

# A FRAMEWORK FOR TEACHING MULTIMODAL INTERFACE CONSTRUCTION

K. Rajendran  
University of Teesside  
Middlesbrough  
UK, TS1 3BA  
k.rajendran@tees.ac.uk

S.C.Lynch  
University of Teesside  
Middlesbrough  
UK, TS1 3BA  
s.c.lynch@tees.ac.uk

---

## ABSTRACT

*In this paper we describe an approach which aims to reduce the complexity associated with building multimodal systems to a level suitable for undergraduate and postgraduate computing students. To achieve this we specify an adaptable multiagent architecture for multimodal systems and provide a suite of general purpose, plug and play agents to handle essential tasks like speech I/O, fusion and semantic analysis. We outline the design of these agents and consider how they may be used by learners who are new to multimodal dialog systems.*

## Keywords

*Multiagent systems, multimodal interfaces, design patterns, teaching.*

## 1. INTRODUCTION

Multimodal dialogue (MMD) systems are currently a specialist domain and, due to steep learning curves, practical multimodal interface construction is not prevalent in computing courses. Even in research environments, the development of prototype systems presents a “demanding challenge” [10]. Much research into multimodality is focused on education [15, 17, 18] but these efforts aim to improve teaching and learning in general by using multimodal systems rather than advance the teaching of multimodal systems construction.

Teaching practical MMD systems presents several difficulties because of the complexity of their development. Firstly, recognition-based technologies are more difficult to process and are more error-prone than the discrete inputs from standard interfaces using devices like keyboard and mouse. Advances in fields such as speech recognition, gaze tracking and gesture recognition have reduced this complexity to some extent but incorporating these technologies into general computing applications is still challenging for educators and even more so for learners. The second difficulty in emulating more human-like communication is in creating a true multimodal interface in which a combination of modes (e.g. speech and facial expression) is used in parallel, either to supplement a primary mode of input/output

or to strengthen the communication through redundancy. This involves processes like multimodal fusion and fission which are specialist research areas. Thirdly, the motivation for these interfaces is often associated with dialog systems which are considerably different to traditional systems (those that process discrete commands normally associated with computer interfaces) since they require processes like conversation tracking and maintenance of context. There is also an inherent difficulty in building systems which are complex, distributed and involve heterogeneous components since these typically require a variety of expertise areas.

In addition, for any significant MMD application, the difficulty is great enough to necessitate support from base technologies, platforms and middleware. A number of architectures and generic components for building MMD systems have been suggested e.g. [6, 9, 10, 12] but although these support technologies have simplified development they are almost entirely in the research domain and there are several challenges to overcome before they may be deployed as easy to learn, usable products that can be adapted for the purposes of MMD education.

All of these issues make it difficult to design learning modules for MMD which provide students with practical experience of building systems as well as covering the theoretical aspects. Additionally it means that MMD systems are typically beyond the scope of project work for undergraduate students and even postgraduate students following taught courses. A compromise solution is to focus practical work on one (or a small number) of specific subcomponents of MMD systems (such as speech processing or gesture for example) but there are disadvantages with this approach:

- (i) students fail to experiment with a wider range of MMD subcomponents;
- (ii) since students only focus on a subset of the components required for MMD they do not have the opportunity to progress to building applications supported by multimodal interfaces and examine important usability issues associated with them;
- (iii) although a course which examines specific MMD components may impart in-depth knowledge

about those components it will not also investigate different design patterns for collaboration between components.

An alternative approach is to allow students to assemble MMD systems using pre-existing components, tailoring and/or reconfiguring them to their specific needs, this requires systems and tools especially designed for teaching.

Our aim is to make the development of multimodal dialogue systems more accessible to educators, students and the general programming community by increasing the quality and usability of the supporting technologies. We approach this in two ways (i) by considering the paradigm of multiagent systems development and its application as a base technology for supporting multimodal dialogue (ii) by supplying a number of agents which provide specific functionality for MMD (agents to handle speech I/O, fusion, etc).

Section 2 gives an overview of multiagent systems as a supporting technology and introduces the agentware used for this research described here. Section 3 identifies a skeletal but extensible agent architecture for MMD systems and also smaller architectural patterns. Sections 4-7 describe a suite of agents to handle some of the key tasks involved with building multimodal interfaces. Sections 8-9 conclude with evaluation of this approach which has been conducted through ethnographic studies of computing students who have no previous knowledge of MAS or MMD systems.

## 2. MULTIAGENT SYSTEMS (MAS)

MAS [22] is a highly active and emerging area of research that is viewed not as a particular technology or platform but as an engineering technique comparable to object oriented software engineering. Agent-orientation provides a level of abstraction above that of object orientation and is supported by several platforms, frameworks, languages and methodologies. Researchers have shown the advantages of applying MAS techniques to complex systems development in general [11] as well as MMD systems in particular [16]. Like MMD systems, MAS development is also only moving slowly into the wider community, it too has barriers to wider adoption which stem from usability factors as well as issues of re-use and extensibility [3].

Marian et.al. [14] discuss various reasons why agent platforms are not in general use; these include poor usability due to complexity and a lack of interoperability and extensibility. At a more specific level, a lack of standard concepts surrounding agents and the specialised nature of agent platforms are major barriers [3]. MMD systems based on agents may have different views on agency which impact on their concepts and capabilities and may not conform to the general

paradigm of agency, specified by agent researchers [7, 11, 22]. This can make it difficult to use MAS platforms for MMD applications and can also require a larger initial learning effort. Many attempts to build MMD systems using MAS begin with a survey of current technologies which often fails to reveal a suitable MAS platform and subsequently results in the construction of a suitable, but tailor-made, middleware solution e.g. [8, 18]. Generality in MAS infrastructure has been the target of research with some results (e.g. Open Agent Architecture, JADE, Boris) and some multimodal systems are based on these [4, 16] but despite these successful applications of MAS technology, MMD systems still require the involvement of subject specialists and are not within the grasp of the student community.

The generic platform Boris ([www.agent-domain.org](http://www.agent-domain.org)) which adheres to the general MAS paradigm was selected as the basis for development work and a number of student groups have been involved in the evaluation of the results. The selected platform offers generality for wider use, extensibility as well as usability. There are specified models of extension for different application areas as well as for adding support for different programming languages. The toolkit associated with the platform [13] provides several visual tools to aid development and may be extended to provide increased support for new application areas. This means that MMD specific tools can be built as extensions of the MAS toolkit.

## 3. ARCHITECTURES

Example architectures are important reference tools for learners, especially for systems with complex internal structures. In a multiagent system, the MMD components are in the form of agents and the structure of the system, defined by the agent groupings and hierarchies as well as interactions between agents and/or agent groups, are of primary importance. Patterns can be seen in the arrangement of agents, interaction scenarios and higher level collaborations which have different advantages in different situations. Architectures may be presented at different levels of abstraction from a high level view of the system to expanded views which show more detail. This is a necessary property since students are often required to make in-depth study of specific areas but still have a general understanding of a whole system. Due to the complexity of MMD, most learners will not gain in depth understanding of all system components and their interactions but an abstract view of the entire system is possible. The contributions that can be made by reusable abstract patterns have also been suggested for other application areas of multiagent systems [13, 14].

There are various criteria to be considered when specifying example architectures:

- i. they should be based on abstract and widely known concepts rather than specific features;
- ii. they should be simple enough for initial learners to comprehend;
- iii. they must be flexible enough to extend to real applications that are complex and large.

We present the diagram below (Fig.1) as a high level generic MAS architecture for MMD. The diagram shows speech input and GUI input but is extendible to include additional and/or alternative input modalities.

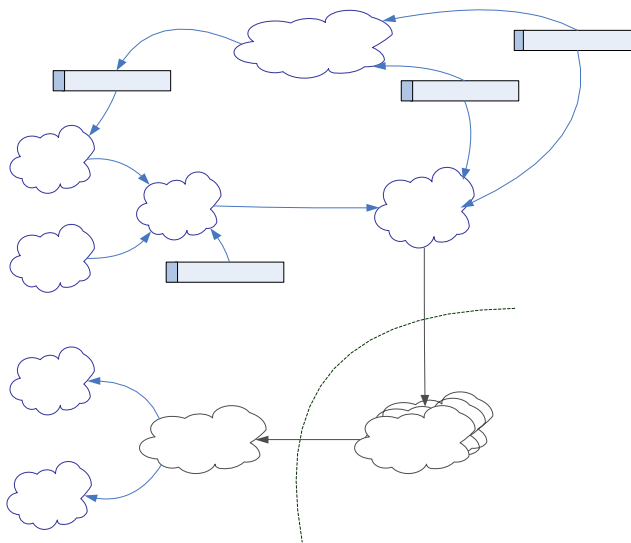


Figure 1. Example architecture for an MMD system.

This diagram shows the system only in terms of high level processes (any databases, models etc used by the processes are excluded). A brief explanation of the agents in the diagram above follows:

- the Grammar Synchronisation utility is a tool to reduce the learning and effort required to produce grammars. As can be seen, the language processor as well as the speech input requires separate grammars. This is because speech processing and language processing are separate areas of research and currently there is little integration. Within the area of speech processing, some standards for grammar have emerged, notably the Java Speech Grammar Format. Language processors are not standardised and because of their complexity are not developed specifically to support individual applications so, almost always, MMD systems use existing language processors and these have their own specific grammar formats;
- fusion receives input data from input modality components and combines them according to a set of specified rules. The result of fusion is forwarded to the component that extracts semantic information (the language processor).

GUI  
out

semantic information (the language processor). All input is sent to the fuser including input that may be uni-modal and does not require fusion. In the discussion which follows, the fuser is driven by a set of rules which allows the process of fusion to be decoupled from the specification of what can be fused and when;

- language processing: the user input, fused into a single input statement is processed to derive semantics. The language processor uses specifications of a lexicon and grammar to extract semantics;
- various tasks may be required after extracting semantics (including pragmatics, dialog management, etc) the nature of these tasks are typically dependant on the application domain and are shown as a single agent cluster in the diagram.

Figure 2 and 3 show "sub-architectures" that can be used as design patterns for building the different parts of the system either by developing them or by using plug and play agents and/or agent clusters.

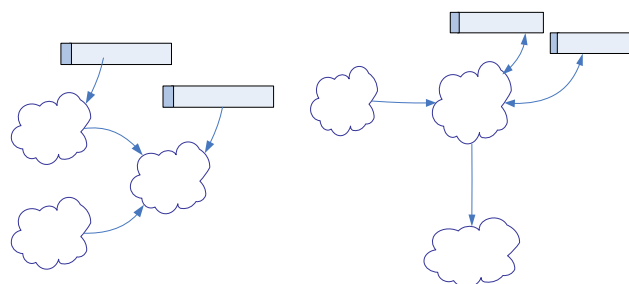


Figure 2. Input fusion.

Figure 3. Semantic analysis.

Each of these are independent architectures in their own right with their own structure and interactions and so they may be considered in isolation and developed as separable agent clusters which can then be linked together to create an entire application. Students are faced with a smaller learning curve if provided with design patterns and a range of pre-existing agents (like those described next) which fill the roles of components in these patterns.

#### 4. SPEECH AGENTS

One way in which MAS technology can support complex modalities that may require specific expertise is to provide plug-and play agents that hide the complexities but also allow for easy extensions. The advantage of using generic platforms and additional individual units of software (for example, objects and agents) to abstract out the complexity of handling input modes has been suggested in dated literature [2, 20], yet easy to use, specialised technology for generic agent platforms is not readily available for multimodal systems.

Dialogue  
manager \*

\* dialog management involves application control

The agents presented here offer some specific advantages when coupled with a usable platform; they are useful for building standard interfaces but perhaps more importantly they are useful as an initial learning tool and for experimentation at advanced levels of courses or project work. The more complex agents are extensions of the simpler agents and adhere to the same models and the extensible agents aid transition from learning to real systems development.

In our experiments with MAS support for MMD, we chose speech input/output which is a popular and highly supported modality as the focus of the study. Speech recognition also introduces the need to develop language artefacts like grammars and lexica which could not be omitted from a MMD architecture which claims to be of general use.

Generic Boris agents for speech input and output are built in Java to communicate with speech engines using the Java Speech API standard and so can be used with any JSAPI compliant engine. IBM's ViaVoice and SpeechForJava were used during the development period. Boris agents are object based and so speech agents are object generalisations of the Boris Agent and form a hierarchy of agents (Figure 4).

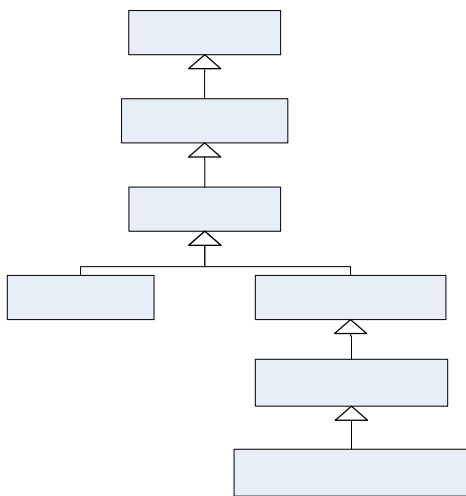


Figure 4. Object hierarchy of speech output agents.

Speech agents have all of the capabilities of generic agents including communicating with other agents in a MAS. The list below describes their additional functionality:

- **SpeechOutAgent.** This agent is not executable but encapsulates the basic functionality of generic speech output. It is capable of receiving agent messages and of communicating with the speech engine. It may be instantiated and used as show here:

```

SpeechOutAgent a
  = new SpeechOutAgent("speaker");
a.speak( "hello world!" );
  
```

This agent provides a number of functions that can be called for tailoring the speech for example to set the pitch of the speech, rate of speech, volume etc.;

- **AutoSpeechOutAgent.** This is an executable plug and play agent that can be added to MAS. It has the capability to communicate with the synthesiser and sends the text of all messages it receives to the synthesiser to be spoken. Once this agent is connected to the MAS, any other agents in the MAS may send it text to be spoken. Applications will generally require more functionality from a speech output agent than simply speaking the text contained in messages so this agent can be extended for different processing of messages but in its default form may be used as shown below...

```

// create an AutoSpeechOutAgent
Portal p = new Portal("p", false );
AutoSpeechOutAgent autoSpeak =
  new AutoSpeechOutAgent("speaker");

// use AutoSpeechOutAgent
someAgent.sendMessage("speaker",
  "hello world");
  
```

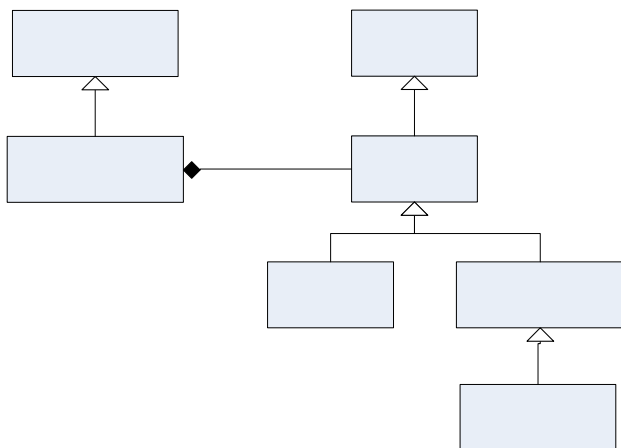
- **GUISpeechOutAgent.** This is an agent that has the capabilities of the AutoSpeechOutAgent but also presents a GUI based interface through which the user of the application can change the properties of speech output, for example: volume, speech of speech and voice.

Multimodal interfaces using speech may synchronise with facial images and lip movements, which are often separate specialised applications on their own although there are benefits to be gained from coupling this with the speech output software. On a simpler level, they may be animations that are not as synchronised as human lip and facial expressions e.g. images of diagrams or simple animations to indicate the agent is speaking. Learners benefit from support of simple synchronisation tasks involving speech and animation. The following are variations of the AutoSpeechOutAgent that provide visual support during speech:

- **AnimatedSpeechOutAgent.** This is a convenience agent that uses a different JAVA thread to produce speech. Programmer-developed animations which may include code that does not release the current thread can then be included in subclasses without interrupting speech output;

- **PaintingSpeechOutAgent.** This agent accepts a drawable object on which graphics painting can be done. The user prepares the object and any graphics based animations for the object. The agent uses a timer to invoke painting on the object;
- **ImagePaintingSpeechOutAgent.** This agent accepts a string of images that will be used for animation while the synthesiser is working. It is tailorable with specification of size and location.

Speech recognition technology is rapidly making significant progress but several challenges can be identified [5]. Some of these are to do with speed and accuracy of recognition, others relate to a lack of standards and suitable architectures that allow reuse across applications. Our work has focused on the latter challenge and considers the development of reusable components in the form of agents. Some components like sample grammars and listener objects that process sound have also been provided to further support learners. In speech enabled conversational systems, grammars are used at two different levels. They are used by the language understanding and generation software as well as dialog managers. These grammars may have semantic as well as lexical information. These tasks are difficult tasks in their own right and increases the level of knowledge students must already possess before building applications involving speech. Components like these release learners from this burden and elements of multimodality can be studied using speech but without requiring in-depth knowledge of grammars, recognition etc. Figure 5 below shows the inheritance hierarchy of speech input agents.



**Figure 5. Object hierarchy of agents for speech input.**

Briefly these agents function as follows:

- **SpeechInAgent.** This is a non-executable agent that contains methods for setting up the recogniser, grammar and for processing input. Speech agents allow the runtime specification of grammar files. These are simple grammars

that do not hold semantic information and are used by speech recognition software for narrowing the range for recognition possibilities which reduces errors in recognition. A grammar will specify a vocabulary of words and grammatical structures like phrases and sentences. The figure below shows an example grammar file written in the JAVA Speech Grammar Format (JSGF). It allows the recognition of sentences like “pick up the yellow box” and “put the red box on the green box”;

```
grammar boxes;
<color> = red | blue | pink |green;
<object> = the <color> box | prism
          | it;

public <sentence>
    = pick up <object> [and
                        <continue>];

public <continue>
    = put <object> on the <object>;
```

- **AutoSpeechInAgent.** This is an executable agent which delivers recognised speech in textual form. To manage speech input, learners may simply add this agent to their system without any knowledge of speech-based programming or of the Java Speech API;
- **AutoTimedSpeechInAgent.** This agent captures timing information such as the time in milliseconds that each recognised word was heard by the microphone. This information is important for multimodal temporal fusion which uses timestamps on words to associate them with other input like lip movements or mouse clicks. Other information like timestamps for unrecognised words and begin and end of grammatical structures like sentences as defined in the grammar file are also available.
- **GUITimedSpeechInAgent.** This agent presents a GUI interface which provides functions like monitoring the recognition process. This input agent is especially easy to use as it can be added to a MAS without any reconfiguration, a grammar file can be specified or re-specified at run time and all information can be sent to another running agent in the MAS. This recipient is specified through the GUI interface at run time making this agent useful in test situations.

## 5. FUSION

Fusion is an intricate process which needs to consider temporal issues as well constraints presented by the use of different modalities. Our observations of students show that, though they can outline ideas about the nature of a fusion process, they require significant support in order to produce a

coherent design. We find that fusion (along with semantic analysis and, when appropriate, dialog management) is one of the main barriers for students' understanding of practical MMD.

The fusing process is necessarily highly dependent on the nature of input modalities used and format of data supplied by agents connected to these sources of input. This implies that a fuser will be closely coupled with input modes and, in many cases, to the application domain. Our aim, however, is to provide a generic fuser which will provide much of the necessary functionality (at least for small to medium scale systems) regardless of the input modes chosen and independent of the application domain. This presents a dilemma.

Broadly there are two approaches to fusion: early and late. Early fusion takes syntactic tokens from each of the input modalities and combines them to form new syntactic tokens. Late fusion assumes that input from each modality has been semantically analysed and combines semantic fragments to produce larger scale (more complete) semantics. In practice many researchers discuss hybrids between early fusion and late fusion, which either take input that has had some semantic processing on some input modes or may repeat fusing at different stages in the process of semantic analysis.

Specifying a generic fuser which will apply late fusion presents problems. Semantics are typically (and often necessarily) application dependent. Conversely, fusing based solely on low-level syntactic tokens is more easily achieved but these tokens are often typically highly related to specific input modes.

After various trials, we have developed a fuser in the following form. The core fusing process is independent of any application domain and can achieve early fusing and/or hybrid fusing (where input sources may have been semantically processed). It operates using a set of fusion rules which drive the fusing process. The advantages of separating information about how to fuse inputs from the actual task of fusing them have been recognised by other authors e.g. [21]. In the model presented here, the ruleset which is application dependant is separate from the fusing agent which means that students working on practical development projects are free to code application specific concepts into their rules without the need to reprogram the fuser itself.

Fusion rules specify how input from different modalities can be fused and the temporal and/or modal constraints which should apply. In a manner similar to the dialog manager described by Portillo et. al. [19], the fuser maintains a *multimodal input pool* of input data. Input from different sources that overlap temporally (or nearly overlap) is considered as potentially multimodal and the fuser will attempt

to combine the input according to its fusion rules. Fused input is then sent to a language processor.

Note: we recognise various issues relating to fragmented speech input (discussed by Bell et. al. [1] and others) and have had positive results from experimentation with feedback loops between the language processor and fuser. In this model the language processor will check the fuser for additional input when it detects incomplete/unusable input fragments, this is an area of continued investigation.

The fuser is implemented as a cluster of agents which provide the core rule-based engine and also the following utilities...

- a collection of small utilities to reformat data from input streams into that required by the fuser. One of these transforms the type of speech data created by the speech input agents, other transform data typically generated by other types of input;
- a rule compiler which allows common types of fusion rule to be more easily specified.

The core fuser agent combines fragments of input data and derives all possible interpretations of the user utterance. In a dialog system, a language processor (or a dialog manager in smaller-scale systems) will then be consulted with the fused results in order to resolve any ambiguities based on semantics, context, etc. In some cases a knowledgebase may also be consulted for resolution of ambiguities regarding application dependant world states.

These tasks are sometimes incorporated into the fusion component but it is important for reusability that complex systems such as these are decoupled as much as possible. This decoupling also aids in the reduction of learning curves for novices who wish to familiarise themselves with a new technology since they are able to focus on each individual processing task in isolation. In addition, novices need architectures which support simple systems development as well as more complex systems. The development of these simple systems can be impossible when they are based on architectures that have been designed to meet specific and intricate requirements.

## 5.1 The Fuser Agent

The fuser agent typically receives information about spoken input and input from other modalities (gesture, etc). Each input of spoken data to the fuser represents one utterance by the user. An utterance relates to one command or sentence the user says. These are identified by the speech input agents through reference to a defined grammar. The input consists of the words that were spoken and their start and end times as recorded by the speech agents. These are the times that each word

was spoken. The gesture input contains details of the objects that were referenced by the user (by pointing, clicking etc.) and the times associated with each reference. With pen-based input for example, this time would be the time the user pointed at an object with the pen.

There are various different ways that the fusion process can be developed but the goals are the same: to combine compatible, time-coincident data from different input sources when that data refers to the same semantic element.

As well as a need for the fusing process to be transparent and understandable (for the benefit of students) there is also a requirement that any resulting solution is reusable across different applications – we are interested in building a generic agent which can be reconfigured as necessary. We achieve this reusability by providing a basic inference mechanism which is guided by fusion rules that are declarative in nature and aim to be reasonably easy to write. In practice (as is examined later) fusion rules can be abbreviated. Initially we consider an unabbreviated rule which specifies the preconditions for combining data from a speech input source with data from GUI input and how the combined structure should be represented.

## 5.2 Rule Structure

Consider the following case... the user says "that" and clicks over an object at the same time. Among other tuples/facts describing the multi-input utterance will be something like...

```
(speech-in (that 1 5 20))
(gui-in g0 type $obj)
(gui-in g0 time 12)
(gui-in g0 target blue-block)
```

The first tuple is from a speech input agent named speech-in and relate to the user speaking "that". The numbers 2, 15, 25 have the following meaning...

- 1 the edge number, in this case it means that "there" was the 2nd word spoken (edge numbers start from zero);
- 5 the start time – the time when the recogniser started to receive the phonemes that make up the word (time data has been simplified for the sake of discussion);
- 20 the end time – the time the last phoneme was completed.

So the spoken word "that" is smeared over times 5-20 with a GUI click arriving at time 12.

A fusion rule could describe the following: when the word "that" is time-coincident with a mouse click and that click is over a GUI feature of type \$obj rewrite the tuples describing this event as a new

fused tuple describing the event, deriving the single tuple below from those shown above.

```
(fused ($obj blue-block 1 5 20))
```

We build fusion rules by defining their preconditions and effects. Preconditions are specified in two ways...

1. matching forms: patterns of tuples which have to be present in the source data;
2. guards: mathematical or other cross pattern/tuple relationships which need further processing to determine their validity.

the preconditions for the rule described above can be written as follows...

```
matching preconditions
(speech-in (that ?edge ?start ?end))
(gui-in ?click type $obj)
(gui-in ?click target ?target)
(gui-in ?click time ?t)
guards
(<= ?start ?t ?end)
```

The effects of a rule can be written in terms of tuples to add and to delete from the input data...

```
add
(fused
($obj ?target ?edge ?start ?end))
delete
(speech-in (that ?edge ?start ?end))
(gui-in ?click ==)
```

Rules encoded in this form can be applied using a mechanism similar to that used for GPS style rules and collections of rules can be used to form a *legal move generator*. A legal move generator (LMG) is a function which takes a state as an argument and returns a set of successor states. Here we consider a LMG which takes a set of tuples describing partially fused input as a state and returns new sets which each have one additional fusing described in them.

A search process using this LMG takes an initial set of unfused tuples and finds new sets of tuples which have all possible fusings satisfied. The search filters out solutions which have any key tuples (those which must be fused) still present. Key tuples are optionally declared (with the rules) as those from some specific sources with specific structure.

## 5.3 Compound Rules

Agents providing input from modalities like GUI input may choose to process that input to varying extent but the fuser agent supplied assumes that the speech input has not been processed (ie: it is a string of words). Though it is possible to write fusing rules (in the form described above) to work with various structures, our default assumption is that all language processing (syntactic and semantic analysis, etc) will occur after fusing. There is one necessary exception to this relating to morphology:

where compounding of words can result in a single form which can be fused with tokens from other inputs. Consider sub-phrases like "this thing" or "that place", these can sensibly be combined and then fused with other input. The process of compounding (combining) words needs to occur before or during the fusing process otherwise fusion may not generate the necessary results. To facilitate this, the fuser provides an alternative style of fusing rule.

The rule below fuses edges from the speech input agent representing the words "that" and "thing" into a single form if they occur in the correct order and have no other speech input tokens between them (ie: their edge numbers are continuous).

```
matching preconditions
  (speech-in (that ?e1 ?st1 ?end1))
  (speech-in (thing ?e2 ?st2 ?end2))
guards
  (chk ((= (1+ ?e1) e2)))
additions
  (speech-in (that ?e1 ?st1 ?end2))
deletions
  (speech-in (that ?e1 ?st1 ?end1))
  (speech-in (thing ?e2 ?st2 ?end2))
```

#### A Rule Compiler

We have identified two types of rule above (i) fusing from multiple sources (ii) fusing from a single source. Both rule types have a stereotypical structure so it is possible to further simplify the process of rule construction. A rule compiler agents allows abbreviated rules to be specified as follows (the first rule is a single modality compound rule, the other two are cross-modality fusion rules).

```
(rule that-thing
  (compound speech-in that thing
    => that))
(rule that-click
  (fuse (speech-in that)(gui-in $obj)
    => (fused (gui-in type)
      (gui-in target))))
(rule there-click
  (fuse (speech-in there)
    (gui-in $place)
    => (fused (gui-in type)
      (gui-in target))))
```

### 5.4 Input Formats

Input agents (like those for speech input and GUI input) simply accept data and send it to the fuser. As described above, the fuser processes sets of tuples and assumes that it will receive data in this format. However this is not the same format that data will always be most obviously gathered by the input agents. To modify the structure of data we provide reformatting utilities which logically sit between the input agent and the fuser but are, in practice, provided with the fuser as part of its cluster of agents. One of the data reformatting agents

transforms data created by the speech input agent, others transform data typically generated by other types of input. The separation of data formatting provides greater modularity and more flexible plug and play architecture so, depending on their objectives, students may choose to use varied agents for input or other fusers without having to modify these agents.

The example below illustrates how speech data is transformed...

```
raw data from speech input
  (speech-in (put 0 5) (that 10 15)
    (there (20 25) ))
transformed speech data
  (speech-in (put 0 0 10) )
  (speech-in (that 1 5 20) )
  (speech-in (there 2 15 25) )
```

Here, the data is in a number of shorter tuples rather than a single structure. Note also that, for speech input the data reformatting also adds edge numbers and stretches start and end times for words to include silent times. These silent times are included in the fusion process to identify clicks that occur in pauses before or after certain words but can, none-the-less, be fused with them. The provision of *input format agents* allows students to experiment with different integration patterns by modifying these agents".

## 6. SEMANTIC ANALYSIS

The language processing agent provided is based on Lkit, a Natural Language Processing toolkit (available at [www.agent-domain.org](http://www.agent-domain.org)). Lkit includes a parser for carrying out syntax analysis and a rule based engine for semantic processing. All language-specific and application dependent structures for syntax analysis are coded into a lexicon and set of grammar rules. The semantic processing rules are also provided as part of the grammar.

A lexicon and grammar suitable for processing simple sentences like "the cat ate the rat" could be written as follows...

```
lexicon
  (a      det  (quantifier any))
  (cat    noun (small feline) )
  (ate    verb (kill consume) )
  (rat    noun (small rodent) )
  (the    det  (quantifier specific))

grammar
(s1 (sentence
  -> noun-phrase verb-phrase)
  (actor . noun-phrase)
  (action . verb-phrase.action)
  (object . verb-phrase.object))
(np (noun-phrase -> det noun)
  (det . noun))
(vp (verb-phrase -> verb noun-phrase)
  (action . verb))
```

For "the cat ate the rat" this would generate semantics as...

```
(actor (quantifier specific)
      (small feline))
(action (kill consume))
(object (quantifier specific)
      (small rodent))
```

The agent provided as the default language processor is a version of Lkit which has been wrapped in the form of a Lisp agent and will accept data in the form produced by the fuser. The other specialised feature involving Lkit is a utility agent which allows grammars and lexica to be shared with a speech recogniser.

## 7. GRAMMAR SYNCHRONISATION

The language processor uses a grammar and lexicon to deconstruct a sentence and ultimately to build some form of semantic representation which captures its meaning. The speech recogniser uses a grammar and lexicon to improve its recognition by placing constraints on the sequence of words which can legally occur. This helps the recogniser select valid labellings for the phonemes which it decodes.

Unfortunately the grammar/lexicon used by language processors is not compatible with that used by recognisers. In addition to allowing grammar builders to specify semantics, language processors typically provide a richer notation for expressing grammar rules and support a greater level of complexity for rule forms. This means that students must become familiar with two different forms of language specification. For researchers who will spend considerable time modifying their language specifications this may not cause significant problems but for students wishing to experiment with MMD concepts as only a part of their wider programme of study it introduces the burden of learning additional syntactic constructions and also increased possibilities for errors – we have observed students using incorrect grammar formats or mixing formats.

To overcome these difficulties we supply an agent which will take a grammar and lexicon specified for the language processor and automatically produce a grammar file for a recogniser. An example of an automatically generated grammar in the format of the Java Speech Grammar Specification (JSGF) is shown below. This is equivalent to language processor grammar presented in the previous section.

```
grammar autogenerated219;
<det> = a | the; <noun> = cat | rat;
<verb> = ate;
<np> = <det> <noun>;
<vp> = <verb> <np>;
```

```
public <sentence> = <np> <vp>;
```

## 8. EVALUATION

Evaluations of MMD systems usually focus on the usability of the final multimodal application from the perspective of the end user, e.g. as in [18]. Therefore, although many systems evaluate successfully, the usability of the architectures they present may be poor from a development perspective making them difficult for learners. If we wish to increase the exposure of MMD in education an alternative focus for evaluation is required. In order to gauge the relevance of our research output for the wider educational communities, we have focused on the usability of the underlying architecture and of the re-usable multimodal components from the perspective of the novice developer. In particular we are interested in whether the approach we have taken will allow students (computing final year undergraduate and postgraduate) to successfully experiment with practical deployment of multimodal systems.

Evaluation was conducted in two phases with small user groups of students who had some programming experience (in Java) but were unfamiliar with the concepts of multiagent systems, speech technology or MMD. In phase-I, groups of students were given tasks which involved developing applications using speech I/O agents. The groups were given an introduction to the general concepts of MAS and to the API of agents provided (including an outline of the speech agents). The group successfully developed applications involving speech without further study of speech technologies or MASs but were unable to present their work in the context of MMD and could only provide theoretical discussion of systems architectures and key processes such as fusion.

In phase-II, groups were given an introduction to MMD and an outline of the example architectures. They were also given instruction on writing fusion rules and the grammar/lexicon used by the language processor. The groups were required to configure the necessary sub-system to process multimodal input, this involved selecting a suitable systems architecture, writing fusion rules and grammar rules but they were not required to manage dialog. In contrast to Phase-I results, Phase-II students were more able to discuss system architectures from a practical perspective and appreciate the complexities of processes like fusion even though they had only used these (not built them).

The problems encountered by students were less concerned with the underlying architecture or the multimodal nature of the input than the difficulty of writing grammar and fusion rules to handle human-like language. The extent of this difficulty varied depending on whether the target group of students

had some prior exposure to the concept of rules based on tuples and using matching expressions to define their effects.

Although more investigation is needed to judge its suitability for larger scale systems, both students and tutors commented positively that providing a suitable MMD architecture had eased the burden of teaching and learning.

## 9. CONCLUSION

Our primary aim is to investigate how we can reduce the complexities of MMD development in order to allow students to experiment with constructing multimodal interfaces. To achieve this we have specified a generic architecture for MMD systems based on a multiagent approach and provided an API of agents for tasks such as speech I/O, fusion and language processing. This allows students to experiment with systems at a higher level of abstraction by reconfiguring systems architectures and their subcomponents. While more discussion is needed within the community about the merits of different architectures and the suitability of different agent-based tools, the results of our work suggest that the barriers for wider development of MMD in education can be reduced.

## 10. REFERENCES

- [1] Bell, L., Boyle, J. and Gustafson, J., *Real-time Handling of Fragmented Utterances*. NAACL Workshop on Adaption in Dialogue Systems, Pittsburgh, PA. (2001).
- [2] Blattner, M., M. and E. P. Glinert. Multimodal Integration. *IEEE MultiMedia* 3(4): 14-24. (1996).
- [3] Bordini, R. H. et al., A Survey of Programming Languages and Platforms for Multi-Agent Systems, *Informatica*. 30, 33-44 (2006).
- [4] Coronato A. and Pietro, D. G., *Middleware Services for Multimodal Interactions in Smart Environments*, In proceedings of the 6th Annual IEEE International Conference on Pervasive Computing and Communicatons, 2008.
- [5] Deng, L. and Huang, X. Challenges in adopting speech recognition. *Communications of the ACM*. 47(1): 69-75. (2004).
- [6] Elting, C. et al., *Architecture and implementation of multimodal plug and play*, in Proceedings of ICMI, Vancouver, British Columbia, Canada, 2003. ACM.
- [7] Ferber, J., *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, Harlow (1999).
- [8] Fernandes, V. T., et al., *Extensible middleware framework for multimodal interfaces in distributed environments*, in Proceedings of ICMI 2007. Nagoya, Aichi, ACM, Japan, 2007.
- [9] Galaxy Communicator, "[http://communicator.sourceforge.net/]."
- [10] Herzog, G., et al., *MULTIPLATFORM testbed: an integration platform for multimodal dialog systems*, in Proc. HLT-NAACL workshop on Software engineering and architecture of language technology systems - Volume 8: ACL, (2003).
- [11] Jennings, N. R. An agent-based approach for building complex software systems, *Communications of the ACM*. 44, 35-41 (2001).
- [12] LuperFoy, S., et al., *An architecture for dialogue management, context tracking, and pragmatic adaptation in spoken dialogue systems*, in Proceedings of the 17th international conference on Computational linguistics - Volume 2. Montreal, Quebec, Canada: Association for Computational Linguistics, 1998.
- [13] Lynch, S. and Rajendran, K., Providing Integrated Development Environments for Multi-Agent Systems, in *Multiagent System Technologies*. LNCS, 5244, 123-134. Springer, Berlin / Heidelberg, 2008.
- [14] Marian, T., et al., *A framework of reusable structures for mobile agent development*, In Proc. of IEEE INES Conference, Cluj-Napoca, Romania, 2004.
- [15] Massaro, W. D., *Just in time learning: implementing principles of multimodal processing and learning for education*, in Proceedings of ICMI. Nagoya, Aichi, Japan, 2007. ACM.
- [16] Oviatt, S. and Cohen, P., Perceptual user interfaces: multimodal interfaces that process what comes naturally, *Communications of the ACM*. 43, 45-53 (2000).
- [17] Oviatt, S., *Human-centered design meets cognitive load theory: designing interfaces that help people think*, in Proceedings of the 14th annual ACM international conference on Multimedia. Santa Barbara, CA, USA, 2006. ACM.
- [18] Pietrzak, T., et al., *The micole architecture: multimodal support for inclusion of visually impaired children*, in Proceedings of ICMI. Nagoya, Aichi, Japan, 2007. ACM.
- [19] Portillo, P. M., Guillermo, P.G., and Carredano, G.A. *Multimodal fusion: a new hybrid strategy for dialogue systems*. Proceedings of ICMI. Banff, Alberta, Canada, 2006. ACM.
- [20] Srinivasan, S. and Vergo, J. *Object oriented reuse: experience in developing a framework for speech recognition applications*.

Proceedings ICSE. Kyoto, Japan, 1998. IEEE Computer Society.

[21] Sun, Y., Chen, F and Chung, V (2006). *QuickFusion: multimodal fusion without time thresholds*. In Proceedings of the Nicta-Hcsnet

Multimodal User interaction Workshop., Sydney, Australia, 2006. ACM.

[22] Sycara, K., Multiagent Systems, *AI Magazine*. **10**, 79-93 (1998).