Interleaved Preparation and Output in the COMIC Fission Module

Mary Ellen Foster
Institute for Communicating and Collaborative Systems
School of Informatics, University of Edinburgh
2 Buccleuch Place, Edinburgh EH8 9LW United Kingdom
M.E.Foster@ed.ac.uk

Abstract

We give a technical description of the fission module of the COMIC multimodal dialogue system, which both plans the multimodal content of the system turns and controls the execution of those plans. We emphasise the parts of the implementation that allow the system to begin producing output as soon as possible by preparing and outputting the content in parallel. We also demonstrate how the module was designed to ensure robustness and configurability, and describe how the module has performed successfully as part of the overall system. Finally, we discuss how the techniques used in this module can be applied to other similar dialogue systems.

1 Introduction

In a multimodal dialogue system, even minor delays in processing at each stage can add up to produce a system that produces an overall sluggish impression. It is therefore critical that the output system avoid as much as possible adding any delays of its own to the sequence; there should be as little time as possible between the dialogue manager's selection of the content of the next turn and the start of that turn's output. When the output incorporates temporal modalities such as speech, it is possible to take advantage of this by planning later parts of the turn even as the earlier parts are being played. This means that the initial parts of the output can be produced more quickly, and any delay in preparing the later parts is partly or entirely eliminated. The net effect is that the overall perceived delay in the output is much shorter than if the whole turn had been prepared before any output was produced.

In this paper, we give a technical description of the output system of the COMIC multimodal dialogue system, which is designed to allow exactly this interleaving of preparation and output. The paper is arranged as follows. In Section 2, we begin with a general overview of multimodal dialogue systems, concentrating on the design decisions that affect how output is specified and produced. In Section 3, we then describe the COMIC multimodal dialogue system and show how it addresses each of the relevant design decisions. Next, in Section 4, we describe how the segments of an output plan are represented in COMIC, and how those segments are prepared and executed in parallel. In Section 5, we discuss two aspects of the module implementation that are relevant to its role within the overall COMIC system: the techniques that were used to ensure the robustness of the fission module, and how it can be configured to support a variety of requirements. In Section 6, we then assess the practical impact of the parallel processing on the overall system responsiveness, and show that the output speed has a perceptible effect on the overall user experiences with the system. Finally, in Section 7, we outline the aspects of the COMIC output system that are applicable to similar systems.

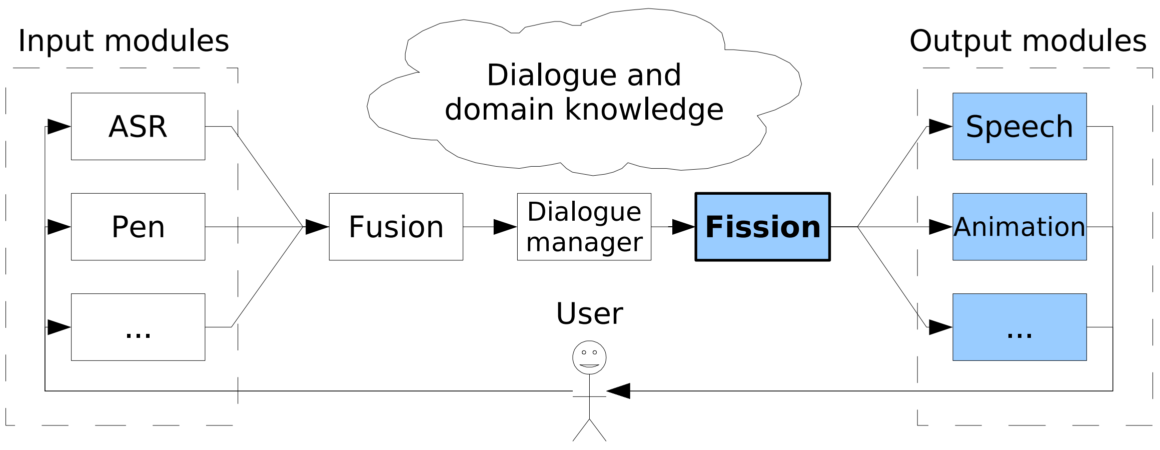


Figure 1: High-level architecture of a typical multimodal dialogue system

2 Output in Multimodal Dialogue Systems

Most multimodal dialogue systems use the basic high-level architecture shown in Figure 1. Input from the user is analysed by one or more input-processing modules, each of which deals with an individual input channel; depending on the application, the input channels may include speech recognition, pen-gesture or handwriting recognition, or information from visual sensors, for example. The messages from the various sources are then combined by a fusion module, which resolves any cross-modal references and produces a combined representation of the user input. This combined representation is sent to the dialogue manager, which uses a set of domain and dialogue knowledge sources to process the user input, interact with the underlying application if necessary, and specify the content to be output by the system in response. The output specification is sent to the fission module, which creates a presentation to meet the specification, using a combination of the available output channels. Again, depending on the application, a variety of output channels may be used; typical channels are synthesised speech, on-screen displays, or behaviour specifications for an animated agent or a robot.

This general structure is typical across multimodal dialogue systems; however, there are a number of design decisions that must be made when implementing a specific system. As this paper concentrates on the output components highlighted in Figure 1, we will discuss the design decisions that have a particular impact on those parts of the dialogue system: the domain of the application, the output modalities, the turn-taking protocol, and the division of labour among the modules. We will use as examples the the WITAS (Lemon et al., 2002), MATCH (Walker et al., 2002), and SmartKom (Wahlster, 2005) systems.

The domain of the system and the interactions that it is intended to support both have an influence on the type of output that is to be generated. Many systems are designed primarily to support information exploration and presentation, and concentrate on effectively communicating the necessary information to the user. SmartKom and MATCH both fall into this category: SmartKom deals with movie and television listings, while MATCH works in the domain of restaurant recommendations. In a system such as WITAS, which incorporates real-time control of a robot helicopter, very different output must be generated to communicate the current state and goals of the robot to the user.

The choice of output modalities also affects the output system—different combinations of modalities require different types of temporal and spatial coordination, and different methods of allocating the content across the channels. Most multimodal dialogue systems use synthesised speech as an output modality, often in combination with lip-synch and other behaviours of an animated agent (e.g., MATCH, SmartKom). Various types of visual output are also often employed, including interactive maps (MATCH, WITAS), textual information presentations (SmartKom, MATCH), or images from visual sensors (WITAS). Some systems also dynamically adapt the output channels based on changing constraints; for example, SmartKom chooses a spoken presentation over a visual one in an eyes-busy situation.

Another factor that has an effect on the design of the output components is the turn-taking protocol selected by the system. Some systems—such as WITAS—support *barge-in* (Ström and Seneff, 2000); that is, the user may interrupt the system output at any time. Allowing the user to interrupt can permit a more intuitive interaction with the system; however, supporting barge-in creates many technical complications. For example, it is crucial that the output system be prepared to stop at any point, and that any parts of the system that track the dialogue history be made aware of how much of the intended content was actually produced. For simplicity, many systems—including SmartKom—instead use half-duplex turn-taking: when the system is producing output, the input modules are not active. This sort of system is technically more straightforward to implement, but requires that the user be given very clear signals as to when the system is and is not paying attention to their input. MATCH uses a click-to-talk interface, where the user presses a button on the interface to indicate that they want to speak; it is not clear whether the system supports barge-in.

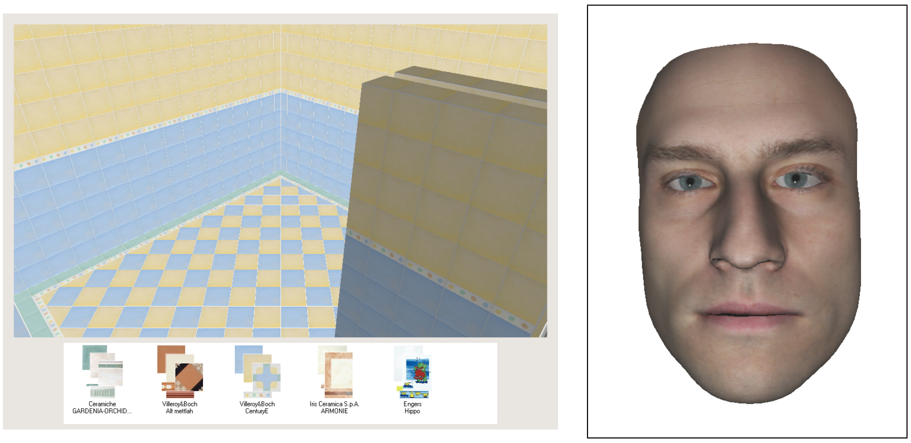
The division of labour across the modules also differs among implemented systems. First of all, not all systems actually incorporate a separate component that could be labelled *fission*: for example, in WITAS, the dialogue manager itself also addresses the tasks of presentation planning and coordination. The components of the typical natural language generation "pipeline" (Reiter and Dale, 2000) may be split across the modules in a variety of ways. When it comes to content selection, for instance, in MATCH the dialogue manager specifies the content at a high level, while the text planner selects and structures the actual facts to include in the presentation; in WITAS, on the other hand, the specific content is selected by the dialogue manager. The tasks of text planning and sentence planning may be addressed by various combinations of the fission module and any text-generation modules involved—SmartKom creates the text in a separate generation module, while in MATCH text and sentence planning is more tightly integrated with content selection.

Coordination across multiple output channels is also implemented in various ways. If the only presentation modality is an animated agent, in many cases the generated text is sent directly to the agent, which then communicates privately with the speech synthesiser to ensure synchronisation. This "visual text-to-speech" configuration is the default behaviour of the Greta (de Rosis et al., 2003) and RUTH (DeCarlo et al., 2004) animated presentation agents, for instance. However, if the behaviour of the agent must be coordinated with other forms of output, it is necessary that the behaviour of all synchronised modules be coordinated centrally. How this is accomplished in practice depends on the capabilities of selected speech synthesiser that is used. In SmartKom, for example, the presentation planner pre-synthesises the speech and uses the schedule returned by the synthesiser to create the full multimodal schedule; in MATCH, on the other hand, the speech synthesiser sends progress messages as it plays its output, which are used to control the output in the other modalities at run time.

3 The COMIC Dialogue System

COMIC[1] (COnversational Multimodal Interaction with Computers) is an EU IST 5th framework project combining fundamental research on human-human dialogues with advanced technology development for multimodal conversational systems. The COMIC multimodal dialogue system adds a dialogue interface to a CAD-like application used in sales situations to help clients redesign their bathrooms. The input to COMIC consists of speech, pen gestures, and handwriting; turn-taking is strictly half-duplex, with no barge-in or click-to-talk. The output combines the following modalities:

[1] http://www.hcrc.ed.ac.uk/comic/


Ceramiche GARDENIA-ORCHID...
Villeroy&Boch Alt mettlah
Villeroy&Boch CenturyE
Iris Ceramica S.p.A. ARMONIE
Engers Hippo

"*[Nod]* Okay. *[Choose design]* *[Look at screen]* THIS design *[circling gesture]* is CLASSIC. It uses tiles from VILLEROY AND BOCH's CENTURY ESPRIT series. There are FLORAL MOTIFS and GEOMETRIC SHAPES on the DECORATIVE tiles."

Figure 2: COMIC interface and sample output

- Synthesised speech, created using the OpenCCG surface realiser (White, 2005a;b) and synthesised by a custom Festival 2 voice (Clark et al., 2004) with support for APML prosodic markup (de Carolis et al., 2004).

- Facial expressions and gaze shifts of a talking head (Breidt et al., 2003).

- Direct commands to the design application.

- Deictic gestures at objects on the application screen, using a simulated mouse pointer.

Figure 2 shows the COMIC interface and a typical output turn, including commands for all modalities; the small capitals indicate pitch accents in the speech, with corresponding facial emphasis.

The specifications from the COMIC dialogue manager are high-level and modality-independent; for example, the specification of the output shown in Figure 2 would indicate that system should show a particular set of tiles on the screen, and should give a detailed description of those tiles. When the fission module receives input from the dialogue manager, it selects and structures multimodal content to create an output plan, using a combination of scripted and dynamically-generated output segments. The fission module addresses the tasks of low-level content selection, text planning, and sentence planning; surface realisation of the sentence plans is done by the OpenCCG realiser. The fission module also controls the output of the planned presentation by sending appropriate messages to the output modules including the text realiser, speech synthesiser, talking head, and bathroom-design GUI. Coordination across the modalities is implemented using a technique similar to that used in SmartKom: the synthesised speech is prepared in advance, and the timing information from the synthesiser is used to create the schedule for the other modalities.

The plan for an output turn in COMIC is represented in a tree structure; for example, Figure 3 shows part of the plan for the output in Figure 2. A plan tree like this is created from the top down, with the children created left-to-right at each level, and is executed in the same order. The planning and execution processes for a turn are started together and run in parallel, which makes it possible to begin producing output as soon as possible and to continue planning while output is active. In the following section, we describe the set of classes and algorithms that make this interleaved preparation and execution possible.

The COMIC fission module is implemented in a combination of Java and XSLT. The current module consists of 18 000 lines of Java code in 88 source files, and just over 9000 lines of XSLT templates. In the diagrams and algorithm descriptions that follow, some non-essential details are omitted for simplicity.

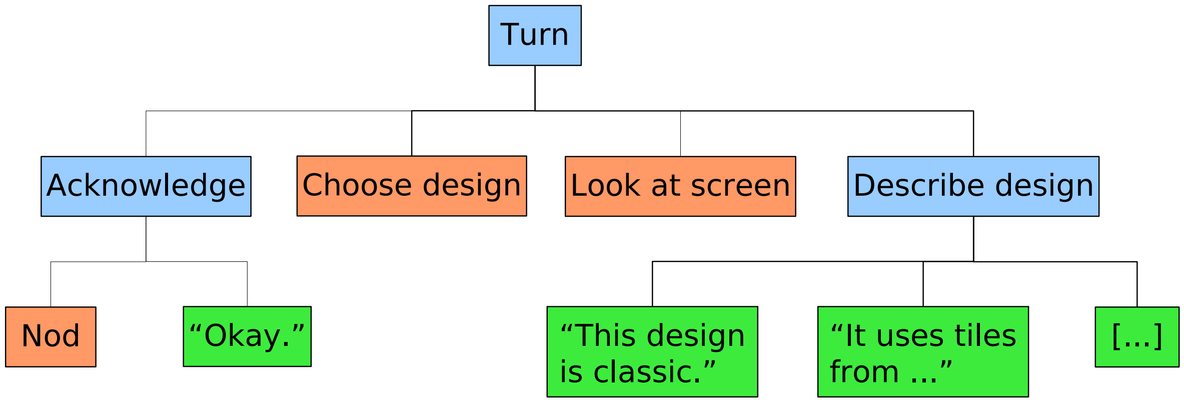


Figure 3: Output plan

4 Representing an Output Plan

Each node in a output-plan tree such as that shown in Figure 3 is represented by an instance of the Segment class. The structure of this abstract class is shown in Figure 4; the fields and methods defined in this class control the preparation and output of the corresponding segment of the plan tree, and allow preparation and output to proceed in parallel.

Each Segment instance stores a reference to its parent in the tree, and defines the following three methods:

- `plan()` Begins preparing the output.
- `execute()` Produces the prepared output.
- `reportDone()` Indicates to the Segment's parent that its output has been completed.

`plan()` and `execute()` are abstract methods of the Segment class; the concrete implementations of these methods on the subclasses of Segment are described later in this section. Each Segment also has the following Boolean flags that control its processing; all are initially false.

- `ready` This flag is set internally once the Segment has finished all of its preparation and is ready to be output.
- `skip` This flag is set internally if the Segment encounters a problem during its planning, and indicates that the Segment should be skipped when the time comes to produce output.

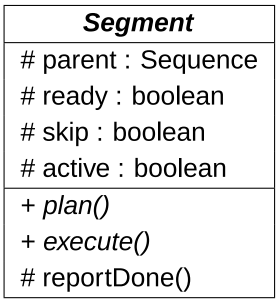


Figure 4: Structure of the Segment class

- `active` This flag is set externally by the Segment's parent, and indicates that this Segment should produce its output as soon as it is ready.

The activity diagram in Figure 5 shows how these flags and methods are used during the preparation and output of a Segment. Note that a Segment may send asynchronous queries to other modules as part of its planning. When such a query is sent, the Segment sets its internal state and exits its `plan()` method; when the response is received, preparation continues from the last state reached. Since planning and execution proceed in parallel across the tree, and the planning process may be interrupted to wait for responses from other modules, the `ready` and `active` flags may be set in either order on a particular Segment. Once both of these flags have been set, the `execute()` method is called automatically. If both `skip` and `active` are set, the Segment instead automatically calls `reportDone()` without ever ex-

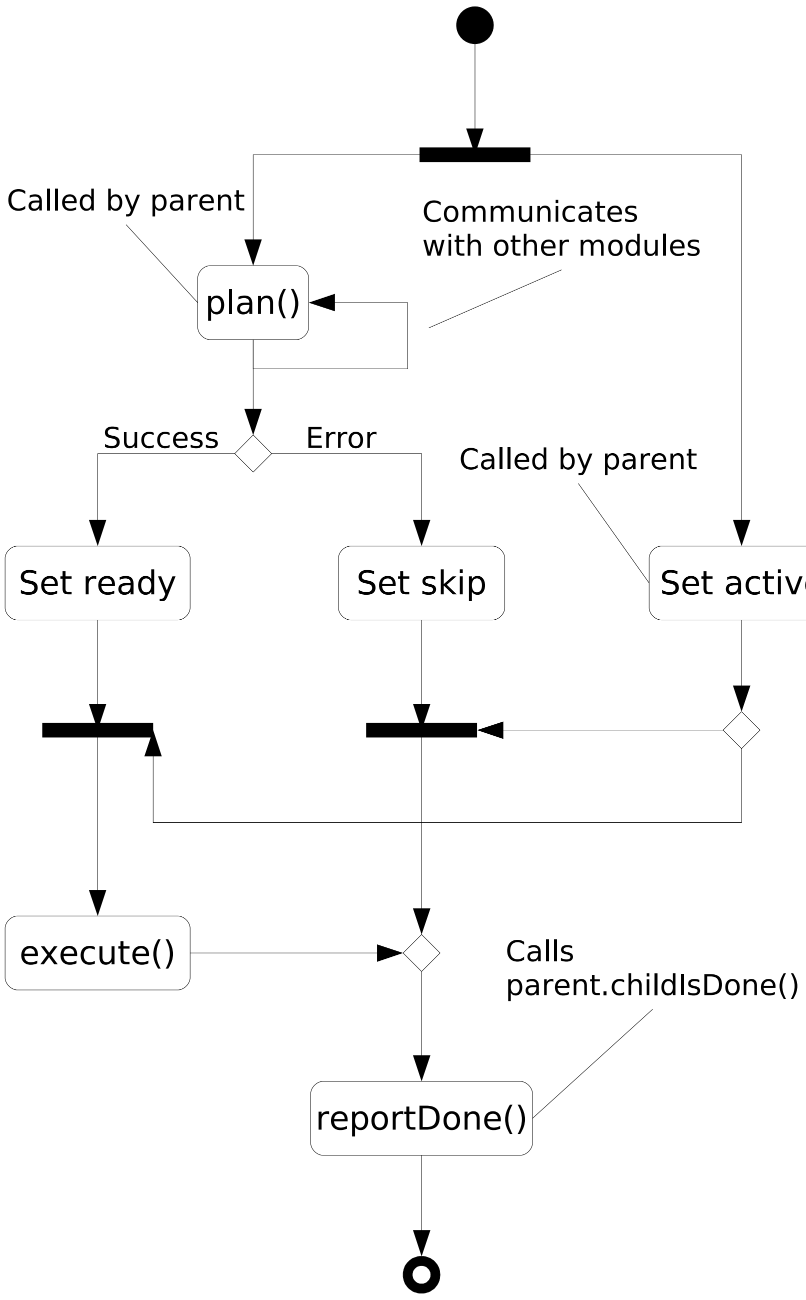


Figure 5: Segment preparation and output

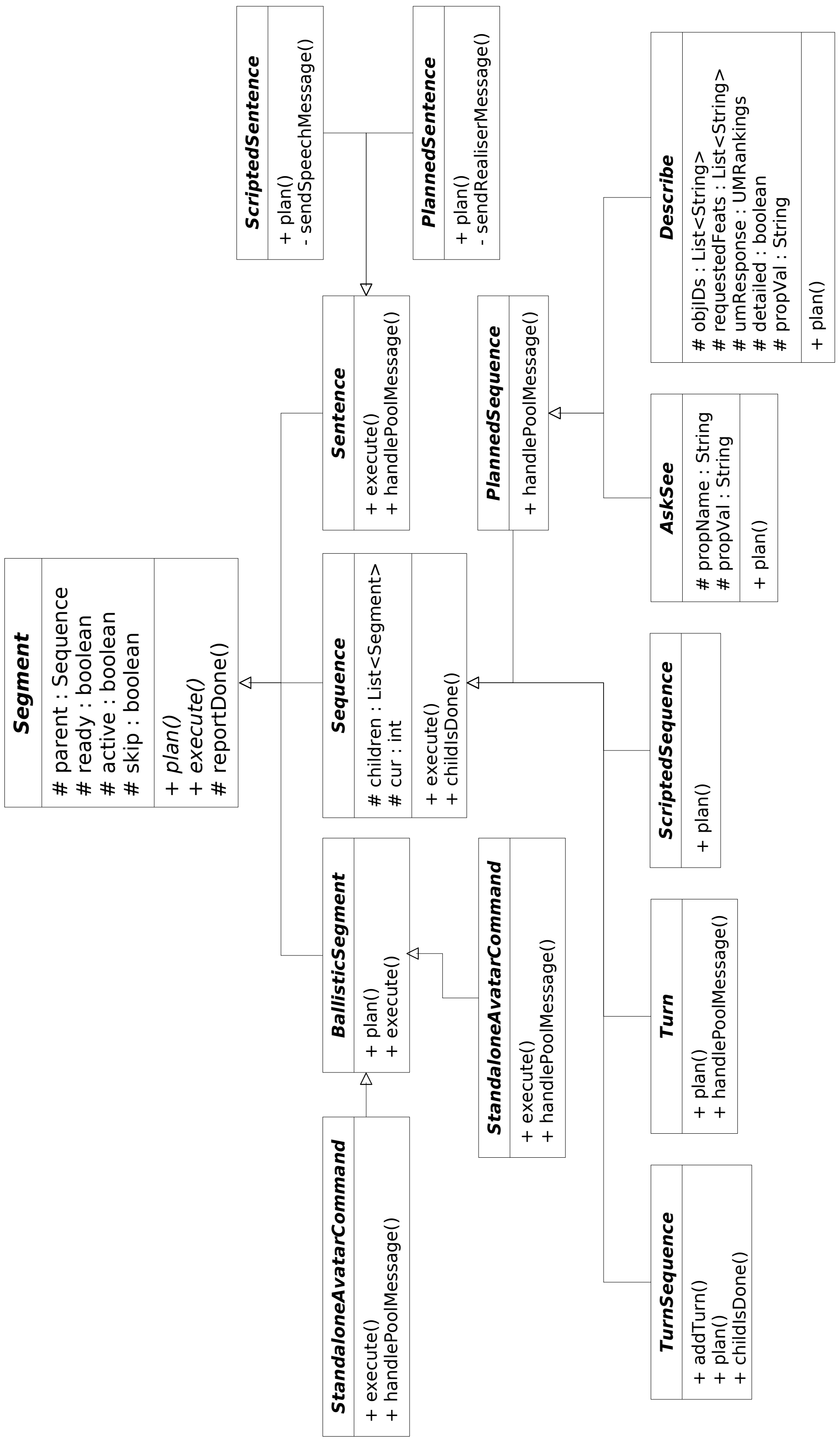


Figure 6: Segment class hierarchy

ecuting; this allows Segments with errors to be skipped without affecting the output of the rest of the turn.

The full class hierarchy under Segment is shown in Figure 6. There are three main top-level subclasses of Segment, which differ primarily based on how they implement `execute()`:

Sequence An ordered sequence of Segments. It is executed by activating each child in turn.

BallisticSegment A single command whose duration is determined by the module producing the output. It is executed by sending a message to the appropriate module and waiting for that module to report back that it has finished.

Sentence A single sentence, incorporating coordinated output in all modalities. Its schedule is computed in advance, as part of the planning process; it is executed by sending a "go" command to the appropriate output modules.

In the remainder of this section, we discuss each of these classes and its subclasses in more detail.

4.1 Sequence

All internal nodes in a presentation-plan tree (coloured blue in Figure 3) are instances of some type of Sequence. A Sequence stores a list of child Segments, which it plans and activates in order, along with a pointer to the currently active Segment. Figure 7 shows the pseudocode for the main methods of a typical Sequence.

Note that a Sequence calls sets its `ready` flag as soon as all of its necessary child Segments have been created, and only then begins calling `plan()` on them. This allows the Sequence's `execute()` method to be called as soon as possible, which is critical to allowing the fission module to begin producing output from the tree before the full tree has been created.

When `execute()` is called on a Sequence, it calls `activate()` on the first child in its list. All subsequent children are activated by calls to the `childIsDone()` method, which is called by each child as part of its `reportDone()` method after its execution is completed. Note that this ensures that the children of a Sequence will always be executed in the proper order, even if they are prepared out of

```
public void plan() {
    // Create child Segments

    cur = 0;
    ready = true;

    for( Segment seg: children ) {
        seg.plan();
    }
}

public void execute() {
    children.get( 0 ).activate();
}

public void childIsDone() {
    cur++;
    if( cur >= children.size() ) {
        reportDone();
    } else {
        children.get( cur ).activate();
    }
}
```

Figure 7: Pseudocode for Sequence methods

order. Once all of the Sequence's children have reported that they are done, the Sequence itself calls `reportDone()`.

The main subclasses of Sequence, and their relevant features, are as follows:

TurnSequence The singleton class that is the parent of all Turns. It is always active, and new children can be added to its list at any time.

Turn Corresponds to a single message from the dialogue manager; the root of the output plan in Figure 3 is a Turn. Its `plan()` implementation creates a Segment corresponding to each dialogue act from the dialogue manager; in some cases, the Turn adds additional children not directly specified by the DAM, such as the verbal acknowledgement and the gaze shift in Figure 3.

ScriptedSequence A sequence of canned output segments stored as an XSLT template. A ScriptedSequence is used anywhere in the dialogue where dynamically-generated content is not necessary; for example, instructions to the user and acknowledgements such as the leftmost subtree in Figure 3 are stored as ScriptedSequences.

PlannedSequence In contrast to a ScriptedSequence, a PlannedSequence creates its children dynamically depending on the dialogue context. The principal type of PlannedSequence is a description of one or more tile designs, such as that shown in Figure 2. To create the content of such a description, the fission module uses information from the system ontology, the dialogue history, and the model of user preferences to select and structure the facts about the selected design and to create the sequence of sentences to realise that content. This process is described in detail in (Foster and White, 2004; 2005).

4.2 BallisticSegment

A BallisticSegment is a single command for a single output module, where the output module is allowed to choose the duration at execution time. In Figure 3, the orange *Nod*, *Choose design*, and *Look at screen* nodes are examples of BallisticSegments. In its `plan()` method, a BallisticSegment transforms its input specification into an appropriate message for the target output module. When `execute()` is called, the BallisticSegment sends the transformed command to the output module and waits for that module to report back that it is done; it calls `reportDone()` when it receives that acknowledgement.

4.3 Sentence

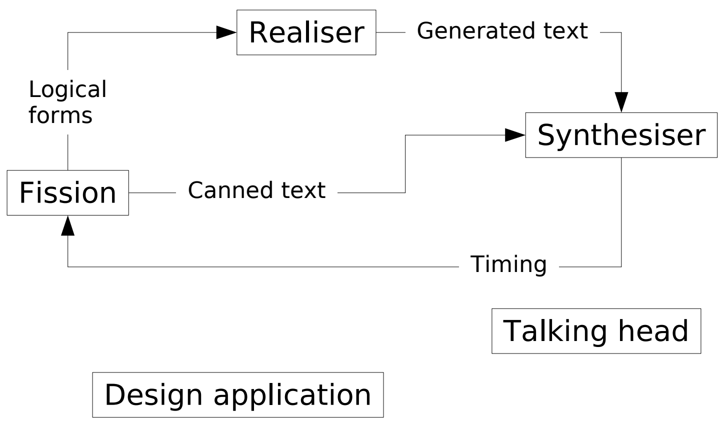
The Sentence class represents a single sentence, combining synthesised speech, lip-synch commands for the talking head, and possible coordinated behaviours on the other multimodal channels. The timing of a sentence is based on the timing of the synthesised speech; all multimodal behaviours are scheduled to coincide with particular words in the text. Unlike a BallisticSegment, which allows the output module to determine the duration at execution time, a Sentence must prepare its schedule in advance to ensure that output is coordinated across all of the channels. In Figure 3, all of the green leaf nodes containing text are instances of Sentence.

There are two types of Sentences: ScriptedSentences and PlannedSentences. A ScriptedSentence is generally created as part of a ScriptedSequence, and is based on pre-written text that is sent directly to the speech synthesiser, along with any necessary multimodal behaviours. A PlannedSentence forms part of a PlannedSequence, and is based on logical forms for the OpenCCG realiser (White, 2005a;b). The logical forms may contain multiple possibilities for both the text and the multimodal behaviours; the OpenCCG realiser uses statistical language models to make a final choice of the actual content of the sentence.

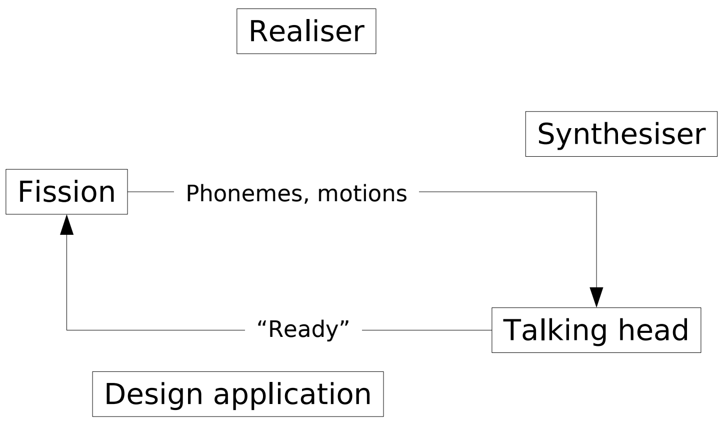
The first step in preparing either type of Sentence is to send the text to the speech synthesiser (Figure 8(a)). For a ScriptedSentence, the canned text is sent directly to the speech synthesiser; for a PlannedSentence, the logical forms are sent to the realiser, which then creates the text and sends it to the synthesiser. In either case, the speech-synthesiser input also includes marks at all points where multimodal output is intended. The speech synthesiser prepares and stores the waveform based on the input text, and returns timing information for the words and phonemes, along with the timing of any multimodal coordination marks.

The fission module uses the returned timing information to create the final schedule for all modalities. It then sends the animation schedule (lip-synch commands, along with any coordinated expression or gaze behaviours) to the talking-head module so that it can prepare its animation in advance (Figure 8(b)). Once the talking-head module has prepared the animation for a turn, it returns a “ready” message. The design application does not need its schedule in advance, so once the response is received from the talking head, the Sentence has finished its preparation and is able to set its `ready` flag.

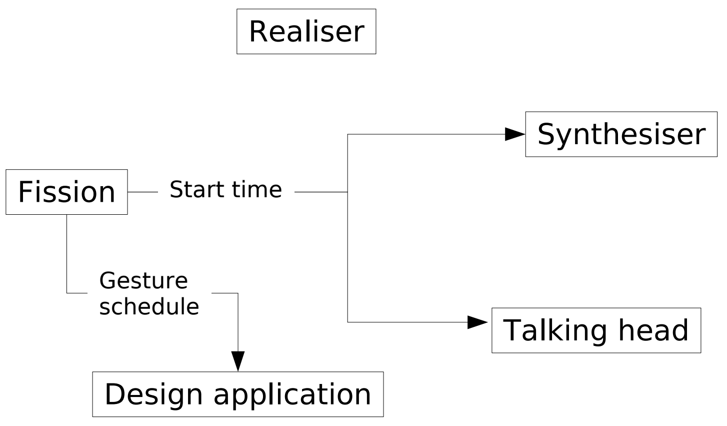
When a Sentence is executed by its parent, it selects a desired start time slightly in the future and sends two messages, as shown in Figure 8(c). First, it sends a “go” message with the selected starting time to the speech-synthesis and talking-head modules; these modules then play the prepared output for that turn at the given time. The Sentence also sends the concrete schedule for any coordinated gesture commands to the bathroom-design application at this point. After sending its messages, the Sentence waits until the scheduled duration has elapsed, and then calls `reportDone()`.



(a) Preparing the speech



(b) Preparing the animation



(c) Producing the output

Figure 8: Planning and executing a Sentence

5 Robustness and Configurability

In the preceding section, we gave a description of the data structures and methods that are used when preparing and executing and output plan. In this section, we describe two other aspects of the module that are important to its functioning as part of the overall dialogue system: its ability to detect and deal with errors in its processing, and the various configurations in which it can be run.

5.1 Error Detection and Recovery

Since barge-in is not implemented in COMIC, the fission module plays an important role in turn-taking for the whole COMIC system: it is the module that informs the input components when the system output is finished, so that they are able to process the next user input. The fission module therefore incorporates several measures to ensure that it is able to detect and recover from unexpected events during its processing, so that the dialogue is able to continue even if there are errors in some parts of the output.

Most input from external modules is validated against XML schemas to ensure that it is well-formed, and any messages that fail to validate are not processed further. As well, all queries to external modules are sent with configurable time-outs, and any Segment that is expecting a response to a query is also prepared to deal with a time-out.

If a problem occurs while preparing any Segment for output—either due to an error in internal processing, or because of an issue with some external module—that Segment immediately sets its `skip` flag and stops the preparation process. As described in Section 4, any Segments with this flag set are then skipped at execution time. This ensures that processing is able to continue as much as possible despite the errors, and that the fission module is still able to produce output from the parts of an output plan unaffected by the problems and to perform its necessary turn-taking functions.

5.2 Configurability

The COMIC fission module can be run in several different configurations, to meet a variety of evaluation, demonstration, and development situations. The fission module can be configured not to wait for "ready" and "done" responses from either or both of the talking-head and design-application modules; the fission module simply proceeds with the rest of its processing as if the required response had been received. This allows the whole COMIC system to be run without those output modules enabled. This is useful during development of other parts of the system, and for running demos and evaluation experiments where not all of the output channels are used. The module also has a number of other configuration options to control factors such as query time-outs and the method of selecting multimodal coarticulations.

As well, the fission module has the ability to generate multiple alternative versions of a single turn, using different user models, dialogue-history settings, or multimodal planning techniques; this is useful both as a testing tool and as part of a system demonstration. The module can also store all of the generated output to a script, and to play back the scripted output at a later time using a subset of the full system. This allows alternative versions of the system output to be directly compared in user evaluation studies such as (Foster, 2004; Foster and White, 2005).

6 Output Speed

In the final version of the COMIC system, the average time[2] that the speech synthesiser takes to prepare the waveform for a sentence is 1.9 seconds, while the average synthesised length of a sentence is 2.7 seconds. This means that, on average, each sentence takes long enough to play that the next sentence is ready as soon as it is needed; and even when this is not the case, the delay between sentences is still greatly reduced by the parallel planning process.

The importance of beginning output as soon as possible was demonstrated by a user evaluation of an interim version of COMIC (White et al., 2005). Subjects in that study used the full COMIC system in one of two configurations: an "expressive" condition, where the talking head used all of the expressions it was capable of, or a "zombie" condition where all of the behaviours of the head were disabled except for lip-synch. One effect of this

[2]On a Pentium 4 1.6GHz computer.

difference was that the system gave a consistently earlier response in the expressive condition—a facial response was produced an average of 1.4 seconds after the dialogue-manager message, while spoken input did not begin for nearly 4 seconds. Although that version of the system was very slow, the subjects in the expressive condition were significantly less likely to mention the overall slowness than the subjects in the zombie condition.

After this interim evaluation, effort was put into further reducing the delay in the final system. For example, we now store the waveforms for acknowledgements and other frequently-used texts pre-synthesised in the speech module instead of sending them to Festival, and other internal processing bottlenecks were eliminated. Using the same computers as the interim evaluation, the fission delay for initial output is under 0.5 seconds in the final system.

7 Conclusions

The COMIC fission module is able to prepare and control the output of multimodal turns. It prepares and executes its plans in parallel, which allows it to begin producing output as soon as possible and to continue with preparing later parts of the presentation while executing earlier parts. It is able to produce output coordinated and synchronised across multiple modalities, to detect and recover from a variety of errors during its processing, and to be run in a number of different configurations to support testing, demonstrations, and evaluation experiments. The parallel planning process is able to make a significant reduction in the time taken to produce output, which has a perceptible effect on user satisfaction with the overall system.

Some aspects of the fission module are specific to the design of the COMIC dialogue system; for example, the module performs content-selection and sentence-planning tasks that in other systems might be addressed by a dialogue manager or text-generation module. Also, aspects of the communication with the output modules are tailored to the particular modules involved: the fission module makes use of features of the OpenCCG realiser to help choose the content of many of its turns, and the implementation of the design application is obviously COMIC-specific.

However, the general technique of interleaving preparation and execution, using the time while the system is playing earlier parts of a turn to prepare the later parts, is easily applicable to any system that produces temporal output, as long as the same module is responsible for preparing and executing the output. There is nothing COMIC-specific about the design of the Segment class or its immediate sub-classes.

As well, the method of coordinating distributed multimodal behaviour with the speech timing (Section 4.3) is a general one. Although the current implementation relies on the output modules to respect the schedules that they are given—with no adaptation at run time—in practice the coordination in COMIC has been generally successful, providing that three conditions are met. First, the selected starting time must be far enough in the future that it can be received and processed by each module in time. Second, the clocks on all computers involved in running the system must be synchronised precisely. Finally, the processing load on each computer must be low enough that timer events do not get delayed or pre-empted.

Since the COMIC system does not support barge-in, the current fission module always produces the full presentation that is planned, barring processing errors. However, since the module produces its output incrementally, it would be straightforward to extend the processing to allow execution to be interrupted after any Segment, and to know how much of the planned output was actually produced.

Acknowledgements

Many thanks to Peter Poller, Tilman Becker, and especially Michael White for helpful advice on and discussions about the implementation of the COMIC fission module, and to the anonymous reviewers for their comments on the initial version of this paper. This work was supported by the COMIC project (IST-2001-32311).

References

M. Breidt, C. Wallraven, D.W. Cunningham, and H.H. Bülthoff. 2003. Facial animation based on 3d scans and motion capture. In Neill Campbell, editor, *SIGGRAPH 03 Sketches & Applications*. ACM Press.

R.A.J. Clark, K. Richmond, and S. King. 2004. Festival 2 – build your own general purpose unit selection speech synthesiser. In *Proceedings, 5th ISCA workshop on speech synthesis*.

B. de Carolis, C. Pelachaud, I. Poggi, and M. Steedman. 2004. APML, a mark-up language for believable behaviour generation. In H. Prendinger, editor, *Life-like Characters, Tools, Affective Functions and Applications*, pages 65–85. Springer.

F. de Rosis, C. Pelachaud, I. Poggi, V. Carofiglio, and B. De Carolis. 2003. From Greta's mind to her face: modelling the dynamics of affective states in a conversational embodied agent. *International Journal of Human-Computer Studies*, 59(1–2):81–118.

D. DeCarlo, M. Stone, C. Revilla, and J.J. Venditti. 2004. Specifying and animating facial signals for discourse in embodied conversational agents. *Computer Animation and Virtual Worlds*, 15(1):27–38.

M.E. Foster. 2004. User evaluation of generated deictic gestures in the T24 demonstrator. Public deliverable 6.5, COMIC project.

M.E. Foster and M. White. 2004. Techniques for text planning with XSLT. In *Proceedings, NLPXML-2004*.

M.E. Foster and M. White. 2005. Assessing the impact of adaptive generation in the COMIC multimodal dialogue system. In *Proceedings, IJCAI 2005 Workshop on Knowledge and Reasoning in Practical Dialogue systems*.

O. Lemon, A. Gruenstein, and S. Peters. 2002. Collaborative activities and multi-tasking in dialogue systems. *Traitement Automatique des Langues (TAL)*, 43(2):131–154.

E Reiter and R Dale. 2000. *Building Natural Language Generation Systems*. Cambridge University Press.

N. Ström and S. Seneff. 2000. Intelligent barge-in in conversational systems. In *Proceedings, ICSLP-2000*, volume 2, pages 652–655.

W. Wahlster, editor. 2005. *SmartKom: Foundations of Multimodal Dialogue Systems*. Springer. In press.

M.A. Walker, S. Whittaker, A. Stent, P. Maloor, J.D. Moore, M. Johnston, and G. Vasireddy. 2002. Speech-plans: Generating evaluative responses in spoken dialogue. In *Proceedings, INLG 2002*.

M. White. 2005a. Designing an extensible API for integrating language modeling and realization. In *Proceedings, ACL 2005 Workshop on Software*.

M. White. 2005b. Efficient realization of coordinate structures in Combinatory Categorial Grammar. *Research on Language and Computation*. To appear.

M. White, M.E. Foster, J. Oberlander, and A. Brown. 2005. Using facial feedback to enhance turn-taking in a multimodal dialogue system. In *Proceedings, HCI International 2005 Thematic Session on Universal Access in Human-Computer Interaction*.