Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Master's Program in Computer Science

Master's Thesis

Realtime generation of multimodal affective sports commentary for embodied agents

submitted by
Martin Strauss
on October 11, 2007

Supervisor
Prof. Wolfgang Wahlster
Advisor
Dr. Michael Kipp

Reviewers
Prof. Wolfgang Wahlster
Dr. Michael Kipp

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, October 11, 2007 ____________________
Martin Strauss

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, October 11, 2007 ____________________
Martin Strauss

Abstract

Autonomous, graphically embodied agents are a versatile platform for information presentation and user interaction. This thesis presents ERIC, a homogeneous agent framework that can be configured to provide real-time running commentary on a dynamic environment of many and frequent events. We have focused on knowledge reasoning with a world model, generating and expressing affect, and generating coherent natural language, synchronised with nonverbal modalities. The graphical and TTS output of the agent is provided by commercial systems.

ERIC is currently implemented to commentate a simulated horse race and a multiplayer tank combat game. With minimal modification the system is configurable to provide commentary in any continuous dynamically changing environment; for example, it could commentate sports matches and computer games, or play the role of "tourist guide" during a self-guided tour of a city.

An elaborate world model is deduced from limited input by an expert system implemented as rules in Jess. Natural language is generated using template-based NLG. Discourse coherence is maintained by requiring semantic relations between the forward-looking and backward-looking centers of successive utterances. The agent uses a set of causal and belief relations to assign appraisals of emotion-eliciting conditions to facts in the world model based on goals and desires. These appraisals are used to generate an affective state according to the OCC cognitive model of emotions; the agent's affect is expressed via his lexical choice, gestures and facial expressions.

ERIC was designed to be domain-independent, homogeneous, behaviourally complex, reactive and affective. Domain-indepence was evaluated by comparing the effort required to implement the ERIC system with the effort required to re-implement the framework for another domain. Complexity, reactivity and affectivity were assessed by independent experts, whose reviews are presented.

Acknowledgements

I would like to thank a number of people for making this thesis possible. Firstly, thanks to Michael Kipp, for not just being an enthusiastic, helpful, and inspiring supervisor, but also volunteering a lot of his own time especially around the GALA competition and in the weeks before submission. Michael, Martin Rumpler and Patrick Gebhard were very helpful with ALMA, Charascript and the Paul interface; Patrick deserves special thanks for making not insignificant changes to the ALMA code when I needed them, quickly and uncomplainingly.

My studies in Saarbrücken were funded by the International Max Planck Research School for Computer Science: my thanks go to the IMPRS not only for the funding but also all their support in countless organisational matters. Thank you also to the Deutsches Forschungszentrum für Künstliche Intelligenz for providing the opportunity to work on this project, the necessary equipment, and the funding to attend the GALA competition and the IVA conference. I am also grateful for Charamel GmbH and Nuance Communications, Inc., for providing the CharaVirld system with the Paul character and the RealSpeak Solo software with the Tom voice respectively.

Finally, a heartfelt thanks to my friends and family: in particular, Rebecca Dridan for being a sounding board in the early stages, and for her unfailingly reliable good advice; Terese Liadal for not only her support, but also for loaning me her laptop when the stress of writing a thesis became too much for mine; and my parents for believing in me from the other side of the world.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents ERIC, a homogeneous framework agent that can be configured to provide running commentary on a continuous event in real-time. In this agent, we have focused on knowledge reasoning with a world model, generating and using emotions and affect, and generating coherent natural language, synchronised with nonverbal modalities. The graphical and text-to-speech output of the agent is provided by commercial systems.

ERIC is currently implemented to commentate a horse race provided by the RaceSim system[1], and a tank battle simulated with the dTank combat environment[2] [Councill et al., 2004, Morgan et al., 2005]. With minimal modification the system is configurable to provide commentary in any continuous dynamically changing environment; for example, it could commentate sports matches and computer games, or play the role of "tourist guide" during a self-guided tour of a city.

1.1 Motivation

Intelligent, graphically embodied agents – a form of intelligent user interface [Maybury and Wahlster, 1998] – are a powerful means of human-computer interaction. Foster [2007] calls face-to-face communication "the most natural method of interaction". Cassell [2000] observes that face-to-face conversation is a primary skill for humans: "When people have something really important to say, they say it in person" [Cassell et al., 2000]. Face-to-face communication allows communication protocols that give a much richer communication channel than other means of communicating. It enables pragmatic communication acts such as conversational turn-taking [Rickel and Johnson, 1998], facial expression of emotions [Ekman and Friesen, 1978], information structure and emphasis [Cassell et al., 2000], visualisation and iconic gestures [McNeill, 1992], and orientation in a three-dimensional environment [Rickel and Johnson, 1998]. This communication takes

[1]Provided at GALA '07, the Gathering of Animated Lifelike Agents: `http://hmi.ewi.utwente.nl/gala/racereporter`.

[2]dTank is available at `http://ritter.ist.psu.edu/dTank/`.

place through non-verbal channels such as gaze, gesture, spoken intonation and body posture [Cassell et al., 2000].

This rich style of communication that characterises human conversation makes conversational interaction with embodied agents ideal for many non-traditional interaction tasks. The most familiar application of embodied agents is computer games; embodied agents are ideal for this setting because the richer communication style makes interacting with the agent enjoyable. Embodied agents have also been used in virtual training environments [Rickel and Johnson, 1998], portable personal navigation guides [Theune, 2001], interactive fiction and storytelling systems [Theune et al., 2003], and automated presenters and commentators [Fielding et al., 2004].

Marsi and van Rooden [2007] found that users prefer a non-verbal visual indication of an embodied system's internal state to a verbal indication, demonstrating the value of additional non-verbal communication channels. As well as this, the face-to-face communication involved in interacting with an embodied agent can be conducted alongside another task without distracting the human participants, instead improving the enjoyment of such an interaction [Kipp et al., 2006]. Furthermore, Beun et al. [2003] found that the use of an embodied presentation agent resulted in improved recall of the presented information.

Embodied agents also provide a social dimension to the interaction. Humans willingly ascribe social awareness to computers [Nass et al., 1994], and thus interaction with embodied agents follows social conventions, similar to human/human interactions. This social interaction both raises the believability and perceived trustworthiness of agents, and increases the user's engagement with the system [Mulken et al., 1998]. Rickenberg and Reeves [2000] found that the presence of an embodied agent on a website increased the level of user trust in that website; as well as this, the presence of the agent increased users' anxiety and affected their performance just as if they were being watched by a real human. Another effect of the social aspect of agents is that presentations given by a embodied agent are perceived as more entertaining and less difficult than the same presentations given without an agent [Mulken et al., 1998].

In all of these situations the users' interaction is enhanced by the non-verbal communication and social interaction provided by embodiment. The use of an embodied agent leads to increased user satisfaction and engagement with the systems [Foster, 2007]. The richer communication channel allows more information to be conveyed, makes the information easier to absorb, and allows the system to be better integrated in a complex situation.

1.2 The ERIC Framework

ERIC is a homogeneous framework agent that can be configured to provide running commentary on a continuous event in real-time. ERIC is implemented as a modular framework of Java-based components, and observes a strict separation of domain-specific and domain-independent knowledge. It uses an expert system to generate a rich world model of declarative knowledge in real-time. The agent features a template-based natural language generation system capable of generating anaphora and coherent discourse, using Centering Theory. It also uses a layered model of emotions, mood and personality to guide its output generation; these affects are generated from dynamic appraisal of events, actions and objects against goals and desires.

Expert system-based architecture The architecture of ERIC was designed to be modular, parallel and homogeneous. To this end, the architecture is based on parallel rule-based reasoning engines, implemented in Java and Jess, a rule-based system itself implemented in Java. With the exception of the affect module, the ERIC modules are identical in design, consisting of a Jess reasoning engine running in a separate Java thread. The affect module adds to this an interface with the ALMA affect model. The various modules communicate via the observer design pattern, to minimise coupling for ease of reconfiguration.

Throughout the architecture of ERIC, domain-specific knowledge is kept separate from domain-independent reasoning: for example, the NLG templates are kept outside the generation reasoning rules, and the goals/desires and cause/effect relations are separate from the affective appraisal rules. This results in an architecture that facilitates shifting the system to a new domain by replacing the domain-dependent modules.

The key modules in ERIC are those concerned with knowledge reasoning (the domain model), natural language generation, and affective appraisal. As well as these, the agent has a world interface module for receiving information from the world and converting it into Jess facts for the world model, and a fusion module for combining generated language, facial expressions and gestures into a form that can be sent to the Charamel character.

Knowledge inference In the knowledge module, an elaborate world model is deduced from limited input by an expert system implemented as rules in Jess. In the current implementations, the input comes from the RaceSim horse race simulation, and the dTank tank combat environment. Both of these systems periodically update the knowledge module with a small number of facts about their world state. In the case of the RaceSim simulation, this consists of the speed and location of each of four horses, once a second; the dTank environment provides slightly more information about each connected tank and the world, on an event-by-event, rather than timed, basis.

From this limited input, the knowledge module produces a world model consisting of a large number of Jess facts describing the world. Thus the world model contains facts such as *horse 3 is currently in second place*, *horse 1 is about to overtake horse 2*, or *horse 4 has fallen over* (for the horse race domain), and *tank Eraser is aiming at tank 2DHuman*, *tank MyTank is moving in a westerly direction*, or *tank 2DHuman is hiding* *behind a boulder* (for the tank domain). In order to maintain consistency over the course of a continuous event, the facts in the world model created by the knowledge inference module are all tagged with a timestamp.

By basing the world model on declarative facts, the natural language generation templates can easily be conditional on the world state, since they are then simply conditional on facts in the world model. Similarly, it is quite easy to implement rules in the affect module to identify facts as events, actions or objects for its reasoning.

The knowledge module is highly dependent on the domain we are commenting. In order to keep the agent as general as possible, the framework places very few constraints on the implementation of the knowledge module: thus an agent author is free to use any fact format, style, or naming conventions he or she wishes. In fact, as long as the module exports declarative Jess facts to the other modules in the framework, the Jess-based module could be replaced with something else entirely, such as a ontology-based reasoner.

Natural language generation The template-based natural language generation in ERIC is similar to that of Theune et al. [2001]. A NLG template in ERIC consists of a priority (for comparison with other templates), some conditions that must be true for the template to be active, and a set of information conveyed by the template, as well as a single backward-looking and a number of forward-looking semantic centers for discourse coherence. Each template is implemented as a Jess rule; thus the conditions are matched against the world state by the Jess engine. The templates are filled with text from a lexicon, which is stored separately from the templates.

ERIC's NLG uses semantic centers (inspired by Grosz et al. [1995]) for macro-level discourse coherence. Each candidate utterance is assigned a single backward-looking and a number of forward-looking centers; by comparing the forward and backward centers of successive utterances, the system determines the strength of their discourse relation and prefers candidate utterances with a stronger relation to their prior utterance.

Affect In order to generate emotional responses, the agent has an affective appraisal module which uses a set of causal and belief relations to assign appraisals of emotion-eliciting conditions (EECs) to facts in the world model based on a set of goals and desires. These appraisals are used by ALMA [Gebhard, 2005] to generate an affective state (medium-term moods and emotion events) according to the OCC cognitive model of emotions [Ortony et al., 1988]. This dynamic appraisal of events removes the need for manual tagging of events with affective stimuli or EEC values.

ERIC's affective state is expressed in his lexical selection, facial expression and gestures. The NLG module's templates are filled with words and phrases from an utterance lexicon; lexicon entries can be tagged with affective states, and the NLG system only uses lexicon entries that are appropriate for ERIC's current affective state. Also, the Charamel character “Paul” supports a wide variety of pre-defined gestures and facial expressions, which are used to express his affective state. As well as these, the ability to express affective state via speech prosody is a possible extension that is outlined in this thesis.

Evaluation They key goals in the design and implementation of ERIC were homogeneity, domain-independence, reactivity, behavioural complexity and affectivity. The domain-independence of the agent was evaluated by implementing two distinct domains: the horse race simulation, and the tank combat game. We recorded the amount of effort required to implement the framework into a new domain, after it was initially developed for a new domain, and used this as an indication of the reusability, and hence domain-independence, of the agent. To evaluate the output goals of complexity, reactivity and affectivity, two independent anonymous judges provided feedback on a video of the agent commentating a single horse race, as part of the GALA challenge. On the basis of this evaluation, the agent was found to be satisfactorily homogeneous and domain-independent, and quite engaging. The affective expression of the agent was limited by his lack of prosody.

1.3 Related Work

There is a considerable body of past work in the field of virtual human presentation agents; some representative systems are described below.

1.3.1 Virtual Human Presenter

The Virtual Human Presenter [Noma et al., 2000] is a system that animates an embodied agent from input speech text and embedded body language commands, called “presentation scenarios”. The presentations are displayed using the JACK animation engine. The Virtual Human Presenter was designed to exhibit natural motions and presentation skills, and generate motions synchronised with speech in real-time. The gestures available to the Virtual Human Presenter are based on human presentation skills, to be as natural as possible. The Presenter is able to present either with a two-dimensional presentation board, or with a three-dimensional environment.

The input to the system – the agent’s presentation – is specified using commands embedded in the speech text that identify gestures to be performed, without defining the movements explicitly. The Presenter is controlled using a kind of state chart called PaT-nets, which enables him to generate the appropriate movements from such a presentation scenario in real-time. This style of presentation description is also intended to be applicable in a wide range of scenarios.

Although the aims of domain-independence and real-time reactivity are shared between the Virtual Human Presenter and ERIC, the Virtual Human Presenter generates a presentation's movements and gestures assuming the text to be presented is already present. Thus he does not reason about a world model to generate output text and gestures; rather, these are specified in the input. The Virtual Human Presenter corresponds more closely to the Charamel output component of a commentary system such as ERIC, leaving the reasoning required to generate the presentation unsolved.

1.3.2 BEAT

The Behaviour Expression Animation Toolkit (BEAT) [Cassell et al., 2001] is a system created by the MIT Media Lab that automatically generates synthesised speech and synchronised nonverbal behaviours from input text. These behaviours are generated based on linguistic and contextual analysis of the input text: the text is first given syntactic and semantic tags, and then behaviours are matched to these tags.

The input text to BEAT is automatically tagged with linguistic and contextual information: syntactically the text is broken up into clauses, which are then each broken into a *theme* and a *rheme*. Within these, the system identifies and tags word phrases representing either an *action* or an *object*. In addition to this, the tagger keeps track of all nouns, verbs, adverbs and adjectives, and marks previously unseen words as *new*; also, contrasting adjectives (spotted using WordNet) are tagged with a *contrast* tag. The mapping of linguistic tags to behaviours is rule-based, derived from observations of human gestures in conversational behaviour research. As well as this, authors can add custom rules to the default BEAT mapping, write their own gestures, and tweak the existing movements.

BEAT fulfils a similar role to the Virtual Human Presenter, although it has a significant advantage over the Presenter in that it can generate nonverbal behaviours automatically rather than requiring them to be specified in the input text. However, like the Virtual Human Presenter, BEAT still requires an input text to be provided, and does no cognitive reasoning like ERIC. Conversely, the input processing performed by BEAT on the input text is superfluous to ERIC's task, since ERIC can generate this information from underlying facts as part of its natural language generation. For instance, the semantic centers used by ERIC for discourse coherence are comparable to the theme/rheme separation produced by BEAT's analysis.

Also, since BEAT only uses analysis of the input text to generate nonverbal behaviours, the variety of its generated behaviours is limited by the sophistication of its linguistic analysis, and the behaviours are likely to be much more limited and less expressive than those of a system such as ERIC that can generate nonverbal behaviours from a world model and an internal affective state as well.

1.3.3 Virtual Storyteller

The Virtual Storyteller [Theune et al., 2003] is a system that generates stories and presents them using an embodied agent. It is able to generate appropriate prosody and gestures to accompany the story, and in future it will be extended to enable embodied characters to act out the generated stories.

The Virtual Storyteller generates stories using a number of agents that act in three levels: first the *plot*, the series of events making up the story; second the *narrative*, a natural language description of the plot; and third the *presentation*, consisting of the narrative rendered into speech and gestures by an embodied agent. Plots are created by a number of agents, by planning for a number of story goals with a set of story world knowledge. Similar to ERIC, the reasoning components of the agents were implemented in JESS.

The upper layers of the Virtual Storyteller (responsible for turning the events of the plot into an embodied presentation) correspond with the functionality of ERIC, and Theune et al. [2006] describe a system for generating expressive prosody in the context of the Virtual Storyteller system. However unlike ERIC, the Virtual Storyteller focuses largely on the generation of believable, consistent and interesting stories; in contrast, ERIC does not need to generate the ‘plot’ it is commentating, rather the events come from an interface to the world. Conversely, ERIC needs to select information from the changing world model for commentary in real-time. Because there is insufficient time to commentate every event in the world, a choice needs to be made. This is in contrast to the presentation of Virtual Storyteller stories, since the stories can be generated to suit telling, and thus no such selection is necessary.

1.3.4 VirtualHuman and COHIBIT

VirtualHuman [Reithinger et al., 2006] is a knowledge-based framework for creating embodied agent applications; in particular applications involving multiple users and multiple agents. The agents’ behaviour is controlled by dialogue engines, as well as behaviour and affective models to produce natural emotional reactions. The narration engine, dialog and behaviour controllers, and 3D rendering engine communicate over shared blackboards using XML-based markup languages.

To produce an unscripted dialogue between multiple participants, VirtualHuman uses a shared knowledge base describing the world, and a Conversational Dialogue Engine for each participant (human or virtual) in the dialogue: these engines interpret the participant’s interaction, and generate the embodied agents’ reactions. As well as these engines, VirtualHuman has a narration engine which controls the overall story.

COHIBIT (COnversational Helpers in an Immersive exhiBIt with a Tangible interface) [Ndiaye et al., 2005, Kipp et al., 2006] is an ambient intelligence edutainment exhibit implemented in the VirtualHuman framework. Visitors interact with the system by assembling a puzzle from three-dimensional pieces; the visitors' actions are observed by two embodied agents who are able to comment on the actions, provide advice and additional information, or make smalltalk.

The COHIBIT system uses a type of augmented transition networks called SceneFlow to model the dialogue interaction. The SceneFlow transition networks are easy to author and read, and thus easy to maintain and debug – something that can be a drawback of large expert systems such as those used in ERIC (see Chapter 2). In contrast, using expert systems for all levels of processing has the advantage that the knowledge representation and behaviour generation are all in the same homogeneous framework, which also makes the system easier to maintain.

The focus of the VirtualHuman framework is producing believable dialogues between any number of embodied agents and users. In contrast, the single-commentator commentary scenario of ERIC does not require any such dialogue management – although a multiple-commentator scenario is not uncommon in real commentary situations, and might make a useful further development of ERIC. Because of its modular architecture, ERIC has been able to reuse a number of components from the VirtualHuman project: in particular, the ALMA module for affective state modelling and the Charamel interface for controlling the Charamel characters are both used in VirtualHuman and COHIBIT.

1.3.5 VITRA and Rocco

The VIrtual TRAnslator (VITRA) [Herzog and Wazinski, 1994, Herzog, 1995] is a commentary generation system developed by the DFKI and the vision group at the Fraunhofer Institute (IITB, Karlsruhe). VITRA generates natural language commentary from a video sequence of a scene. VITRA assumes that its listener(s) cannot see the scene it is commentating, but have some prior knowledge about static properties of the scene. VITRA has been implemented to describe traffic scenes, provide route descriptions through a 3D model, communicate with an autonomous mobile robot, and commentate on soccer games. VITRA produces its commentary incrementally, first tracking objects in the scene to determine the spatial relations between them, then reasoning over these spatial relations to recognise first events and then plans and intentions, and finally generating output natural language to describe both recognised events and plans. The incremental approach allows the system to generate the commentary as the scene progresses, rather than only once the entire scene has been viewed.

The RoboCup robotic soccer has included a commentary challenge, and several systems have been developed to commentate a RoboCup soccer match [André et al., 2000]. The Rocco system of André et al. [1998], Voelz et al. [1999] is one such system developed at DFKI, as an extension of the soccer commentator in the VITRA project. Like VITRA, Rocco generates multimedia commentary output from an input image sequence. Rocco has three layers of processing: low-level visual analysis of the input scenes, high-level scene analysis producing a scene description (including player positions, actions and intentions), and multimedia presentation generation, which plans the multimedia output.

Both of these systems are focussed mainly on producing commentary from visual input, and thus have significant emphasis on image processing and scene recognition. In contrast, ERIC's scenarios assume a given world interface; conceivably this could take the form of such a visual analysis module, however the extraction of information from the world is out of the scope of this thesis. Also, the high-level scene analysis performed by these systems – especially plan recognition to detect strategies and player intentions – is not necessarily applicable in a simple domain such as the RaceSim simulation, and has thus only been touched upon in ERIC; however (as described in Section 4.4) the knowledge module does allow authors to implement some plan recognition, and can easily be extended to support much more complex reasoning.

Conversely, both these systems have less focus on generating believable affect and non-verbal expression. Although both VITRA and Rocco include a mechanism for multimedia report generation, the available media are restricted to generated speech, and the video sequences used as input. The ERIC agent has a much richer multimodal repertoire. The affective model used in ERIC allows the commentator to exhibit emotional engagement with the commentated events, and maintain an emotional state which can then guide his output and be expressed in his non-verbal actions. Further, the fact that ERIC is presented as an embodied agent enables the additional use of facial expressions and hand and body gestures in his commentary. Also, the modular framework of ERIC easily enables additional modalities such as prosody or camera movement to be added to the system.

1.4 Application Domains

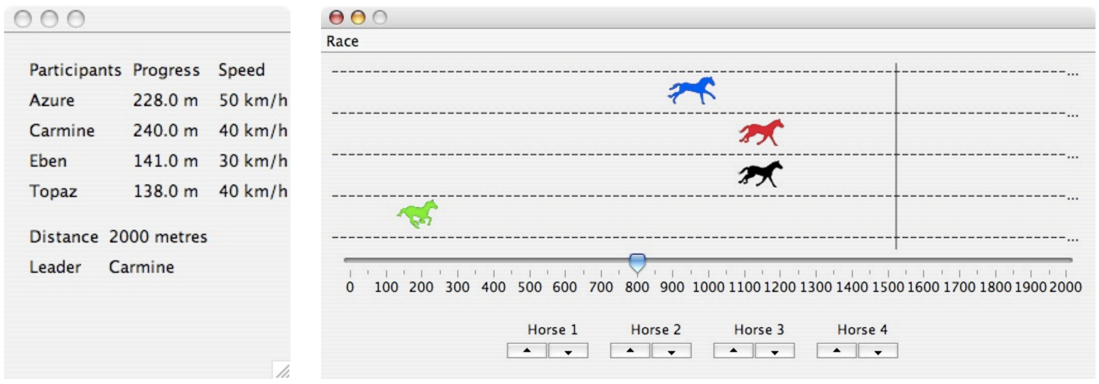
ERIC is currently implemented to commentate on two scenarios: a horse race simulation, and a tank combat game. These two different domains were chosen to verify the domain-independence of the ERIC framework: one of our research goals (see Section 1.5).

Both domains are a type of competitive sport: they both feature adversarial competition among multiple participants, only one of whom can win. Thus both are suitable for a sports-commentary style of commentating. However they also differ in a number of regards. The horse race is a relatively stylised genre of commentary, with certain expectations about its format; on the other hand, a tank combat game allows an author more freedom in commentary style. More concretely, the horse race simulation is a very well-defined domain: the race is fixed at 4 horses whose attributes are known in advance, and has a clear finish line after 2000 metres. By contrast, the dTank game is a much more open domain: any number of previously-unknown tank commander clients can connect to the environment, and tanks can join and leave during the battle. As well as this, there is no clear "finish" for the dTank scenario, so unlike the horse race simulation there can be no estimation of remaining time. Also, the tanks in the dTank environment have much more freedom of movement than the horses in the RaceSim: they can move in two dimensions, rotate, rotate their turrets, raise shields and fire – six degrees of freedom – whereas the horses have only two degrees of freedom, their position along a straight line and their speed.

We hope to demonstrate the domain-independence of the ERIC framework across these contrasting domains. Both domains call for "sporting event"-style commentary, as distinct from other conceivable applications of this framework, such as a tourist guide or a news reporter. Nevertheless, the differences between the domains should be sufficient to reveal aspects of the ERIC framework that do not meet the aim of domain-independence.

1.4.1 Horse Race Simulation

The RaceSim horse race simulation is provided by the organisers of GALA '07[3] In the RaceSim simulation, four horses race along a 2000m track (shown in Figure 1.1), and every second each horse's location and speed is output over a network socket interface.



Participants	Progress	Speed
Azure	228.0 m	50 km/h
Carmine	240.0 m	40 km/h
Eben	141.0 m	30 km/h
Topaz	138.0 m	40 km/h

Distance 2000 metres
Leader Carmine

(a) The info window (b) The main window

Figure 1.1: The RaceSim horse race simulator

During the course of the race, the speed of each of the four horses ranges between 0 and 90 km/h. A speed of 0 during the race represents a horse that has fallen over. The race ends when either all horses reach the finish line, or all horses have not moved for 10 seconds (that is, in the extreme case where all four horses have stayed "fallen over" for 10 seconds).

[3]Gathering of Animated Lifelike Agents, at the 7th International Conference on Intelligent Virtual Agents – IVA 2007. The race reporter challenge using the RaceSim system is described at `http://hmi.ewi.utwente.nl/gala/racereporter`. ERIC was a winning entry in the race reporter challenge: the GALA entries can be viewed at `http://hmi.ewi.utwente.nl/gala/finalists`.

The progress of the race can be controlled by issuing one of three commands: increase a horse's speed by 10 km/h, decrease a horse's speed by 10 km/h, and cause a horse to fall over (that is, set its speed to 0 km/h). At most one of these commands can be issued for each horse in any second. These commands can be issued via the RaceSim GUI, and they can also be specified in an XML file. The GUI can also record a race to such an XML file, so that past races can be played back. Commands can also be issued via the GUI while a XML file is being played back, as long as the restriction of one command per horse per second is kept; thus a past race can be altered during playback.

1.4.2 dTank Combat Simulation

dTank is a simulation of a tank battle on a two-dimensional battlefield, intended for competitive evaluation of adversarial real-time cognitive models, and as an education tool in cognitive science [Councill et al., 2004, Morgan et al., 2005]. The latest version of dTank (version 4, see Figure 1.3) simulates a sophisticated battle environment, with two armies of any number of battalions of up to four tanks, each battalion commanded by a commander agent. For ERIC's commentary an earlier version of dTank was used (see Figure 1.2); in this version any number of individual tanks battle against each other. In the simplest version a single tank battles against one other adversarial tank.

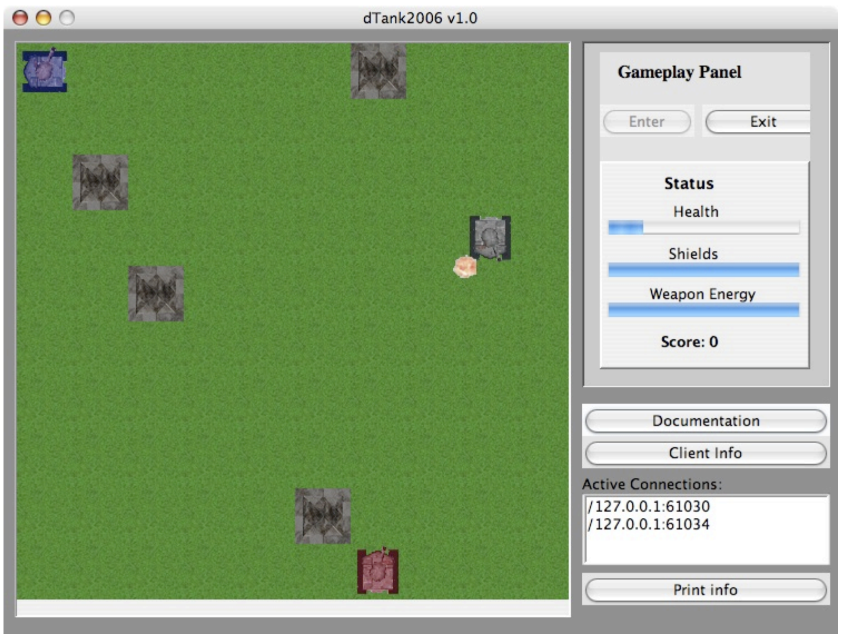
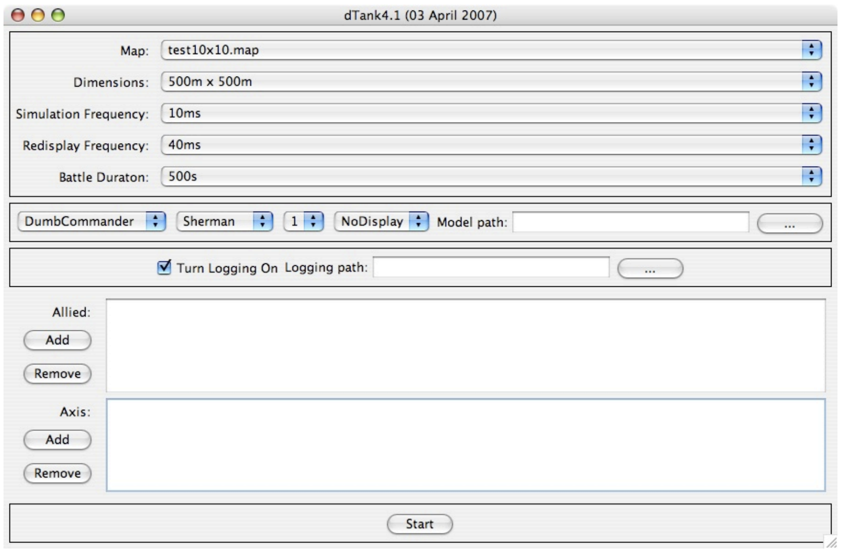
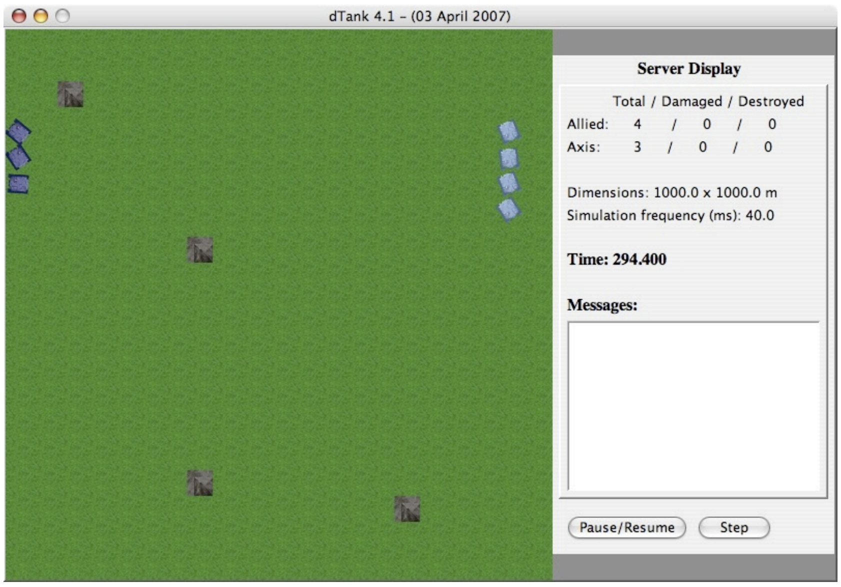


Figure 1.2: A screenshot of the dTank environment

The dTank environment runs as a server, to which tank clients can connect via a network socket. Each client is represented in the environment as a single tank with a distinct colour (assigned by the environment). The tank is controlled by the client sending one



(a) The setup window



Total / Damaged / Destroyed
Allied: 4 / 0 / 0
Axis: 3 / 0 / 0

Dimensions: 1000.0 x 1000.0 m
Simulation frequency (ms): 40.0

Time: 294.400

Messages:

(b) The battlefield view

Figure 1.3: The latest version of the dTank environment

of a number of commands via the network connection: turn 90° right, turn 90° left, move forward, turn the turret to a particular orientation, raise the shields for a brief period, and fire. The intent of the dTank environment is for the tanks to be controlled by different cognitive models implemented in modelling languages such as SOAR or expert systems such as JESS, however there is also a three-dimensional first-person-shooter-style client with which a human can control a tank interactively, and any client that is able to produce the tank commands can connect and control a tank.

1.5 Research Aims

The design and implementation of ERIC followed a number of aims which are detailed below. The results of an evaluation of the system against these aims, and a discussion of this evaluation, are presented in chapter 7.

Homogeneity In order to facilitate easy reuse of the agent's framework across many domains, it was desired that the varied functions of the agent (such as natural language generation, domain reasoning, and affective appraisal) are all implemented in a similar style. Thus a user who is adapting the framework to his or her own domain need only understand this one style, rather than a variety of styles chosen by the various implementers of the individual components. Also, a system of homogeneous modules allows for simpler inter-module communication, without the need for middleware such as CORBA: thus such a system is faster and less error-prone than a system of heterogeneous components. A homogeneous framework also has the advantage that it is ideal for rapid prototyping of a commentary system: rather than having to juggle a whole lot of different components to demonstrate the feasibility of a domain or a new component, the homogeneous framework can be reused, replacing only the domain or single component being prototyped. In line with this aim, the components of the system each follow a common thread-based structure, enclosing an expert system written in Jess.

Domain-independence The agent is intended to require minimal additional work to implement it for a new domain, so that it can be easily configured to provide commentary on any situation. To this end, the components of the system have been designed as much as possible to separate domain-dependent reasoning from domain-independent reasoning; where this separation has been impossible on the component level, domain-dependent information is stored in a configuration file, separate from the domain-independent information.

Reactivity The agent should be able to react to new or changing events in a dynamically changing environment. Thus he must be able to change topic in his speech without prior planning, decide on the most appropriate comment in real-time, and possibly even be able to interrupt himself when a more important event occurs than his current commentary. Additionally, the agent should also be able to predict important events before they occur, thus allowing him to generate shorter or more easily interruptible utterances, or even avoid speaking at all, just prior to such an important event to enable the important event to be commentated immediately.

Behavioural complexity In order to be engaging, the agent must exhibit sufficiently complex behaviour, such as we would expect from a human commentator. Thus he must generate variety in his speech utterances, and use gestures and facial expressions to engage with the commentated events. Behavioural complexity is essential to creating a believable character; without variety in responses, a wide emotional range, and interaction with its environment, an agent appears 'wooden' and unrealistic [Vinayagamoorthy et al., 2006].

Affectivity The agent should respond emotionally to the events, action and objects of its environment. Further than this, we wish the agent to have a theoretically well-founded cognitive model of affect, encompassing both short-term affect (emotions) and longer-term affect (mood and personality). In this way some of the atmosphere of the event can be conveyed to the agent's audience. Also, affect conveys information to the audience: for example, the closeness of the finish line of a horse race is conveyed by the commentator's excitement.

These five aims fall naturally into two categories: homogeneity and domain-independence are design goals, and reactivity, complexity and affectivity are goals for believability of the virtual commentator.

1.6 Thesis Outline

This thesis presents the ERIC commentary agent framework: both the design and algorithms of the agent and an evaluation of the agent against our aims are presented. First we will introduce the concept of expert systems, and the Java expert system shell (Jess), in Chapter 2. Chapter 3 covers the design of the system and the system architecture. We will describe the overall modular architecture of the system, showing how it answers the stated aims; we will also describe the components of the system that were used off-the-shelf, and the design of the implemented modules comprising the system.

In Chapter 4 we describe the knowledge inference module, and show how a sophisticated domain model is produced from the limited input of the two implemented domains. Chapter 5 describes the natural language generation algorithm implemented in ERIC, covering both the generation of individual utterances and generation of coherent discourse. ERIC's affect generation is described in Chapter 6. We first describe the means by which ALMA generates an affective state from input emotion-eliciting conditions, and then describe how ERIC generates these emotion-eliciting conditions from the world model. Then, we show the means by which ERIC expresses his affective state. In Chapter 7, we evaluate the ERIC agent against the aims stated above, and discuss our results.

Chapter 2

Expert Systems and Jess

The fundamental component of ERIC is an expert system. It is quite natural to model deductive knowledge about the world as facts and rules such as in an expert system; and the main task of ERIC is to deduce various forms of knowledge – a world model, an affective state, appropriate gestures, utterances and facial expressions – from limited input facts. An expert system provides a standard 'black box' algorithm for processing facts and rules, allowing a programmer to implement this knowledge declaratively (specifying just the facts and rules, rather than the algorithm itself). We will first illustrate this style of programming, and then describe Jess, the Java-based expert system language and engine that we have used in the ERIC system.

2.1 Expert Systems

Consider the problem in Figure 2.1. How do we solve it? You can buy books full of this sort of problem; they are usually called "logic puzzles" or something similar. They come accompanied with a n-dimensional table of combinations of variables (the colour of Bob's pants, for instance; or the name of the golfer who is second in line): an example is shown in Figure 2.2. To solve the puzzle, the reader enters the facts from the clues into this table (using ticks and crosses, for example), and then by elimination the table can be filled, producing the answer. So we have a specific, repeatable algorithm for solving such a problem. But what if we only had to specify the clues from the problem in some standard way, and could let a computer solve it for us?

Specifically, we would like to transform the clues in Figure 2.1 into the facts in Figure 2.3 (for simplicity, these facts are in first-order logic); and then ask a program to tell us for each value of $g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\}$, for which values of $p \in \text{pants}, c \in \{\text{Plaid}, \text{Red}, \text{Blue}, \text{Orange}\}$ $\text{position}(g, p)$ and $\text{pants}(g, c)$ are true.

We call such a system an *expert system*: that is, an expert system is a system that can reason about the world, and take appropriate actions based on some kind of knowledge about the world. Since the program is specified using facts and rules, and letting a reasoning engine operate on them, an expert system is also called a *rule-based system*.

1. A foursome of golfers is standing at a tee, in a line from left to right. Each golfer wears different colored pants; one is wearing red pants. The golfer to Fred's immediate right is wearing blue pants.
2. Joe is second in line.
3. Bob is wearing plaid pants.
4. Tom isn't in position one or four, and he isn't wearing the hideous orange pants.

In what order will the four golfers tee off, and what color are each golfer's pants?

Figure 2.1: A logic puzzle, taken from Hill [2003]

	Fred	Joe	Bob	Tom	red	blue	plaid	orange
first								
second								
third								
fourth								
red								
blue								
plaid								
orange								

Figure 2.2: A table to help solve the logic puzzle in Figure 2.1

1. a) $\forall g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} : \text{golfer}(g)$
   b) $\forall g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} \exists p \in \{1,2,3,4\} : \text{position}(g,p)$
   c) $\forall p \in \{1,2,3,4\} \exists g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} : \text{position}(g,p)$
   d) $\forall g,h \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} \forall p,q \in \{1,2,3,4\} : \text{position}(g,p) \wedge \text{position}(h,q) \rightarrow (g = h) \vee (p \neq q)$
   e) $\forall g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} \exists c \in \{\text{Plaid}, \text{Red}, \text{Blue}, \text{Orange}\} : \text{pants}(g,c)$
   f) $\forall c \in \{\text{Plaid}, \text{Red}, \text{Blue}, \text{Orange}\} \exists g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} : \text{pants}(g,c)$
   g) $\forall g,h \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} \forall c,d \in \{\text{Plaid}, \text{Red}, \text{Blue}, \text{Orange}\} : \text{pants}(g,c) \wedge \text{pants}(h,d) \rightarrow (g = h) \vee (c \neq d)$
   h) $\exists g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} : \text{pants}(g, \text{Red})$
   i) $\exists g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} \exists p,q \in \{1,2,3,4\} : \text{position}(\text{Fred},p) \wedge \text{position}(g,q) \wedge q = p+1 \wedge \text{pants}(g, \text{Blue})$
2. $\text{position}(\text{Joe}, 2)$
3. $\text{pants}(\text{Bob}, \text{Plaid})$
4. a) $\text{position}(\text{Tom}, p) \wedge p \neq 1 \wedge p \neq 4$
   b) $\text{pants}(\text{Tom}, t) \wedge t \neq \text{Orange}$

Figure 2.3: The facts from the logic puzzle in Figure 2.1 as first order logic facts

Since expert systems are programmed by specifying facts about the world rather than steps in an algorithm, they are a kind of declarative programming. Declarative programming is much more intuitive than imperative programming in many situations, in particular situations requiring this sort of reasoning; here, we no longer need to specify an algorithm for solving logic puzzles, instead we only need to formally state the puzzle, allowing us to focus on the problem itself. Admittedly, such a logic puzzle can be solved using a pretty simple algorithm; but if you've ever found yourself buried in complex if/then/else constructs covering lots of complicated and varying cases, you can appreciate how much easier it would be to just specify a list of rules and let the system do all the hard work for you. Of course there are other advantages to this style of coding as well: for a problem that consists mainly of rules and facts (such as this logic puzzle), the format of rule-based code is much more closely related to the format of the problem, and is therefore much easier to read and also easier to maintain or change if the problem changes.

2.1.1 Motivation

This example shows that an expert system is ideal for generating a world model from limited facts. Admittedly the facts provided in the example are rather cryptic, since the example is intended as a puzzle that is difficult for humans to solve; but the reasoning process is identical if we have less cryptic, but no less limited, facts. In general, it is quite natural to model deductive knowledge about the world as expert system rules.

We have already observed that separating the knowledge declaration from the algorithm makes the system easier to read and maintain than if the two were jumbled together. This separation has another advantage: it allows the code to be much more easily understood by non-programmers, thus making it possible for expert systems to be produced by experts in the domain being modelled (which is not necessarily software engineering).

In addition to these general advantages, expert systems are well suited to the needs of ERIC in a few specific ways. The above example has shown that the knowledge module can use Jess to generate a world model from limited facts; the cognitive appraisal of emotion-eliciting conditions in the affect module involves similar reasoning, generating an affective model from the world facts. In addition to this, representing the world model as a set of facts allows easy transfer of this model (or parts thereof) between modules, simply by transferring the facts. Also, the ability of Jess rules to execute functions (as well as modify the working memory) when a rule is fired makes Jess rules a natural way to make the character's output conditional on the internal world model.

As well as such knowledge reasoning, an expert system turns out to be an excellent way to implement the natural language generation module. The natural language generation of ERIC is a template-based system (see Chapter 5). The process of selection and application of NLG templates is very similar to the way rules are activated and fired in an expert system, and in fact the NLG templates are very naturally implemented as Jess rules.

2.1.2 Disadvantages

There are a number of disadvantages and limitations to expert systems. In particular, is it difficult to reason quantitatively using rules: there is no easy way to make a rule dependent on the number of facts matching a particular pattern, for instance. By clever design of facts, it is possible to count some phenomena: for example, the rule in Figure 2.4 counts the time that has elapsed since a horse's position last changed. However, this is not a natural way to express this knowledge, and such rules need to be specially designed. Comparative reasoning (asking whether there are more stones than grassy squares in the dTank world, for example) is an especially difficult case of this.

```
(nopositionchange (timestamp ?t) (duration ?d))
(position (timestamp ?t) (horse 1) (position ?p1))
(position {timestamp == (+ ?t 1)} (horse 1) (position ?p1))
=>
(assert (nopositionchange (timestamp ?timestamp)
    (duration (+ ?d 1))))
```

Figure 2.4: An example of quantitative reasoning in Jess

As well as this, the fact-based nature of an expert system means that facts in our knowledge are either true or false; there is no mechanism for reasoning about things we do not know, or things we are uncertain of. We can work around this limitation in a similar way to the problem of quantitative reasoning, by adding facts to our knowledge that reason about other facts; but again this is a special measure, not a natural representation.

Large expert systems can also become difficult to read and maintain, due to their declarative nature. Since the system chooses which rules to activate and fire, it can be difficult to predict the behaviour of a complex expert system with many rules and facts, thus making testing and debugging difficult. This can be counteracted by dividing the system into smaller modules. Each module then contains a subset of the rules of the whole system, and can therefore be more easily tested and debugged than the system as a whole.

Despite these drawbacks, an expert system is still very well suited to the needs of ERIC. The advantages of expert systems, and the ability to work around these limitations to some degree, suggest an expert system as the best building block for the reasoning modules in ERIC.

2.2 Jess

Jess is a language for specifying expert systems, and an interpreter for this language implemented in Java. Jess originated as a reimplementation of another expert system language and reasoning engine, CLIPS, into Java. Since then Jess has undergone a number of improvements over CLIPS, resulting in a very fast reasoning engine – faster even than the C implementation of CLIPS. Jess was chosen for the ERIC framework because its Java implementation makes it easy to integrate Jess with other components written in Java. This section will briefly describe the architecture and syntax of Jess. It is mostly taken from Hill [2003]; that book, as well as the online Jess manual [Hill, 2006], describe Jess in much more detail.

2.2.1 Architecture

A rule based system generally follows the architecture in Figure 2.5; this is the architecture of Jess. The working memory stores facts about the world; it is also sometimes called the *fact base*.

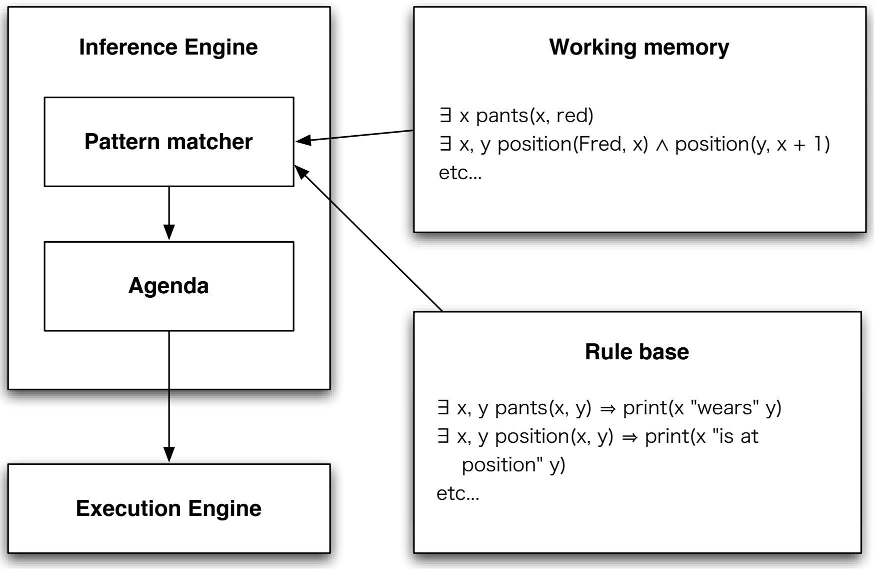

Inference Engine
Pattern matcher
Agenda
Execution Engine
Working memory
∃ x pants(x, red)
∃ x, y position(Fred, x) ∧ position(y, x + 1)
etc...
Rule base
∃ x, y pants(x, y) ⇒ print(x "wears" y)
∃ x, y position(x, y) ⇒ print(x "is at position" y)
etc...

Figure 2.5: The architecture of a typical rule based system

The rule base stores the system's rules. The rules and the initial facts (together with some functions) form a Jess "program". The inference engine is the "black box" that performs reasoning over the facts and rules; the left hand side of Figure 3.1 is called the *rule engine*. Inside the rule engine, the pattern matcher selects rules that are applicable given the facts in the fact base, and activates these rules – that is, places them on the agenda. Then the execution engine fires the rules on the agenda in a particular order.

The agenda is also called the *conflict set*, since all the rules on the agenda are applicable now, and are therefore in conflict; the algorithm by which the execution engine decides on an order in which to fire the rules is called *conflict resolution*.

If we have a standard way of specifying rules and facts for the rule base and the working memory then the rule engine and the execution engine together form a standalone system. Jess is such a system. In fact, Jess is both a rule engine and a language; the Jess engine is implemented in Java, and the Jess language is LISP-like in appearance.

2.2.2 The Language

The Jess language is mostly a declarative language – that is, a Jess program is made up of a set of statements about the world, rather than a list of commands to execute. In fact, since Jess is a rule-based system, these statements about the world are either facts, or rules.

Syntax

On the surface, Jess has a very LISP-like syntax: every component (facts, rules, function calls, etc) takes the form of a list. Also, like in many languages, whitespace is ignored.

Symbols

The basic unit of Jess is the symbol. These perform a similar function to identifiers in Java – they can represent values or variables. The following characters are valid characters for a symbol:

```
a-z A-Z 0-9 $ * . = + / < > _ ? #
```

However, a symbol may not begin with a number, or with the characters `$`, `?`, `&`, and = : these characters have special meanings which will be detailed below. Also, symbols are case sensitive. A number of symbols are reserved and have special meaning in the Jess language:

- `nil` is a special symbol that represents the absence of a value, rather than a value itself; it corresponds to Java's `null`.
- `TRUE` and `FALSE` are the Boolean true and false values

In addition, some other symbols are special only in a particular context; for example, `crlf` is transformed into a newline character when it is output as part of a string.

Values

Jess values can be either symbols, numbers, or strings, or lists of symbols, numbers or strings. Numbers can be either integers or doubles. Unlike Java, a string containing newlines, tab characters, etc. can be specified by typing those characters in the program; thus the Java escapes for these characters do not work in Jess (`\n` produces the characters `\` and `n`, for example); the exception is `\"`, since the quotation mark is used to enclose a string and thus needs to be escaped within a string. Any line beginning with a semicolon is considered a comment.

Variables

Variables in Jess are denoted by strings beginning with a question mark (which is why, as we saw above, symbols cannot begin with the character `?`). Despite the fact that Jess values have types, Jess variables are untyped. Generally only alphanumeric characters, dashes and underscores are used in a variable name, although other punctuation marks are allowed.

To assign a value to a variable, Jess provides the `(bind)` function, which has the syntax `(bind <variable> <value>)`, for example:

```
(bind ?length (str-length ?string))
```

A multifield is a special kind of variable that is identical to normal variables except in the argument list when defining a function, and on the left-hand-side of rules. In these situations, a standard variable would match only a single value, whereas a multifield allows us to permit one or more values. A multifield is denoted by the initial characters `$?` (instead of just `?`), for example `$?referents`.

The `(reset)` function clears all non-global variables. Global variables are persistent variables that are not cleared by `(reset)` (although they can be optionally reset to their default values). A variable is marked as global by enclosing its name in asterisks, for example `?*question*`. A global variable is defined using the `(defglobal)` function, which takes the form `(defglobal variable = default-value)`, for example:

```
(defglobal ?*x* = 3)
```

Lists

A list is a ordered sequence of values. Although all snippets of Jess code look like lists (as in LISP), Jess differs from LISP in that they are not in fact interpreted as lists – by default, a “list” typed into the Jess prompt is considered a function, with the head of the list the functor and the body its arguments. Also, other data types such as rules do not obey the same syntax as lists – in fact, to define a list you need to explicitly create a list data type. Lists are defined using the `(create$)` function; usually they are then immediately bound to a variable so we can access them again later, for example:

```
(bind ?effects (create$ ?unknown1 ?unknown2 ?unknown3))
```

In addition to this function, Jess provides other list handling functions such as `(nth$)` (which returns the *n*th element of a list), `(first$)` (which returns the first element of a list), and `(rest$)` (which returns all of the list except the first element). Lists in Jess cannot be nested: Jess flattens nested lists into a single list, so that the list `(a b c (d e) f)` is the same as (and is in fact represented as) the list `(a b c d e f)`.

Functions

When you write a simple list at the Jess prompt, Jess treats it as a function call: for example, `(eq 3 5)` calls the function `(eq)` which tests if its two arguments (in this case `3` and `5`) are equal. To define functions, Jess provides the function `(deffunction)`, which has the syntax `(deffunction <name> (<parameter>*) [<comment>] <expression>*)`.

We have already seen that functions and variables are defined using function calls; we will soon see that facts and rules are also defined this way. In addition, Jess provides a number of useful built-in functions for diagnostic purposes, in particular:

- `(facts)` lists all the facts currently in working memory;
- `(rules)` lists all the rules currently in the rule base;
- `(watch)` turns on debug output. This has the syntax `(watch <what>)`; you can watch `facts`, `rules`, `focus`, or `all`; debug output is then produced when the `watch`-ed items change. To turn watch off again, Jess provides the function `(unwatch)` with the same syntax as `watch`.

Facts

Facts in Jess contain a *head* and optionally one or more *slots*, and are either *ordered* or *unordered* facts. In an unordered fact, the slots are labelled and thus can come in any order (hence "unordered"); in an ordered fact the slots are unlabelled and get their meaning from their order (hence "ordered"). Examples of ordered and unordered facts (respectively):

```
(name 1 "Azure")
(name (horse 1) (name "Azure"))
```

Because ordered facts can quickly become difficult to keep track of, it is usual to use unordered facts for all facts with more than one slot. We need to somehow define the format our unordered facts will take; similarly to functions, Jess gives us a function `(deftemplate)` that defines an unordered fact's slots, with the syntax
`(deftemplate <name> [<comment>]`
`(slot|multislot <slotname> (<slotoptions>)*)*)`, for example:

```
(deftemplate name
  (slot horse (type INTEGER) (default 0))
  (slot name))
```

There are a number of different options that a slot can have; the most useful (seen here) is the `default` option, which specifies the default value a slot should take if we ever do not specify a value when asserting this fact. The `type` options allows us to specify a type of the slot; although future versions of Jess may enforce slot types, Jess 7 does not. Alternatively, we can declare a `multislot` instead of a `slot`, which specifies that this slot will match a multifield rather than a simple variable, and can accept one or more values (like a multifield).

Facts and functions live in the working memory in Jess. Facts are added to our knowledge by the `(assert)` function, and removed from our knowledge by `(retract)`. Rules, functions and our initial facts form the Jess program.

We can clear Jess's working memory with the function `(clear)`. This resets Jess to its initial state, wiping the facts, functions and rules from working memory – that is, it deletes our program. Thus this is not something we want to do often: in fact, it is mainly used to swap out an entire program before loading a different one. Similar to this function is the `(reset)` function (which we already saw cleared all non-global variables). In fact `(reset)` resets our program to its initial state, clearing non-global variables, setting global variables to their default values, putting only our initial facts into working memory, and leaving the rules and functions intact. Thus our Jess program is ready to run from the beginning after a call to `(reset)`.

When we first start Jess, before loading any files, a compulsory initial call to `(reset)` sets the working memory to contain only one fact: `(initial-fact)`. We can make the initial state contain other facts as well as `(initial-fact)` by using the `(deffacts)` function. `(deffacts)` takes as an argument a number of facts, and thereafter every call to `(reset)` resets the working memory to contain exactly those facts. The syntax of `(deffacts)` is `(deffacts <name> [<comment>] <fact>*)`. An example is given in Figure 2.6.

Conditional Elements Now we are able to specify facts, variables and values; however in order to build up more complex logical statements we need to be able to combine these. Jess provides four conditional elements for this purpose: `and`, `or`, `not`, and `exists`,

```
(deffacts background-knowledge
    (weather "foggy")
    (tank-commander (tank "2DHuman")
        (commander "our visitor"))
    (tank-history (tank "2DHuman")
        (history successful)))
```

Figure 2.6: An example of the use of `deffacts` in Jess

which correspond to the Boolean operators "and", "or", "not" and the existential quantifier respectively. If the existential quantifier is absent from a logical expression, Jess considers that expression to be implicitly universally quantified.

In addition to the Boolean operators, two more conditional elements are necessary due to the way rules are defined. The conditional element `test` calls a function and evaluates to `TRUE` if the function returns anything except `FALSE`, allowing the result of a function call to be used as part of a logical expression. The other conditional element is the `logical` element, which is required for particular kinds of rules and will be elaborated below.

Rules

Now we come to the core component of our rule-based system: rules. In Jess a rule takes the form of a left-hand side (consisting of facts) and a right-hand side (consisting of a sequence of function calls), separated by the characters =>. The Jess engine attempts to match each rule's left-hand side against facts in the working memory, and if it can successfully match the left-hand side of a rule it executes the functions in the rule's right-hand side. Rules are defined using the `(defrule)` function, which has the syntax `(defrule <name> [<comment>] <fact>* => <function>*)`; the sequence `<fact>* => <function>*` is the actual rule. An example of a rule is shown in Figure 2.7.

```
(defrule knowledge::fallen
  "a horse has fallen when its speed is 0"
  (speed (timestamp ?t1) (horse ?h1) (speed 0))
  (speed {timestamp == (- ?t1 1)} (horse ?h1) {speed > 0})
  (not (raceover))
  =>
  (assert (fallen (timestamp ?t1) (horse ?h1)))
  )
```

Figure 2.7: An example of a Jess rule

The slots in the facts in the left hand side of the rule don't have to be specified exactly; if we specify variables instead of values here, then the rule will match any fact that has a value in that slot, and the variable will be set to the value of that slot. We can then access this value on the right hand side of the rule using that variable. If the variable already has a value, however, then using the variable is the same as using its value.

Sometimes it may be useful to be able to refer to not just slot values but the entire fact on the right hand side of a rule. Jess allows us to bind a fact to a variable using the <− symbol:

```
?boring <- (boringrace)
=>
(retract ?boring))
```

Sometimes we may wish to place conditions on slot values, for example to allow a rule to match facts with slots containing a range of values rather than any value. Jess allows us to place such conditions using the `&:` symbol, as in Figure 2.8a. Since Jess 7, there is a shorthand version of this condition using curly braces: in this case we write {`odds < ?o`} as shorthand for `(odds ?odds&:(< ?odds ?o))`; the slot value is implicitly bound to a variable of the same name as the slot (which may cause problems if not used carefully). Such conditions can also be combined, for example in Figure 2.8b.

```
(location (timestamp ?t) (horse ?h1)
    (loc ?l1&:(< ?l2 ?l1)))
```

(a) Using the `&:` characters

```
(location (timestamp ?t) (horse ?h1)
    {loc < ?l1 && loc > (- ?l1 20)})
```

(b) Using the {} brackets

Figure 2.8: Restrictions on the values a slot may take

Now we can introduce the final conditional element mentioned (but not explained) above: `logical`. This asserts a *logical dependency* between the left hand side and the right hand side of a rule; specifically, a rule marked as `logical` is not only fired when the left hand side matches facts in the fact base, but also any facts asserted by the rule's right-hand-side are retracted when the left hand side no longer matches, as in Figure 2.9.

```
(logical (odds (horse ?h) (odds ?o))
    (not (odds (horse ?h1) \{odds < ?o\})))
 =>
(assert (favourite ?h))
```

Figure 2.9: Using `logical` to assert a logical dependency

Without the `logical` element, this rule would merely assert the fact `(favourite ?h)` every time the fact `(odds (horse ?h) (odds ?o)` appeared in the knowledge base without the fact `(odds (horse ?h1)` {`odds < ?o`}`)`; after adding `logical`, this rule will also retract the `favourite` fact every time the first `odds` fact disappears from the knowledge base, or the second `odds` fact appears in the knowledge base. We say that the fact `recommendation` is *logically dependent* on the facts in the left-hand side of this rule.

2.2.3 Execution Cycle

The typical execution cycle of an expert system (and the one used by Jess) consists of the following steps:

1. Match facts from the fact base against the left hand sides of the rules in the rule base; move matched rules onto the agenda
2. Order the rules on the agenda according to some conflict resolution strategy
3. Execute the right hand sides of (*fire*) the rules on the agenda in the order decided by step 2

Every time the command `(run)` is entered at the Jess prompt, Jess performs these steps, matching each rule against each fact once, until there are no more combinations of rules and facts to attempt matching. By default, Jess uses a "first matched, first executed" policy to order the agenda; however there are a number of ways in which the programmer can influence the conflict resolution.

Conflict resolution strategy

The "first matched, first executed" policy is an example of a *conflict resolution strategy* – an algorithm for deciding in which order to fire the rules on the agenda (also called the *conflict set*). By default, rules are fired in the order in which they are activated; Jess also provides another conflict resolution strategy where rules are fired in the reverse order of activation. In addition to this, it is possible to write your own conflict resolution strategy by subclassing the Java class `jess.Strategy`; we have used this mechanism to implement a random strategy for the language generation module (see Chapter 5).

Salience

Every rule in Jess has a *salience* value, which specifies how urgently a rule should be fired. Rules with a higher salience value are fired earlier than rules with a lower salience value; rules with the same salience value are compared according to the current conflict resolution strategy. The programmer can manually set a rule's salience value as an option to the `defrule` function.

Modules

Jess also provides a less intrusive method to influence the order of a program's execution: it allows us to partition the rule base and fact base into named modules. Thus we can group together rules and facts that operate on the same stage of the program's execution. Jess allows us to give *focus* to specific modules, thus temporarily enabling and disabling groups of rules and facts, while still letting the rule engine's own conflict resolution decide which rules from this module to fire.

By default, all Jess rules and facts live in the module `MAIN`. To define a new module, Jess provides the function `(defmodule)`, with the syntax `(defmodule <name>)`:

```
(defmodule appraisal)
```

Focus movement between modules is implemented using a stack; modules are pushed onto the stack, and then popped off and processed one by one. To manipulate the focus stack, we have several functions: `(focus <modulename>*)` pushes one or more modules onto the focus stack; `(clear-focus-stack)` clears the stack; `(pop-focus)` pops the next module off the stack and gives it focus; and `(get-current-module)` gets the current module.

For example, in the affect appraisal module, we have separated the rules into "appraise EECs" rules and "invoke ALMA" rules, so we can have the system appraise all its events, actions and objects against all EECs at once first, and then generate all its emotions together at the end.

Namespaces

By organising rules and facts into modules, we are also defining namespaces for our rules and facts. A name qualified with a namespace has the syntax `<namespace>::<name>`. All names that are not explicitly defined in a module are defined in the current module. If no modules have yet been defined, this is `MAIN`. If a name is explicitly referenced, then only that module will be searched. Otherwise, the current module will first be searched, and if no name is matched then `MAIN` will be searched.

2.2.4 Interaction with Java

Jess is not only implemented in Java, but also provides mechanisms for Jess code to interact with Java code. This is especially useful in the modular threaded design used in ERIC (see Chapter 3). The two ways Jess code in ERIC interacts with Java code are by storing Java objects in Jess and calling their methods from within Jess, and custom Jess user functions written in Java. Jess provides a third mechanism, called *shadow facts*, where a Java Bean object appears in Jess as a Jess Fact; this mechanism is not used in ERIC and therefore not explained here. For more information, see Hill [2003].

Calling Java methods from within Jess

The key to this style of interaction is that Jess variables can have not only numbers, strings, and symbols as their value, but also Java objects. Jess provides a number of functions to interact with Java objects. The function `(new)` creates a new object; it takes as arguments the class name, and any arguments to the class constructor:

```
(defglobal ?*eecManager* =
    (new de.dfki.racereporter.affect.CreateAffectInputEEC
        "HorseRaceCommentator"))
```

Now that we have a Jess variable that refers to a Java object, we can call its methods with the `(call)` function, and get and set its public members with the `(get-member)` and `(set-member)` functions respectively:

```
(bind ?logger (get-member de.dfki.racereporter.util.Logger
    log))
(call ?logger facts (str-cat "retracting \"" ?text
    "\" : redundant."))
```

These three functions can also take a class name in place of the object reference argument (as in the call to Logger); in this case Jess will access static members and methods. To make method calls read more like Java, the keyword `call` is optional when calling non-static methods:

```
(?*eecManager* setAppealingness ?el ?a)
(?*eecManager* processEECs)
```

User functions

Alternatively, we can write a Jess function in Java (rather than in Jess), and then import this into a Jess instance. Such a function is called a *user function*. A user function is used in Jess exactly the same way as a normal function that was defined with `(deffunction)`; however since the function was defined in Java rather than inside the Jess instance, it can interact with other Java classes, maintain state information, and take advantage of the full power of Java. To add a user function to a Jess instance, we need to be able to access that Jess instance from within Java: to do this we use the jess.Rete class.

The `jess.Rete` class Every Jess instance is in fact an instance of the `jess.Rete` class. Thus we can instantiate instances of Jess inside Java code by creating new objects of this type. The `jess.Rete` class provides useful methods for interacting with and controlling the Jess instance, such as `reset()`, `clear()`, `assertFact()`, and `run()`, which correspond to the Jess functions `(reset)`, `(clear)`, `(assert)`, and `(run)` respectively. Also, it has methods for adding `deffacts`, `defrule`, and `deftemplate` statements, as well as various diagnostic and debugging methods. A whole file of Jess code is loaded into a `jess.Rete` object with the method `batch()` (which corresponds to the Jess function `(batch)`).

The `jess.UserFunction` class A Jess user function takes the form of a Java class that extends `jess.UserFunction`. We make this new function available in a Jess engine with the `addUserFunction(UserFunction)` method of `jess.Rete`. An example of such a user function is the `de.dfki.racereporter.jess.commands.Speak` class, which provides a function called `(charamel-speak)` to send a TTS command containing its string argument to the Charamel system. The Java package `de.dfki.racereporter.jess.commands` contains a number of such user functions to interact with Charamel.

Chapter 3

System Architecture

The system architecture of ERIC is composed of a number of homogeneous modules, running in individual Java threads. A modular architecture was chosen to make the agent easy to reuse, modify and extend, by adding or removing modules as appropriate. Parallel execution in multiple threads was chosen for performance: when commentating on a real-time continuous event, the agent must be responsive. Since the main focus of ERIC was to be on the knowledge processing, the components used for 3D visualisation of the agent and speech generation were used off-the-shelf.

3.1 Design Aims

Three of the overall aims of the agent are particularly relevant to the design: homogeneity, domain-independence and reactivity.

Homogeneity For ease of use and maintenance, it was desired that the framework consist of homogeneous modules. Thus, the agent's modules each consist of an expert system written in Jess. The structure of a module is detailed in Section 3.4.

Domain-independence The agent should require minimal additional work to implement it for a new domain, so that it can be easily configured to provide commentary on any situation. Thus the different functionalities of the system were modularised in a way that attempted to minimise the number of domain-specific modules. In the case where a module was partly domain-specific and partly domain-independent, the domain-specific parts are kept in a separate file from the domain-independent parts.

Reactivity The agent should be able to react to new or changing events in a dynamically changing environment. Thus each module runs in its own Java thread, allowing the different kinds of reasoning (for example affect appraisal and knowledge generation) to occur in parallel.

3.2 Modular Architecture

Figure 3.1 shows the modular architecture of the system. Modules in orange are provided/used off-the-shelf; modules in violet are domain-independent, modules in green are domain-dependent. Dotted lines and modules are future work. The large dotted box surrounds the core of the ERIC framework – not only the inputs but also the output embodied agent player could be exchanged.

The interface module transforms the input from the world into Jess facts suitable for the knowledge module. In this diagram it is labelled the RaceSim interface, but this module is exchanged for an appropriate interface module for each new domain that ERIC will commentate, such as dTank.

The knowledge inference module deduces a world model from the input facts it receives from the world interface module. This world model also consists of Jess facts, and is deduced from the input facts by a Jess expert system. The world model is sent to the natural language generation, affect observer, and gesture modules.

The affect observer module finds events, actions and objects in the world model, and sends facts describing these to the affect module. The affect module appraises these events, actions and objects against a number of emotion-eliciting conditions, and sends these appraisals to ALMA, which maintains an affective state. The current mood and emotion of the affective state are in turn represented by the affect module as Jess facts, which are sent to the natural language generation, gesture, and facial expression modules, and in future also to a prosody module.

The natural language generation module generates candidate speech utterances from the world model. Templates are matched against the facts in the world model, and then matching templates are filled from an utterance lexicon. Lexicon entries are chosen based on the current affective state. The candidate utterances are sent to the fusion module, which maintains a discourse state and selects a single output utterance from the candidate utterances based on the utterances' priorities and discourse coherence.

The gesture module generates candidate gestures from the world model and the affective state. These are also sent to the fusion module which selects the most appropriate candidates. Similarly, the facial expression module generates facial expressions from the affective state, and sends these to the fusion module. In future the prosody module will also generate prosody from both the affective state and the candidate utterances, and send this to the fusion module for coordinating with the selected candidate utterances.

The fusion module is responsible for coordinating coherent output from all the candidate utterances, gestures, facial expressions and prosody that it receives from the various modules. The fusion module maintains discourse coherence by selecting appropriate utterances from all the candidates. It also coordinates messages from different modules that must match, for example matching a candidate utterance with its prosody or its associated gesture, or matching associated gestures and facial expressions. As well as this, the fusion module can check multiple simultaneously generated gestures or facial expressions to ensure they do not conflict, and to combine them or select one depending on the capabilities of the output system. Since Charamel can blend gestures and facial expressions smoothly, the fusion module currently does not need to perform this processing, but conceivably a different output system might require it.

The fusion module provides feedback to the natural language generation, gesture generation, and affective appraisal module in the form of context information and knowledge state information. This informs the natural language module of the current discourse coherence state and what information has been conveyed thus far, and the gesture and facial expression modules of the current semantic focus of the discourse. In addition, this channel allows for output quality evaluation mechanisms such as anticipation feedback [Alassa and Jameson, 1996], or conversational status modelling such as that of Rumpler [2007 (to appear)].

3.3 Off-the-shelf Components

Because the main focus of the work was on the architecture and knowledge processing, several components were used off-the-shelf. In particular, the graphical and text-to-speech functionalities were provided by Charamel and Nuance RealSpeak Solo respectively; both were treated as a black box. In addition, Patrick Gebhard's Java implementation of the ALMA affective model was used in the affect module.

3.3.1 Charamel

Charamel[1] [Charamel GmbH, 2007] provides a 3D animated humanoid character, which is capable of facial expressions, hand and body gestures, and lip movements synchronised to speech. The character is animated in real-time using a skeleton model; it is able to not only play back prefabricated animations, but also generate smooth transitions between these animations in real-time. Facial expressions are generated using morph targets, again allowing smooth transitions between expressions in real-time. Charamel can be connected to commercial text-to-speech engines, and using the output from these engines lip movements are automatically generated to match both the timing and the phonemes of the generated speech.

The character can be controlled via a scripting language called CharaScript. A network socket connection listens for commands from a controller system; these commands call up functions in the CharaScript scripts that (for instance) start the playback of a gesture, change the character's facial expression, or send a text string to the text-to-speech interface. As well as calling specific gestures, the CharaScript maintains a list of "idle"

[1] `http://www.charamel.de`

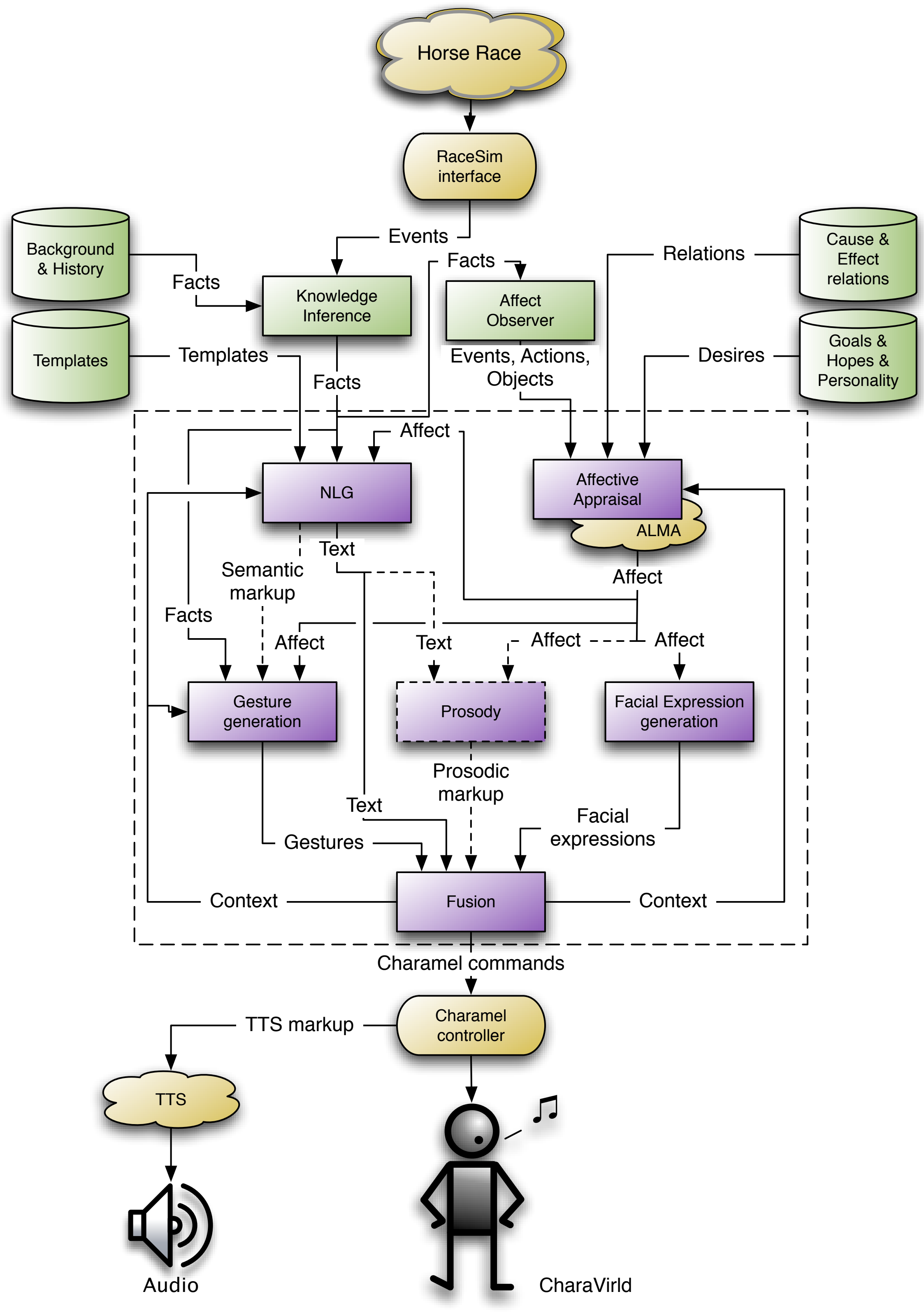


Figure 3.1: The modular architecture of the system

gestures, and whenever no specific gesture has been requested the CharaScript randomly chooses from this list. This has the effect that the character is never frozen perfectly still, which would be unrealistic. A similar list of "speaking" gestures fill the same role while the character is speaking; this list is distinct from the "idle" list, for realism: a human would use different gestures while speaking than while listening.

The character model used for the ERIC system is called "Paul". The character is rendered in an application called CharaVirld, shown in Figure 3.2.

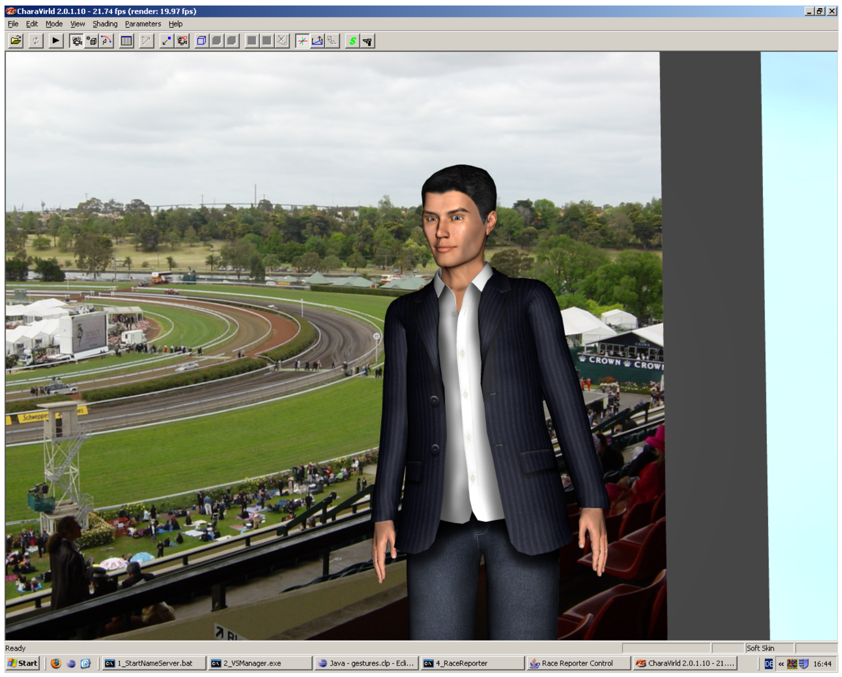


Figure 3.2: A screenshot of Charamel's CharaVirld environment

3.3.2 Nuance RealSpeak Solo

Nuance RealSpeak Solo[2] [Nuance Communications Inc, 2007] is a commercial concatenative text-to-speech system. The input text can contain limited prosodic markup, to affect the average pitch, rate and volume of the speech (although these possibilities are voice-dependent). The voice used is an American English voice called "Tom". RealSpeak Solo is interfaced with the Charamel system via a CGI script running on a web server: this produces audio files and viseme files (used by Charamel's lip movement synchronisation) from text input.

[2]http://www.nuance.com/realspeak/solo/

3.3.3 ALMA

ALMA [Gebhard, 2005] is a model of affect, implemented as a Java library. ALMA models affect in terms of three layers: the Big Five model for personality [Goldberg, 1990], a three-factor (pleasure, arousal, dominance) mood model [Mehrabian, 1996], and the OCC cognitive model of emotions [Ortony et al., 1988] that are generated from input emotion-eliciting conditions. The model is explained in detail in Chapter 6.

ALMA generates an affective state from input emotion-eliciting conditions. The affect model of ERIC generates these emotion-eliciting conditions and passes them to ALMA for further processing. ALMA is linked with the affect module in Java, and Java methods are used to communicate emotion-eliciting-conditions and affective states. This interface is described in detail in Section 3.5.

3.4 ERIC Standard Module Design

To enable the agent to react in a timely way to new or changing events, each of ERIC's components runs in a parallel Java thread. Because the various components were desired to be homogeneous, a standard module design was used: the Java class implementing this module is called `JessThread`. This module wraps around an instance of a Jess engine (a `jess.Rete` object), and implements the `java.lang.Runnable` interface so it can be run in its own thread. For low coupling between the different modules of ERIC, the observer pattern is used to exchange information between modules; this way additional modules can be added to, or some modules left out of, ERIC without changing any module's code. The design of an ERIC module is shown graphically in Figures 3.3 and 3.4.

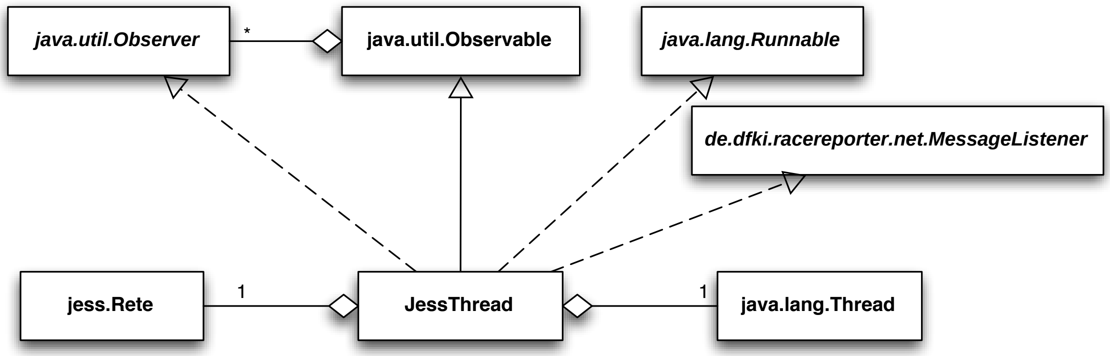


Figure 3.3: A UML inheritance diagram of the standard ERIC module

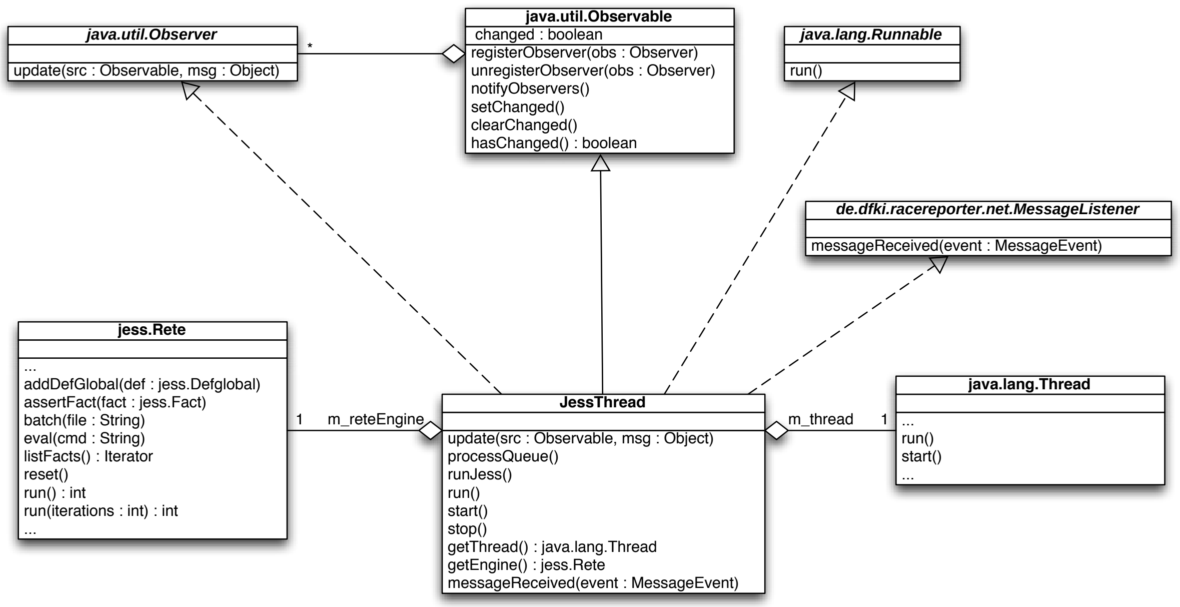

java.util.Observer
update(src : Observable, msg : Object)
*
java.util.Observable
changed : boolean
registerObserver(obs : Observer)
unregisterObserver(obs : Observer)
notifyObservers()
setChanged()
clearChanged()
hasChanged() : boolean
java.lang.Runnable
run()
de.dfki.racereporter.net.MessageListener
messageReceived(event : MessageEvent)
jess.Rete
...
addDefGlobal(def : jess.Defglobal)
assertFact(fact : jess.Fact)
batch(file : String)
eval(cmd : String)
listFacts() : Iterator
reset()
run() : int
run(iterations : int) : int
...
1
m_reteEngine
JessThread
update(src : Observable, msg : Object)
processQueue()
runJess()
run()
start()
stop()
getThread() : java.lang.Thread
getEngine() : jess.Rete
messageReceived(event : MessageEvent)
m_thread
1
java.lang.Thread
...
run()
start()
...

Figure 3.4: A UML class diagram of the standard ERIC module

3.4.1 Enclosed Jess Instance

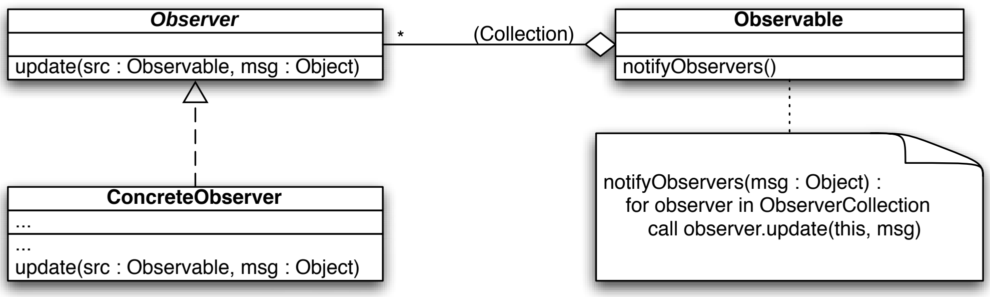
As we saw in Section 2.2.4, Jess provides a Java API through which Jess can be invoked. We can create an instance of Jess by instantiating an object of the class `jess.Rete`; this class has methods for interacting with the Jess instance from Java. Each module in the ERIC framework contains such a Jess instance; the facts generated by Jess reasoning are passed between modules using the observer pattern. Whenever a module receives facts from another module, these facts are asserted inside the enclosed Jess instance, and then `(run)` is called on the Jess instance.

3.4.2 ERIC Module as a Java Thread

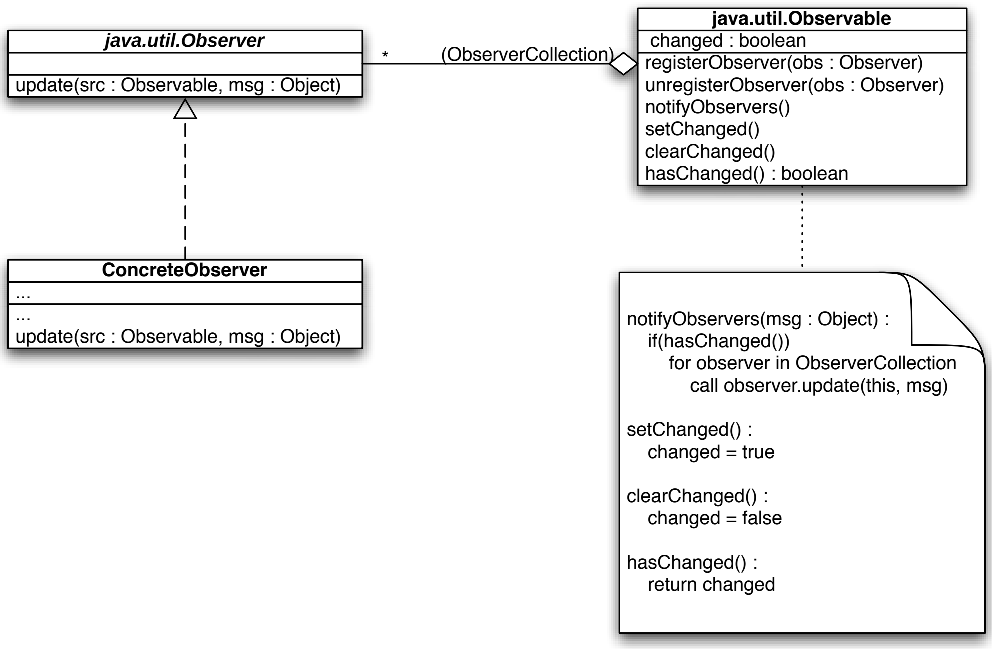
In order to run in its own thread, a class must implement the `java.lang.Runnable` interface; this interface specifies a single method `run()`. The `java.lang.Thread` class has a constructor taking a `Runnable` as an argument; a thread thus created will run the `Runnable` object's `run()` method when it is run. Aside from the initialisation of the Jess instance and the observer pattern methods (see below), all of the processing of an ERIC modules (in particular, calling `(run)` on the Jess engine) is done inside this `run()` method.

3.4.3 Communication via the Observer Pattern

The observer pattern is a way of exchanging information between classes without linking the classes together in code (before runtime) [Gamma et al., 1995, Wikipedia, 2007]. Figure 6.6 illustrates the structure of the observer pattern in UML. At runtime an *observer* object registers itself with the *observable* object, and the observable object sends notifications to all classes that have registered themselves with it. In this way the observer object receives notifications from the observable object. This pattern leads to low coupling between classes, and easy interchangeability (due to the clearly defined interface between the classes). The Java implementation of the pattern adds a “changed” flag to the Observable object: the flag can be set and cleared, and a notification is only sent to registered observers if the flag is set.


Observer
update(src : Observable, msg : Object)
*
(Collection)
Observable
notifyObservers()
ConcreteObserver
...
...
update(src : Observable, msg : Object)
notifyObservers(msg : Object) :
for observer in ObserverCollection
call observer.update(this, msg)

(a) The basic observer pattern



(b) The observer pattern as implemented in Java

Figure 3.5: UML class diagrams of the observer pattern

ERIC's modules are both observable and observers; thus they can both send events to and receive events from other modules. Since the `update()` method in the observing module is called from the observed module's thread, that method also runs in this thread. To keep the threads separate, the facts received by the observer are placed in a queue; on its next loop the observing module's thread will add these facts to Jess before running the Jess engine.

The message that is sent by an observed module to its observing modules is an object of the `NewFactsEvent` class. This class encloses the facts from a Jess engine, allowing the modules in the ERIC framework to interchange facts between Jess instances.

3.4.4 Execution Cycle of a Module

The Jess engine is initialised with the templates file, the Jess files, and any user functions. Then the engine is `reset` so that any `deffacts` come into effect.

Then we loop infinitely (until the program is terminated):

- `run` the Jess engine
- list facts and build a NewFactsEvent from them
- notify observers of these new facts with the NewFactsEvent
- wait for new facts to arrive in our queue
- move the facts from the queue into the Jess engine

This cycle is illustrated in Figure 3.6.

3.5 The Affect Module

The affect module is a slightly enhanced version of the basic ERIC module, since it also needs to interact with ALMA. In addition to the above behaviour, the affect thread launches the ALMA module at startup, and can assert emotion facts from ALMA-generated emotions; the thread also has the ability to pass emotion-eliciting condition appraisals and tag appraisals to ALMA, from which ALMA generates emotions.

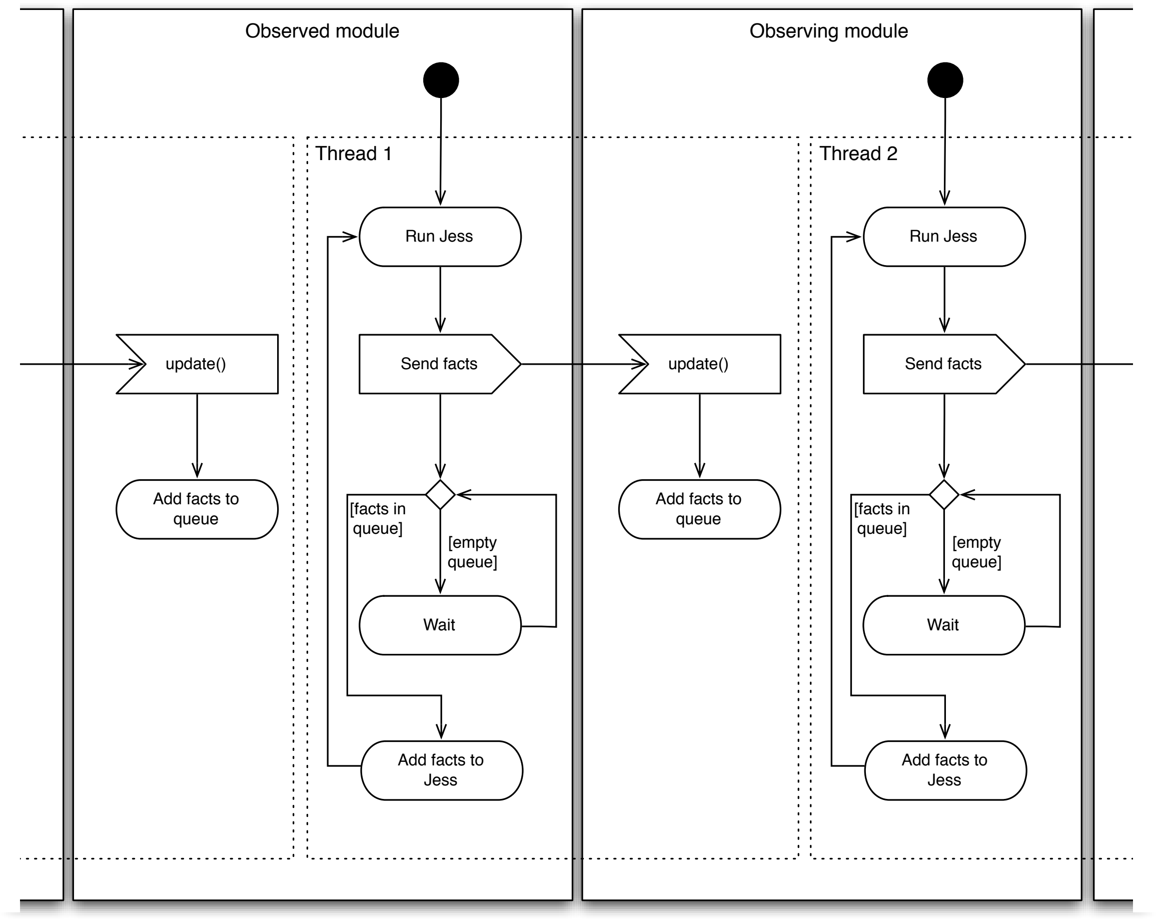


Figure 3.6: A UML activity diagram of the ERIC modules. The `update()` and “Add facts to queue” actions occur in the thread of the observed module.

3.5.1 AffectThread Class

The affect module is implemented as a Java class that extends the usual ERIC module class. It extends the functionality of the parent class in two ways: by initialising an ALMA object at creation, and by listening for emotion events from the ALMA object. The AffectThread class is illustrated in Figures 3.7 and 3.8. At object creation, the affect module initialises an ALMA *affect manager* object. The expert system in the affect module reasons about the world state to appraise emotion-eliciting conditions (EECs), either tags specifying sets of EEC values, or the raw values themselves. The affect manager receives these EECs and uses them to compute emotions and update the agent's affective state. (See Chapter 6 for a detailed description of this process.) This affective state is then communicated to listeners (again via the observer pattern): at initialisation the affect module adds itself as an observer to the affect manager. The affect module maintains a queue for affect state updates from the affect manager, similar to the queue for incoming facts. Incoming affect state objects are placed in this queue. When the affect module checks the facts queue for facts to add to the Jess engine, it also checks the affect state queue, and generates emotion and mood facts from the queued affective state.

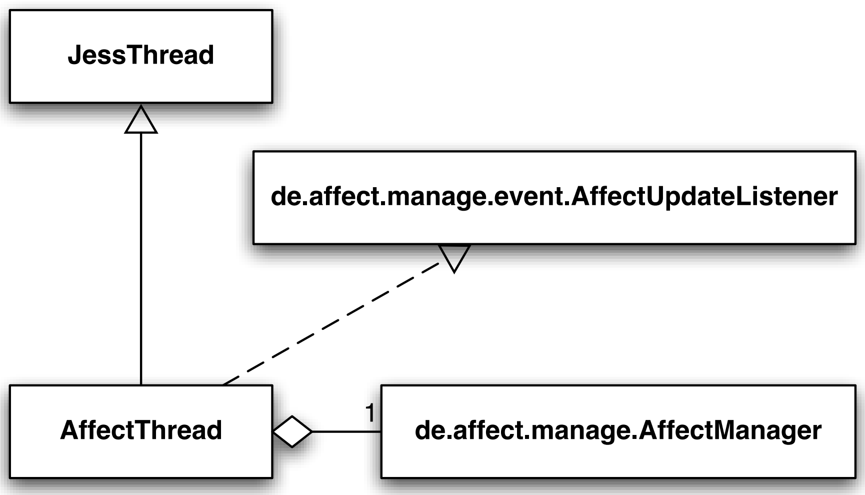

JessThread
de.affect.manage.event.AffectUpdateListener
AffectThread
1
de.affect.manage.AffectManager

Figure 3.7: A UML inheritance diagram of the ERIC affect module

3.5.2 Affect User Functions

In order for the emotion-eliciting condition tags generated by the expert system (see Section 6.2.4) to be passed to the affect manager, some Jess user functions were implemented in Java: `CreateAffectInputEvent`, `CreateAffectInputAction`, and `CreateAffectInputObject`, all subclasses of the abstract class `CreateAffectInput`.

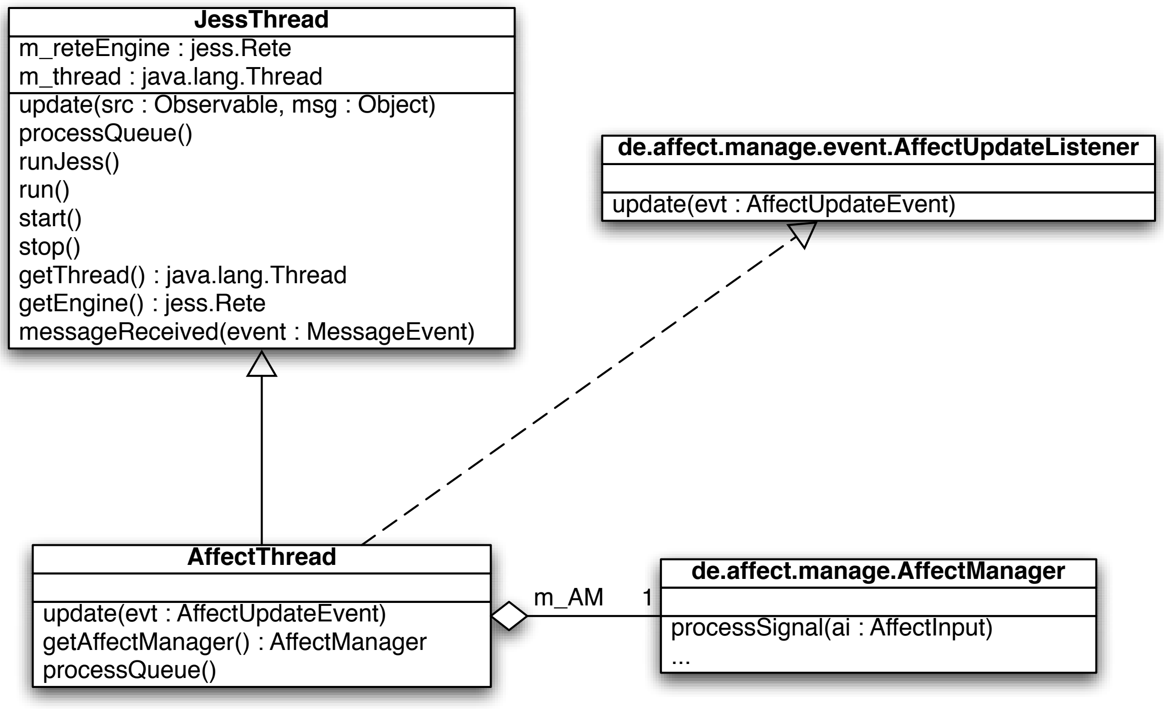

JessThread
m_reteEngine : jess.Rete
m_thread : java.lang.Thread
update(src : Observable, msg : Object)
processQueue()
runJess()
run()
start()
stop()
getThread() : java.lang.Thread
getEngine() : jess.Rete
messageReceived(event : MessageEvent)
de.affect.manage.event.AffectUpdateListener
update(evt : AffectUpdateEvent)
AffectThread
update(evt : AffectUpdateEvent)
getAffectManager() : AffectManager
processQueue()
m_AM
1
de.affect.manage.AffectManager
processSignal(ai : AffectInput)
...

Figure 3.8: A UML class diagram of the ERIC affect module

Each of these user functions takes as arguments the agent performer, the eliciting event, action or object, the elicited tag, and the elicited intensity. From this information it generates an *affect input document* in an XML format compatible with ALMA, and passes this document to the affect manager.

3.5.3 Emotion-Eliciting-Condition Manager Class

Sending plain emotion-eliciting condition values (see Section 6.2.5) to the affect manager is slightly more complicated than sending EEC tags, since there are more possible combinations of EEC values than there are tags, and not all possible combinations are valid. Thus rather than using a set of user functions, the interface for the EEC values was designed as a manager class that is instantiated inside Jess, and whose methods are then called by the Jess rules.

The EEC manager class maintains an EEC value for every elicitor it has seen. It has *setter* methods for each of the possible emotion-eliciting conditions, which are called from Jess to update these EEC values; it also has a `clear()` method for clearing all its known EEC values. Finally, it provides a `processEECs()` method, which builds an affect input document for each known elicitor out of its stored EEC values and sends this document to the affect manager, before clearing the EEC values. In the Jess module, rules match valid combinations of emotion-eliciting conditions; when such a rule fires, it adds the EEC values one by one to the EEC manager via the setter methods, and finally calls `processEECs()` to send these values to the affect manager.

3.6 Design Considerations

Through the implementation process, two aspects of this design have emerged as requiring special care and attention: the transferring of `jess.Fact` objects between modules, and the question of timing.

3.6.1 Copies of Fact Objects

When Jess facts are transferred between modules, it is necessary to make a copy of the underlying `jess.Fact` Java objects before adding the fact to a new `jess.Rete` engine. This is due to the phenomenon that if a `jess.Fact` object is present in more than one Jess engine, then the Jess pattern matcher will continue to activate rules which have that fact on their left-hand side, even after the fact has been retracted from that engine, until the fact is retracted from all engines in which it is present. Making a copy of the underlying Java object (using the `copy()` method) avoids this problem, since the facts in different Jess engines are now represented by different underlying Java objects. However this means that the facts no longer have the same fact ID. This could present a problem, as the "audience knowledge model" used in the natural language generation module stores the IDs of previously-conveyed facts. Currently this model is only used within the natural language generation module, so the ID inconsistency across modules is not a problem; however it could easily cause problems in the future, and must be kept in mind when implementing an agent using ERIC.

3.6.2 Timing Issues

The second issue is that of the time ERIC takes to reason. Since we wish ERIC's commentary to be produced in real-time, it is necessary for ERIC to reason about world events faster than they occur. For the currently implemented domains, new world events are presented to the agent every second; this means that all of ERIC's threads must complete their reasoning in less than one second. In some situations this can be a difficult requirement to meet: in particular, because of the way the rule selection process is implemented in Jess, the time taken to process the rules increases with the number of facts matched against rules: thus rules that partially match can have a significant effect on the speed of Jess's processing. To avoid this situation, the Jess manual [Hill, 2006] suggests:

- Put the most specific patterns near the top of each rule's LHS.

- Put the patterns that will match the fewest facts near the top of each rule's LHS.
- Put the most transient patterns (those that will match facts that are frequently retracted and asserted) near the bottom of a LHS.
- use the `view` command to find out how many partial matches your rules generate.

However sometimes it can be impossible or infeasible to follow these guidelines. To reduce the number of partial matches, the current implementations of the ERIC framework contain *cleanup* rules that retract out-of-date facts, thus keeping the number of facts in the knowledge base somewhat constant, and preventing a gradual slowdown in Jess's processing over time. These cleanup rules have the form:

```
(time ?t)
?fact <- (fact {timestamp < ?t})
=>
(retract ?fact)
```

and their source code is in the file `templates.clp`, so that the same rules are imported into every Jess engine along with the Jess templates.

This measure carries with it the significant disadvantage that knowledge about past events in the world is lost, and therefore the agent cannot reason about history, or trends across time. The solution we have used to mitigate this disadvantage is to formulate the condition `{timestamp < (- ?t 5)}` instead of `{timestamp < ?t}`; thus we can store a limited amount of history, and we can also tweak the amount of stored history on a fact-by-fact basis.

A final control over the timing of modules has been implemented in the Jess thread class itself. Here, the execution of the Jess engine may be interrupted if it exceeds a predefined time limit. This kind of hard interrupt compromises the completeness of the agent's reasoning process, since not all Jess rules are able to fire. Thus this functionality is currently disabled. However it reamins an option when all other timing measures fail.

Chapter 4

Knowledge Inference and Domain Modelling

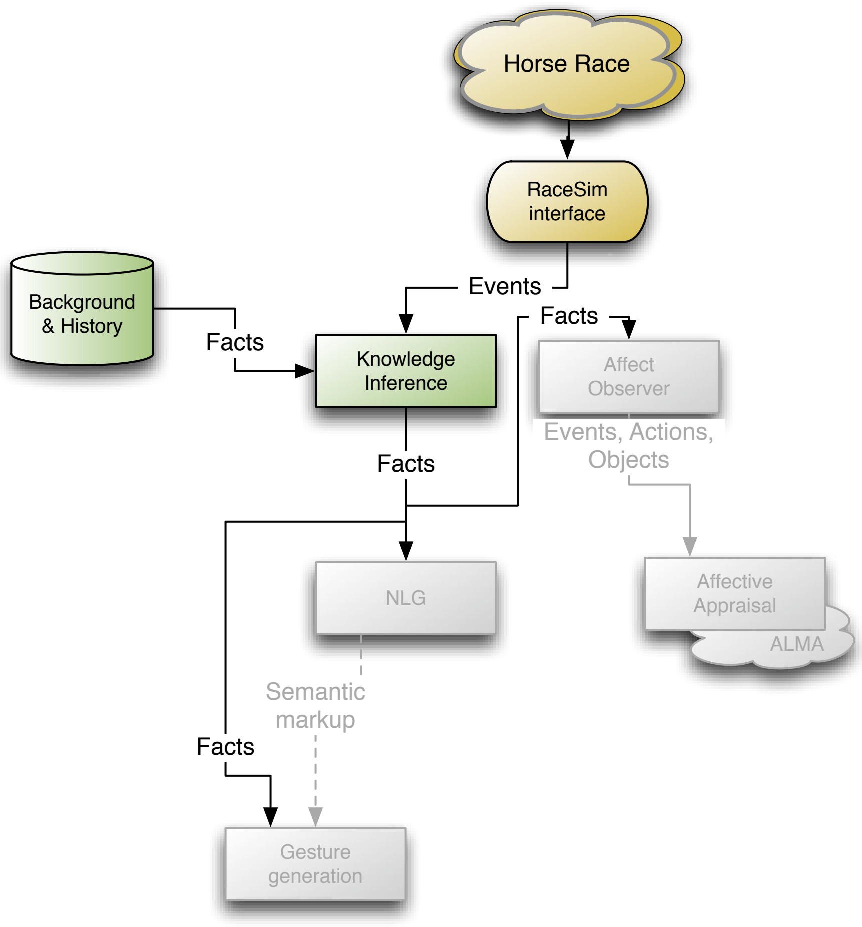


Figure 4.1: The knowledge module in the system architecture

The knowledge module of ERIC is responsible for elaborating the limited incoming information from the domain interface into a rich world model which then forms the basis for the agent's natural language and affect generation. This limited information consists of the location and speed of each horse every second for the horse race, and information about the location and actions of each tank in the dTank environment. The knowledge module's elaboration may also use background knowledge that has been specified when configuring the agent, such as the names of the horses and tanks, the weather, the style of the track or battlefield, or the historic performance of a horse or a tank. From this information the knowledge module can deduce facts and events such as a horse overtaking another horse, the order of the horses in the race, or a tank aiming its guns at another tank.

The knowledge module is implemented as an expert system in Jess (see Chapter 2): information from the world interface is represented as Jess facts, and then by logical inference over these facts (via Jess rules) many more facts are inferred from the input information. The set of all facts in the knowledge module make up the world model.

4.1 Aims

The relevant research aims for the knowledge module are domain-independence, reactivity and behavioural complexity. For the last two, it is important that the knowledge module produce a rich world model, not only to allow the agent to commentate a variety of aspects of the world, but also to remove the need for the other modules in the architecture to do their own knowledge reasoning. For the first, although it is clear that the knowledge module will be domain-dependent (as the world model necessarily describes the domain), it is desired that as much as possible of the system's domain-dependent processing takes place in the knowledge module, to allow the other components to be reused across domains almost unchanged.

4.2 Domains

Currently the ERIC commentator has been implemented for two domains: a simulated horse race, and a tank combat simulation.

4.2.1 The RaceSim Horse Race Simulation

The RaceSim horse race simulation is provided by the organisers of GALA '07 for the race reporter challenge[1]. In the RaceSim simulation, four horses ("Azure", "Carmine",

[1]The race reporter challenge was one of the categories at the Gathering of Animated Lifelike Agents, at the 7th International Conference on Intelligent Virtual Agents – IVA 2007. The race reporter challenge using the RaceSim system is described at `http://hmi.ewi.utwente.nl/gala/racereporter`. ERIC was a winning entry in the race reporter challenge: the GALA entries

"Eben" and "Topaz", wearing blue, red, black and green respectively) race along a 2000m track (shown in Figure 1.1). Every second the simulation sends a message to an open network socket containing the speed and location of each of the four horses. The RaceSim Interface module of ERIC connects to the RaceSim simulation via this network socket and transforms the information in the message into Jess facts.

As well as the speed and location data from the interface, the RaceSim world model uses background knowledge about the horses' names, ages, odds, handicaps, owners, jockeys and colours, the horses' past success (races recently won), the length of the race, the prizemoney, the weather, the condition of the track, the horses' preferred track condition, and future races the horses are entered in. As the RaceSim is a simulation, this information is purely fictional, and specified as part of the domain-specific configuration.

From this background information and the incoming data, the knowledge module builds a world model consisting of facts representing various information about the race and the horses before, during and after the race. These facts include the race favourite, the race order, the distance between adjacent horses, a change in this distance, a change in horse speeds, overtaking events, change-of-leader events, horse falling events, getting up events, the final placings, and events for when the horses pass various landmarks along the racetrack. As well as this, the world model contains facts to predict overtaking events, and facts that summarise the race as surprising, predictable, boring or exciting, and that categorise the finishing field as spread or close.

4.2.2 The dTank Tank Combat Simulation

dTank is a simulation of a tank battle on a two-dimensional battlefield, intended for competitive evaluation of adversarial real-time cognitive models [Councill et al., 2004, Morgan et al., 2005]. The latest version of dTank (version 4, see Figure 1.3) simulates a sophisticated battle environment, with two armies of any number of battalions of up to four tanks, each battalion commanded by a commander agent. For this simulation an earlier version of dTank was used (see Figure 1.2); in this version any number of individual tanks battle against each other.

The dTank environment was modified slightly to output world information as Jess facts via the Java observer pattern; thus unlike the RaceSim simulator, dTank does not run in its own process but is opened by the dTank Interface module of ERIC, which then registers itself as an observer to receive the Jess facts from the world.

The dTank environment provides slightly more facts about the world than the RaceSim simulation: we are given the location and orientation of each tank, as well as the orientation of its gun turret, and its remaining health, shields and ammunition. Also, we are given a Jess fact whenever a tank turns, turns its gun turret, moves, raises its shields,

can be viewed at `http://hmi.ewi.utwente.nl/gala/finalists`.

or begins to shoot. As well as facts about the tanks, we are also given the locations of the stones in the simulated battlefield.

As in the RaceSim scenario, the knowledge module uses background knowledge to supplement the information coming from the dTank environment. The knowledge used covers the weather, the commander of the tanks, and a summary of a tank's past performance. Unlike the RaceSim scenario, where the simulation provides exactly four horses with predetermined colours and names, the dTank environment allows any number of tanks to join the battle, and participating tanks can specify their own names; this means that many kinds of knowledge that were specified as facts in the RaceSim knowledge module must be specified as rules in the dTank module, so that the appropriate facts are "generated" when a new, unknown tank joins the battle.

The world model produced by the dTank knowledge module contains facts representing both the situation of the tanks and the world, and the behaviour of the tanks. From the location of stones in the battlefield map, the knowledge module characterises the map as sparse or dense, and can identify rows, columns and clusters of stones in the map. As well as all the status facts obtained from the dTank interface, the situation facts include additional facts representing the same information but with directions given as compass points rather than angles from the horizontal, and with tank locations given relative to other tanks rather than just absolute in the world. Also, the world model contains facts that record for how long each of these facts has remained unchanged. The behaviour facts include events such as a tank aiming its guns at another, a tank moving towards or away from another, and a general direction trend in a tank's movement; also the knowledge module is able to detect when a tank has been hit, and in some cases can identify the attacker corresponding to such a hit.

In addition to this reasoning, the dTank knowledge module is also responsible for summarising each second of activity into a single *quantised* fact: whereas in the RaceSim simulation the horse facts are received regularly every second, the dTank environment sends facts only when they are relevant, with a configurable update time (currently at 50ms).

4.3 Declarative World Model

The facts from the domain interface are asserted in a Jess expert system, the knowledge module. The rules in the knowledge module deduce a large number of more elaborate facts from the input domain events and background knowledge. For instance, the RaceSim knowledge module deduces facts representing the current order of the horses, the distance between pairs of horses, and when a horse is about to overtake another horse. Similarly, the dTank knowledge module deduces facts representing the location of tanks relative to each other, the compass-point orientation of each tank and each turret, and successful shots.

Table 4.1 lists the facts that ERIC receives from the two input domains. Table 4.2 lists the facts that ERIC assumes as background knowledge for the two domains; these facts are mostly invented by the domain author, although they could in theory also contain some real world knowledge.

RaceSim	**dTank**
Timestamp	Location of each stone on the field
Location of each horse	The name of each tank
Speed of each horse	The location of each tank
	The orientation of each tank
	The orientation of each tank's gun turret
	Each tank's health
	Each tank's shields status
	Each tank's ammunition levels
	A tank is commanded to fire
	A tank is commanded to raise its shields
	A tank is commanded to move forward
	A tank is commanded to rotate
	A tank is commanded to turn its turret

Table 4.1: Facts provided by the two input domains

RaceSim	**dTank**
Odds of each horse	The current weather
Each horse's name	Each tank's commander
Each horse's age	A summary of a tank's past success
Each horse's owner	
Each horse's jockey	
Each horse's colour	
Each horse's handicap	
The length of the racetrack	
The prizemoney available	
The condition of the track	
Each horse's preference in track condition	
A past win of a horse	
A future race of a horse	

Table 4.2: Background knowledge facts assumed for the two input domains

Table 4.3 lists the facts that constitute the world model for each of the two domains. These facts are generated from the input facts and the background knowledge using declarative rules. Thus for example, a horse is deduced to be about to overtake another horse when it is less than 10 metres behind the other horse and has a higher speed. Similarly, a tank is judged to be aiming at another tank if it is turning its turret, and

RaceSim	**dTank**
Whether the race is over	Each tank's kills score
The distance between adjacent horses	A summary of the terrain style
A horse is about to overtake another horse	A row of stones on the map
A horse has overtaken another horse	A column of stones on the map
A horse failed to overtake after being about to	A cluster of stones on the map
A horse is about to take the lead	The map quadrant in which each tank is located
A horse has taken the lead	Relative location of pairs of tanks
A horse failed to take the lead after being about to	The compass direction of a tank's orientation
A horse has drawn level with another	The compass direction of a tank's gun turret
A horse has broken free of another	The compass direction of a tank's movement
A horse leads the pack just before the finish	A tank's turret is currently turning
Each horse's cardinal position	A tank is aiming at another tank
Each horse's final placing in the race	A tank's shields are currently up
A horse has fallen over	A tank is under fire
A horse has gotten up again	A tank has been destroyed
The duration in which the order of the horses hasn't changed	A tank's shot hit another tank
The duration in which the speed of the horses hasn't changed	A tank has moved into a new map quadrant
An increasing gap between adjacent horses	A tank's location is unchanged
A decreasing gap between adjacent horses	A tank's orientation is unchanged
An increase in the speed of a horse	A tank's gun turret orientation is unchanged
A decrease in the speed of a horse	A tank's health is unchanged
The remaining distance left to race	A tank's shields status is unchanged
The horses have left the start gates	A tank's ammunition levels are unchanged
The lead horse is approaching the first turn	A tank's kills score is unchanged
The lead horse has entered the first turn	
The lead horse is approaching the back straight	
The lead horse has entered the back straight	
The lead horse is approaching the second turn	
The lead horse has entered the second turn	
The lead horse is approaching the home straight	
The lead horse has entered the home straight	
The lead horse is approaching the finish	
The winner took the lead just before the finish	
A horse lost the lead just before the finish	
The winner held the lead position for a lot of the race	
The race was exciting	
The race was boring	
The field was spread out at the finish	
The field was very close at the finish	

Table 4.3: Generated knowledge facts for the two input domains

the destination orientation of the turret would point it at the other tank. Examples of these rules are shown in Figures 4.2 and 4.3.

```
(Horse $h_1$ is in location $l_1$)
(Horse $h_2$ is in location $l_2$, with $l_2 \leq l_1$)
(Horse $h_3$ is in location $l_3$, with $l_3 \leq l_1$)
(Horse $h_4$ is in location $l_4$, with $l_4 \leq l_1$)
  $\rightarrow$
(Horse $h_1$ is in first place)
```

```
(Horse $h_1$ is in $p_1$-th place)
(Horse $h_2$ is in $p_2$-th place, where $p_2 = (p_1 + 1)$)
(Horse $h_1$ is in location $l_1$)
(Horse $h_2$ is in location $l_2$)
  $\rightarrow$
(Horse $h_1$ leads horse $h_2$ by $(l_1 - l_2)$ metres)
```

```
(At time $t_1$, horse $h_1$ is in location $l_1$)
(At time $t_1$, horse $h_2$ is in location $l_2$, with $l_2 < l_1$)
(At time $t_2$, where $t_1 < t_2 < (t_1 + 3)$, horse $h_1$ is in location $l_3$)
(At time $t_2$, horse $h_2$ is in location $l_4$, with $l_4 > l_3$)
  $\rightarrow$
(At time $t_2$, horse $h_2$ has overtaken horse $h_1$)
```

```
(The race is over)
(Horse ?h is in fourth place)
(Horse ?h is in location ?l, with ?l > 1900 metres)
  $\rightarrow$
(The race had a “blanket finish”)
```

Figure 4.2: Pseudocode examples of rules generating the facts in the RaceSim world model

The facts generated by the knowledge module make up the world model, and are sent to the other modules of ERIC for generation of affect, language and gesture. In order to maintain consistency over the course of a continuous event, the facts in the world model created by the knowledge inference module are all tagged with a timestamp. This allows ERIC to use only current information for output generation, without having to retract facts that are no longer true. Also, by using these timestamps the knowledge module can deduce behaviour patterns over a period of time, or correct past predictions that turned out to be incorrect: for example, when commentating the RaceSim ERIC can predict that a horse will soon overtake another horse based on its speed, and can also observe that the pursuing horse failed to overtake as predicted.

The knowledge module is completely free-form, with no prescribed Jess templates. The use of an expert system is the only restriction: by using facts to represent the world model, it is easy to transfer parts of the world model between modules, simply by transferring the appropriate facts. Also, the templates used for natural language generation

```
(There is a stone at x = $x_1$, y = $y$)
(There is a stone at x = $x_2$, y = $y$, where $x_1 < x_2 < (x_1 + 3)$)
(There is a stone at x = $x_3$, y = $y$, where $x_2 < x_3 < (x_2 + 3)$)
  $\rightarrow$
(There is a column of stones at y = $y$)
```

```
(Tank $t$ is at x = $x$, y = $y$, where $2 < x < 6$ and $y < 3$
  $\rightarrow$
(Tank $t$ is in the North sector of the map)
```

```
(Tank $t_1$ is at x = $x_1$, y = $y_1$)
(Tank $t_2$ is at x = $x_2$, y = $y_2$, where $x_2 > (x_1 + 1)$ and $(y_1 - 1) < y_2 < (y_1 + 1)$)
  $\rightarrow$
(Tank $t_2$ is East of tank $t_1$)
```

```
(Tank $t_1$ is turning its turret to angle $\theta$)
(Tank $t_1$ is at x = $x_1$, y = $y_1$)
(Tank $t_2 \neq t_1$ is at x = $x_2$, y = $y_2$)
($\cos\theta = \frac{y_2 - y_1}{\sqrt{(y_2-y_1)^2+(x_2-x_1)^2}}$)
($\sin\theta = \frac{x_2 - x_1}{\sqrt{(y_2-y_1)^2+(x_2-x_1)^2}}$)
  $\rightarrow$
(Tank $t_1$ is aiming its turret at tank $t_2$)
```

Figure 4.3: Pseudocode examples of rules generating the facts in the dTank world model

are conditional on the world model: such conditions are easily implemented as expert system rules if the world model is already represented as facts. As well as this, it is easy to write affect module rules that extract events, actions and objects from the world model, since each of these will already be represented by a fact in the world model.

The free-form nature of the knowledge module is intended to make it easy for an author to implement reasoning for any domain. If an expert system already exists for a domain, this can be re-used as the knowledge module, as there are no restrictions on the types of facts that must be exported from this module. Alternatively, the Jess expert system could be replaced by a different reasoner, such as an ontology reasoner, provided the knowledge module still exports its world model in the form of Jess facts.

4.4 Plan Recognition

Obviously this module can not only perform event recognition, but also plan recognition [Schmidt et al., 1978]. Using this reasoning, the commentator can not only narrate the events as they happen, but also provide higher level analysis of tactics or actors' intentions – a particularly important aspect in visual commentary where the audience can see the event themselves, and therefore don't particularly need plain narration of the events.

Plan recognition refers to the task of inferring an agent's plans or intentions from observations of its actions. A simple example of this kind of reasoning is implemented in the dTank knowledge module: here, a tank is claimed to be aiming at another tank if it is turning its turret towards an orientation that would have it pointing at that tank; similarly, a tank can be considered to be hiding from another tank if it is staying near a rock, on the opposite side to another tank. Both of these are examples of intentions that can be inferred using rules; however there are many behaviours that are difficult to identify using the kind of forward reasoning used in expert systems. One of the challenges in plan identification is that the reasoner may need to make several observations over a period of time to identify a plan: both the RaceSim and dTank knowledge modules support this kind of reasoning by tagging each fact in the world model with a timestamp. Another challenge is the identification of plans that have failed, or plans that may vary slightly from a standard form: these require more flexible reasoning than the rigid "if-then" rules of expert systems. We have already seen that due to the homogeneous interface, the knowledge module can be easily replaced by another reasoning system: for example (for this task) a dedicated plan recogniser, such as those of Maybury [1988, 1995], or the incremental recognisers used in VITRA [Herzog and Wazinski, 1994, Herzog, 1995, Herzog et al., 1996] and Rocco [André et al., 1998, Voelz et al., 1999].

4.5 Domain Independence

Clearly the knowledge module cannot be domain-independent, since we require the world model to describe the domain. To keep the framework as flexible as possible, we do not proscribe standard Jess facts and templates for the knowledge module to use; rather, the knowledge module defines its own templates, which are then imported to all the modules that need them along with the domain-independent templates. This affects the extent to which a number of other modules can be domain-independent; in particular, the affect processing needs a domain-dependent module (the *affect observer*) that is responsible for generating standardised event, action and object facts (required for the domain-independent affect reasoning) from the knowledge module's facts.

As well as the affect processing module, the natural language generation module and the gesture module both use the world model: in the gesture model there are a few rules producing gestures to match specific world events, whereas in the natural language generation system the templates are conditional upon facts in the world model (see Chapter 5).

To determine our success in making ERIC a domain-independent framework, the domain-independence was evaluated after both commentators had been implemented. The results of this evaluation are in Chapter 7.

Chapter 5

Natural Language Generation

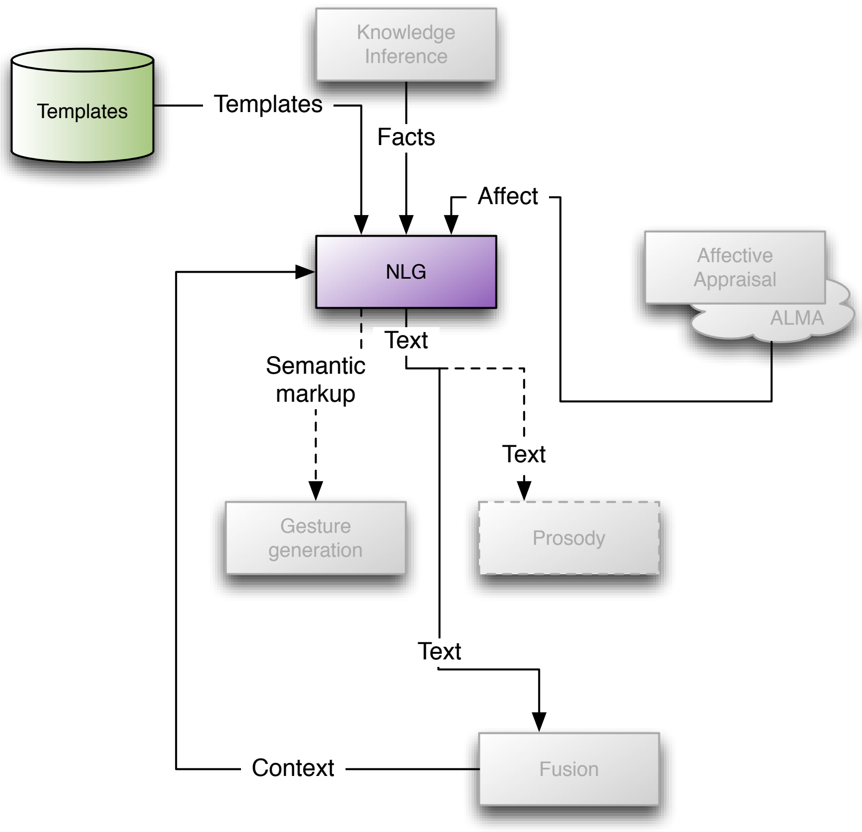


Figure 5.1: The natural language generation module in the system architecture

The automatic generation of human language from an underlying representation is a large challenge, and there are many different approaches and proposed systems in the literature. Reiter and Dale [1997] provide a good overview of the field. In general, NLG systems can be categorised along a scale of theoretical sophistication, ranging from naïve “canned text” at the low end to pure implementations of syntactic and semantic theory at the upper end. In between these two extremes lie a variety of systems combining pattern matching of templates and linguistic theory. Despite the seeming opposition between these two approaches, van Deemter et al. [1999, 2005] argue that template-based natural language generation need not preclude such a system being grounded in linguistics, and give a demonstration of template-based systems that are also supported by linguistic theory.

The natural language generation module of ERIC uses a template-based algorithm to generate prioritised candidate utterances from a world model consisting of Jess facts from the knowledge inference module and an emotional state from the affect module. These candidates are then ranked according to their priority and discourse coherence to select the single best utterance, which is sent to the text-to-speech system. The ranking and selection is performed by the fusion module

5.1 Aims

The most important aim for the natural language generation in the ERIC framework is reactivity: it is important that the generation occur in real-time and is able to spontaneously react to changes in the environment. Also, the generated language must be salient and informative, providing the most important information, so a listener is able to visualise the world. As well as these aims, it was desired that the generated language exhibited good coherence between utterances and variety in the generated utterances, thus making it pleasant to listen to.

5.2 Template-based NLG

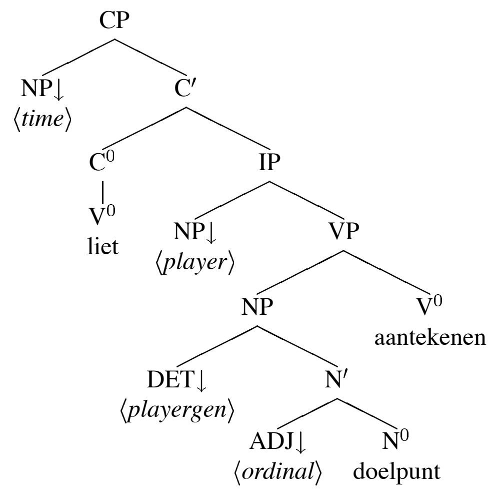
Template-based natural language generation systems are defined by van Deemter et al. [1999, 2005] as systems that map their non-linguistic input directly to the linguistic surface structure (that is, the output text), rather than via some intermediate representation. This gives template-based systems a number of advantages over more complex systems: in particular, their simplicity makes them easier to implement and maintain, particularly for domain experts [Reiter and Dale, 1997]. For our purposes, template-based natural language systems have the additional advantage that the only domain-dependent parts of such a system are the templates themselves, and since these are expressed in terms of the output text, it should be quite easy to implement a new set of templates for any domain, once the algorithm is given.

5.2.1 Previous Work

The D2S natural language generation system described in Odijk [1995], Theune et al. [2001] provided the main source for the NLG module of ERIC. D2S has been implemented to generate summaries of football games, as a natural language-based interface to a jukebox system, and as an information system for train travel. As well as generating text from arbitrarily structured input data, D2S generates syntax trees which are then used in a special text-to-speech engine to identify phrase boundaries, and thus produce more natural-sounding prosody.

D2S is a template-based natural language generation system. It takes structured input data representing facts about the world: for example, in the train information system and the jukebox this was in the form of a database lookup, whereas in the football summary system it was in the form of broadcast teletext. A template in the D2S system consists of a syntax tree with a number of open slots, links to additional syntax trees (possibly as simple as merely a leaf) that can be used to fill these slots, a condition on when the template is applicable, and a set of topics of the sentence (for coherence). The syntax tree is quite similar to the trees of tree-adjoining grammars (TAG) in that its interior nodes are all labelled by non-terminal symbols, the leaf nodes are labelled by either terminal or non-terminal symbols, and non-terminal leaf nodes represent gaps that are open for substitution. However they differ from TAG trees in that whereas TAG trees are usually 'minimal', in the sense that only the head of the tree is lexicalised, with the gaps in the tree representing the arguments of the head, the D2S syntactic trees often contain more words, leaving less gaps to be filled.

$S =$



$E =$ *time* ← ExpressTime (*currentgoal.time*)
player ← ExpressObject (*currentgoal.player, P, nom*)
playergen ← ExpressObject (*currentgoal.player, P, gen*)
ordinal ← ExpressOrdinal (*ordinalnumber*)

$C =$ Known (*currentmatch.result*) ∧ *currentgoal* = First (*notknown,goallist*) ∧ *currentgoal.type* ≠ *owngoal*

$T =$ goalscoring

Figure 5.2: An example of a template in D2S

Slots in the D2S trees are filled by `Express` functions: each slot is associated with a particular function that generates the various possible trees to fill that slot. For example, the `ExpressPlayerGen` function in the football summaries application, to generate a reference to a player in the genitive case, might generate the set of adjectival phrases {*his, Hamming's, the forward Hamming's*} for the player called Hamming.

The conditions on a template specify the situations in which a template is allowed to be used. Conditions are of two types: conditions on the knowledge state, and conditions on the world state. The first type are conditions of the form "we already know X, and we don't already know Y". This allows the system to ensure that facts are presented to the user in a partial order: for instance, it is necessary for a listener to already know the teams who played before it makes sense to talk about the score. The second type allow us to specify that a particular template is only applicable if certain facts about the world are true: for instance, it only makes sense to describe a player's second goal if that player in fact scored more than one goal.

Whereas the knowledge state conditions allow us to specify a partial (temporal) order for templates, larger-scale control over the discourse structure is provided by the topics. Once a template has been used, D2S will attempt to use all other templates with the same topic as that first template, only changing topic when no templates are applicable. This way the discourse does not constantly change topic, ensuring a level of coherence.

5.2.2 NLG Templates in ERIC

The natural language generation module in ERIC uses templates similar to the D2S system. Templates in the ERIC NLG consist of a set of conditions, a discourse state, and an index into an utterance lexicon. Like D2S the conditions are both on the knowledge state and the world state: in ERIC they are expressed as four sets of Jess facts: facts that must be known, facts that must be unknown (for the knowledge state), facts that must be true, and facts that must be false (for the world state). The discourse state consists of a single backwards center and multiple forwards centers for each template.

```
# A horse overtakes another horse
BasicTemplate:   overtaken
UnknownFact:     (overtaken (timestamp ?t) (overtaker ?h1)
                    (overtakee ?h2))
BackwardCenter:  event
Priority:        40
ForwardCenter:   call
ForwardCenter:   overtaken
ForwardCenter:   ?h1
Referent:        ?h1
Referent:        ?h2
```

Figure 5.3: An example of a template in ERIC

In contrast to D2S's syntax-tree based structure, the ERIC NLG uses a lexicon of plain text utterance templates; however these templates can contain slots that will be filled by Jess variables that have been bound by the conditions. The discourse state allows a finer control of discourse structure than D2S's templates, and is based on Centering Theory of Grosz et al. [1995]; this is described in detail in Section 5.3.2.

The utterance lexicon

The utterance lexicon of ERIC's NLG system consists of a list of phrases or utterances, one or more for each template, that are output by the NLG system when the template is activated. The utterances in the lexicon contain slots for referring expressions and other variables, for example horse distances or tank orientations. These slots are filled by the templates before the utterances are output. As well as these, each entry in the lexicon is associated with one or more affective states; the NLG system only selects lexicon entries that match the agent's current affective state. Figure 5.4 shows some examples of lexicon entries.

```
(deffacts lexicon
   (lex (id welcome) (affect ?*any-moods*)
       (text "\"Welcome to the dTank challenge\""))
   (lex (id terrain-col) (affect ?*any-moods*)
       (text
        "(str-cat \"A column of stones at column \" ?y)"))
   (lex (id tank-health) (affect Exuberant Dependent)
       (text
        "(str-cat ?horse1 \" has \" ?health \" points of health\")"
        ))
    )
```

Figure 5.4: Examples of lexicon entries in ERIC

The lexicon is stored as a `deffacts` statement of Jess facts, with each lexicon entry represented as a single fact. The lexicon entry fact has an ID, a multislot of associated affects, and the utterance text itself. The utterance is evaluated by the Jess rules using Jess's `(eval)` function, to allow concatenation of strings, generated referring expressions, and other bound variables.

Referring expression generation

Most of the variables in the lexicon entry slots are bound by the facts on which the template is conditional: the distance between horses, or the health points of a tank, for instance. For referring expressions however we desire more variety than these simple facts; thus we use a more complex process to generate referring expressions.

The entities for which we need referring expressions are identified in the template. The generated referring expressions are then stored in numbered variables, in the order in which they are specified in the template, corresponding to the referring expression slots in the lexicon entry. These numbered variables are currently named `?horse1`, `?horse2`, etc., a domain-specific result of the first implemented domain; in future they will be renamed with more general names such as `?expression1`. The referring expressions are generated on the basis of the world state and knowledge state, so that a horse can be referred to as “the horse in blue”, “the leader”, or “the favourite”, depending on the current world state and what the listener already knows about that horse.

```
(referring-expression (timestamp ?t) (referent 1)
    (expression "The favourite"))

(referring-expression (timestamp ?t) (referent 2)
    (expression "Carmine"))
```

Figure 5.5: Examples of referring expression facts in ERIC

Like the lexicon, allowed referring expressions are stored as Jess facts in working memory; as well as a timestamp, each referring expression fact contains the ID of the referred entity (for the horses, a horse number; for the tanks, the tank name) and a single referring expression for that entity. Unlike the lexicon entries, the referring expressions are generated by rules in the expert system, so that a horse is only described as the leader when it is in fact in first place, for example, or a tank is only described as ”Jason’s tank” after its commander has been conveyed in the commentary.

```
(defrule refexp::name
    (time ?t)
    (name (tank ?w) (name ?n))
    =>
    (assert (referring-expression (timestamp ?t) (referent ?w)
        (expression ?n)))
    )

(defrule refexp::yearling
    (time ?t)
    ?age <- (age (horse ?h) (age 1))
    (known ?age)
    =>
    (assert (referring-expression (timestamp ?t) (referent ?h)
        (expression "the yearling")))
    )
```

Figure 5.6: Examples of rules generating referring expressions in ERIC

As well as these rules, any referring expression can be extended with an adjective describing the commentator’s affective response to that entity, so that “Carmine” can become "the wonderful Carmine" if ERIC is currently experiencing liking or love for Carmine, or "2DHuman" can become "the horrible 2DHuman" if ERIC is feeling disliking or hatred.

The other slot type that is filled by a more complex process than simple variable binding is that representing distances: these are specified in the templates in the same way as referring expression slots, and a Jess function transforms raw distance measures to a more fuzzy form, such as "two lengths" or "just over 200 metres". This function is also rather specific to the horse race domain, and similar functionality in the dTank domain – representing tanks' positions as compass points – is provided by the knowledge module as additional facts in the world model.

5.2.3 Implementation in Jess

The process by which this kind of template-based NLG generates text – find all the templates that match, and generate their candidate utterances – is the same as the execution cycle of an expert system – select all matching rules, and execute their right-hand sides. Thus it is natural to model each template as a rule in an expert system, with the conditions forming the left-hand sides of the rules, and the utterance output (as well as the necessary updates to the knowledge state and discourse state) forming the right-hand side of the rules. To keep the lexicon separate from the templates, we store it in a `deffacts` declaration. In this way we can also later add other conditions on the lexicon lookup, without writing extra templates: for example, the agent's affective state is reflected in his choice of lexicon entry.

Since the Jess rules representing NLG templates share lots of common structure (in fact, the only parts that change are the components representing the conditions and the discourse state), the rules are generated by a Java class

```
de.dfki.racereporter.language.NLGTemplates
```

from a configuration file that lists only the above elements for each template. Figure 5.7 shows the Jess rule produced by this class for the template in Figure 5.3.

Conflict resolution strategy

To maximise variety in the agent's utterances, the Jess instance in the NLG module uses a resolution strategy that chooses rules at random, rather than the default "first-activated, first-fired" resolution strategy of Jess. Thus the order in which the candidate utterances are produced is randomised.

There is one caveat with this random Jess conflict resolution: the resolution strategy chooses from the entries on the activation list, rather than from the rules. Since the activation list contains an entry for every set of facts that match a rule's left-hand side,

```
(defrule generation::overtaken
    (declare (salience 40))
    ?unknown1 <- (overtaken (timestamp ?t) (overtaker ?h1)
        (overtakee ?h2))
    (not (MAIN::known ?unknown1))
    (MAIN::discourse-state (timestamp ?context-t)
        (referents $?usedreferents)
        (forward-centers $?forward-centers)
        (used-references $?usedreferences)
    (MAIN::mood (timestamp ?moodt) (name ?emotion))
    (not (MAIN::mood {timestamp > ?moodt})
    (MAIN::lex (id overtaken)
        (affect $?lexaffect&:(member$ ?emotion ?lexaffect))
        (text ?lexentry))
    (MAIN::referring-expression (referent ?h1)
        (expression ?horse1))
    (not (test (member$ ?horse1 ?usedreferences)))
    (MAIN::referring-expression (referent ?h2)
        (expression ?horse2))
    (not (test (member$ ?horse2 ?usedreferences)))
    =>
    (bind ?effects (create$ ?unknown1))
    (bind ?newcenters (create$ call overtaken ?h1))
    (bind ?referents (create$ ?h1 ?h2))
    (bind ?references (create$ ?horse1 ?horse2))
    (bind ?what (eval ?lexentry))
    (assert (say (timestamp ?t) (priority 40)
        (backward-center event)
        (forward-centers ?newcenters)
        (affect ?lexaffect) (conveys ?effects)
        (referents ?referents) (text ?what)
        ))
    )
```

Figure 5.7: The generated Jess rule corresponding to the template in Figure 5.3

rules that are matched by more combinations of facts appear more frequently on the activation list, and are therefore more likely to fire earlier according to the random conflict resolution strategy. As long as the facts that match the rule are facts from the world model or knowledge model, this is the desired behaviour; however since the lexicon is also implemented as facts, an author can inadvertently cause this behaviour by writing more lexicon entry alternatives for some templates than for other. This problem could be avoided by using the Jess `exists` conditional element to quantify the lexicon fact; however this has the disadvantage that only the first lexicon fact would ever be matched, since this affects the activation rather than the firing of rules. An alternative solution would be to rewrite the resolution strategy to randomly choose between rules rather than activation list entries.

5.3 Discourse Coherence

Generating appropriate individual utterances for a given world state and knowledge state however is not sufficient: we still need to organise the utterances into a coherent discourse. The usual approach to the question of ordering a set of utterances into a coherent discourse is to use a discourse planner. Discourse planning generates a text plan from the set of utterances to be conveyed and the discourse relations between them (for example Rhetorical Structure Theory: see below). The output text plan is a structure for the discourse; it can be as simple as an ordered list of utterances, or it can be a tree describing the discourse relations between the utterances. However this kind of offline planning requires all the utterances to be available at the time of planning: this is not feasible for ERIC, since we want the commentator to be able to spontaneously react to ongoing events, rather than summarise the events after everything has finished. Alternatively, discourse coherence reasoning could be performed online using dynamic replanning; however this is complicated and not particularly fast, and thus also unsuitable for ERIC. To be satisfactorily reactive, ERIC must be able to choose a suitably coherent next utterance from the list of available utterances using only local information, rather than an overall plan for his discourse.

5.3.1 Coherence Theories

Discourse Structure Relations

Rhetorical Structure Theory [Mann and Taboada, 2007] is a theory explaining the coherence and structure of a text. RST explains coherence in terms of two kinds of structures: coherence relations, and schemas. The first is most relevant to our requirements, since relations describe local phenomena.

RST posits quite a number of different relations. Most are *nucleus::satellite* relations: a piece of text (the *satellite*) has a role relative to another (the *nucleus*). Examples of such relations are shown in Table 5.1a. Other relations do not have one piece of text more central than the other; these are called *multinuclear*, and examples are shown in Table 5.1b.

Relation Name	**Nucleus**	**Satellite**
Background	text whose understanding is being facilitated	text for facilitating understanding
Elaboration	basic information	additional information
Preparation	text to be presented	text which prepares the reader to expect and interpret the text to be presented.

(a) Nucleus::satellite relations

Relation Name	**Span**	**Other span**
Contrast	one alternate	the other alternate
Joint	(unconstrained)	(unconstrained)
List	an item	a next item
Sequence	an item	a next item

(b) Multinuclear relations

Table 5.1: Examples of relations of Rhetorical Structure Theory, taken from Mann and Taboada [2007]

Hovy [1993] describes the use of discourse structure relations such as RST in the generation of coherent discourses. However, this work focuses on the offline planning approach to discourse generation, which is not ideal for our real-time commentary scenario.

DLTAG

Lexicalised TAG for discourse [Webber, 2004] is an extension of the tree-adjoining grammar (TAG) formalism to describe discourse coherence rather than single sentence structure. In lexicalised TAG grammars, tree structures have words as *anchors*, and represent either the relationship between a functor and its arguments (*initial* trees) or recursive relationships between other trees (*auxiliary* trees). The substitution slots in the LTAG trees are then filled by other trees that represent the functor or its arguments (for an initial tree), or subclauses (for auxiliary trees).

In DLTAG, this means of modelling relations as trees has been extended to discourse relations; thus the substitution slots in DLTAG trees are filled by trees representing sentences or clauses. Thus, the construction shown in the sequence

(5.1) On the one hand, John is generous. On the other hand, he's hard to find.

would be described with the tree in Figure 5.8. In keeping with standard TAG notation, the arrow $\downarrow$ represents a substitution site; D_c stands for "discourse clause".

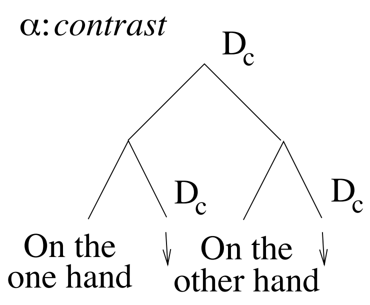


Figure 5.8: A contrast discourse relation, represented in DLTAG

DLTAG trees describe relationships between discourse clauses, and could thus be used to order generated clauses into coherent discourse. However often fitting the candidate utterances into a DLTAG tree requires processing multiple utterances before outputting any of them; even the simple relation in Figure 5.8 requires both utterances to be available before either are output in order that the system can add the phrase "on the one hand". However in a real-time reactive commentary situation we want to be able to output utterances as soon as they are generated, without requiring the surrounding utterances for discourse coherence. The relationships described by DLTAG are not local enough for this situation, and are thus more suitable to offline discourse planning than the online processing required for ERIC.

Centering Theory

Centering Theory is a theory of discourse coherence proposed by Grosz et al. [1995]. It claims that we can describe the global coherence of a discourse entirely in terms of local coherence relationships. From a natural language generation perspective, this claim implies that offline text planning is unnecessary for producing a coherent discourse: we merely need to ensure that the discourse is locally coherent from utterance to utterance. This is precisely what we are looking for in ERIC.

We consider a discourse as a set of utterances: U_n. Centering Theory assigns each utterance U a number of semantic focal points called *centers*: a single *backward-looking* center $C_b(U)$ and a partially ordered set of *forward-looking* centers $C_f(U)$. The forward-looking centers are partially ordered by *prominence*, that is, the likelihood that a forward-looking center of an utterance will be the backward-looking center of the following utterance. In effect, the backward-looking center of a sentence describes its topic, and the forward-looking centers describe possible topics for a coherent following sentence. Given this semantic information, Centering Theory defines three relations between subsequent utterances, ordered by decreasing coherence:

Center continuation $C_b(U_{n+1}) = C_b(U_n)$ and also $C_b(U_{n+1})$ is the most highly ranked element of $C_f(U_{n+1})$; so, $C_b(U_{n+1})$ is the most likely candidate for $C_b(U_{n+2})$

Center retaining $C_b(U_{n+1}) = C_b(U_n)$ but $C_b(U_{n+1})$ is not the most highly ranked element of $C_f(U_{n+1})$; so, $C_b(U_{n+1})$ is not the most likely candidate for $C_b(U_{n+2})$

Center shifting $C_b(U_{n+1}) \neq C_b(U_n)$.

For example, consider the sentences:

(5.2) a. John has been having a lot of trouble arranging his vacation.

b. He cannot find anyone to take over his responsibilities.

c. He called up Mike yesterday to work out a plan.

d. Mike has annoyed him a lot recently.

e. He called John at 5 AM on Friday last week.

The two sentences 5.2b and 5.2c share a backward-looking center (John), which is also the most highly ranked of the second sentence's forward-looking centers (John, Mike), so they are related by center continuation. The two sentences 5.2c and 5.2d are less coherent: although these sentences share a backward-looking center (John), this is not the most highly ranked of the second sentence's forward-looking centers (Mike, John), so the relation between these two is center retaining. Least coherent are the two sentences 5.2d and 5.2e. The backward-looking centers of these sentences are different (John for 5.2d, Mike for 5.2e), so here we have center shifting, the least coherent relation.

5.3.2 Coherence in ERIC

The template NLG system in ERIC generates all utterances that are applicable to the current world state and knowledge state. Each of these utterances are assigned a priority, a backward-looking center and a number of forward-looking centers; the fusion module then selects a single utterance from all the possibilities on the basis of these attributes. The backward-looking and forward-looking centers are compared using a variation of the Centering Theory relations described below; the priority allows ERIC to compare utterances on the significance of their commentated events as well as on their discourse coherence, to allow the agent to commentate a event regardless of discourse coherence if it is judged sufficiently significant.

In order to compare utterances by their discourse coherence, we use three relations between subsequent utterances similar to those of Centering Theory; once the relationship of each candidate utterance to the previously spoken utterance is known, we can select the most coherent candidate (or select randomly among multiple equally-coherent candidates). Unlike in Centering Theory, the forward-looking centers in the ERIC NLG templates are not ordered: this makes authoring templates simpler, since an author merely needs to identify the forward-looking centers of an utterance, not order them as well. The three relations used to compare utterances, in decreasing order of coherence are:

Center retaining $C_b(U_{n+1}) = C_b(U_n)$

Smooth center shifting $C_b(U_{n+1}) \in C_f(U_n)$ but $C_b(U_{n+1}) \neq C_b(U_n)$

Abrupt center shifting $C_b(U_{n+1}) \neq C_b(U_n)$ and $C_b(U_{n+1}) \notin C_f(U_n)$

In a sense, the first two are the the relations between U_{n+1} and U_{n+2} that correspond to the continuation and retaining relations of Grosz et al. [1995]; the third is a stricter version of shifting (since the second relation is also a subset of shifting).

These relations are assessed by comparing the forward-looking and backward-looking centers of the previously spoken utterance with the backward-looking centers of each candidate utterance; the candidate utterance with the strongest coherence is then spoken. In the case when multiple utterances have equally strong coherence, a single utterance is chosen at random.

For example, consider the sentences:

(5.3) a. Carmine has overtaken the favourite.

b. She is in first place.

c. Eben has overtaken the favourite.

d. The favourite is in third place.

e. Topaz is wearing green today.

The two sentences 5.3a and 5.3b share a backward-looking center (Carmine), so they are related by center retaining. The two sentences 5.3c and 5.3d are less coherent: these sentences do not share a backward-looking center (Eben for 5.3c and the favourite for 5.3d), but the backward-looking center of 5.3d is in the forward-looking centers of 5.3c (Eben, the favourite), so the relation between these two is smooth center shifting. Least coherent are the two sentences 5.3d and 5.3e. The backward-looking centers of these sentences are different (the favourite for 5.3d and Topaz for 5.3e), and the backward-looking center of 5.3e is not in the forward-looking centers of 5.3d (the favourite), so here we have abrupt center shifting, the least coherent relation.

5.4 Challenges

There are a number of additional challenges in natural language generation that are not addressed by the templates or coherence theory. In particular, we need to generate referring expressions for entities in the world, and we need to ensure variety in the generated utterances.

Referring expressions, pronouns and ellipsis

A significant challenge in generation of natural language commentary is that of how to reference objects in the world. Depending on the user's knowledge state and the current discourse state, an object may be referred to by its name, a short description (such as "the leader" or "the favourite"), or a pronoun. The approach taken in ERIC is to assert Jess facts to store valid referring expressions for entities; these are then matched by the template rules and used to fill the slots in the utterance template. For example, depending on the knowledge state and the discourse state, a horse could be referred to as "the leader" or "the blue horse" or "Azure".

In addition to these Jess facts, any named entity that is a forward-looking center of the previous utterance can be replaced by a pronoun. Currently the pronouns are hard-coded in the `NLGTemplates` Java class, and do not take into account factors such as the gender of the entity. In future, it may be desirable to replace this with a system to generate a pronoun based on facts in the world state; similar to the referring expression generation, this would be a combination of predefined "pronoun" facts, representing the correct pronoun to use for every named entity in the system, and rules asserting such "pronoun" facts. Such a mechanism could also be used to generate ellipses, which are currently not generated at all by the NLG system. Clearly the hard-coded definition of pronouns is an obstacle to reimplementation: see Section 7.1 for further discussion of its domain-independence.

Repetition of utterances

It is vital that the generated natural language is interesting to the listener, and remains interesting throughout the commentary; one of the aims of the natural language generation system is a high variety in the generated utterances. However, many facts in the world model remain true over a period of time; if these facts are judged to be important by the NLG templates, then there is a risk the agent will get "stuck in a groove" and continuously repeat this important fact. Of course we cannot simply ignore such a continuous fact once it has first been mentioned: it may be that a fact is worth repeating after a period of time, to remind listeners of its currency.

Currently to avoid repetition, the agent is not allowed to repeat an utterance immediately. However this does not completely eliminate the repetition, since he has a number of ways of conveying each fact, and he may cycle through these. Thus two additional approaches have been used.

For a number of facts in the world, we have added "meta-facts" to the knowledge module: facts about how recently another fact has changed. For example, the `nopositionchange` fact for the horse race commentator records how long it has been since the relative positions of the horses changed; thus we can write templates that only apply when a fact is new, or when a fact is older than a particular age. In addition, such meta-facts allow us to draw conclusions about less immediate facts such as how often the lead changed in the race, or how interesting or exciting the race was. These meta-facts are part of the world model, and thus like the world model domain-specific.

A similar effect can be achieved without such meta-facts, by using the template priorities: we split their NLG templates into two templates, one (with a higher priority) for the situation where the fact is true for the first time recently, and one (with a lower priority) for the situation where the fact remains true. For example, the utterances that convey that a tank is firing are in two templates: “tank-firing” and “tank-stillfiring”, the first with a higher priority but the added condition that the tank must not have been firing in the previous second. Thus the agent will certainly comment on a fact when it is new, but a fact that has already been true for at least one second is equal in priority to all the other utterances, and thus the random selection ensures variety.

Chapter 6

Affect

In the commentary domain, affective responses to events is vital to an engaging commentator. The emotional dimension enhances the character's believability and the amount we engage with the character [Gebhard et al., 2003, 2004]: emotional expression is essential to immersing the audience in the commentary. Additionally, the emotional state of a commentator can convey information to an audience, such as the significance of a currently presented piece of information, or its personal relevance to the commentator (for example, the horse he favours in the horse race). In fact, human listeners will attribute some emotional state to a presenter even if there is no such state directing its behaviour [Picard, 1997]. The emotions in a horse race commentary even tend to follow a common format, described by Trouvain and Barry [2000].

The presence of affect answers one of the key aims of the ERIC commentator, affectivity. It also heightens the behavioural complexity of the agent, since he can vary his behaviour depending on his affective state. Both the model used to compute ERIC's affective state, and the various means in which he can express this affective state are described in this chapter.

ERIC generates affective responses by first using a set of cause-effect and belief relations to assign appraisals of emotion-eliciting conditions (EECs) to facts in the world model based on a set of goals and desires. These appraisals are used by ALMA [Gebhard, 2005] to generate an affective state (medium-term moods and emotion events) according to the OCC cognitive model of emotions [Ortony et al., 1988]. The generated affective state is then expressed by the ERIC agent in his lexical selection, his gestures, and his facial expressions; in addition, expression through his prosody is planned as future work. The process of appraising emotion-eliciting conditions is described in Section 6.2; the ALMA model is described in Section 6.1, and the expression capabilities are described in section 6.3.

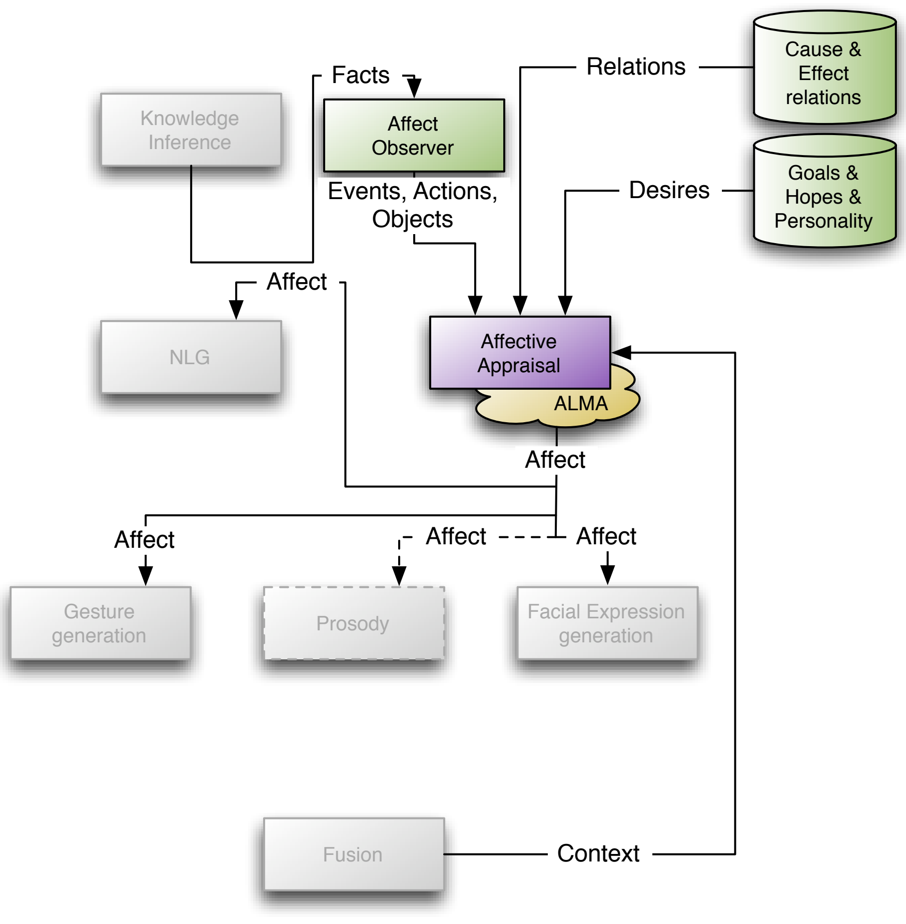


Figure 6.1: The affect module in the system architecture

6.1 Affective Computation in ALMA

The character's emotional state is modelled by ALMA [Gebhard, 2005]. ALMA models three distinct types of affect: emotions (short-term), moods (medium-term) and personality (long-term). Specifically, emotions are usually bound to a specific event, action or object, and decay through time; moods are generally not related to a concrete cause, and are more stable than emotions; and personality is a long-term description of the agent's affective behaviour.

6.1.1 Personality Model

The agent's personality is modelled using the Big Five personality model of Goldberg [1990], which defines a personality along five factors: openness, conscientiousness, extraversion, agreeableness and neuroticism. These five factors are used to calculate an initial mood (see below); in addition, the values of these traits affect both the intensity of the character's emotional reactions, and the decay of his emotion intensities [Gebhard et al., 2004].

6.1.2 Mood Model

Mood is modelled as an average of emotional states across time. Each mood is described by a value on the three scales pleasure, arousal and dominance, according to the PAD model of Mehrabian [1996]; these three values form a three-dimensional mood space (PAD space). Each scale ranges from -1.0 to 1.0. Thus each mood represents a point in the PAD space: the moods corresponding to the eight octants are shown in Table 6.1, and the PAD space is shown graphically (along with some emotions and a highlighted current mood octant) in Figure 6.2. The intensity of the mood is the magnitude of the vector describing its PAD point.

Since the character's mood is changed, not set, by his emotions, he needs a initial default mood; this is calculated from the Big Five personality traits as follows:

$$\begin{aligned}
\text{Pleasure} :=&\ 0.21 \times \text{Extraversion} + 0.59 \times \text{Agreeableness} + 0.12 \times \text{Neuroticism} \\
\text{Arousal} :=&\ 0.15 \times \text{Openness} + 0.30 \times \text{Agreeableness} - 0.57 \times \text{Neuroticism} \\
\text{Dominance} :=&\ 0.25 \times \text{Extraversion} + 0.17 \times \text{Conscientiousness} \\
&\ + 0.60 \times \text{Extraversion} - 0.32 \times \text{Agreeableness}
\end{aligned}$$

Given this initial state, a character's emotion can then affect his mood. For this to be possible, the emotions need to be expressed in (or at least mapped to) PAD space as well: such a mapping is provided in Gebhard [2005], and reproduced in Table 6.2.

Using these mappings, the character's emotion changes his mood according to the *pull and push mood change function.* This works as follows: firstly, it computes the geometric

			Pleasure	
			+	−
Arousal +	Dominance	+	Exuberant	Hostile
		−	Dependent	Anxious
Arousal −	Dominance	+	Relaxed	Disdainful
		−	Docile	Bored

Table 6.1: The corresponding moods to the octants of PAD space

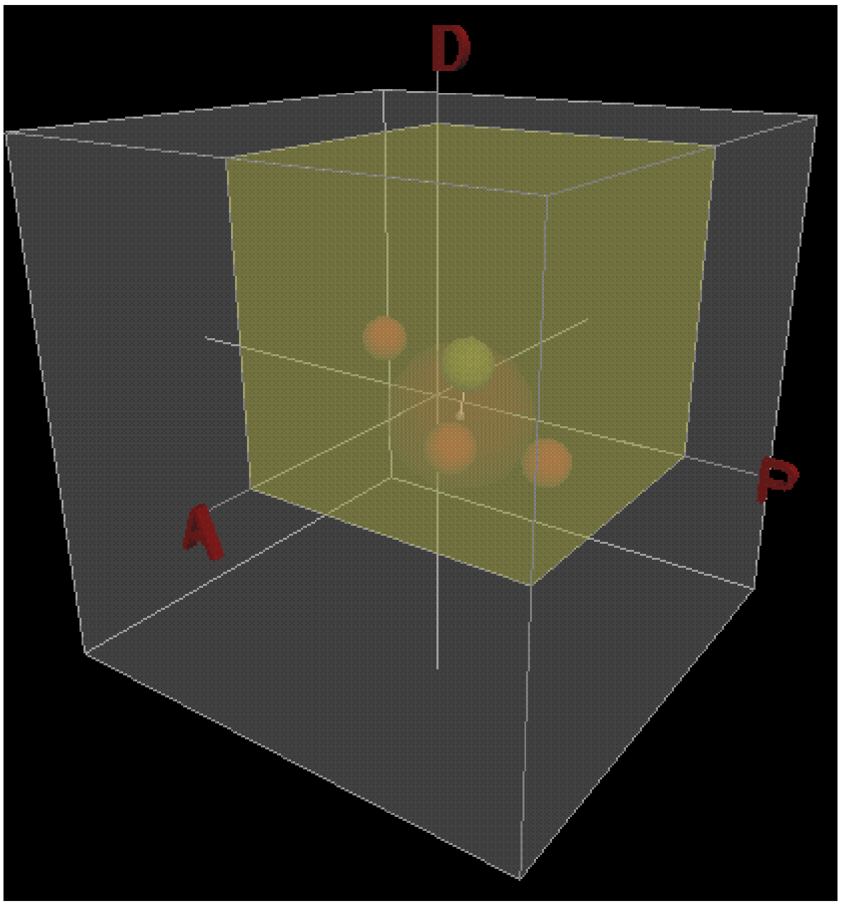


Figure 6.2: A graphical representation of PAD space

Emotion	**P**	**A**	**D**	**Mood Octant**
Admiration	0.5	0.3	–0.2	Dependent
Anger	–0.51	0.59	0.25	Hostile
Disliking	–0.4	0.2	0.1	Hostile
Disappointment	–0.3	0.1	–0.4	Anxious
Distress	–0.4	–0.2	–0.5	Bored
Fear	-0.64	0.6	–0.43	Anxious
FearsConfirmed	–0.5	–0.3	–0.7	Bored
Gloating	0.3	–0.3	–0.1	Docile
Gratification	0.6	0.5	0.4	Exuberant
Gratitude	0.4	0.2	–0.3	Dependent
HappyFor	0.4	0.2	0.2	Exuberant
Hate	–0.6	0.6	0.3	Hostile
Hope	0.2	0.2	–0.1	Dependent
Joy	0.4	0.2	0.1	Exuberant
Liking	0.4	0.16	–0.24	Dependent
Love	0.3	0.1	0.2	Exuberant
Pity	–0.4	–0.2	–0.5	Pity
Pride	0.4	0.3	0.3	Exuberant
Relief	0.2	–0.3	0.4	Relaxed
Remorse	–0.3	0.1	–0.6	Anxious
Reproach	–0.3	–0.1	0.4	Disdainful
Resentment	–0.2	–0.3	–0.2	Bored
Satisfaction	0.3	–0.2	0.4	Relaxed
Shame	–0.3	0.1	–0.6	Anxious

Table 6.2: Mapping OCC emotions into PAD space

center of all current emotions (in PAD space). If the current mood in PAD space is between the origin and the emotion center, then the mood is drawn towards the emotion center (the *pull phase*); alternatively if the current mood is beyond or at the emotion center, then the mood is shifted away from the center, further into the current mood octant (the *push phase*). The strength of the push or pull is determined by the intensity of the emotion center. Finally, the mood tends over time to slowly move back towards the default mood.

6.1.3 Emotion Model (OCC)

Emotions are deduced from stimuli according to the OCC cognitive model of emotions, from Ortony et al. [1988]. The inputs to this model are appraisals of the world, called emotion-eliciting conditions. Events, actions and objects are appraised: desirability and realisation of events, liking of others affected by events, praiseworthiness and agency of actions, and appealingness of objects. In addition, the appraisals distinguish between desirability of a past event and desirability of a future event. The mapping of these appraisals to emotions is given in Table 6.3. Although these tables show only positive/negative appraisals, all the emotion-eliciting conditions except realisation and agency are in fact continuous values. The value of each appraisal, along with the Big Five personality model, is used to compute the intensities of the emotions. In addition to these simple emotions, various combinations of emotions give rise to compound emotions: these are shown in Table 6.4. When they are generated, compound emotions are elicited instead of, rather than in addition to, their component simple emotions.

		Desirability for self	
		desirable	undesirable
(timing)	future	Hope	Fear
	present	Joy	Distress

		Desirability for other	
		desirable	undesirable
Liking	like	HappyFor	Pity
	dislike	Resentment	Gloating

		Realisation	
		realised	not realised
(prior emotion)	Hope	Satisfaction	Disappointment
	Fear	FearsConfirmed	Relief

		Praiseworthiness	
		praiseworthy	unpraiseworthy
Agency	self	Pride	Shame
	other	Admiration	Reproach

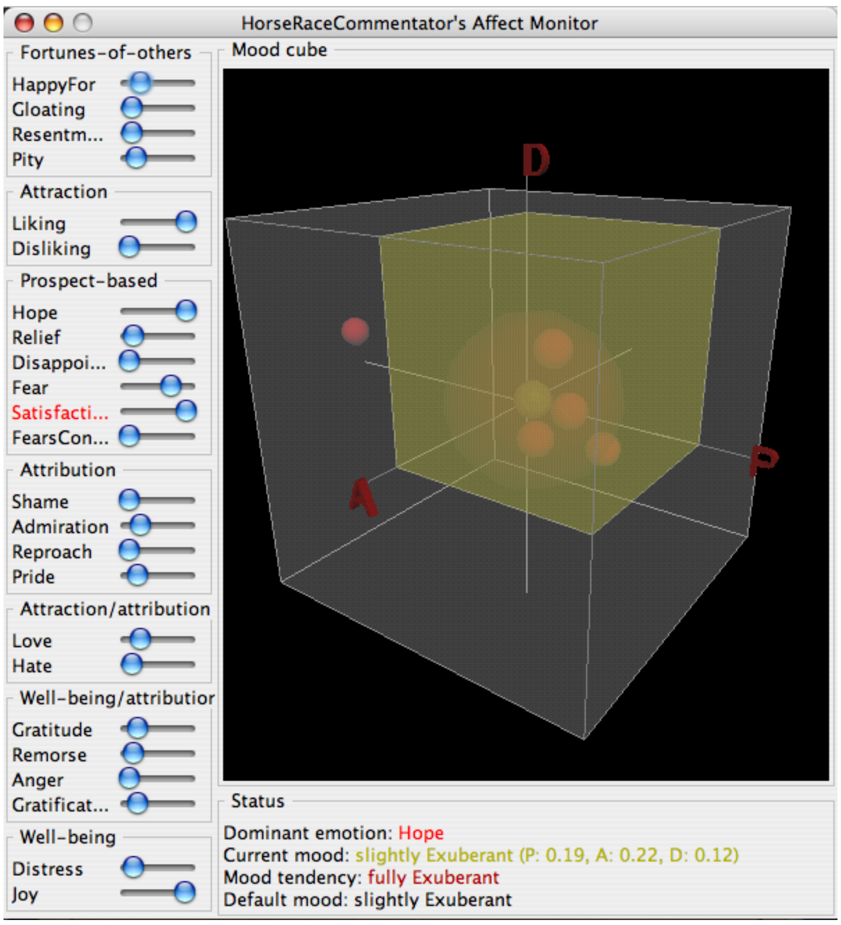
Appealingness	
appealing	not appealing
Liking	Disliking

Table 6.3: Emotions corresponding to various emotion eliciting conditions

The agent's current affective state can be graphically displayed, showing the current emotions, the current mood, and the current mood tendency in PAD space. An example of this output is shown in Figure 6.3.

Basic emotions	Compound emotion
Liking and Admiration	Love
Disliking and Reproach	Hate
Joy and Pride	Gratification
Distress and Shame	Remorse
Joy and Admiration	Gratitude
Distress and Reproach	Anger

Table 6.4: The definition of compound emotions in ALMA



Dominant emotion: Hope
Current mood: slightly Exuberant (P: 0.19, A: 0.22, D: 0.12)
Mood tendency: fully Exuberant
Default mood: slightly Exuberant

Figure 6.3: The graphical output of the agent's affective state in ALMA

6.2 Appraisal of Emotion-Eliciting Conditions

The appraisals of the emotion-eliciting conditions are made by comparing events, actions and objects observed in the world against the agent's goals and desires. In the goals and desires, we specify events or actions we desire, events or actions we desire to avoid, and objects we like or dislike. If an event or action we desire occurs, this is appraised positively (desirable for events, or praiseworthy for actions); conversely, if an event or action we desire to avoid occurs, this is appraised negatively (undesirable for events, or unpraiseworthy for actions). Also, when an object we like occurs in the discourse state (because it has just been mentioned in the commentary), it is appraised positively (liking); conversely, when an object we dislike occurs in the discourse state, it is appraised negatively (disliking).

Appraisals of agency and realisation (as well as a past/future judgement on desirability) are made by observing the world model. Agency is appraised by identifying the agent of each action. An event is appraised as realised if it has occurred in the world; in this case its desirability is also past desirability. An event that has not yet occurred in the world is considered to have only future desirability.

As well as these basic appraisals, cause-effect relations allow us to appraise events, actions and objects that are not specified in the goals and desires. ERIC defines four such relations: `leadsto` and `hinders`, describing an event or action's effect on the likelihood of future events and actions, and `supports` and `contradicts`, describing an event or action's effect on our belief in other events or actions. Using these relations, we can propagate an event or action's appraisal to other events or actions, thus appraising every event, action and object in the world, not just the ones specified as goals and desires.

6.2.1 Goals and Desires

To be able to compare events, actions and objects against goals and desires, we need to specify events or actions we would like to see occur (*goals*), events or actions we would not like to see occur (*antigoals*), and objects we like or dislike. Additionally, the agent maintains a set of beliefs about other actors' goals and desires, to enable him to judge the desirability of events for the players in the scenario.

In a commentary scenario the commentator's own goals and desires may be less influential on his commentary than those of his audience: for example, commentary on an international sporting game for broadcast in one of the competing nations will naturally show a preference for the "home" nation. Since the goals and desires are represented as facts in the world model they are easily adjusted to match characteristics of the audience, and the affect model does not require a distinction between the commentator's desires and those of his audience.

The goals and desires are specified in a separate configuration file to the main affect module, since they are dependent on the world model and thus domain-specific. Thus they can easily be altered to match both a change in domain and a change in intended audience.

6.2.2 Cause-Effect Relations

In order to reason about events, actions or objects that are not directly part of our goals, but still somehow related to (or influencing) our goals, we need to codify the relations between non-goal events, actions and objects and goals. This is done by means of the following relations.

leadsto and hinders

The relations `leadsto` and `hinders` model events or actions that affect the likelihood of some future event or action: a precedent `leadsto` a consequence if it increases the likelihood of the consequence; conversely, a precedent `hinders` a consequence if it decreases the likelihood of the consequence. Both these relations are parameterised with a `certainty` value, which models our certainty that the relation holds (or put another way, the estimated probability that the precedent will result in the consequence occurring or not occurring).

```
(leadsto (event decreases-lead) (eventagent 1)
    (goal is-overtaken) (goalagent 1) (certainty 0.8))
(hinders (event increases-lead) (eventagent 1)
    (goal is-overtaken) (goalagent 1) (certainty 0.8))
(leadsto (event aiming) (eventagent ?w)
    (goal attack) (goalagent ?w) (certainty 0.7))
(hinders (event taking-fire) (eventagent ?w)
    (goal score) (goalagent ?w) (certainty 0.5))
```

Figure 6.4: Examples of the `leadsto` and `hinders` relations

supports and contradicts

The relations `supports` and `contradicts` model events or actions that affect our knowledge of other events or actions: an observation `supports` a conclusion if we can deduce the conclusion from the observation; conversely, an observation `contradicts` a conclusion if we can deduce the falsity of the conclusion from the observation. Like the `leadsto` and `hinders` relations, these are parameterised with a `certainty` value, which models our confidence in the logical relation between the observation and the conclusion.

```
(supports (event overtakes) (eventagent 1)
    (goal interesting) (goalagent race) (certainty 0.8))
(contradicts (event wins) (eventagent 3)
    (goal wins) (goalagent 4) (certainty 1.0))
(supports (event score) (eventagent ?w)
    (goal interesting) (goalagent battle) (certainty 0.7))
```

Figure 6.5: Examples of the `supports` and `contradicts` relations

Using these relations, we can propagate an event or action's positive or negative appraisal to each event or action that `leadsto` it, and propagate the opposite appraisal to each event or action that `hinders` it. Given an event that is realised, we can also appraise all events it `supports` as realised, and appraise all events it `contradicts` as not realised. Finally, we can use `leadsto` and `hinders` to appraise the likelihood of future events: whenever we observe an action or object, we increase our appraisal of the likelihood of each event or action it `leadsto`, and decrease the likelihood of each event or action it `hinders`.

Like the goals and desires, these relations describe elements of the world model and are thus domain-specific. Therefore they are also stored in a domain-specific configuration file, separate from the main domain-independent code of the affect module.

6.2.3 The Affect/Knowledge Module Interface

Since there are no restrictions on the templates and rules that may be used in the knowledge model, there must be a means to transform the facts from knowledge model (unrestricted in format) into `event` and `action` facts for use in the affect module. This role is fulfilled by a set of Jess rules called the *affect observer*; these rules match facts from the knowledge model on their left-hand side, and assert corresponding `event` or `action` facts for use in the affect module on their right-hand side. Since they are closely tied to the knowledge model, these rules are necessarily domain-dependent.

As well as these rules, two rules ensure that if an event or action is observed that `supports` another event or action, then the appropriate `event` or `action` fact for the consequence is also asserted.

6.2.4 Generation of ALMA Tags

ALMA tags

ALMA can generate emotions from a number of appraisal tags, which are a shorthand way of specifying particular EEC values. The available tags are shown in table 6.5.

Tag	Description
GoodEvent	An event with positive desirability for me
BadEvent	An event with negative desirability for me
GoodEventForGoodOther	An event with positive desirability for an other that is liked by me
BadEventForGoodOther	An event with negative desirability for an other that is liked by me
GoodEventForBadOther	An event with positive desirability for an other that is disliked by me
BadEventForBadOther	An event with negative desirability for an other that is disliked by me
GoodLikelyFutureEvent	A future event with positive desirability for me and positive likelihood
BadLikelyFutureEvent	A future event with negative desirability for me and positive likelihood
GoodUnlikelyFutureEvent	A future event with positive desirability for me and negative likelihood
BadUnlikelyFutureEvent	A future event with negative desirability for me and negative likelihood
EventConfirmed	An event with realization TRUE
EventDisconfirmed	An event with realization FALSE

(a) Tags describing events

Tag	Description
GoodActSelf	An action with positive praiseworthiness and agency self
BadActSelf	An action with negative praiseworthiness and agency self
GoodActOther	An action with positive praiseworthiness and agency other
BadActOther	An action with negative praiseworthiness and agency other

(b) Tags describing actions

Tag	Description
NiceThing	An object with positive appealingness
NastyThing	An object with negative appealingness

(c) Tags describing objects

Table 6.5: ALMA emotion-eliciting condition tags

```
(defrule appraisal::attacking
    (attack (timestamp ?t) (tank ?w))
    =>
    (assert (action (timestamp ?t) (action attack)
        (agent ?w)))
    )

(defrule appraisal::overtakes
    (overtaken (timestamp ?t) (overtaker ?h1)
        (overtakee ?h2))
    =>
    (assert (action (timestamp ?t) (action overtakes)
        (agent ?h1)))
    (assert (event (timestamp ?t) (event is-overtaken)
        (agent ?h2)))
    )
```

Figure 6.6: Example of the rules transforming free knowledge module facts into standard affect module facts

Tags across relations

To assign tags to goals, the following rules are observed: a goal is a GoodEvent, and a antigoal a BadEvent, if they occur; similarly an object we like is a NiceThing, and an object we dislike is a NastyThing. All other appraisals, including appraisals of non-goal events and actions, and appraisals of likelihood, require the "propagation" of tags through the relations.

Events To propagate desirability for self through the relations, the following rules apply:

- An event that `leadsto` or `supports` a GoodEvent, a GoodActSelf or a GoodActOther becomes a GoodEvent
- An event that `leadsto` or `supports` a BadEvent, a BadActSelf or a BadActOther becomes a BadEvent
- An event that `hinders` or `contradicts` a GoodEvent, a GoodActSelf or a GoodActOther becomes a BadEvent
- An event that `hinders` or `contradicts` a BadEvent, a BadActSelf or a BadActOther becomes a GoodEvent

For example, the horse Carmine taking the lead from another horse `leadsto` Carmine winning the race, which is one of our goals: thus Carmine taking the lead from another horse is a GoodEvent. Similarly, the tank 2DHuman being shot at `leadsto` 2DHuman dying, which `hinders` 2DHuman winning, which is one of our goals: thus 2DHuman being shot at is a BadEvent.

The following rules propagate desirability for a liked other:

- An event that `leadsto` or `supports` a GoodEventForGoodOther becomes a GoodEventForGoodOther
- An event that `leadsto` or `supports` a BadEventForGoodOther becomes a BadEventForGoodOther
- An event that `hinders` or `contradicts` a GoodEventForGoodOther becomes a BadEventForGoodOther
- An event that `hinders` or `contradicts` a BadEventForGoodOther becomes a GoodEventForGoodOther

Similarly for desirability for a disliked other:

- An event that `leadsto` or `supports` a GoodEventForBadOther becomes a GoodEventForBadOther
- An event that `leadsto` or `supports` a BadEventForBadOther becomes a BadEventForBadOther
- An event that `hinders` or `contradicts` a GoodEventForBadOther becomes a BadEventForBadOther
- An event that `hinders` or `contradicts` a BadEventForBadOther becomes a GoodEventForBadOther

For example, we presume that the horse Eben, who we dislike, wants to win the race; also, Eben falling over `hinders` Eben winning. Thus Eben falling over is a BadEventForBadOther. Similarly, we dislike the tank Eraser, who we also presume wants to win; and Eraser shooting at another tank `leadsto` Eraser killing that tank, which `leadsto` Eraser winning. Thus, Eraser shooting at another tank is a GoodEventForBadOther.

Likelihood is propagated with the following rules:

- When an event or action `leadsto` a GoodEvent, the consequence becomes also a GoodLikelyFutureEvent
- When an event or action `hinders` a GoodEvent, the consequence becomes also a GoodUnlikelyFutureEvent
- When an event or action `leadsto` a BadEvent, the consequence becomes also a BadLikelyFutureEvent
- When an event or action `hinders` a BadEvent, the consequence becomes also a BadUnlikelyFutureEvent

For example, since the horse Carmine winning the race is in our goals, it is a GoodEvent. When Carmine takes the lead from another horse, which `leadsto` Carmine winning, Carmine winning also becomes a GoodLikelyFutureEvent. Similarly, the tank 2DHuman winning is one of our goals, so it is a GoodEvent. 2DHuman being shot at `leadsto` 2DHuman dying, which in turn `hinders` 2DHuman winning. Thus when 2DHuman is shot at, 2DHuman winning becomes a GoodUnlikelyFutureEvent.

Realization is appraised as follows:

- When an event is observed, that event is also appraised EventConfirmed
- When an event is observed that `supports` another event, the deduced event is also appraised EventConfirmed
- When an event is observed that `contradicts` another event, the deduced event is appraised EventDisconfirmed.

Depending on the nature of the tag, we can either send the tag to ALMA (thus "making the appraisal") as soon as the tag is generated, or we wait until the associated elicitor is observed:

- GoodLikelyFutureEvent, BadLikelyFutureEvent, GoodUnlikelyFutureEvent and BadUnlikelyFutureEvent are sent to ALMA as soon as they are appraised.
- GoodEvent, BadEvent, GoodEventForGoodOther, BadEventForGoodOther, GoodEventForBadOther, and BadEventForBadOther are sent to ALMA when the associated elicitor is observed.

Actions To propagate praiseworthiness and agency, we use the following rules:

- An action that `leadsto` or `supports` a GoodEvent, a GoodActSelf or a GoodActOther becomes a GoodActSelf if it has agent "me" or a GoodActOther otherwise
- An action that `leadsto` or `supports` a BadEvent, a BadActSelf or a BadActOther becomes a BadActSelf if it has agent "me" or a BadActOther otherwise
- An action that `hinders` or `contradicts` a GoodEvent, a GoodActSelf or a GoodActOther becomes a BadActSelf if it has agent "me" or a BadActOther otherwise
- An action that `hinders` or `contradicts` a BadEvent, a BadActSelf or a BadActOther becomes a GoodActSelf if it has agent "me" or a GoodActOther otherwise

Since we do not reason about future actions, only future events, all action tags (GoodActSelf, BadActSelf, GoodActOther and BadActOther) are sent to ALMA when the associated elicitor is observed.

Objects NiceThing and NastyThing do not need to be propagated; they are appraised from our liking and disliking goals, and sent to ALMA when the associated elicitor is mentioned in the generated speech.

6.2.5 Generation of EEC Values

In contrast to the generation of tags, the EEC values for an elicitor are sent to ALMA as soon as a valid combination has been appraised. This is because whereas an event gets different tags depending on whether it has been observed or is still a future event, the single 'desirability' EEC represents the appropriate choice of "desirability for self", "desirability for other" or "future desirability for self" depending on what other appraisals have been made.

Allowed combinations of EEC appraisals

As we have seen above, not all combinations of EECs produce valid emotions. We restrict ourselves to the following combinations:

- Events
  - Desirability + Agency (implicitly: desirability for self)
  - Desirability + Likelihood + Agency (implicitly: future desirability for self)
  - Desirability + Liking + Agency (implicitly: desirability for other)
  - Realization
- Actions
  - Praiseworthiness + Agency
  - Realization
- Objects
  - Appealingness

EECs across relations

Desirability Similarly to the GoodEvent and BadEvent tags, a goal has positive desirability and an antigoal has negative desirability. Desirability is also propagated across the `leadsto` and `hinders` relations as follows:

- if an event `leadsto` another event or action, it inherits some of its desirability
- if an event `hinders` another event or action, it inherits some desirability opposite to that of the consequence.

For example, the horse Carmine winning the race is one of our goals, and thus has positive desirability. Carmine taking the lead from another horse `leadsto` Carmine winning the race: thus the desirability of Carmine taking the lead from another horse increases. Similarly, the tank 2DHuman winning is one of our goals, and thus has positive desirability. 2DHuman being shot at `leadsto` 2DHuman dying, which in turn `hinders` 2DHuman winning. Thus, the desirability of 2DHuman dying decreases, and in turn the desirability of 2DHuman being shot at decreases.

Liking Liking or disliking are specified as goals, similarly to the NiceThing and NastyThing tags.

Likelihood Likelihood is calculated on the basis of the `leadsto` and `hinders` relations. An initial likelihood can be specified in the goals; otherwise all events have a neutral initial likelihood. The likelihood is then calculated as follows:

- if an event or action is observed which `leadsto` another event or action, the likelihood of the consequence is increased
- if an event or action is observed which `hinders` another event or action, the likelihood of the consequence is decreased.

For example, the horse Carmine taking the lead from another horse `leadsto` Carmine winning; thus, when Carmine takes the lead from another horse, the likelihood of Carmine winning increases. Similarly, the tank 2DHuman dying `hinders` 2DHuman winning; thus when 2DHuman is killed, the likelihood of 2DHuman winning decreases.

Agency Agency is calculated for the EEC values the same way as it is for tags: the observing module observes events and actions with their agent; if this agent is "me" then agency is *self*, otherwise it is *other*.

Appealingness Appealingness is specified in the goals; in fact, appealingness and liking are the same value.

Praiseworthiness Praiseworthiness is calculated for actions in the same way as desirability is calculated for events. A goal action has positive praiseworthiness, and an antigoal action has negative praiseworthiness. Praiseworthiness is then propagated across the `leadsto` and `hinders` relations:

- if an action `leadsto` another event or action, it inherits some of its praiseworthiness
- if an action `hinders` another event or action, it inherits some praiseworthiness opposite to that of the consequence.

Praiseworthiness and desirability are interchangeable in the calculations, depending on whether the precedent and antecedent are events or actions.

Realization Realization models whether we have observed an action or event or not: thus if we observe an action or event, it has realization *TRUE*. If we observe an action or event that supports another, then the other also has realization *TRUE*, and if we observe an action or event that contradicts another, then the other has realization *FALSE*.

6.3 Expression

The agent's affective state can be expressed in one of four ways:

- his selection of words and phrases from the utterance lexicon to fill a NLG template,
- his hand and body gestures,
- his facial expressions, and
- the prosody of his speech.

The first three of these have been fully implemented; a prosody module has been outlined and is foreseen as future work.

Each expression modality is generated by a separate module. The fusion module is responsible for matching the generated non-verbal modalities to each other and to the speech output, as well as resolving semantically conflicting outputs and ensuring the outputs are within the capabilities of the output animated character.

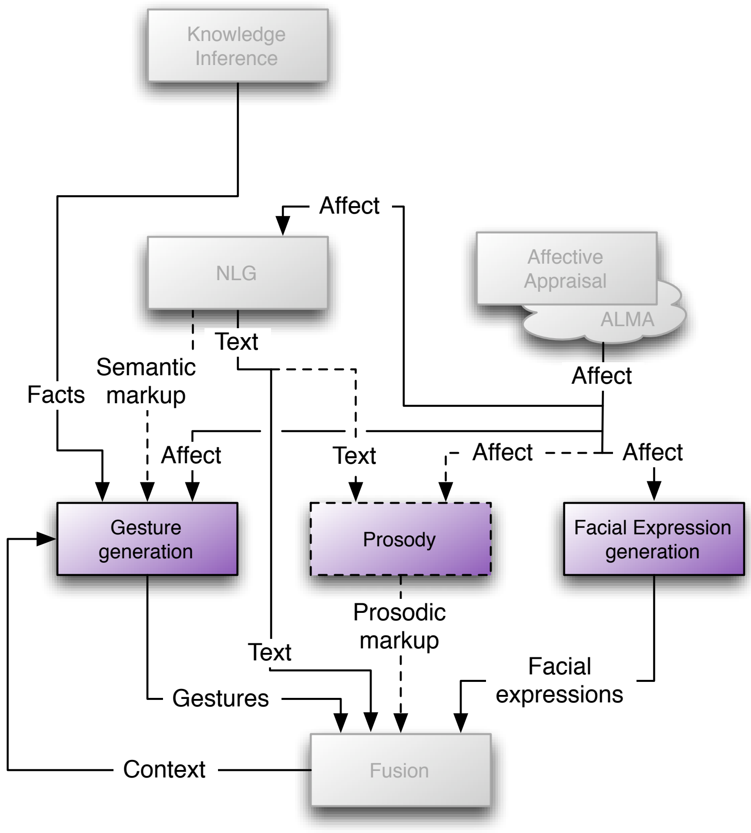


Figure 6.7: The expression modules in the main system architecture

6.3.1 Lexical Selection

Each NLG template can be filled by a number of phrases from the utterance lexicon. These utterances can be tagged with specific emotional states, and such tagged utterances are then only used when the character is in one of the specified emotional states. Currently this selection is performed using the character's mood (medium-term affect).

As well as this, the agent is able to generate emotionally loaded referring expressions and pronouns. Also, since the affective state is represented as Jess facts just like the world model, the agent can comment directly on his current affect.

6.3.2 Gesture

The Charamel framework allows us to specify a set of "idle" and "speaking" gestures which the character plays back when he is idle and speaking, respectively: the controlling CharaScript script chooses randomly from these gestures when no other gesture has been specifically requested. The gesture module of ERIC varies these gesture sets according to the character's emotional state (specifically his mood), so that ERIC moves his hands more vigorously when he is excited, for example. The Paul model supports over three hundred different gestures, ranging from pointing, waving, and turning gestures to more complex gesture sets such as "more or less" or "speaking" or "waiting".

As well as these background gestures, it is possible to command the agent to perform a specific gesture. The gesture module contains a number of rules triggering gestures based on specific emotions: for example, disappointment or distress will be expressed in a shake of the head.

6.3.3 Facial Expression

The Charamel character provides a variety of facial expressions which are used to depict ERIC's emotions. The facial expression module is a map of the ALMA-generated emotions onto the Charamel facial expressions: when ALMA generates any of the mapped emotions, the corresponding facial expression command is sent to Charamel. The Paul model supports 13 different facial expressions, grouped into categories such as neutral, happy, disgust, surprise and fear.

6.3.4 Prosody

A prosody module is planned as future work; we will now outline how such a module could be implemented. Schröder [2004] give a mapping from PAD values to speech parameters, which is shown in Table 6.6. The values in this table are factors; if we assume a scale along each affective dimension from –100 to 100, then the percentage change in each prosodic parameter from its default value is the product of the affective dimension value and the entry in the table. For example, if we have an activation of +50, then the accent prominence is 25% above its default value. (Note that since in ERIC we are using a scale from –1 to 1, rather than –100 to 100, the appropriate scale adjustments need to be made.)

		Coefficients		
	Prosodic parameter	**Activation** (Arousal)	**Evaluation** (Pleasure)	**Power** (Dominance)
fundamental frequency	pitch	0.3	0.1	–0.1
	pitch-dynamics	0.3%		–0.3%
	range	0.4		
	range-dynamics	1.2%		0.4%
	accent-prominence	0.5%	–0.5%	
	preferred-accent-shape		$E \leq -20$: falling $-20 < E \leq 40$: rising $E > 40$: alternating	
	accent-slope	1%	-0.5%	
	preferred-boundary-type		$P \leq 0$: high $P > 0$: low	
tempo	rate	0.5%	0.2%	
	number-of-pauses	0.7%		
	pause-duration	–0.2%		
	vowel-duration		0.3%	0.3%
	nasal-duration		0.3%	0.3%
	liquid-duration		0.3%	0.3%
	plosive-duration		0.5%	–0.3%
	fricative-duration		0.5%	–0.3%
	volume	0.33%		

Table 6.6: Emotion dimension prosody rules, from Schröder [2004]. The activation, evaluation and power dimensions correspond to arousal, pleasure and dominance, respectively.

Unfortunately the Nuance RealSpeak Solo engine does not provide such fine control over so many parameters: it merely allows us to control absolute volume, absolute (average) pitch and absolute rate. Thus the prosody module uses a simplified version of this mapping, shown in Table 6.7. In fact the availability of these parameters is voice-dependent, and the Tom voice does not support changing the pitch. Once the volume, pitch and rate have been calculated using these factors, we can add RealSpeak Solo markup to the text string which is sent to the text-to-speech system. Currently however there is a problem with the interface between the controlling CharaScript and RealSpeak Solo: RealSpeak Solo markup uses a special escape character to mark a prosody command, and if this character is the first character in the string sent to text-to-speech then it is removed by the interface. Finding a solution or workaround to this problem remains as future work; alternatively, we could replace RealSpeak Solo with an affective text-to-speech system such as the MARY system[1] used by Schröder (and for which these mappings were designed).

Prosodic parameter	**Coefficients**		
	Pleasure	**Arousal**	**Dominance**
pitch	0.1	0.3	–0.1
rate	0.2%	0.5%	
volume		0.33%	

Table 6.7: Simplified emotion dimension prosody rules used in the current prosody module of ERIC

6.3.5 Additional modalities

The modular nature of the ERIC architecture means it is simple to implement additional output modalities for the agent: once an expert system describing the generation of the modality has been written, an author merely needs to instantiate another ERIC module containing this expert system, and add the appropriate input and output modules as observers. Thus for example the prosody modality would be generated by a prosody module (as shown in the design diagram); similarly, a camera module could be implemented to control camera movements in the virtual commentary studio, dependent on the intensity of the agent's affect, or on the progress of the commentated event.

[1]`http://mary.dfki.de/`

Chapter 7

Evaluation

The ERIC agent was designed with a number of goals: homogeneity, domain-independence, behavioural complexity, reactivity and affectivity. Homogeneity and domain-independence are design goals, and the others are goals for the desired output. An evaluation of these goals was carried out, and is described in this chapter.

We can satisfy ourselves that the goal of homogeneity has been achieved by observing the agent's design: the core of the system consists entirely of identical modules (with only slight variations for the affect module). The non-homogeneous parts of the system are those concerned with external interfaces: the input world interface, the Charamel actor, and the interface with the ALMA affect model. The ALMA interface was implemented as a simple extension of the standard homogeneous module, so that it differs as little as possible from this standard module. The other two interfaces are specified by the input domain and output actor respectively. For these modules to be homogeneous, the agent could not be domain-independent, since the agent could then only be implemented for input domains and output actors that presented this standard interface; here, the aim of domain-independence was judged more important than the aim of homogeneity.

The other goals (domain-independence, complexity, reactivity and affectivity) however require more detailed evaluation. To evaluate the degree of domain-independence of the agent, we have attempted to quantify the amount of effort required to implement the framework for a new domain. For the output goals, two independent anonymous judges provided feedback on a video of the agent commentating a single horse race, as part of the GALA challenge.

7.1 Domain Independence

The desire underlying the goal of domain-independence is for the agent to be easily implemented for any given domain. Thus we evaluate our success at this goal by measuring the effort required to implement ERIC for the dTank domain. As a baseline, we compare this effort against the effort required to implement ERIC from scratch (for the RaceSim domain).

We have used three measurements to quantify the effort: time taken, final lines of code, and final number of Jess definitions. For the first measurement, both domains were implemented by a single coder working approximately full-time on development: this allows us to estimate the number of man-hours used. The other two measurements are chosen because effort is directly related to the size of the software produced [van Vliet, 2000, Chapter 7]: lines of code is an easy to obtain and easy to understand measurement of software size, and Jess definition count was chosen since it is slightly less dependent on irrelevant factors such as code formatting style.

In addition to these comparative measurements, the process of re-implementing the agent for a new domain revealed a number of places where the agent was not as domain-independent as intended. Many of these were modified to improve domain-independence; we discuss these findings below.

7.1.1 Results

For the man-hours measurement, the time taken to implement the initial system in the RaceSim domain was compared to the time taken to reimplement the system in the dTank domain. These measurements are shown in Table 7.1.

Implementation from scratch	82 days
Reimplementation for new domain	38 days

Table 7.1: Time taken to implement the agent

For the lines of code measurement, we counted the lines of code that were common to both domains and the lines of code that were unique to each domain. This way we are not counting the lines of code modified during the improvements to domain-independence as domain-dependent code. The results of this tally are shown in Table 7.2, and as percentages in Table 7.3. The same measurement, in terms of Jess declarations rather than lines of code, is shown in Table 7.4, and again as percentages in Table 7.5.

On a module level, we observe that three modules out of seven – less than half – required substantial changes. The knowledge module is almost entirely domain-dependent (as expected); the language model similarly required a lot of reimplementation. The affect module, although being mostly domain-independent, also contained significant domain-dependent parts. In contrast, the prosody, facial expression and fusion modules were able to be reused without any modification; and although the gesture module had some domain-dependent rules in both domains, these were few and small, since the main bulk of the gestures are generated by the domain-independent rules.

Module		LOC unique to domain		LOC unchanged
		Horse race	dTank	
Knowledge	templates	466	532	275
	knowledge rules	549	932	15
	total	1015	1464	290
Language	referring expressions	97	324	99
	lexicon	216	134	0
	templates	1012	536	0
	template Java code	0	0	755
	total	1325	994	854
Affect	causality relations	98	32	1091
	goals and desires	128	20	0
	observation rules	24	49	0
	Java functions	0	0	755
	total	248	101	1846
Gesture		97	54	219
Prosody		0	0	109
Facial Expression		0	0	76
Fusion		0	0	371

Table 7.2: Lines of code changed vs unchanged by module

Module		LOC unique to domain		LOC unchanged	
		Horse race	dTank	Horse race	dTank
Knowledge	templates	62.9%	65.9%	37.1%	34.1%
	knowledge rules	97.3$	98.4%	2.7%	1.6%
	total	77.8%	83.5%	22.2%	16.5%
Language	referring expressions	49.5%	76.6%	50.5%	23.4%
	lexicon	100%	100%	0%	0%
	templates	100%	100%	0%	0%
	template Java code	0%	0%	100%	100%
	total	60.8%	53.8%	39.2%	46.2%
Affect	causality relations	8.2%	2.8%	91.8%	97.2%
	goals and desires	100%	100%	0%	0%
	observation rules	100%	100%	0%	0%
	Java functions	0%	0%	100%	100%
	total	11.8%	5.2%	88.2%	94.8%
Gesture		30.7%	19.8%	69.3%	80.2%
Prosody		0%	0%	100%	100%
Facial Expression		0%	0%	100%	100%
Fusion		0%	0%	100%	100%

Table 7.3: Lines of code changed vs unchanged by module (relative percentages)

Module		unique to domain		unchanged
		Horse race	dTank	
Knowledge	templates	164	97	79
	knowledge rules	54	108	2
	total	218	205	81
Language	referring expressions	7	30	14
	lexicon	6	7	3
	templates	81	37	0
	total	94	74	17
Affect	causality relations	1	3	114
	goals and desires	2	2	0
	observation rules	19	8	0
	total	22	13	114
Gesture		11	6	33
Prosody		0	0	14
Facial Expression		0	0	9
Fusion		0	0	41

Table 7.4: Jess declarations changed vs unchanged by module

Module		unique to domain		unchanged	
		Horse race	dTank	Horse race	dTank
Knowledge	templates	67.5%	55.1%	32.5%	44.9%
	knowledge rules	96.4%	98.2%	3.6%	1.8%
	total	72.9%	71.7%	27.1%	28.3%
Language	referring expressions	33.3%	68.2%	66.7%	31.8%
	lexicon	66.7%	70.0%	33.3%	30.0%
	templates	100%	100%	0%	0%
	total	84.7%	81.3%	15.3%	18.7%
Affect	causality relations	0.9%	2.6%	99.1%	97.4%
	goals and desires	100%	100%	0%	0%
	observation rules	100%	100%	0%	0%
	total	16.2%	10.2%	83.8%	89.8%
Gesture		25.0%	15.4%	75.0%	84.6%
Prosody		0%	0%	100%	100%
Facial Expression		0%	0%	100%	100%
Fusion		0%	0%	100%	100%

Table 7.5: Jess declarations changed vs unchanged by module (relative percentages)

7.1.2 Discussion

The implementation effort measurement, although it is a very rough estimate, clearly shows that shifting the ERIC agent into a new domain required significantly less effort than implementing it from scratch. Of the modules that required changing, the most effort was involved in the knowledge module: this required identification of events and facts on which to comment, and designing rules to identify and deduce these facts. Once the knowledge module was populated with facts, the NLG templates were almost trivial to write; the main effort here is in authoring appropriate utterances for each fact in the world model. Similarly, the affect observer is trivial to write: it just needs to classify each fact in the world model as an event, action, or object. The agent's goals and desires, while not trivial, are quite intuitive to write; and although the cause/effect relations may be slightly less simple, they should still be easy for a domain expert.

The gesture module is a special case: a number of domain-dependent rules occur in the gesture module in both domains, linking events in the world directly with gestures. These rules are however only additional cosmetic touches; almost all of the character's gesture behaviour is generated by the domain-independent rules from the agent's affective state, and this alone would suffice to give the agent satisfactory gestures. In future it is planned to replace the domain-dependent gesture rules with similarly effective, yet domain-independent, rules that generate gestures on the basis of semantic information from the NLG module, rather than facts from the world model: so abstract concepts such as "comparison", "progress", or "direction" could be reflected in the gestures, as in the BEAT system of Cassell et al. [2001].

During the reimplementation of ERIC into the dTank domain, a number of components were found that were domain-dependent after all: especially on the interfaces between components:

Affect module Aside from the goals and desires and the cause/effect relations, the affect module was intended to be entirely domain-independent. However the affect module processes domain-independent `event` and `action` facts, whereas the knowledge module facts that are input to the affect module are left up to the implementer to specify. Thus it is necessary to have a separate set of domain-dependent rules that assert appropriate `event` or `action` facts for every incoming knowledge module fact.

NLG module As well as the templates and lexicon entries themselves, the NLG module contains a few other pieces of domain-dependent information: the generation of referring expressions, and the centers used for discourse coherence. Whereas these could be easily separated from the domain-independent Jess file, two further domain-dependent issues remain unresolved. Firstly, the variable names used in the generation of templates were chosen with the first domain in mind, and thus the referring expressions are bound to `?horse1`, `?horse2`, `?horse3`, etc. Secondly, the pronouns are hard-coded in the Java file that generates the templates; as mentioned in Chapter 5, in future this should be replaced by Jess facts and rules to allow dynamic generation of the appropriate pronoun, similar to generation of referring expressions.

Gesture module As mentioned above, as well as the gestures representing mood and emotion, each domain includes a number of domain-dependent gesture rules generating gestures directly from knowledge module facts.

Modifications to the Java As well as changing parts of the agent's Jess reasoning, the transfer into a new domain required the framework to be connected to a new domain interface to obtain facts from the world. This required some rewriting of the Java code; to reduce coupling, every world interface is a Java class that extends the abstract class de.dfki.racereporter.racesim.WorldInterface, however changing domain still requires that we instantiate the appropriate interface class in the controlling method, and catch any exceptions that it could throw. Ideally transferring the agent to a new domain would not require any recompilation of the Java code; however implementing and connecting a new domain interface clearly requires this.

7.2 GALA Competition Feedback

In contrast to the effort required to transfer ERIC to a new domain, the goals of engagement, reactivity and affectivity are much more subjective and hence difficult to measure. We can obviously claim that, by design, ERIC has affective responses to events, and reacts to events in the world; but these observations make no claims about the quality of the agent's responses.

As part of the GALA challenge, a video was produced of the agent commentating a three-minute horse race. This video was evaluated by two independent judges; their comments are shown below, and their ratings are in Table 7.6. As well as this feedback, ERIC won a judge's prize in the race reporter category[1].

7.2.1 Results

Reviewer 1:

> Overall a very nice commentator. At times some of the behavior, however, did appear repetitive and random.

Reviewer 2:

[1]All the GALA entries can be viewed at `http://hmi.ewi.utwente.nl/gala/finalists`.

Component	Reviewer 1	Reviewer 2
Engagement	⋆⋆⋆⋆	⋆⋆⋆⋆
Quality of speech	⋆⋆⋆⋆⋆	⋆⋆⋆
Quality of nonverbal signals	⋆⋆⋆	⋆⋆⋆
Aesthetic value of the animated agent model	⋆⋆⋆⋆	⋆⋆⋆⋆⋆
Natural language content and variety	⋆⋆⋆⋆	⋆⋆⋆⋆
Interactivity, reactivity	⋆⋆⋆	⋆⋆⋆
Overall judgment	⋆⋆⋆⋆	⋆⋆⋆⋆

Table 7.6: Feedback from the GALA reviewers (stars out of five)

The overall impression is engaging, the character and the environment design are very nice (though not the merit of the author, but of industrial partner). The speech sounds good, but does not reflect emotions, increasing excitement. In general, the emotions are not displayed well enough. The multimodal behavior of the presenter reminds a bit of a polite shopkeeper than of an excited sport reporter. This has a lot to do with the noise applied for body and head animation: it is not believable, after some time disturbing. But of course, completely frozen posture and head are not the solution. It would be not so difficult to sync posture changes and head movements to (emotional) events.

The hand gestures are not adequate (e.g. funny way of pointing at tracks with two hands, for farewell a pointing gesture is used).

I am eager to see the interactive real-time performance.

7.2.2 Discussion

This feedback indicates that overall the goal of affective engagement has been satisfied. However the agent's emotions were judged to be insufficiently well displayed; in particular emotional prosody was missing from his speech. The use of prosody to display the agent's emotional state would probably address the main criticisms of affectivity.

The low score on the "quality of nonverbal signals" component is reflected in the comments observing lack of speech prosody, and inadequate gestures. Again, reflecting the agent's affective state in his prosody would address the first criticism. The second is a result of the random process by which the agent's gestures are selected. As well as the improvement suggested in the comment (align posture changes and head movements to events), the agent's behaviour could be made less random by generating synchronised gestures from his utterances using a system such as BEAT [Cassell et al., 2001] or the system of Kipp et al. [2007].

The other component judged poor by both reviewers was "interactivity and reactivity". There is no comment by either commentator that explicitly refers to this criterion;

however the ERIC does sometimes exhibit a delay between the occurrence of an event and hist commentary on it, and it is likely that the score reflects this phenomenon. There are a number of factors that contribute to this effect. Firstly, events in the world occur faster than the agent can complete his utterances, thus sometimes an event may occur while the agent is still commenting on a past event. Secondly, the computational performance of the system itself may lead to delays.

In the situation where events are occurring faster than the agent can speak, the agent's reactivity could be improved by enabling him to interrupt a current utterance with a new one, if the new event is more salient than his current commentary. Alternatively, a prediction model might be added to the knowledge modelling, to allow the agent to anticipate salient events and thus avoid beginning a long utterance just before a likely salient event.

As well as the delay due to speaking, a delay between events and their commentary may also be introduced by poor computational performance of the system. The key bottlenecks in the system are the text-to-speech processing, and the individual reasoning components. The text-to-speech processing in the Charamel character is performed offline to produce a wav or mp3 file, which is then played back by the agent. This introduces a delay before each utterance which increases in size with the length of the utterance. To minimise this artifact, the utterances have been kept relatively short. The speed of the text-to-speech rendering could be further improved by performing the processing online rather than offline, that is by playing the generated speech while it is produced rather than writing it to a file and then playing that back.

We observed in Section 3.6.2 that the time taken for each component to process is critical. Several measures have already been described to optimise the performance of the reasoning components. These measures mostly require an author implementing the ERIC agent for a new domain to carefully craft his or her Jess rules to optimise performance. As well as the Jess rules, the performance can be affected by the component interaction: for example, the ALMA module currently takes a relatively long time to deduce emotions from the raw EEC values appraised by the affect module, which is reflected in the performance of the affect module. In this situation, optimisation of the bottleneck methods (to speed them up) or the component interaction (to call the bottleneck methods less frequently) is the only solution.

Chapter 8

Conclusion

8.1 Summary

In this thesis we have presented ERIC, a a homogeneous framework agent that can be configured to provide running commentary on a continuous event in real-time. ERIC has been implemented to commentate a simulated horse race, and a tank combat game. In the Gathering of Animated Lifelike Agents at at the 7th International Conference on Intelligent Virtual Agents, ERIC won the judge's prize in the race reporter challenge[1]. With minimal modification the system is configurable to provide commentary in any continuous dynamically changing environment; for example, it could commentate sports matches and computer games, or play the role of "tourist guide" during a self-guided tour of a city.

ERIC is implemented as a modular framework of Java-based components. It uses an expert system to generate a rich world model of declarative knowledge from limited input information in real-time. The agent features a template-based natural language generation system capable of generating anaphora and coherent discourse. It also uses a layered model of emotions, mood and personality to guide its output generation; these affects are generated from dynamic appraisal of events, actions and objects against goals and desires.

The architecture of ERIC was designed to by modular, parallel and homogeneous. To this end, the architecture is based on parallel rule-based reasoning engines, implemented in Java and Jess, a rule-based system itself implemented in Java. Throughout the architecture of ERIC, domain-specific knowledge is kept separate from domain-independent reasoning: for example, the NLG templates are kept outside the generation reasoning rules, and the goals/desires and cause/effect relations are separate from the affective appraisal rules. This results in an architecture that is highly reusable across domains.

In ERIC's knowledge module, an elaborate world model is deduced from limited input by an expert system implemented as rules in Jess. The systems on which ERIC commentates periodically update the knowledge module with a small number of facts about

[1]The GALA entries can be viewed at `http://hmi.ewi.utwente.nl/gala/finalists`.

their world state. From this input, the knowledge module produces a world model consisting of a large number of Jess facts describing the world. By basing the world model on declarative facts, the natural language generation templates can easily be conditional on the world state, since they are then simply conditional on facts in the world model. Similarly, it is quite easy to implement rules in the affect module to identify facts as events, actions or objects for its reasoning.

ERIC uses a template-based natural language generation system to generate output utterances. Each template is implemented as a Jess rule; thus the conditions are matched against the world state by the Jess engine. ERIC's NLG uses semantic centers for macro-level discourse coherence. Each candidate utterance is assigned a single backward-looking and a number of forward-looking centers; by comparing the forward and backward centers of successive utterances, the system determines the strength of their discourse relation and prefers candidate utterances with a stronger relation to their prior utterance.

In order to generate emotional responses, the agent has an affective appraisal module which uses a set of causal and belief relations to assign appraisals of emotion-eliciting conditions (EECs) to facts in the world model based on a set of goals and desires. These appraisals are used to generate an affective state. This dynamic appraisal of events relieves the need for manual tagging of events with affective stimuli or EEC values. ERIC's affective state is expressed in his lexical selection, facial expression and gestures. As well as these, the ability to express affective state via speech prosody is planned.

8.2 Evaluation

ERIC was designed to be homogeneous, domain-independent, behaviourally complex, affective and reactive. These aims were evaluated, on the one hand by measurement of the effort required to re-implement ERIC for a new domain, and on the other hand by a subjective assessment by independent judges.

The effort required to implement the agent for a new domain occurred mainly in the expected modules: namely, the knowledge module and the NLG templates. However the affect module also required some additional rules to transform the domain-dependent knowledge module into the domain-independent facts required for the EEC-generating reasoning. The affect module has been adjusted as a result of this finding to separate these domain-dependent rules from the domain-independent part of the module. Overall the effort required to implement the agent for a new domain was significantly lower than the effort required to implement the agent in the first instance.

The GALA feedback indicated that overall the agent was engaging and varied. However ERIC's emotions were judged to be insufficiently well displayed. In particular, emotional prosody was missing from his speech; the addition of a prosody module to generate this affective expression would answer this criticism.

8.3 Future work

The evaluation of the ERIC system revealed a number of avenues for future work. In particular, a prosody module, and the association of gestures with utterances is planned, to improve the agent's nonverbal output. In addition, we will add a mechanism for allowing the agent to either interrupt current utterances with more important ones, or avoid uttering the less important ones entirely in this situation.

The beginnings of a prosody module were described in Section 6.3.4. Using the emotion dimension prosody rules developed by Schröder [2004], the prosody module can specify a number of properties of the desired output prosody, which can then be matched with its utterance by the fusion module. The fusion module will transform the prosody specification output from the prosody module into prosody markup suitable for the text-to-speech system. In a similar way, the gesture module can tag its generated gestures with the utterances with which they are intended to coincide, and then the fusion module can send the matching speech and gesture commands to Charamel simultaneously.

In order to improve the reactivity of the agent, we plan to augment ERIC's fusion module with the ability to compare a current candidate utterance with the utterance that is currently being spoken, and then interrupt the current utterance if the new candidate is sufficiently important. This way significant events can be commentated immediately, rather than once the current utterance is finished. Additionally, the natural language generation and fusion modules will be extended to support a "null utterance", a signal for the agent to not speak: then using facts predicting that a important event is likely to occur soon, the natural language generation can generate such a "null utterance" to inform the fusion module to stay silent, thus leaving the system free to comment on the important event when it occurs.

As well as closing these open issues, many aspects of ERIC are intended to be extended. For instance, the gesture generation module currently generates semantic gestures based on the world model; in future it is planned to use semantic information from the NLG module instead, so that the gestures can be more easily matched to appropriate utterances, and also to allow more abstract concepts such as "comparison" or "progress" to be shown as gestures. Similarly, the prosody generation module can be extended to use semantic and syntactic information from the utterance as well as affective information. Finally, the discourse coherence rules in the fusion module are currently relatively simple, and use only a few relations between successive sentences; the commentator's output may be improved by extending these rules to support other, more complex discourse theories than Centering Theory such as DLTAG or RST.

Bibliography

Alassa and Anthony Jameson. Predictive role taking in dialog: Global anticipation feedback based on transmutability. In Sandra Carberry and Ingrid Zuckerman, editors, *Proceedings of the Fifth International Conference on User Modeling*, pages 137–144, 1996.

Elisabeth André, Gerd Herzog, and Thomas Rist. Generating multimedia presentations for robocup soccer games. In *RoboCup-97: Robot Soccer World Cup I*, pages 200–215, London, UK, 1998. Springer-Verlag.

Elisabeth André, Kim Binsted, Kumiko Tanaka-Ishii, Sean Luke, Gerd Herzog, and Thomas Rist. Three robocup simulation league commentator systems. *AI Magazine*, 21(1):57–66, 2000.

Robbert-Jan Beun, Eveliene de Vos, and Cilia Witteman. Embodied conversational agents: Effects on memory performance and anthropomorphisation. In *Proceedings of the International Conference on Intelligent Virtual Agents 2003*, LNAI 2792, pages 315–319, Berlin, Heidelberg, 2003. Springer-Verlag.

Justine Cassell. More than just another pretty face: Embodied conversational interface agents. *Communications of the ACM*, 43(4):70–78, 2000.

Justine Cassell, Timothy W. Bickmore, Hannes Hoge Vilhjalmsson, and H. Yan. More than just a pretty face: affordances of embodiment. In *Intelligent User Interfaces*, 2000.

Justine Cassell, Hannes Högni Vilhjálmsson, and Timothy Bickmore. BEAT: the Behavior Expression Animation Toolkit. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 477–486, New York, NY, USA, 2001. ACM Press.

Charamel GmbH. Charamel, 2007. URL `http://www.charamel.de/`. [accessed 23-August-2007].

Isaac G. Councill, Geoffrey P. Morgan, and Frank E. Ritter. dTank: A competitive environment for distributed agents. Technical Report ACS 2004 - 1, Applied Cognitive Science Lab, School of Information Sciences & Technology, The Pennsylvania State University, Applied Cognitive Science Lab, School of Information Sciences & Technology, University Park, PA 16802, 30 March 2004.

Paul Ekman and Wallace V Friesen. *Facial action coding system: A technique for the measurement of facial movement.* Consulting Psychologists Press., Palo Alto, CA, USA, 1978.

Dan Fielding, Mike Fraser, Brian Logan, and Steve Benford. Reporters, editors and presenters: Using embodied agents to report on online computer games. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1530–1531, Washington, DC, USA, 2004. IEEE Computer Society.

Mary Ellen Foster. Enhancing human-computer interaction with embodied conversational agents. In *Proceedings of HCI International 2007*, Beijing, July 2007.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Patrick Gebhard. ALMA – A Layered Model of Affect. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05)*, pages 29–36, Utrecht, 2005.

Patrick Gebhard, Michael Kipp, Martin Klesen, and Thomas Rist. Adding the emotional dimension to scripting character dialogues. In *Proceedings of the 4th International Working Conference on Intelligent Virtual Agents (IVA'03)*, pages 48–56, 2003.

Patrick Gebhard, Martin Klesen, and Thomas Rist. Coloring multi-character conversations through the expression of emotions. In *roceedings of the Tutorial and Research Workshop on Affective Dialogue Systems (ADS'04)*, pages 128–141, 2004.

Lewis R. Goldberg. An alternative "description of personality": The Big-Five factor structure. *Journal of Personality and Social Psychology*, 59(6):1216–1229, 1990.

Barbara J. Grosz, Aravind K. Joshi, and Scott Weinstein. Centering: A framework for modeling the local coherence of discourse. *Computational Linguistics*, 21(2):203 –225, 1995.

Gerd Herzog. From visual input to verbal output in the visual translator. Technical Report 124, Universität des Saarlandes, 1995.

Gerd Herzog and Peter Wazinski. VIsual TRAnslator: Linking perceptions and natural language descriptions. *Artificial Intelligence Review*, 8(2–3):175–187, 1994.

Gerd Herzog, Anselm Blocher, Klaus-Peter Gapp, Eva Stopp, and Wolfgang Wahlster. VITRA: Verbalisierung visueller Information. *Informatik Forschung und Entwicklung*, 11(1):12–19, 1996.

Ernest Friedman Hill. *Jess in Action: Java Rule-Based Systems.* Manning Publications Co., Greenwich, CT, USA, 2003.

Ernest Friedman Hill. Jess, the rule engine for the Java platform, 2006. URL `http://herzberg.ca.sandia.gov/jess/docs/70/`. [accessed 24-August-2007].

Eduard H. Hovy. Automated discourse generation using discourse structure relations. *Artificial Intelligence*, 63(1–2):341–385, 1993.

Michael Kipp, Kerstin H. Kipp, Alassane Ndiaye, and Patrick Gebhard. Evaluating the tangible interface and virtual characters in the interactive COHIBIT exhibit. In *Proceedings of the International Conference on Intelligent Virtual Agents (IVA'06)*, LNAI 4133, pages 434–444, Berlin, Heidelberg, 2006. Springer-Verlag.

Michael Kipp, Michael Neff, Kerstin H. Kipp, and Irene Albrecht. Toward natural gesture synthesis: Evaluating gesture units in a data-driven approach. In *Proceedings of the 7th International Conference on Intelligent Virtual Agents*, LNAI 4722, pages 15–28. Springer, 2007.

William C. Mann and Maite Taboada. Intro to RST /rhetorical structure theory/, 2007. URL `http://www.sfu.ca/rst/01intro/intro.html`. [accessed 15-August-2007].

Erwin Marsi and Ferdi van Rooden. Expressing uncertainty with a talking head in a multimodal question-answering system. In *Proceedings of the Workshop on Multimodal Output Generation (MOG 2007)*, pages 105–116, Aberdeen, UK, 25–26 January 2007.

Mark T. Maybury. A computational model of explanation for a tactical mission planner. In *Proceedings of the 1988 Symposium of Command and Control Research*, pages 382–386, Naval Post Graduate School and Monterey Resort Inn, Monterey, CA, 7–9 June 1988.

Mark T. Maybury. Generating summaries from event data. *International Journal of Information Processing and Management : Special Issue on Text Summarization.*, 31 (5):735–751, 1995.

Mark T. Maybury and Wolfgang Wahlster, editors. *Readings in Intelligent User Interfaces.* Morgan Kaufmann, San Francisco, 1998.

David McNeill. *Hand and Mind: What Gestures Reveal About Thought.* The University of Chicago Press, Chicago, IL, USA, 1992.

Albert Mehrabian. Pleasure-arousal-dominance: A general framework for describing and measuring individual differences in temperament. *Current Psychology*, 14(4):261–292, December 1996.

Geoffrey P. Morgan, Frank E. Ritter, William E. Stevenson, and Ian N. Schenck. dTank: An environment for architectural comparisons of competitive agents. In L. Allender and T. Kelley, editors, *Proceedings of the 14th Conference on Behavior Representation in Modeling and Simulation*, 05-BRIMS-043, pages 133–140, Orlando, FL, 2005. University of Central Florida, University of Central Florida.

Susanne Van Mulken, Elisabeth André, and Jochen Müller. The persona effect: How substantial is it? In *HCI '98: Proceedings of HCI on People and Computers XIII*, pages 53–66, London, UK, 1998. Springer-Verlag.

Clifford Nass, Jonathan Steuer, and Ellen R. Tauber. Computers are social actors. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 72–78, New York, NY, USA, 1994. ACM Press.

Alassane Ndiaye, Patrick Gebhard, Michael Kipp, Martin Klesen, Michael Schneider, and Wolfgang Wahlster. Ambient intelligence in edutainment: Tangible interaction with life-like exhibit guides. In *Proceedings of the Conference on INtelligent TEchnologies for interactive enterTAINment (INTETAIN'05)*, LNAI 3814, pages 102–111, Berlin, Heidelberg, 2005. Springer-Verlag.

Tsukasa Noma, Liwei Zhao, and Norm Badler. Design of a virtual human presenter. *IEEE Journal of Computer Graphics and Applications*, 20(4):79–85, July/August 2000.

Nuance Communications Inc. Nuance RealSpeak Solo, 2007. URL `http://www.nuance.com/realspeak/solo/`. [accessed 23-August-2007].

Jan Odijk. Generation of coherent monologues. In *CLIN V: Proceedings of the Fifth CLIN Meeting*, Netherlands, 1995. University of Twente.

Andrew Ortony, Allan Collins, and Gerald L. Clore. *The Cognitive Structure of Emotions.* Cambridge University Press, Cambridge, UK, 1988.

Rosalind W. Picard. *Affective Computing.* MIT Press, Cambridge, MA, USA, 1997. ISBN 0-262-16170-2.

Ehud Reiter and Robert Dale. Building applied natural language generation systems. *Journal of Natural Language Engineering*, 3(1):57–87, 1997.

Norbert Reithinger, Patrick Gebhard, Markus Löckelt, Alassane Ndiaye, Norbert Pfleger, and Martin Klesen. VirtualHuman—Dialogic and affective interaction with virtual characters. In *ICMI '06: Proceedings of the 8th international conference on Multimodal interfaces*, pages 51–58, New York, NY, USA, 2006. ACM Press.

Jeff Rickel and W. Lewis Johnson. STEVE: A pedagogical agent for virtual reality. In Katia P. Sycara and Michael Wooldridge, editors, *Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98)*, pages 332–333, New York, 1998. ACM Press.

Raoul Rickenberg and Byron Reeves. The effects of animated characters on anxiety, task performance, and evaluations of user interfaces. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 49–56, New York, NY, USA, 2000. ACM Press.

Martin Rumpler. *Statusbasierte Verhaltenssteuerung von virtuellen Charakteren.* PhD thesis, Universität des Saarlandes, 2007 (to appear).

C. F. Schmidt, N. S. Sridharan, and J. L. Goodson. The plan recognition problem: An intersection of psychology and artificial intelligence. *Artificial Intelligence*, 11(1–2): 45–83, 1978.

Marc Schröder. Dimensional emotion representation as a basis for speech synthesis with non-extreme emotions. In *Proc. Workshop on Affective Dialogue Systems*, pages 209–220, Kloster Irsee, Germany, 2004.

Mariët Theune. ANGELICA: Choice of output modality in an embodied agent. In *International Workshop on Information Presentation and Natural Multimodal Dialogue (IPNMD-2001)*, Verona, Italy, Dec 2001.

Mariët Theune, Esther Klabbers, Jan Odijk, Jan-Roelof de Pijper, and Emiel Krahmer. From data to speech: A general approach. *Natural Language Engineering*, 7(1):47–86, 2001.

Mariët Theune, Sander Faas, Anton Nijholt, and Dirk Heylen. The virtual storyteller. In *Proceedings of the Technologies for Interactive Digital Storytelling and Entertainment (TIDSE) Conference*, pages 204–215, 2003.

Mariët Theune, Koen Meijs, Dirk Heylen, and Roeland Ordelman. Generating expressive speech for storytelling applications. *IEEE Transactions on Audio, Speech and Language Processing*, 14(4):1137–1144, 2006.

Jürgen Trouvain and William J. Barry. The prosody of excitement in horse race commentaries. In *Proceedings of the ISCA Workshop on Speech and Emotion*, pages 86–91, Northern Ireland, 2000.

Kees van Deemter, Emiel Krahmer, and Mariët Theune. Plan-based vs. template-based NLG: a false opposition? In *'May I Speak Freely?', workshop associated with 23rd German Conference on Artificial Intelligence*, September 1999.

Kees van Deemter, Emiel Krahmer, and Mariët Theune. Real vs. template-based natural language generation: a false opposition? *Computational Linguistics*, 31(1):15–23, 2005.

Hans van Vliet. *Software engineering (2nd ed.): principles and practice.* John Wiley & Sons, Inc., New York, NY, USA, 2000.

Vinoba Vinayagamoorthy, Marco Gillies, Anthony Steed, Emmanuel Tanguy, Xueni Pan, Celine Loscos, and Mel Slater. Building expression into virtual characters. In *Eurographics Conference State of the Art Report*, Vienna, 2006.

Dirk Voelz, Elisabeth André, Gerd Herzog, and Thomas Rist. Rocco: A RoboCup soccer commentator system. In *RoboCup-98: Robot Soccer World Cup II*, pages 50–60, London, UK, 1999. Springer-Verlag.

Bonnie Webber. DLTAG: Extending lexicalized tag to discourse. *Cognitive Science*, 28: 751–779, 2004.

Wikipedia. Observer pattern — Wikipedia, the free encyclopedia, 2007. URL `http://en.wikipedia.org/w/index.php?title=Observer_pattern&oldid=147874069`. [accessed 9-August-2007].