payload>
</event>
```

In the following, a possible response issued by the receiving site (positive answer, in case it was awaiting such a message) (200 OK header):

```
<eventSendAcknowledge
  version="1.0" eventName="recognisedkey"
  stateName="RecKeyword_CompleteCommand"
  stateMachineFile="/SM/dialogue_main_7_EN"/>
```

Note that the absence of any error events does not mean that the event was successfully delivered to its target, but only that the executor was able to dispatch it.

**<invoke>**

The <invoke> element is used to create and refer to an instance of an external service.

The <invoke> tag is used to execute a child process on the same processing node where the MIA-XML executor runs; option can be passed to the child process via the <param> tag; the same <param> tag can take returned values back into the State Machine (see below the <finalize> tag).

If the 'name' of a <param> element in the <invoke> matches the 'id' of a <data> element in a <datamodel>, the value of the <param> element will be set to the value of the data element when passed to the invoked service. If there is no data in datamodels matching the param name that <param> element will not be passed to the service.

If the invoking state machine exits the state containing the invocation before the invoked service terminates, it cancels the invoked session aborting the invoked service.

**<finalize>**

The <finalize> element enables an invoking session to update its data model with data when it terminates its execution. <finalize> contains executable content that is executed whenever
the external service terminates its execution. This content is applied before the system looks for transitions that match the event. In the case of parallel states, only the finalize code in the original invoking state is executed.

### 4.6 Release 1.0 of the MIA-XML executor

A first version of the MIA-XML executor has been implemented and tested; this will be referred to as release 1.0; the whole MIA-XML language is only partially supported; however its actual coverage is large enough to run the first DIRHA prototype; Appendix 1 contains the reference manual for the complete designed language; tags identified as “not yet implemented” will be added in the coming months so that all the tags reported in Appendix 1 will be implemented by the end of the project. Release 1.0 already implements parallelism.

**Implementation details**

The MIA-XML executor has been implemented as a C++ program for maximal execution efficiency, minimal memory footprint and maximal portability to different environments, including mobile and embedded systems and uninterrupted operability.

A specialized regression test-suite has been developed; the following pictures report the most common state machine structures, captured into specific test cases, starting from the simple ones (pure sequential ones) to more complex ones, with parallelism (with reference to the following picture, the objects with yellow background contain parallel threads).

The current Release 1.0 of the MIA-XML executor is able to handle fairly large State Machines (up to 65k tags per state machine); some initial evaluations have been carried out in order to assess its efficiency in terms of two orthogonal performance figures of evaluation speed and communication speed, measured on a single CPU computer running the MS-Windows XP operating system; there are no known benchmarks for this kind of software; however such performances exceed any reasonable set of requirements for human to machine interfaces.

<table>
<thead>
<tr>
<th>Example</th>
<th>Evaluation speed (states/S)</th>
<th>Communication Speed (events/S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>16 7100</td>
<td>n.a. (there is no ext. communication)</td>
</tr>
<tr>
<td>Parallel3</td>
<td>8300</td>
<td>n.a. (there is no ext. communication)</td>
</tr>
<tr>
<td>Circular</td>
<td>n.a. (there is only communication)</td>
<td>530</td>
</tr>
</tbody>
</table>
Figure 4-1 – examples of Sequential Test cases
Figure 4-2: examples of Parallel Test cases
5. House+User Profile and House+User State

Defining and keeping up-to-date House Profile and House State, as well as User Profile allows implementing more effective dialogues, as they represent knowledge that can be used a-priori to direct dialogues, instead of asking the user or deriving from the context; within the DIRHA project such information will be exploited as much as possible; keeping advantage of the concurrent nature of the dialog State Machines the House State within the Dialog Manager is always kept in-sync with the real house, through the House Automation system.

The House Profile is a data structure abstracting and encapsulating all the unique items contained into the specific house managed by one Dialogue instance; encapsulating such information into a single place will be of great help in adapting the dialogue to a variety of different Houses where the DIRHA system would be installed.

The User Profile is a data structure with the same objectives of the House Profile, but regarding each and every user of a given house; it is fairly smaller than the House Profile; for this reason it has been associated with the House Profile; from this point on the term House Profile (and hence House State) will be used to refer to both House and User Profiles (and States).

The House State is an augmentation to the House Profile holding the specific state variable(s) for each and every object in the House Profile whose state is known, at least partially.

In order to make the dialogue independent enough from the specific house (and also the grammars, understanding and prompts), a House Profile is being defined (according to XML syntax).

5.1 The House+User Profile

In the following, an example of House Profile data structure is reported; a formal definition of the data structure is reported later in this section.

```xml
<house name="ITEA flat" address="192.168.1.2">
  <room name="kitchen" id="R1" synonyms="food; cooking;">
    <window name="small" id="W1" synonyms="north" confirm="Y" default="Y"/>
    <window name="big" id="W2" synonyms="garden" confirm="Y"/>
    <blinds name="shutter" id="S1" synonyms="garden"/>
    <door name="entrance" id="D1" synonyms="hallway" confirm="Y"/>
    <light name="chandelier", id="L1" synonyms="main" default="Y"/>
    <light name="neon" id="L2" synonyms="little"/>
    <temperature name="heater" id="H1" synonyms="thermostat"/>
    <appliance name="owen" type="owen" id="A1" synonyms=""/>
    <media name="TV" id="TV1" synonyms=""/>
    <telephone name="phone" id="T1" synonyms="" mode="handsfree"/>
  </room>
  <room name="bathroom" id="R2" synonyms="washroom; restroom; …">
    <window name="garden", id="W3" synonyms="" confirm="Y"/>
    <door name="entrance", id="D2" synonyms="hallway" confirm="Y"/>
    <light name="hanging", id="L3" synonyms="main" default="Y"/>
    <light name="mirror", id="L4" synonyms="little"/>
    <temperature name="heater" id="H2" synonyms="thermostat"/>
  </room>
  <room name="bedroom" id="R3" synonyms="bed; sleep; …">
    <window name="south", id="W4" synonyms=""/>
    <blinds name="shutter" id="S2" synonyms=""/>
```

DIRHA_D5.1_20130220 40
The following remarks can be done with respect to the above example:

1. Some items have the attribute “confirm”: for the cases where it is defined and its value is “Y” the dialogue requests specific confirmation to execute the command (i.e. “do you really want to open the front door?”).

2. In case where more than one item of the same class lays within the same room, at most one of them can have the attribute “default” set to “Y”; in this case, when a user utterance is received for that type and that room the default item is picked without asking more questions.

3. Items have a mandatory “id” attribute: this is the key known by the House Automation system.

Some grammars could be built dynamically out of the House+User Profile to help the recognition and understanding phases through a more specific data set; this information, extracted from the House State could be part of an extended context passed to the ASR.

### 5.2 The House+User State

The House Profile is the ideal place to host the dialog abstraction of the state of the various resources in the house and its inhabitants (i.e. the users).

The “House State” is an augmented version of the “House profile”, holding the current state of each and every resource whose state is known.
The House profile is constantly updated in an event-driven way as soon a change of state is communicated by either the House Automation system of the other parallel thread in the Dialog State Machine. In the following, the previous example of House State is reported (in bold face) with added the state information.

```xml
<house name="ITEA flat" address="192.168.1.2"
  state="users:1; temp:20;"
><room name="kitchen" id="R1" synonyms="food; cooking;"
  state="s:50;"
><window name="small" id="W1" synonyms="north" confirm="Y" default="Y"
  state="s:100;"
><blinds name="shutter" id="S1" synonyms="garden" state="s:100;"
><door name="entrance" id="D1" synonyms="hallway" confirm="Y" state="s:0;"
><light name="chandelier", id="L1" synonyms="main" default="Y"
  state="s:100;"
><light name="heater" id="H1" synonyms="thermostat" state="s:21;"
><appliance name="oven" type="oven" id="A1" synonyms=""
  state="s:100; temp:180; togo:1800;"
><media name="TV" id="TV1" synonyms=""
  state="s:100; prog:3; vol:40;"
><telephone name="phone" id="T1" synonyms=""
  state="Off;"
></room>
<room name="bathroom" id="R2" synonyms="washroom; restroom; ..."
><window name="garden", id="W3" synonyms=""
  state="s:10;"
><door name="entrance", id="D2" synonyms="hallway" confirm="Y"
  state="s:0;"
><light name="hanging", id="L3" synonyms="main" default="Y"
  state="s:10;"
><temperature name="heater" id="H2" synonyms="thermostat"
  state="s:22;"
></room>
<room name="bedroom" id="R3" synonyms="bed; sleep;"
><blinds name="shutter" id="S2" synonyms=""
  state="s:0;"
><door name="entrance", id="D3" synonyms="hallway"
  state="s:0;"
><light name="chandelier", id="L5" synonyms="main"
  state="s:0;"
><light name="abat-jour", id="L6" synonyms="little"
  state="s:20;"
><media="" id="TV2" synonyms=""
  state="s:0; prog:3; vol:40;"
><telephone name="phone" id="T2" synonyms=""
  state="Off;"
></room>
<room name="entrance" id="R4" synonyms="hallway"
><door name="front", id="D4" synonyms="front door"
  state="s:0;"
><door name="kitchen", id="D1" synonyms=""
  state="s:100;"
><door name="bathroom", id="D2" synonyms=""
  state="s:0;"
><door name="bedroom", id="D3" synonyms=""
  state="s:0;"
><temperature name="heater" id="H4" synonyms="thermostat"
  state="s:20;"
></room>
<user name="John" id="U1" synonyms="Johnny"
  state="room:R1;RmConf:50"
><preference room="R1"
  <set item="S1" to="s:100;"/>
  <set item="T1" to="mode:headset"/>
</preference>
<user name="Mary" id="U2"
  state="room:--;RmConf:100"
<preference room="R1"
  <set item="S1" to="s:10;"/>
</preference>
```

In the following a formal definition of the House+User Profile and State is reported, as DTD.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!ELEMENT House (room | user)+>
<!ATTLIST House
  name CDATA #REQUIRED
  address CDATA #REQUIRED
  state CDATA #REQUIRED >
<!ELEMENT room ( door | window | blinds | light | temperature | appliance | media | telephone)>>
<!ATTLIST room
  name CDATA #REQUIRED
  id CDATA #REQUIRED
  synonyms CDATA #IMPLIED >
<!ELEMENT door >
<!ATTLIST door
  name CDATA #REQUIRED
  id CDATA #REQUIRED
  synonyms CDATA #IMPLIED
  confirm (yes|no) "no"
  default (yes|no) "no"
  state CDATA #IMPLIED >
<!ELEMENT window >
<!ATTLIST window <!-- same as door --> >
<!ELEMENT blinds >
<!ATTLIST blinds <!-- same as door --> >
<!ELEMENT light >
<!ATTLIST light <!-- same as door --> >
<!ELEMENT temperature >
<!ATTLIST temperature <!-- same as door --> >
<!ELEMENT appliance >
<!ATTLIST appliance <!-- same as door --> >
<!ELEMENT media >
<!ATTLIST media <!-- same as door --> >
<!ELEMENT telephone>
<!ATTLIST telephone
  name CDATA #REQUIRED
  id CDATA #REQUIRED
  synonyms CDATA #IMPLIED
  mode CDATA #IMPLIED
  confirm (yes|no) "no"
  default (yes|no) "no"
  state CDATA #IMPLIED >
<!ELEMENT user ( preference )*>
<!ATTLIST user
  name CDATA #REQUIRED
  id CDATA #REQUIRED
  state CDATA #IMPLIED >
<!ELEMENT preferences >
<!ATTLIST preferences
  room CDATA #REQUIRED >
<!ELEMENT set >
<!ATTLIST set
  item CDATA #REQUIRED
  to CDATA #REQUIRED >
```
5.3 Synchronization among physical resources and House State

The real house resources (i.e. lights, ...) will be under control of the Home Automation system, whose purpose is to control and observe their state; the wall switches, as well as other interaction devices (e.g. centralized console, where a graphic interface on a touch screen allows the visualization and change of the state of the resources, or remote control) send their commands to the House Automation system which, in turn controls the house resource.

The House Automation system will be interfaced to the Dialog Manager in a bidirectional way:

1. from House Automation system to Dialog Manager, to let the DM update its House State; such events will be sent as soon as the state of a house resource changes; in a starting phase the House Automation system shall send a stream of events to let the DM change the state of all the resources from the initial (i.e. off) position.

2. from Dialog Manager to House Automation system, to execute a spoken command provided to the DIRHA system.

Within the Dialog Manager the House+User State will be updated according to the following events:

1. actions over house resources done through the House Automation system (i.e. user turned on a light using the wall switch, or the console)

2. actions over house resources done through the Dialogue Manager: this is the case of spoken commands; in such a case the Dialog Manager will issue a command event to the House Automation system, to request for the needed action. From the House Management system point of view such kind of commands are logically identical to the ones coming from the wall switches and the other control devices.

3. change in the state of Users (e.g. movement from one room to another one) notified to the Dialog Manager by other subsystems; for example the position of one user could be updated after each utterance provided that the system is able to recognize the user from his/her voice and its position. In such a case one or more some command events could be issued to the House Automation system to request actions specified in the user preferences (i.e. change of blind position when a specific user is into the room).
6. Integration of the CDM within the DIRHA environment

After the detailed discussion of the MIA-XML language and House state and Profiling, some more insight can be added to what introduced in §2.2 (Design choices in the DIRHA Dialog Management), concerning the integration of the Concurrent Dialog Manager within the specific ecosystem of the DIRHA project.

Figure 6-1 the target Dialog Manager ecosystem for the DIRHA project

Figure 6-2 typical message exchange among modules
Figure 6-1 reports the CDM and its ecosystem as designed for the final prototype in DIRHA (the oval hides all the signal processing algorithms to be developed and integrated within the project).

Figure 6-2 shows another view of the same modules, in relation with a typical to the message exchange sequence; notice the thick black bar, which represents the group of several concurrent dialogs managed simultaneously; events exchanged with ASR and Prompt Generator carry the room tag along with the other attached data.

**Roles of the various modules in the interaction**

The event nature of the exchange of messages among the MIA-XML executor and the other elements in its ecosystem does not pre-configure which role (e.g. master or slave) will be played by each element; this is decided according to each specific case. In the following, the role attribution for the main modules is reported for each interaction.

1. **Dialog Manager vs. ASR**: encapsulating most of the rest of the DIRHA system, such as the Speech Understanding, Source Localization, Speaker Recognition, ...): for this interaction the Dialog Manager is master and the ASR is slave: first the DM activates the ASR recognition, against a specific recognition context and then the ASR returns a recognition frame, with its fields filled with and understanding of the recognized utterance; then the ASR is stopped.

   This assumption is fairly straightforward, as the actual context to be passed to the ASR is dependent upon the state of the dialogue and this knowledge is kept by the Dialog Manager; in case multiple utterances are expected in different rooms, the ASR will receive multiple activations with different room designators.

   The above statement does not imply that the Dialog State Machine will be the sole place where information about the state of the system is stored: while this is the most natural room for storing the state of the dialogue, and easy to implement also for other kinds of data, information about details of the acoustic scene, or other, could be stored into other modules of the system if this is more efficient or effective.

   For the above reason the context information kept by the Dialog State Machines and passed to the ASR (and the other modules it encapsulates) has not been completely identified; a complete list will be produced in the coming phases of the project, when the capabilities and needs of all the encapsulated modules will be clearer.

2. **Prompt Generator**: for this interaction the Dialog Manager is master and the Prompt Generator is slave; the DM activates the Prompt Generator with a file name containing the recorded message (or the text of the string to synthesize) if needed the Prompt Generator could be stopped in the middle of the playback of a previous message. Prompts could be of three different kinds: (i) dynamically generated via TTS, out of the text phrase; (ii) spoken by a professional speaker and recorded into a file; (iii) jingles, stored into a file. While the choice of the kind of prompt is transparent to the Dialog Manager, it is good practice for the Dialog State Machine to send along with the event...
both the textual message to be synthesized in case of TTS and the file name in case a pre-recorded message or jingle is chosen.

The choice of the prompt phrases is crucial to lead the user to give his/her answer in the expected way (i.e. aka “linguistic inductors”), hence to improve the overall accuracy of the system.

3. Home Automation system: for this interaction they are Peer-to-Peer (i.e. they can both send commands to each other): the DM will send commands to the Home Automation system as soon a command has been received (and confirmed if requested to); on the other hand, the Home Automation system will send a change of State of (some device in) the House as soon as entities outside DIRHA produce a state change; the House State maintained by the Dialog State Machine will be updated accordingly to help understanding future user commands.

Initial Development Ecosystem

While awaiting for the whole system to be complete, with the to-be ASR + Speech Understanding + Source Localization + Speaker Recognition, a temporary development ecosystem has been put in place; this will help to:

1. optimize and tune the Concurrent Dialog Manager, challenging it with long iterative stimuli with test State Machines
2. develop and optimize the Dialogue State Machines
3. develop and optimize the handcrafted Grammars
4. develop end optimize the Prompts

Figure 6-3 reports this development environment.
Figure 6-4 reports the ASR Emulator page, developed in javascript/jquery; it is able to send the ASR events as xml fragments sent in the payload of POST invocations to the CDM; the same page, on the left hand side reports the events sent from the CDM to start the recognition according to the requested context.

Finally, Figure 6-5 reports the simulator of the House Automation system, and is able to show the current state of the house as set up by the dialogue. (only doors, windows and light are shown, at the moment).

![Figure 6-4 ASR Emulator](image)
Figure 6-5 House State Display
7. Design of the User Interface Dialogue Flow

This section discusses a dialog flow able to handle a subset of the house resources that will be dealt with by the real DIRHA prototypes; its functionalities are also limited. It has been reported here to show through a concrete example how the various modules interact with each other to implement the desired behaviour.

However the User interface of the DIRHA prototype #1 is being developed evolving the reported design; since the prototype is due by M18, the design will be improved and completed in the next few months. A specific deliverable (D1.3) is planned to document it.

General guidelines in the design of the dialogue

The DIRHA system will have an “always listening” ASR; however, in order to avoid false starts upon utterances not directed to it or coming from TV or other sources, the ASR will be directed to recognize against to a very specific grammar or Language model, i.e. one containing only a very specific “keyword”, not easy to be misrecognized; after this initial turn of dialog has passed, the real grammars or language models are activated; at the end of the iteration the ASR will be reverted to recognize only the “keyword”. In the initial turn of dialog the identification of the speaker could take place (when available), so that the system could answer “Tell me <username>”. Should the identification be unavailable, or if the user was not identified, a simple “welcome jingle” or prompt (like “Here I am”) will be played instead.

An alternative implementation could be to join the two phases (i.e. activation and usage) into a single phrase; in this case the “keyword” will be the first word to be spoken in the phrase; such an approach is also considered in the user interface shown below; of course the speaker identification will take place during the recognition of the composite phrase.

The DIRHA system will always wait for user requests, the only case in which the system will spontaneously start a dialogue (hence skipping the initial “keyword” check) is when the intercom or telephone is ringing. However such a case is not dealt with in the User Interface shown below.

The User Interface is aware of each and every physical resource of the house (i.e. which rooms, doors, lights, … lay in every room) and of the state of each one of them, through the House+User Profile and House+User State, according to what discussed in Section 6; however, for sake of simplicity, in the User Interface presented here, only the command flow from the Dialog Manager to the House Automation system is designed.

Names (and synonyms) for all the resources of the house are taken from the House Profile, that would be different for each users’ home; on the other hand the dialogue flow defines other voice commands independent on the house profile: these are some “general commands”, as Cancel, Exit or Stop, to end the interaction on the voice interface, and verbs related to the actions allowed to operate upon each item.

The actions allowed for each device could be not only verbs representing binary action modes (e.g. open/close, switch on/off, etc…), but also “gradual” ones, to specify the desired position. The dialogue flow will allow not only to change the state of one item but also to obtain information about its state.
However, in the general case, it is preferable that the understanding data/knowledge base was based on the specific House profile, in order to avoid recognizing some classes or objects which are not available in that house.

The localization of the speaker is crucial to know which item he/she is talking about but also to determine which loudspeakers shall be used when playing back the responses.

The dialogue assumes that the position of the speaker is provided with the granularity of the "room" (i.e. no finer grain localization is assumed).

The DIRHA system will be used continuously on a daily basis; this must be taken into account also when designing prompts: while they should be informative enough in the first period of use they could soon be perceived as redundant (some form of parameterization could also be adopted).

### 7.1 Introduction to the Dialog Flow

The dialog must figure out the following parameters:

1. **class** - the class of object that the user is talking about (i.e. door, light, ...);
2. **action** - the action that the user wants to apply on the object (including no action, meaning to know its actual state)
3. **attribute** – any qualifier that uniquely identifies an object within its class (i.e. its name); the identification of one object could be determined by the system in different ways:
   a. the attribute is provided explicitly
   b. the room name is provided explicitly and it has one object only of that class
   c. the room name is provided explicitly, it has many objects of that class but one is marked as the default one
   d. nothing is provided but the user localization has detected with enough confidence the position of the speaker: in this case the dialog assumes that the room of the item is the one which he/she is in and b or c hold;

With respect of the triple \{**class**, **action**, **attribute**\}, the first element is the most important one: if the recognized utterance contains only this one the dialogue enters a “refinement” procedure requesting the other ones in further turns of dialogue; in case this one is missing the dialogue enters a “recovery” procedure which asks it again.

The following list reports the utterances that will generate a reaction of the system:

- recognition of object **class** + **attribute** (in one of the 4 forms above listed): the refinement procedure is invoked, saying the status of the item and asking if change in state is wanted (i.e. “the kitchen light is off; do you want to turn it on?”);
- recognition of object **class** + **action** + **attribute** (in one of the 4 forms above listed): the system executes the desired action on the requested object; in case explicit confirmation is desired for that object, the “confirmation” procedure is will ask a confirmation before executing the action; if the action required would place the object in the same state it is already, (e.g. the user asks to open a window already open), the system would ask if he/she wants to change it to the opposite state.

### Interaction with the ASR+Speech Understanding

All the interactions between the Dialog Manager and the ASR+ Speech understanding occur exchanging of “events”, each of one carrying some attached data: the attached data sent from DM to ASR+SU are the type of recognition (i.e. the activation key or a complete utterance)
and the recognition context: for Grammar based Understanding such context is represented by the set of grammars relevant for that turn of dialog while, for Language Model based Understanding such context is represented by the goal to be pursued.

Table 7.1 shows the data attached to events that sent by the DM to the ASR + SU.

<table>
<thead>
<tr>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recognisedkey Only the activation keyword recognized</td>
</tr>
<tr>
<td>recognisedanswer Recognized utterance containing at least one of the semantic elements listed in the table below.</td>
</tr>
</tbody>
</table>

Table 7.1

Table 7.2 describes the variables (called Semantics_Slot) expected by the DM as attached to recognition events, used to manage the dialog; the table contains a brief description of the content of each slot, the values the DM expects for the showcase and some examples of words that produce these values if contained in the user utterance.

<table>
<thead>
<tr>
<th>Semantic Slot</th>
<th>Description</th>
<th>Expected values</th>
<th>Examples of synonyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics_object</td>
<td>House device</td>
<td>door / light / window</td>
<td>door, light, window, chandelier</td>
</tr>
<tr>
<td>Semantics_obj_attr</td>
<td>Attribute of the device</td>
<td>small / large</td>
<td>small, little, large, great, round, square, front</td>
</tr>
<tr>
<td>Semantics_action</td>
<td>Action to be done on the device</td>
<td>open / close / state / turnOn / turnOff /</td>
<td>open, close, turn on, turn off, how is,</td>
</tr>
<tr>
<td>Semantics_location</td>
<td>Location of the device (room where the device is)</td>
<td>kitchen / bathroom / livingRoom / entrance / closet / bedroom / littleBedroom / hallway</td>
<td>kitchen, kichenette, bathroom, toilet, living room sitting room, closet, broom closet, utility room, entrance, bedroom, large bedroom, little bedroom</td>
</tr>
<tr>
<td>Semantics_position</td>
<td>user position in the house detected by the system</td>
<td>kitchen / bathroom / livingRoom / entrance / closet / bedroom / littleBedroom / hallway</td>
<td>yes, ok, certain, right, that’s right, no, that’s wrong, not at all, cancel</td>
</tr>
<tr>
<td>Semantics_confirm</td>
<td>confirm/negation of the DM ask for confirmation or user stop to the dialog</td>
<td>yes/no/cancel</td>
<td>%, from 0 to 100</td>
</tr>
<tr>
<td>ASR_confidence</td>
<td>ASR confidence of the recognized sentence</td>
<td>%, from 0 to 100</td>
<td></td>
</tr>
<tr>
<td>SP_confidence</td>
<td>User Position confidence</td>
<td>%, from 0 to 100</td>
<td></td>
</tr>
</tbody>
</table>
Table 7.2

The two slots Semantics_location and Semantics_position share the same domain; in the same turn of dialog the two variables can assume different values; for example if the user is in the kitchen and says: “open the bathroom door” the value of the variable Semantic_location is “bathroom” while the value of the variable Semantic_postion is “kitchen”.

Table 7.3 shows the names of the grammars that will be used by the ASR in the reported dialog flow; for each grammar the table points out the events and variables (see tables 7.1, 7.2) that are filled and shows some sentence examples.

<table>
<thead>
<tr>
<th>Grammar file names</th>
<th>Returned Event and variables</th>
<th>Utterance examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>grammarkey.xml</td>
<td>event: recognisedkey</td>
<td></td>
</tr>
<tr>
<td>grammar_complete.xml</td>
<td>event: recognisedanswer</td>
<td>open the door;</td>
</tr>
<tr>
<td></td>
<td>variables:</td>
<td>close the kitchen window;</td>
</tr>
<tr>
<td></td>
<td>Semantics_object</td>
<td>turn on the closet light;</td>
</tr>
<tr>
<td></td>
<td>Semantics_obj_attr</td>
<td>close the living room little window;</td>
</tr>
<tr>
<td></td>
<td>Semantics_action</td>
<td>how is the front door?</td>
</tr>
<tr>
<td></td>
<td>Semantics_location</td>
<td></td>
</tr>
<tr>
<td>grammar_cancel.xml</td>
<td>event: recognisedanswer</td>
<td>Cancel, stop, annul</td>
</tr>
<tr>
<td></td>
<td>variable: Semantics_confirm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>event: recognisedanswer</td>
<td>the bedroom door</td>
</tr>
<tr>
<td></td>
<td>variable: Semantics_location</td>
<td>the bathroom one</td>
</tr>
<tr>
<td>grammar_yesno.xml</td>
<td>event: recognisedanswer</td>
<td>Yes, no, ok, not at all</td>
</tr>
<tr>
<td></td>
<td>variable: Semantics_confirm</td>
<td></td>
</tr>
<tr>
<td>grammar_attribute.xml</td>
<td>event: recognisedanswer</td>
<td>the round window</td>
</tr>
<tr>
<td></td>
<td>variable: Semantics_obj_attr</td>
<td>the large one</td>
</tr>
</tbody>
</table>
7.2 Discussion of the dialogue flow

In the following a possible dialogue for the DIRHA scenario is discussed; it can be seen as a starting point of the user interface that will be implemented for prototype 1.

While the dialogue is represented as the single user interface taking control of the House Automation system, thanks to the Concurrent Dialogue Manager, it can be instantiated several times, i.e. one per room; however, the strategy to handle concurrency will be addressed in the coming months, to be delivered in the final prototype.

Regarding the addressed services, only those facilities whose dialogue flow really differs from each other, are reported, like the voice interaction to manage the doors and the lights (a description of all the services that will be implemented for the first prototype will be provided in D1.3). The sub-flow dealing with the windows (i.e. “windows management”) is not detailed in the following, as it is very similar to the “Doors management” one (i.e. only the prompts change).

<table>
<thead>
<tr>
<th>N</th>
<th>symbol</th>
<th>explanation</th>
<th>N</th>
<th>symbol</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Start of flow/procedure</td>
<td>5</td>
<td></td>
<td>Decision among alternatives</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>contained in the user response</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Logical Decision</td>
<td>6</td>
<td></td>
<td>Invoke sub-flow</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Prompt issued to the user</td>
<td>7</td>
<td></td>
<td>Return from sub-flow</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Wait for user response (context reported in the box)</td>
<td>8</td>
<td></td>
<td>Invoke external procedure</td>
</tr>
</tbody>
</table>

Figure 7.1: Legend

Figure 7-2 shows the top level flow of dialogue: the system, always listening, has to be “activated” saying the activation keyword; after the activation keyword the real first turn of dialogue can occur: the user can say any command related to the facilities automatized in his/her home; however, in the reported example only doors, windows and light management are dealt with.

Notice the use of blocks of type 4 (Wait for user response); this blocks activates the ASR+SU with the specified context (i.e. the recognition goal or grammar names written inside the block); the flow is blocked until the ASR+SU returns a result (including TimeOut or No Match if it is needed to).

The check at the beginning of the dialogue flow shows the availability of a feature that allows the user also to say the activation keyword immediately followed by a command to operate on one item.
Of course there is always an escape command available (e.g. Cancel or Exit), as well there is always an Error recovery procedure to manage possible recognizer’s errors. After one user request has been recognized and processed the system remains “active” for a few seconds to let users say another request; after this the system returns to wait for the activation keyword.
According requirements expressed by interviewed users (see Deliverable D1.1), the Error recovery procedure (that manages both misunderstanding problems and missed commands) tries for two times to recover the “error”, then suggest to use the haptic interface.

Figure 7-2: House facilities procedure

Figure 7-3: General Error recovery
Each procedure listed in the “House facilities” flow is related to a class of objects (i.e. doors, …). Handling of some of them have a similar dialogue structure which can vary depending by the fact that a confirmation request is needed or not before to operate the item. As can be seen in the following dialogue flow, in the “Doors management” procedure is foreseen a confirmation because door movements can be “dangerous” for an impaired person.

Figure 7-4: Doors management
A precise identification of the object that the user wants to operate is needed each time that it is not possible to use a “default”. For this reason, the system propose this “Request location” procedure if the users does not specify the location of the object and/or the recognizer does not get its location. Before to try to localize the item that the user wanted to operate, the system should check if the action required by the user is coherent with the object got by the recognizer (e.g. it is possible to combine the action “to open” with the object "door”, while should not be allowed to combine the verb “to open” with the item "light"). If there is no coherence, the dialogue proposes some error recovery prompts (see the right-hand prompt following the “false” branch).

![Diagram](image-url)
In the “Door management” before to activate a door, the system requires a confirmation. The first check of this dialogue procedure is related to a part of the dialogue shown on the “Door management” procedure (see fig. 2): if the user asks to change the status of the door in a state equal to the one in which the door already is (e.g. asking to open a door already opened), the dialogue asks if he/she wanted to do the opposite action. At the opposite, if the required action is coherent with the status of the item, in this procedure, the system just requires to confirm all the collected data.

![Diagram of Door Management Dialogue]

Figure 7-6: Confirm Door
As there is a “General error procedure” in the “House facilities” dialogue, also in the “Door management” service there is an error recovery procedure which manages both misunderstanding problems and missed commands.

![Diagram of Doors Error Recovery](image)

**Figure 7-7: Doors Error Recovery**
The “Lights management” procedure is reported here because differs from the “Doors management” procedure, since it does not require a confirmation before to operate the item (see the right low branch at the bottom of the figure). Nevertheless, also for this service, a confirmation procedure is still foreseen to manage the cases in which the requested light is in a location far from the user (see the left low branch at the bottom of the figure).

Figure 7-8: Lights management
8. Conclusions

This document discussed the modules of the DIRHA project involved in the handling and fulfilment of the user requests expressed through spoken utterances, namely the Dialog Manager, the Speech Understanding the Prompt Producer and the house State Keeper.

Such subsystems are devoted to gathering users’ requests through an interactive process and fulfilling them, issuing the proper commands to the House Automation system; the Dialog Manager is the module in charge of conducting this interaction, iteratively asking the proper question to elicit the needed information form the user in the expected form. To this end, the choice of the prompt phrase is crucial to “linguistically induce” the user to give its answer in the expected way, hence to improve the overall accuracy of the system.

The dialog process can take from one to several iterations, ranging from the case of the trained user who says all the relevant information at once (and the system correctly understands it) to several turns of dialog, where the Dialog Manager tries to complete the needed set of information pieces asking dedicated questions – of course a well designed dialog should set a limit in the length of the interaction and give up in case such a limit is exceeded.

After a short survey of the approaches to dialog management the report highlighted the approach to be followed in the DIRHA project; in order to fulfil the challenging constraint of handling more than one session taking place in different rooms, a state based approach has been chosen for the dialog management and, in particular the formalism proposed by D. Harel in its State Charts methodology has been adopted: an executor of concurrent state machines has been developed; its input formalism, MIA-XML has been defined subsetting the SCXML language being standardized (not yet approved at the date of writing) at W3C (see Appendix 1 for the MIA-XML reference manual). Version 1.0 of the executor has been released and is ready to be used in the first prototype (M18 and M24); in the while an improved version will be developed in the M12-M24 timeframe.

The developments in Speech Understanding have been discussed in Section 3; actually two approached are being investigated, the grammar-based and the data-driven with the intention to compare the safer but limited use of grammars with the more generalized paradigm based on statistics. The work on the data-driven approach is considered as medium-term research activity as it requires a preliminary implementation of some auxiliary tools: as such, the resulting component will not be directly integrated in the intermediate prototype. Moreover, the possible advantage of this approach will be measured not only in terms on pure performance but also considering time and resources for the development as it is believed that the initial major effort can be later compensated by a faster porting to new users and settings.

The aspects of User and House Profile and State have been addressed and a data model for their handling has been discussed.

After the discussion of each and every single component (i.e. the Dialog Manager, the Speech Understanding, the Prompt Producer, and the House State Keeper) their mutual integration has been discussed, in Section 6; some intermediate integrations (i.e. where not all the components are available) have already been presented; in particular the one with an emulated version of the ASR+SU and of the House Automation are of interest, as they allow to start implementing advanced state machines ahead of time.
This phase has actually already started: starting from the requirements gathered in WP1 (see D1.1) an initial system has been designed to be demonstrated in the Dialog Showcase (see D6.1) and is currently being evolved according to the results of the Wizard of Oz experiments to become the base for the first DIRHA prototype. Such a system has taken as a case study in Section 7 to discuss the process of implementing a system using the DM and its “ecosystem”.
References


In the following the MIA-XML language is documented. Some tags are not yet implemented in Release 1.0; they will be shown with shaded background; they have been reported in this manual since they have been included in the language by design; they will be included in the next release.

A1.1 Core Constructs of the MIA-XML language

<mia_xml>

The top-level wrapper element, which carries version information. The actual state machine consists of its children. Note that only one of the children is active at any one time.

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Attribute Constr.</th>
<th>Type</th>
<th>Def. It Value</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>false</td>
<td>none</td>
<td>IDREFS</td>
<td>none</td>
<td>A valid id</td>
<td>The id of the initial state(s) for the document. If not specified, the default initial state is the first child state in document order.</td>
</tr>
<tr>
<td>name</td>
<td>false</td>
<td>none</td>
<td>NMTOKEN</td>
<td>none</td>
<td>Any valid NMTOKEN</td>
<td>The name of this state machine. It is for purely informational purposes.</td>
</tr>
<tr>
<td>xmlns</td>
<td>true</td>
<td>none</td>
<td>URI</td>
<td>none</td>
<td></td>
<td></td>
</tr>
<tr>
<td>version</td>
<td>true</td>
<td>none</td>
<td>decimal</td>
<td>none</td>
<td>&quot;1.0&quot;</td>
<td></td>
</tr>
<tr>
<td>datamodel</td>
<td>false</td>
<td>none</td>
<td>NMTOKEN</td>
<td>&quot;proprietary”</td>
<td></td>
<td>&quot;proprietary” denotes the one used by MIX-XML</td>
</tr>
<tr>
<td>binding</td>
<td>false</td>
<td>none</td>
<td>enum</td>
<td>&quot;early&quot;</td>
<td>&quot;early&quot;, &quot;late&quot;</td>
<td>The data binding to use.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>times</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;state&gt;</td>
<td>&gt;= 0</td>
<td>A compound or atomic state</td>
</tr>
<tr>
<td>&lt;parallel&gt;</td>
<td>&gt;= 0</td>
<td>A parallel state.</td>
</tr>
<tr>
<td>&lt;datamodel&gt;</td>
<td>&gt;= 0</td>
<td>Defines part or all of the datamodel</td>
</tr>
</tbody>
</table>
<final> >= 0 A top-level final state in the state machine.

<state>
Holds the representation of a state.

<table>
<thead>
<tr>
<th>Attribute Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>id</td>
</tr>
<tr>
<td>initial</td>
</tr>
</tbody>
</table>

Note(s): Cannot be specified in conjunction with the <initial> element. Cannot occur in atomic states.

<table>
<thead>
<tr>
<th>Children</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>&lt;onentry&gt;</td>
</tr>
<tr>
<td>&lt;onexit&gt;</td>
</tr>
<tr>
<td>&lt;state&gt;</td>
</tr>
<tr>
<td>&lt;transition&gt;</td>
</tr>
<tr>
<td>&lt;initial&gt;</td>
</tr>
<tr>
<td>&lt;final&gt;</td>
</tr>
<tr>
<td>&lt;parallel&gt;</td>
</tr>
<tr>
<td>&lt;datamodel&gt;</td>
</tr>
</tbody>
</table>

Note(s):
1. An atomic state is one that has no <state> or <parallel> children.
2. A compound state is one that has <state> or <parallel> children (or a combination of these).
3. either an "initial" attribute or an <initial> element can be specified, but not both. If neither the "initial" attribute nor an <initial> element is specified, the MIA-XML executor will use the first child state in document order as the default initial state.

<parallel>
The `<parallel>` element encapsulates a set of child states which are simultaneously active when the parent element is active.

<table>
<thead>
<tr>
<th>Attribute Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>id</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Children</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td><code>&lt;onentry&gt;</code></td>
</tr>
<tr>
<td><code>&lt;onexit&gt;</code></td>
</tr>
<tr>
<td><code>&lt;state&gt;</code></td>
</tr>
<tr>
<td><code>&lt;transition&gt;</code></td>
</tr>
<tr>
<td><code>&lt;initial&gt;</code></td>
</tr>
<tr>
<td><code>&lt;parallel&gt;</code></td>
</tr>
<tr>
<td><code>&lt;datamodel&gt;</code></td>
</tr>
</tbody>
</table>

**<transition>**

Transitions between states are triggered by events and conditioned by guard conditions. They may contain executable content, which is executed when the transition is taken.

<table>
<thead>
<tr>
<th>Attribute Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>event</td>
</tr>
<tr>
<td>cond</td>
</tr>
<tr>
<td>target</td>
</tr>
</tbody>
</table>
D5.1 - Design of components for understanding, dialogue management and feedback to the user

<table>
<thead>
<tr>
<th>type</th>
<th>false</th>
<th>emun</th>
<th>&quot;external&quot;</th>
<th>none</th>
<th>&quot;external&quot;, &quot;internal&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Determines whether the source state is exited in transitions whose target state is a descendant of the source state</td>
</tr>
</tbody>
</table>

**Children**

<table>
<thead>
<tr>
<th>Name</th>
<th>times</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>executable cont.</td>
<td>&gt;= 0</td>
<td>such executable content is run after all the &lt;onexit&gt; handlers and before the all &lt;onentry&gt; handlers that are triggered by this transition.</td>
</tr>
</tbody>
</table>

**Note(s):** Transition a must specify at least one of 'event', 'cond' or 'target'.

**<initial>**

This element represents the default initial state for a complex <state> element (i.e. one containing child <state> or <parallel> elements).

**Children**

<table>
<thead>
<tr>
<th>Name</th>
<th>times</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;transition&gt;</td>
<td>0 : 1</td>
<td>A transition whose 'target' specifies the default initial state(s). This transition cannot contain 'cond' or 'event' attributes, and shall specify a non-null 'target' whose value is a valid state specification. This transition can contain executable content.</td>
</tr>
</tbody>
</table>

**<final>**

represents a final state of an <mia_xml> or compound <state> element.

**Attribute Details**

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Attribute Constr.ts</th>
<th>Type</th>
<th>Def. It Value</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>false</td>
<td>none</td>
<td>ID</td>
<td>none</td>
<td>A valid id</td>
<td>The identifier for this state.</td>
</tr>
</tbody>
</table>

**Children**

<table>
<thead>
<tr>
<th>Name</th>
<th>times</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;onentry&gt;</td>
<td>&gt;= 0</td>
<td>holds executable content to be run upon entering this &lt;state&gt;.</td>
</tr>
<tr>
<td>&lt;onexit&gt;</td>
<td>&gt;= 0</td>
<td>holds executable content to be run when exiting this &lt;state&gt;</td>
</tr>
</tbody>
</table>
<onentry>
A wrapper element containing executable content to be executed when the state is entered.

<table>
<thead>
<tr>
<th>Name</th>
<th>times</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable content</td>
<td>&gt;= 0</td>
<td>The &lt;onentry&gt; handlers of a state are executed in document order when the state is entered. In doing so, it treats each handler as a separate block of executable content.</td>
</tr>
</tbody>
</table>

<onexit>
A wrapper element containing executable content to be executed when the state is exited.

<table>
<thead>
<tr>
<th>Name</th>
<th>times</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable content</td>
<td>&gt;= 0</td>
<td>The &lt;onentry&gt; handlers of a state are executed in document order when the state is entered. In doing so, it treats each handler as a separate block of executable content.</td>
</tr>
</tbody>
</table>

<history>
The <history> pseudo-state allows a state machine to remember its state configuration. A <transition> taking the <history> state as its target will return the state machine to this recorded configuration.

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Attribute Constr. ts</th>
<th>Type</th>
<th>Def. It Value</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>false</td>
<td></td>
<td>ID</td>
<td>none</td>
<td>A valid id</td>
<td>Identifier for this pseudo-state</td>
</tr>
<tr>
<td>type</td>
<td>false</td>
<td>enum</td>
<td>Boolean expression</td>
<td>&quot;shallow&quot;, &quot;deep&quot;</td>
<td>&quot;deep&quot;, &quot;shallow&quot;</td>
<td>Determines whether the active atomic substate(s) of the current state or only its immediate active substate(s) are recorded</td>
</tr>
</tbody>
</table>

<children>
A transition whose 'target' specifies the default history configuration. This transition cannot contain 'cond' or 'event' attributes, and shall specify a non-null 'target' whose value is a valid state specification. This transition can contain executable content. If 'type' is "shallow", then the 'target' of this <transition> must contain only immediate children of the parent state.

Note(s): if the 'type' of a <history> element is "shallow", the MIA-XML executor records the immediately active children of its parent before taking any transition that exits the parent. If the 'type' of a <history> element is "deep", the executor records the active atomic descendants of the parent before taking any transition that exits the parent. After the parent state has been visited for the first time, for each <history> element, we define the set of states that the processor has recorded to be the 'stored state configuration' for that history state. We also define the states specified by the 'target' of the <history> element's <transition> child to be the 'default stored state configuration' for that element.

If a transition is executed that takes the <history> state as its target, the behavior depends on whether the parent state has been visited before. If it has, the MIA-XML executor behaves as if the transition had taken the stored state configuration for that history state as its target. If it has not, the executor behaves as if the transition had taken the default stored state configuration for that history state as its target (Note that a single <state> or <parallel> element can have both "deep" and "shallow" <history> children).

**A1.2 Executable Content in the MIA-XML language**

<log>
<log> allows an application to generate a logging or debug message on the file named <stateMachine>.slg, where <stateMachine>.xml contains the MIA-XML of the State Machine.

<table>
<thead>
<tr>
<th>Attribute Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>Label</td>
</tr>
<tr>
<td>Expr</td>
</tr>
</tbody>
</table>

Note(s): The manner in which the message is displayed or logged is platform-dependent. The MIA-XML executor works in a way that <log> has no side-effects on document interpretation.
The `<assign>` element is used to modify the data model.

### Attribute Details

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Attribute Constr. ts</th>
<th>Type</th>
<th>Def. It Value</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>True</td>
<td></td>
<td>path</td>
<td>none</td>
<td>Any valid location expression.</td>
<td>The location in the data model into which to insert the new value.</td>
</tr>
<tr>
<td>expr</td>
<td>false</td>
<td></td>
<td>value</td>
<td>none</td>
<td>Any valid value expression</td>
<td>An expression returning the value to be assigned</td>
</tr>
</tbody>
</table>

### Children

<table>
<thead>
<tr>
<th>Name</th>
<th>times</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subfields</td>
<td>≥ 0</td>
<td>element provide an in-line specification of the legal data value to be inserted into the datamodel at the specified location.</td>
</tr>
</tbody>
</table>

Note(s): `<assign>` must specify either "expr" or children of `<assign>`, but not both.

Assignment to a data model is done by using a location expression to denote the part of the data model where the insertion is to be made. If the location expression does not denote a valid location in the datamodel or if the value specified (by 'expr' or children) is not a legal value for the location specified the assign is not performed.

The `<raise>` element raises an event in the current execution session. Note that the event will not be processed until the current block of executable content has completed and all events that are already in the internal event queue have been processed. For example, suppose the `<raise>` element occurs first in the `<onentry>` handler of state S followed by executable content elements ec1 and ec2. If event e1 is already in the internal event queue when S is entered, the event generated by `<raise>` will not be processed until ec1 and ec2 have finished execution and e1 has been processed.
### Attribute Details

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Attribute Constr.ts</th>
<th>Type</th>
<th>Def. It Value</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>true</td>
<td></td>
<td>NMTOKEN</td>
<td>none</td>
<td></td>
<td>Identifier for this pseudo-state</td>
</tr>
<tr>
<td>type</td>
<td>false</td>
<td>enum</td>
<td>Boolean</td>
<td>&quot;shallow&quot;</td>
<td>&quot;deep&quot;, &quot;shallow&quot;</td>
<td>Specifies the name of the event. This will be matched against the 'event' attribute of transitions.</td>
</tr>
</tbody>
</table>

Note: the event that will be placed at the back end of the session's internal event FIFO queue.

### `<if>`  `<elseif>`  `<else>`

- `<if>` is a container for conditionally executed elements.
- `<elseif>` is an empty element that partitions the content of an `<if>`, and provides a condition that determines whether the partition is executed.
- `<else>` is an empty element that partitions the content of an `<if>`. It is equivalent to an `<elseif>` with a "cond" that always evaluates to true.

### Attribute Details of `<if>`

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Attribute Constr.ts</th>
<th>Type</th>
<th>Def. It Value</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond</td>
<td>True</td>
<td></td>
<td>Conditional expression</td>
<td>'true'</td>
<td></td>
<td>A boolean expression</td>
</tr>
</tbody>
</table>

### Children of `<if>`

<table>
<thead>
<tr>
<th>Name</th>
<th>times</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;elseif&gt;</code></td>
<td>&gt;= 0</td>
<td>See below</td>
</tr>
<tr>
<td><code>&lt;else&gt;</code></td>
<td>&gt;= 0</td>
<td>See below</td>
</tr>
<tr>
<td>Executable content</td>
<td>&gt;= 0</td>
<td>Note that since <code>&lt;if&gt;</code> itself is executable content, nested <code>&lt;if&gt;</code> statements are allowed</td>
</tr>
</tbody>
</table>

Note(s): the behavior of `<if>` is defined in terms of partitions of executable content. The first partition consists of the executable content between the `<if>` and the first `<elseif>`, `<else>` or `/` tag. Each `<elseif>` tag defines a partition that extends from it to the next `<elseif>`, `<else>` or `/` tag. The `<else>` tag defines a partition that extends from it to the closing `/` tag. A partition may be empty. `<else>` must occur after all `<elseif>` tags.
When the `<if>` element is executed, the executor executes the first partition in document order that is defined by a tag whose 'cond' attribute evaluates to true, if there is one. Otherwise, it executes the partition defined by the `<else>` tag, if there is one.

The following is an example:

```xml
<if cond="cond1">
    <!-- selected when "cond1" is true -->
    <elseif cond="cond2"/>
    <!-- selected when "cond1" is false and "cond2" is true -->
    <elseif cond="cond3"/>
    <!-- selected when "cond1" and "cond2" are false and "cond3" is true -->
    <else/>
    <!-- selected when "cond1", "cond2", and "cond3" are false -->
</if>
```

**<foreach>**

The `<foreach>` element allows the executable content of a state to iterate through a collection in the data model and to execute the actions contained within it for each item in the collection.

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Attribute Constr.ts</th>
<th>Type</th>
<th>Def.It Value</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>true</td>
<td></td>
<td>Value expression</td>
<td>none</td>
<td>A value expression that evaluates to an iterable collection.</td>
<td></td>
</tr>
<tr>
<td>item</td>
<td>true</td>
<td>xsd:string</td>
<td>none</td>
<td>Any variable name that is valid in the specified data model.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>index</td>
<td>false</td>
<td>xsd:string</td>
<td>none</td>
<td>Any variable name that is valid in the specified data model.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Children**

<table>
<thead>
<tr>
<th>Name</th>
<th>times</th>
<th>Description</th>
</tr>
</thead>
</table>

The `<foreach>` element will iterate over a shallow copy of this collection.

A variable that stores a different item of the collection in each iteration of the loop.

A variable that stores the current iteration index upon each iteration of the foreach loop.
Executable content  >= 0  items of executable content. (Note that they are considered to be part of the same block of executable content as the parent <foreach> element.)

Note(s): The executor declares a new variable if the one specified by 'item' is not already defined. If 'index' is present, the executor declares a new variable if the one specified by 'index' is not already defined. If 'array' does not evaluate to a legal iterable collection, or if 'item' does not specify a legal variable name, the executor terminates execution of the <foreach> element and the block that contains it.

The executor acts as if it has made a shallow copy of the collection produced by the evaluation of 'array'. Specifically, modifications to the collection during the execution of <foreach> shall affect the iteration behavior. The executor starts with the first item in the collection and proceed to the last item in the iteration order that is defined for the collection (This order depends on the data model in use). For each item in turn, the processor assigns it to the item variable. (Note that the assigned value may be null or undefined if the collection contains a null or undefined item.) After making the assignment, the executor evaluates its child executable content. It then proceeds to the next item in iteration order. If the evaluation of any child element causes an error, the processor ceases execution of the <foreach> element and the block that contains it. (Note that there is no break functionality to interrupt <foreach>, however targetless and/or eventless transitions can provide sophisticated iterative behavior within the state machine itself).

### A1.3 The Data Model in the MIA-XML language

**<datamodel>**

<datamodel> is a wrapper element which encapsulates any number of <data> elements, each of which defines a single data object. Top-level <datamodel> is one occurring directly under the <mia_xml> element.

| Children |
|----------|----------|----------------|
| Name     | times    | Description   |
| <data>   | >= 0     | Each instance defines a named data element |

**<data>**

The <data> element is used to declare and populate portions of the datamodel.

| Attribute Details |
|-------------------|-------------------|----------------|-------------------|------------------|
| Name   | Required | Attribute Constr. | Type | Def. It Value | Valid Values | Description                  |
| id     | true     | ID                | none |             |             | The name of the data item   |
| expr   | false    | Expression        | none | Any valid value |             | Evaluates to provide the value of the data item. |
D5.1 - Design of components for understanding, dialogue management and feedback to the user

<table>
<thead>
<tr>
<th>type</th>
<th>enum</th>
<th>“number”</th>
<th>“string”</th>
<th>Provides the type of the data item</th>
</tr>
</thead>
</table>

**Children**

<table>
<thead>
<tr>
<th>Name</th>
<th>times</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;data&gt;</td>
<td>&gt;= 0</td>
<td>Nested data</td>
</tr>
</tbody>
</table>

Note(s):
1. If the 'expr' attribute is present, the executor evaluates the corresponding expression at the time specified by the 'binding' attribute of <mia_xml> and assigns the resulting value as the value of the data element. If the value specified for a <data> element is not correct, the executor flags an error.
2. The MIA-XML executor uses any values provided by the environment at activation time as defaults, in place of those provided in the “expr” attribute, for those elements contained in the <data> elements of the top-level <datamodel>s.

**<content>**

A container element holding data to be passed to an external service: by means of <send> or <invoke> tags. When evaluating the <content> element, if the 'expr' value expression is present it is evaluated first and the result is taken as the <content> element. If the evaluation of ‘expr’ produces an error, the empty string is used as the value of the <content> element. If the ‘expr’ attribute is not present, the children of <content> is considered (see explanation of the children of the tag).

**Attribute Details**

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Attribute Constr.</th>
<th>Type</th>
<th>Def.It Value</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>false</td>
<td>must not occur with child content</td>
<td>Value expression</td>
<td>none</td>
<td>Any valid value expression</td>
<td>A value expression</td>
</tr>
</tbody>
</table>

Note(s): the 'expr' attribute and child contents are alternative.

**<param>**

The <param> tag provides a general way of identifying a key and a dynamically calculated value which can be passed to an external service or included in an event.
Attribute Details

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Attribute Constr.</th>
<th>Type</th>
<th>Def. It Value</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>true</td>
<td>NMTOKEN</td>
<td>none</td>
<td>A string literal</td>
<td>The name of the key</td>
<td></td>
</tr>
<tr>
<td>expr</td>
<td>false</td>
<td>value expression</td>
<td>none</td>
<td>Valid value expression</td>
<td>A value expression</td>
<td></td>
</tr>
<tr>
<td>location</td>
<td>false</td>
<td>Variable name</td>
<td>none</td>
<td>Valid location expression</td>
<td>The name of the key</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>false</td>
<td>Type name</td>
<td>none</td>
<td>number</td>
<td>The type of the parameter</td>
<td></td>
</tr>
</tbody>
</table>

Note(s): the 'expr' and 'location' attributes are alternative. If the 'location' attribute does not refer to a valid location in the data model, or if the evaluation of the 'expr' produces an error, the <parameter> tag is ignored.

System Variables

- _sessionid. This is a system-generated id for the current MIA-XML session; such a variable is valid and constant until the session terminates.
- _name. This variable is bound at load time to the value of the 'name' attribute of the <mia_xml> element. Such a variable is valid and constant until the session terminates.

A1.4 External Communication in the MIA-XML language

<send>

<send> is used to send events and data to external systems, including other MIA-XML executors, or to raise events in the current MIA-XML session.

Attribute Details

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Attribute Constr.</th>
<th>Type</th>
<th>Def. It Value</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>False</td>
<td>Must not occur with 'eventexpr'</td>
<td>EventType.datatype</td>
<td>none</td>
<td>A string literal</td>
<td>A string indicating the name of message being generated</td>
</tr>
<tr>
<td>eventexpr</td>
<td>False</td>
<td>Must not occur with 'event'.</td>
<td>Value expression</td>
<td>none</td>
<td>Valid value expression</td>
<td>A dynamic alternative to 'event'. See note 1 below</td>
</tr>
<tr>
<td>Attribute</td>
<td>Default</td>
<td>Must not occur with</td>
<td>Value expression</td>
<td>Check</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>---------------------</td>
<td>-----------------</td>
<td>-------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>target</td>
<td>False</td>
<td>'targetexpr'</td>
<td>none</td>
<td>A valid target URI</td>
<td>The unique identifier of the message target that the platform should send the event to.</td>
<td></td>
</tr>
<tr>
<td>targetexpr</td>
<td>False</td>
<td>'target'.</td>
<td>Value expression</td>
<td>Valid value expression</td>
<td>A dynamic alternative to 'target'. See note 1 below</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>False</td>
<td>'typeexpr'</td>
<td>string</td>
<td>&quot;xml&quot;</td>
<td>The type of formatting of the POST payload</td>
<td></td>
</tr>
<tr>
<td>typeexpr</td>
<td>False</td>
<td>type.</td>
<td>string expression</td>
<td>Valid value expression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>False</td>
<td>'eventexpr'</td>
<td>EventType.datatype</td>
<td>A string literal</td>
<td>A string indicating the name of message being generated</td>
<td></td>
</tr>
<tr>
<td>idlocation</td>
<td>False</td>
<td>'event'.</td>
<td>Location expression</td>
<td>Any valid location expression</td>
<td>A dynamic alternative to id. See note 1 below</td>
<td></td>
</tr>
<tr>
<td>delay</td>
<td>false</td>
<td>'delayexpr'</td>
<td>Duration.datatype</td>
<td>A time designation</td>
<td>A string indicating the name of message being generated</td>
<td></td>
</tr>
<tr>
<td>delayexpr</td>
<td>False</td>
<td>delay.</td>
<td>Value expression</td>
<td>Valid value expression</td>
<td>A dynamic alternative to delay. See note 1 below</td>
<td></td>
</tr>
<tr>
<td>namelist</td>
<td>False</td>
<td>See note 3</td>
<td>List of location expressions</td>
<td>List of data model locations</td>
<td>A space-separated list of one or more data model locations to be included as attribute/value pairs with the message.</td>
<td></td>
</tr>
</tbody>
</table>

Note(s):
1. If this attribute is present, its actual value will be evaluated when the parent <send> element is evaluated and treat the result as if it had been entered as the static value.
2. Must not occur with 'delayexpr' or when the attribute 'target' has the value "_internal".
3. Must not be specified in conjunction with the <param> or <content> elements.
Children

<table>
<thead>
<tr>
<th>Name</th>
<th>times</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;param&gt;</td>
<td>&gt;= 0</td>
<td>this element is evaluated when the parent &lt;send&gt; element is evaluated: the resulting data is passed to the external service when the message is delivered.</td>
</tr>
<tr>
<td>&lt;content&gt;</td>
<td>&gt;= 0</td>
<td>this element is evaluated when the parent &lt;send&gt; element is evaluated: the resulting data is passed to the external service when the message is delivered.</td>
</tr>
</tbody>
</table>

Note(s):
1. Only one of 'event', 'eventexpr' and <content> must be provided; "namelist" or <param> cannot be provided along with <content>.
2. If 'idlocation' is present, an id is generated when the parent <send> element is evaluated; such an id is stored in the provided location.
3. If a delay is specified via 'delay' or 'delayexpr', such time interval will be waited for before sending the event: note that the evaluation of the send tag will return immediately. However all arguments to <send> are evaluated when the <send> element is evaluated, and not when the message is actually dispatched. If the evaluation of <send>’s arguments produces an error, the message will be discarded before attempting to deliver it. If the MIA-XML session terminates before the delay interval has elapsed, such message will be discarded without attempting to deliver it.

<invoke>
The <invoke element is used to create an instance of an external service.

Attribute Details

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Attribute Constr.ts</th>
<th>Type</th>
<th>Def. It Value</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>false</td>
<td>Must not occur with 'typeexpr'</td>
<td>URI</td>
<td>none</td>
<td>A string literal</td>
<td>The URI that identifies the transport mechanism for the message.</td>
</tr>
<tr>
<td>typeexpr</td>
<td>false</td>
<td>Must not occur with type.</td>
<td>Value expression</td>
<td>none</td>
<td>Valid value expression</td>
<td>A dynamic alternative to 'type'. See note 1 below</td>
</tr>
<tr>
<td>src</td>
<td>false</td>
<td>Must not occur with srcexpr see note 2</td>
<td>Duration.datatype</td>
<td>none</td>
<td>A time designation</td>
<td>A string indicating the name of message being generated</td>
</tr>
<tr>
<td>srcexpr</td>
<td>false</td>
<td>Must not occur with srcexpr see note 2</td>
<td>Value</td>
<td>none</td>
<td>Valid value</td>
<td>A dynamic alternative to</td>
</tr>
</tbody>
</table>
D5.1 - Design of components for understanding, dialogue management and feedback to the user

<table>
<thead>
<tr>
<th>namelist</th>
<th>false</th>
<th>See note 3</th>
<th>List of location expressions</th>
<th>none</th>
<th>List of data model locations</th>
<th>A space-separated list of one or more data model locations to be included as attribute/value pairs with the message.</th>
</tr>
</thead>
</table>

**Children**

<table>
<thead>
<tr>
<th>Name</th>
<th>times</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;param&gt;</td>
<td>&gt;= 0</td>
<td>Element containing data to be passed to the invoked service</td>
</tr>
<tr>
<td>&lt;finalize&gt;</td>
<td>0,1</td>
<td>Element containing executable content to massage the data returned from the invoked component.</td>
</tr>
<tr>
<td>&lt;content&gt;</td>
<td>&gt;= 0</td>
<td>this element is evaluated when the parent &lt;invoke&gt; element is evaluated: the resulting data is passed to the invoked service.</td>
</tr>
</tbody>
</table>

Note: Exactly one of src, param and <content> must be provided; However <param> may occur multiple times.

**<finalize>**

The <finalize> element enables an invoking session to update its data model with data contained in events returned by the invoked session. <finalize> contains executable content that is executed whenever the external service returns an event after the <invoke> has been executed. This content is applied before the system looks for transitions that match the event. In the case of parallel states, only the finalize code in the original invoking state is executed.

**Children**

<table>
<thead>
<tr>
<th>Name</th>
<th>times</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable content</td>
<td>&gt;= 0</td>
<td></td>
</tr>
</tbody>
</table>

Note(s):
1. the executable content inside <finalize> must not raise events or invoke external actions. In particular, the <send> and <raise> elements must not occur.
2. If one or more elements of executable content is specified, they will be executed each time an event is received from the child process that was created by the parent <invoke> element.
3. If no executable content is specified, the executor updates the data model each time an event is received from the child process that was created by the parent <invoke> element.
Specifically if the parent `<invoke>` element contains or one or more `<param>` children containing 'location' attributes, then for each such `<param>` element, the Processor will update the corresponding location with any return value that has a name that matches the 'name' of the `<param>` element. Thus the effect of an `<invoke>` with an empty `<finalize>` element and a `<param>` element with a 'location' attribute is first to send the part of the data model specified by 'location' to the invoked component and then to update that part of the data model with any returned values that have the same name (this implements parameter passing by “value and return” among the state machine and the external service). Note that the automatic update does not take place if the `<finalize>` element is absent as opposed to empty.

### A1.5 The DTD of the MIA-XML language as of Release 1.0

In the following the DTD of the mia-xml language supported by Release 1.0 is reported.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!ELEMENT mia_xml (state | parallel | final | datamodel)+>
<!ATTLIST mia_xml
  initial CDATA #IMPLIED
  name CDATA #IMPLIED
  xmlns CDATA #REQUIRED
  version "1.0"
  datamodel (proprietary) "proprietary"
  binding (early|late) "early">
<!ELEMENT state (onentry | onexit | transition | initial | state | parallel
  | datamodel | final)*)>
<!ATTLIST state id CDATA #REQUIRED
  initial CDATA #IMPLIED >
<!ELEMENT parallel (onentry | onexit | transition | state | parallel)*)>
<!ATTLIST parallel id CDATA #IMPLIED >
<!ELEMENT executableContent (send | assign | log)>
<!ELEMENT transition (executableContent)*>
<!ATTLIST transition
  event CDATA #IMPLIED
  cond CDATA #IMPLIED
  target CDATA #IMPLIED
  type (external | internal) "internal" >
<!ELEMENT initial (transition)>
<!ELEMENT final (onentry, onexit)*)>
<!ATTLIST final id CDATA #REQUIRED >
<!ELEMENT onentry (executableContent)*)>
<!ELEMENT onexit (executableContent)*)>
<!ELEMENT datamodel (data)*>
<!ATTLIST data
  id CDATA #REQUIRED
  type (integer|string) "number"
  expr CDATA #IMPLIED >
```
<!ELEMENT send (parameter)>
<!ATTLIST send
  event CDATA #IMPLIED
  eventexpr CDATA #IMPLIED
  target CDATA #IMPLIED
  targetexpr CDATA #IMPLIED
  type (xml | basichttp) "xml"
  typeexpr CDATA #IMPLIED
  id CDATA #IMPLIED
  idlocation CDATA #IMPLIED
  delay CDATA #IMPLIED
  delayexpr CDATA #IMPLIED
  namelist CDATA #IMPLIED>

<!ELEMENT param>
<!ATTLIST param
  name CDATA #REQUIRED
  expr CDATA #IMPLIED
  location CDATA #IMPLIED
  type (integer | string) "integer">

<!ELEMENT assign>
<!ATTLIST assign
  location CDATA #REQUIRED
  expr CDATA #IMPLIED>

<!ATTLIST log
  label CDATA #IMPLIED
  expr CDATA #REQUIRED >