A Framework for Rapid Development of Multimodal Interfaces

Frans Flippo * †
fflippo@caip.rutgers.edu

Allen Krebs *
krebs@caip.rutgers.edu

Ivan Marsic *
marsic@caip.rutgers.edu

* Rutgers University
CAIP Center
Piscataway, NJ 08854-8088
+1 732 445 0542

† Delft University of Technology
Dept. Information Technology and Systems
2628 CD Delft, The Netherlands
+31 15 278 7504

ABSTRACT

Despite the availability of multimodal devices, there are very few commercial multimodal applications available. One reason for this may be the lack of a framework to support development of multimodal applications in reasonable time and with limited resources. This paper describes a multimodal framework enabling rapid development of applications using a variety of modalities and methods for ambiguity resolution, featuring a novel approach to multimodal fusion. An example application is studied that was created using the framework.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**]: User InterfacesInteraction styles, Natural language, Voice I/O; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems

General Terms

Algorithms, Design, Human Factors

Keywords

Multimodal interfaces, multimodal fusion, application frameworks, command and control, direct manipulation

1. INTRODUCTION

Multimodal interfaces provide a very natural way for humans to perform tasks on a computer, using direct manipulation and speech: interaction methods that are used daily in human-to-human communication. However, despite the availability of high-accuracy speech recognizers and the maturing of multimodal devices such as gaze trackers, touch screens, and gesture trackers, very little applications take advantage of these technologies. One reason for this may be that the cost in time of implementing a multimodal interface is prohibitive. One desiring to equip an application with such an interface must start from scratch, implementing access to external sensors, developing ambiguity resolution algorithms, and making calls to the application's API based on the determined user intention.

However, when properly implemented, a large part of the code in a multimodal system can be reused. This aspect was identified and used to implement a multimodal application framework. The framework uses a novel and parallelisable application-independent fusion technique that can be easily augmented to support application-specific demands as well as new modalities. The fusion algorithm separates the three parts of fusion: obtaining data from modalities, fusing that data to come to an unambiguous meaning, and calling application code to take an action based on that meaning. Separation of these three tasks makes the framework applicable to a wide range of applications and modalities.

1.1 Requirements and Constraints

The framework enables existing applications to be equipped with a multimodal interface. Therefore the design is to be minimally intrusive on existing application code, but rather function alongside it, calling application code when data is needed from or actions need to be performed in the application; additionally, the interface accepts callbacks from the application when changes occur that change the discourse context or that need to be reported to the user.

To gain user acceptance for a multimodal system, response times must be reasonably small. If the system takes too long to process a user's spoken command, the user will think the command was not understood and repeat it, resulting in confusion when the command then gets carried out twice. All this results in annoyance and should be avoided at all cost. Ideally, response times should be below a second.

Many multimodal systems are geocentric in nature [6]. Combining speech with gesture, gaze, and mouse serves to link spoken references to spatial data (i.e. objects and locations on-screen) with their antecedents as derived from the aforementioned modalities. Therefore our framework is optimized for this type of fusion.

Direct manipulation is currently the most popular mode of interaction. It appeared in the late 1970's [9]. Its software design was described in [12]. It is a proven form of interaction that should not be replaced, but rather be com-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICMI'03, November 5–7, 2003, Vancouver, British Columbia, Canada.
Copyright 2003 ACM 1-58113-621-8/03/0011 ...$5.00.

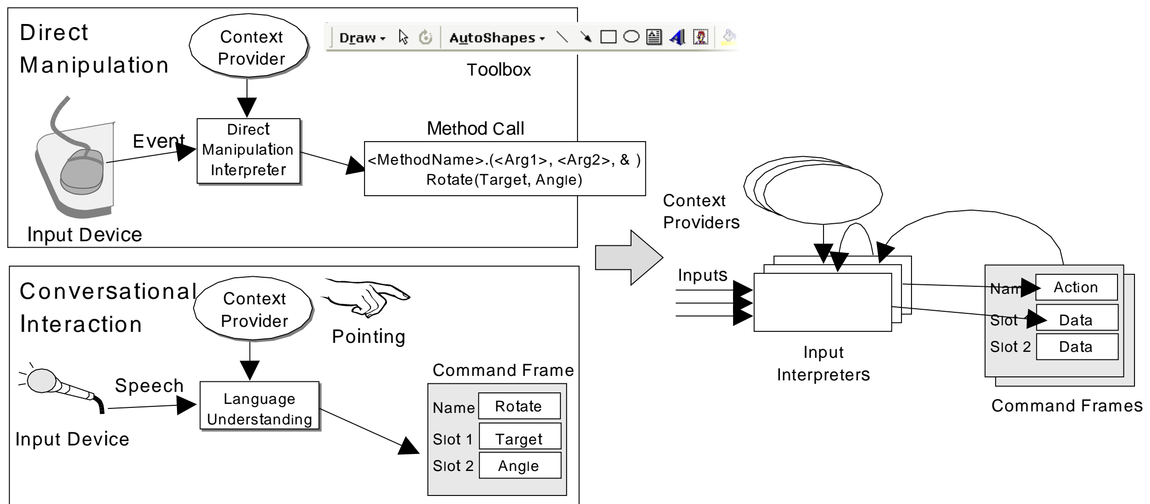

Figure 1: A high-level view of software architectures for direct manipulation and conversational interaction. In direct manipulation, the toolbox (in this figure we show a part of a PowerPoint toolbox) determines the context of the manipulation and the end result is a method call on the application. Similarly, in conversational interaction, pointing helps to resolve ambiguities in speech. The architecture on the right hand side shows a generalisation of these two paradigms.

plemented by conversational interfaces. The key components of the software architecture for direct manipulation are summarized in Figure 1. Primary inputs are mouse clicking and dragging and command selection through menus or toolboxes. On the other hand, recent multimodal interfaces focus on conversational interaction (also summarized in Figure 1). Unfortunately, almost none of these interface include direct manipulation in conversational interaction. Pointing is used only to resolve deictic references in speech. Our objective is to provide an architecture which will support both. This architecture is summarized in Figure 1 on the right hand side. Creating an architecture which supports direct manipulation and conversational interaction *in parallel* combines the strengths of both interaction styles while compensating for the weaknesses. By maintaining direct manipulation as a choice of interaction, we are not limiting user actions to those that are speech-centric. The user can choose between traditional direct manipulation, speech, or a combination of both.

Our focus is on single-user multimodal interfaces. Multi-user interfaces have many more problems, such as having to recognize multiple voices, determine the source of gestures, etc. Our system *does* allow collaboration where users are on different machines, and the application described at the end of the paper is an example of this.

We will first discuss some related work. In Section 2 we describe the conceptual design of the framework, followed by its implementation in Section 3. We then look at the multimodal interface that was made for Flatscape, our collaborative situation map tool using the framework, followed by conclusions and ideas for future work.

1.2 Related Work

The first known multimodal interface was built in 1980 by R. Bolt [1]. It provided an interface in which shapes could be created, moved, copied, removed, and named using a combination of speech and pointing, for example "put that to the left of the green triangle," "copy that there," "call that the calendar," "move the calendar here". Fusion was done at the parse level. Every time an anaphor or deictic reference was recognized, the system would immediately see where the user was pointing and resolve the reference. The system also had an ability to learn new words. When the user said "call that <*name*>," the system would tell the speech recognizer to switch from recognition mode to training mode so that the name that the user gave the object would be learned. The components of the system necessarily had to be tightly integrated because of the way the system was designed. While performing fusion during speech yields a straightforward implementation of fusion, gestures and speech are in general not synchronized, that is, gesture precedes or follows a spoken reference, and assuming that they are demands the user to change his normal behavior to use the system: the system trains the user. This is not desirable.

Krahnstoever [6] describes a multimodal framework targeted specifically at fusing speech and gesture with output being done on large screen displays. Several applications are described that have been implemented using this framework. The fusion process is not described in great detail, but appears to be optimized for and limited to integration of speech and gesture, using inputs from cameras that track a user's head and hands.

The W3C has set up a multimodal framework specifically for the web [7]. This does not appear to be something that has actually been implemented by the W3C. Rather, it proposes a set of properties and standards — specifically the Extensible Multimodal Annotation Markup Language (EMMA) — that a multimodal architecture should adhere to.

The QuickSet system [4], built by the Oregon Graduate Institue, integrates pen with speech to create a multimodal system. QuickSet goes beyond simple point-and-speak commands and recognizes more complex pen gestures,

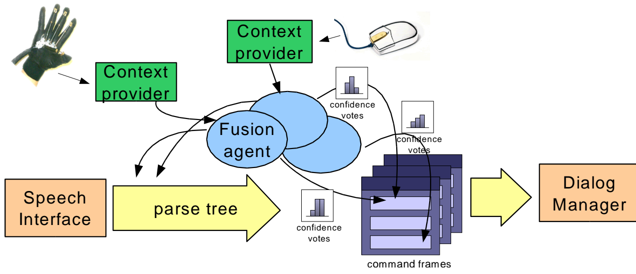

Figure 2: The fusion process – from parse tree to frames

like arrows and lines to better support direct manipulation. This allows for richer and more natural multimodal interaction. The system employs a Members-Teams-Committee technique very similar to the fusion technique described in this paper, using parallel agents to estimate a posteriori probabilities for various possible recognition results, and weighing them to come to a decision. However, our approach is more reusable as it separates the data – or feature – acquisition from the recognition. Also, it supports a variety of simultaneous modalities whereas QuickSet seems to be built solely for pen and speech-based interaction.

2. FRAMEWORK DESIGN

2.1 An Object-Oriented Framework

The fusion system described in this paper has been implemented as a framework. A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications [5]. Developing a framework involves determining what functionality is common to the applications in the target domain and abstracting away from application-specific functionality. This common functionality is the framework's immutable *core*, while *"hot spots"* in the framework are places in which a developer must "plug in" code to come to a specific, working application [8]. An important aspect of an application framework is the inversion of control, or "old code calls new code". In an application created with a framework, the framework core (old code) makes calls to the code that is plugged in to the hot spots (new code). It is this inversion that makes a framework an attractive paradigm for application development: the developer does not need to have knowledge of the framework's internals, but only needs to implement the interfaces that define the hot spots.

Configuration of the implemented framework is largely declarative: the user specifies (declares) structure, not procedure. The framework uses the user's specification to perform fusion, but the user needs to have no knowledge of *how* the framework does this. The user provides the "what" knowledge, while the framework contains the "how" knowledge. This makes the framework an ideal tool for non-experts. It also allows the framework to be changed without affecting existing applications, since the same declarations will still apply for another framework implementation.

2.2 Fusion

Our framework features a new approach to fusion that is reusable across applications and modalities. The process is depicted in figure 2. The input to the fusion process is a semantic parse tree with time stamps as generated by the natural language parser component of the speech interface. This parse tree needs to be transformed into frames that the dialog manager can use to make calls to the application. To accomplish this, the natural language concepts in the parse tree need to be mapped to application concepts. In addition, ambiguity needs to be resolved. Ambiguity exists when the user uses pronouns or deictic references, for example "remove *that*', or "do reconnaissance *here*". Another case of ambiguity is ellipsis, a linguistic construct in which words that are implied by context are omitted, such as "rotate this clockwise ... *and this too*". The last phrase can be expanded to "and rotate this clockwise, too".

Resolving agents operate on the parse tree to realize the aforementioned mapping of concepts and resolution of ambiguity. The implementation details of resolving agents are not specified by the framework. All that is expected is that the agents take a fragment from the parse tree, perform some transformation on it, and use it to fill a slot in the semantic frame that is sent to the dialog manager. The agents can use data from a modality (through an access object we call "context provider") to give them a context in which to perform their task. Context providers can provide data from an external sensor, such as a gaze tracker, but also from more abstract data sources such as dialog history or application state (e.g. which toolbox button is selected). An agent performing pronoun resolution might have access to gaze or gesture input to resolve a pronoun to an object on the screen that the user pointed or looked at. Any agent will typically have access just one such input. This keeps the design of the agents simple, as they do not need to be concerned with combining data from multiple sources. This combining is done by the fusion manager. It *is* possible for resolving agents to share the same modality, and the framework is designed so that this is possible, even when a device has an exclusive use policy.

The agents themselves do not actually perform fusion. Their task is to perform an assessment of what they think the contents of a slot in the frame should be. Each agent will provide zero or more possible solutions with corresponding probability scores. The whole of the solutions provided by all agents will finally determine what the slot will contain. This is shown in Figure 3.

To make resolving agents reusable, the resolution process is separated from the acquisition of data from modalities. The resolution process is implemented in the resolving agents, while the acquisition of data is the responsibility of the context providers. Resolving agents merely specify the type of data they expect to receive from their context provider. In this way, an agent that requires (x, y)-data points to do its work can accept data from any context provider that provides (x, y)-data, such as a mouse, a gaze tracker, or a haptic glove. In a system with a mouse and a gaze tracker, for instance, two copies of the same pronoun resolution agent might be active, one using data from the mouse, and another using data from the gaze tracker. Each will give its resolutions along with corresponding probability scores, based on the data they have access to.

Thus, resolving agents operate *locally* with only the information they have access to, namely the fragment of the parse tree they use and the data they receive from their modality, if any. However, all agents together create a *global* result that takes into account all of the parse tree and all of the

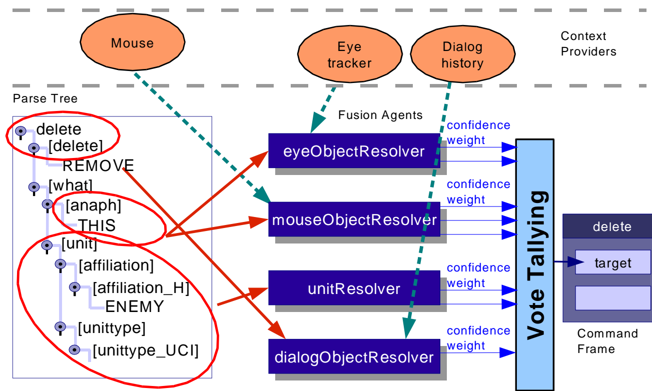

Figure 3: Combining results from different resolving agents (resolvers) to fill a slot

available modalities. Because each resolving agent works independently of the others, the agents can work in parallel, taking advantage of multiprocessor hardware to increase performance.

Context providers provide timestamps along with their data. These can be used by the resolvers so select data that are applicable to the parse tree fragment they are handling, using the timestamps that the natural language parser provides. For instance, the pronoun resolver agent mentioned before will look at data points that were generated around the time that the pronoun was spoken. Timestamps for speech data and context data ensure that the modality streams are properly synchronized.

2.2.1 Fusion Manager

The fusion manager controls the fusion process. It is responsible for:

1. choosing a frame for the parse tree and creating it
2. spawning resolving agents and passing them parse tree fragments to work with
3. taking the possible values for each slot from the resolving agents and choosing one, based on the probability scores provided and the weight assigned to each resolving agent.
4. merging frames from the conversational interface with method calls from the application's GUI, resolving ambiguities to create a frame with unambiguous meaning
5. handing a completed frame to the dialog manager

Direct manipulation and conversational interaction run in parallel and may influence each other. Commands uttered by the user can change the interpretation of mouse events. Conversely, the state of an application's GUI elements, such as a toolbox, can influence the meaning of spoken commands or gestures. The fusion manager implements the merging of direct management and conversational input.

If direct manipulation is done by itself, it must be done with the mouse or a device emulating a mouse. Currently, multimodal actions and actions using other devices than the mouse need speech to drive them. Having speech as the 'primary modality' in this way avoids delays imposed by systems that allow other modalities to be used by themselves and need time thresholds to define the end of a dialog action, as in [11].

The fusion manager is configured through an XML file and will be described in more detail in Section 3.

2.3 Dialog Management

The dialog manager receives frames from the fusion manager. First it updates the dialog history with the contents of the frame's slots. Then it calls an action script that is defined for the frame. This script will ultimately make a call to the application to perform a certain task, but first it will typically check whether the frame is 'complete', that is, whether all the slots that are required to be filled are indeed filled. If not, it can send feedback to the user to request him to provide the missing data. In the future we intend to implement this as part of the dialog manager, so that specifying which slots are required is all that is needed to have the dialog manager check for missing slots and report missing data to the user. However, slot requirements can be conditional, so implementation is not straightforward. The current solution is flexible, at the cost of some extra implementation effort.

As the rest of the framework, the dialog manager contains no application-specific code. All application-specific data is in the configuration file, which the dialog manager shares with the fusion manager.

2.4 Fission

Fission [2] is currently unimplemented in the framework. However, conceptually it is the inverse of fusion and it will be implemented as such. Given a semantic frame, the fission manager will distribute its slots to different fission agents, which create output to send to a modality as well as placing corresponding text in the semantic tree that is sent to the natural language generator. Where resolving agents *resolve* ambiguity by using multimodal input, fission agents *create* it for a single multimodal output. For instance, an anaphor generator adds a pronoun to the parse tree while creating a command for its modality to point to or hilight the object being referred to on the screen. The various fission agents will be responsible for sending output at the correct time, with the fission manager driving the fission agents.

```
<frame name="delete" test="delete/delete" uses="delete">
 <slot name="glyph">
  <source select="delete/what/anaph">
   <resolve resolver="mouseObjectResolver" weight="0.5" />
   <resolve resolver="eyeObjectResolver" weight="0.4" />
  </source>
  <source select="delete/what/unit">
   <resolve resolver="unitResolver" weight="0.3" />
  </source>
  <source select="delete">
   <resolve resolver="dialogObjectResolver" weight="0.1" />
  </source>
 </slot>
 <action language="javascript">
  if (frame.glyph) {
   application.api.deleteGlyph(frame.glyph.glyph);
  }
 </action>
</frame>
```

Figure 4: A sample frame declaration

3. IMPLEMENTATION

The framework is implemented in Java. Java was chosen due to its strong typing, extensive class library, dynamic object instantiation and object reflection capabilities, and the fact that Java applications can run on different platforms without recompiling. The first two aspects result in quicker development time and less errors, the third gives the framework much of its power, while the last allows applications created with the framework to be deployed on any Java 1.4 capable platform.

3.1 Fusion Manager

It is the fusion manager's task to take the possible values for a slot from the resolving agents and choose the one that is optimal, based on the resolving agents' probabilities, and a confidence value or weight for each agent. Currently a very simple voting algorithm is employed, in which the scores for each value are summed and the one with the highest total sum is selected. However, the framework can accomodate any algorithm desired, and we are looking into using fuzzy reasoning and/or Bayesian networks for this purpose, using models trained on empirical data, to obtain better predictions of the user's intent.

The fusion manager, resolving agents, and context providers are configured through an XML file containing declarations for frames, resolvers and context providers. An example frame declaration is shown in Figure 4. The declaration specifies the name of the frame, an XPath test on the parse tree that must succeed for the frame to be used, and a set of slots with XPath expressions for each slot specifying their data source and a list of one or more resolvers (resolving agents) for each source.

The fusion manager uses the XPath test to determine which frame to instantiate. If multiple XPath expressions evaluate to 'true', the fusion manager prefers the frame that was used in the previous utterance, if possible. This ensures that multiple-step dialogs continue as intended. If no frame is of the same type as the previous one and there are multiple frames to choose from, feedback can be generated asking the user to be more specific. The actual implementation of this is left up to the developer, so any message can be generated, but output on the screen is also possible, for instance.

3.1.1 Resolving Contradictory Inputs

Contradiction between modalities can arise. For example, a user may say "move that infantry squad" while pointing to or looking at an infantry army. On the slot level, speech is treated like any modality, so the results from resolving "infantry squad" — that is, a list of all infantry squads — will participate in the voting process along with the result of resolving "that" using a pointing modality and possibly the dialog history, which will also result in a list of objects — those in the vincinity of the location the user was pointing. Ultimately, the developer decides which modality 'wins' in this case by the weights that he or she allocates to each modality. By adjusting the weights appropriately, the developer can achieve that the pointing modality wins if the object being pointed at is under the mouse cursor, but if the pointer is a certain threshold away from any object, the speech modality will win instead. Since the scores from each resolving agent are summed, the fusion manager may choose an object that is somewhat close to the pointer and is (in our example) an infantry squad over an object directly under the pointer.

A better approach in case of unresolvable ambiguity could be, again, to ask the user for clarification, e.g. "That is an infantry army, not an infantry squad. What do you want to move?".

3.2 Fusion Interfaces

Four interfaces are crucial to the fusion process:

- **Resolver** – All resolving agents implement the `Resolver` interface. A resolving agent is initialized by the fusion manager, which passes the initalization parameters as read from its configuration file to the `init()` method.

  `Resolvers` will usually be created by subclassing the `AbstractResolver` class, which implements the `Resolver` interface and inherits from `AbstractSessionObject`. The latter class provides common functionality for objects that operate within a multimodal dialog session and can be dynamically instantiated. The key method of the `Resolver` interface is the `resolve()` method, which takes a DOM `Element` representing a parse tree fragment and returns an `Iterator` containing the agents' resolutions along with their assessed probabilities.

  The following standard resolvers are provided:

  - `AnaphorResolver` — Resolves a pronoun to the on-screen object it refers to as determined from the input modality. The probabilities that accompany the solutions are proportional to the object's distance from the (x, y)-data point that is selected. This resolver does not actually look at the data in the parse tree fragment, it just uses the time stamps.
  - `DeicticResolver` — Resolves a deictic reference to its location on the screen as determined from the input modality. The probabilites that accompany the solutions are determined by the frequency of the point in the (x, y) data list. This resolver does not actually look at the data in the parse tree fragment, it just uses the time stamps.

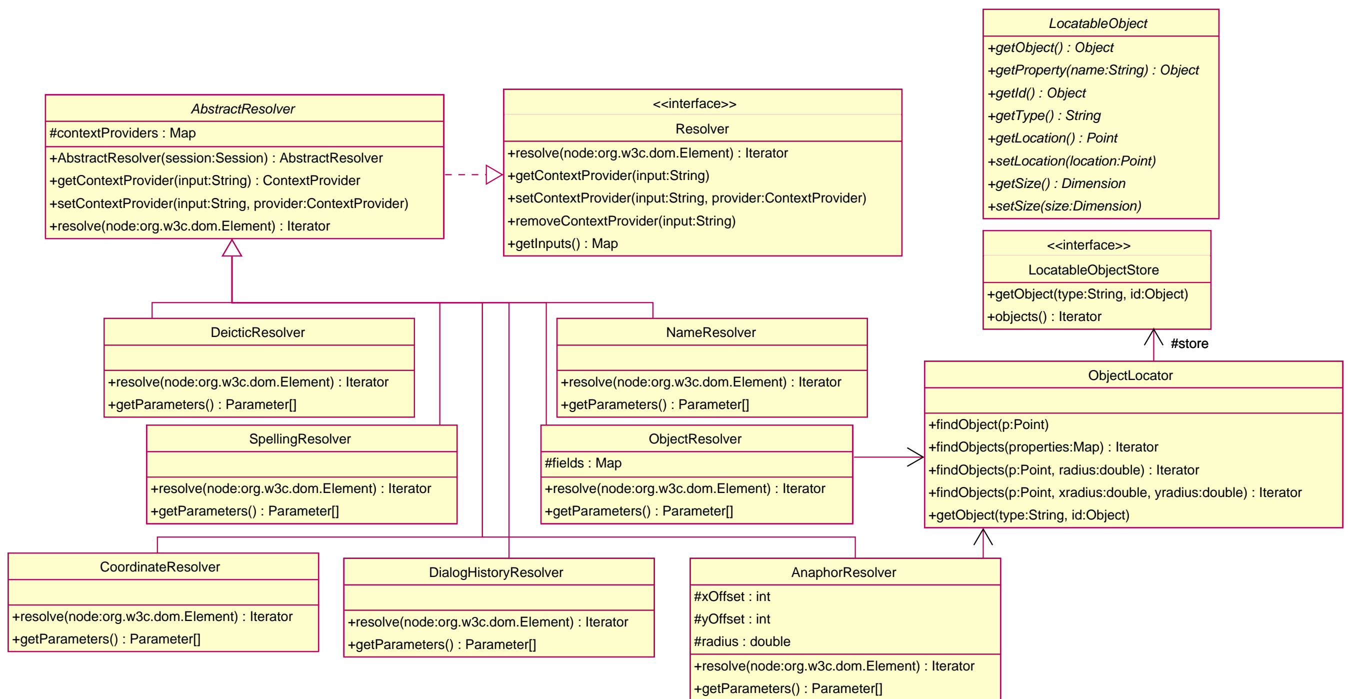

Figure 5: The `Resolver` class hierarchy and related classes

- `ObjectResolver` — Resolves a set of object attributes to the objects that match those attributes. For example, when the user is to be able to select an object using speech only with an expression such as "remove the hostile infantry army", this resolver is used to return all objects matching the attributes named: "hostile", "infantry", and "army" in the example.

- `SpellingResolver` — Takes a parse tree fragment containing nodes that represent spelled letters and returns the word they spell out. Spelling can be very useful when a parser or speech recognizer is used that requires an a priori vocabulary or grammar and cannot learn new words while running. In this way, out of vocabulary terms, such as names, can be spelled out. Spelling using a specialized alphabet, such as the NATO alphabet, is also very accurate. This resolver returns a single String, with a probability of 1.

- `DialogHistoryResolver` – Returns the first item in the dialog history for the slot. This is the most recently used value for the slot. Slots are identified by name, so this also looks for slots with the same name used in other frames.

- `CoordinateResolver` – Resolves a coordinate specification, such as "five hundred comma two fifty" to a Java Point object. One Point is returned, with a probability of 1.

- `NameResolver` – A trivial resolver that simply returns the name of the parse tree fragment's top level node. Some simple transformations can be done on the returned name, including stripping fixed leading and trailing strings, and applying a translation.

- **ContextProvider** – Classes that provide access to modalities implement the `ContextProvider` interface. The key method is `getData()`, which returns a `ContextData` instance that provides the modality's data along with a timestamp for which the data is valid. The context providers supplied by the framework are currently `EyeTracker`, `Mouse`, and `DialogHistory`.

- **ContextData** – `ContextData` implementations represent data from a modality. Three implementations are supplied with the framework: `PositionContextData`, `Entity`, and `ContextDataList`. `ContextDataList` is a container for other `ContextData` objects, and is returned by the `BufferedContextProvider`, which polls another context provider and caches its data for the duration of an utterance, so that multiple resolvers can use the same data through a single `BufferedContextProvider` instance. Most resolvers expect a `ContextDataList` as input.

- **LocatableObjectStore** – An `LocatableObjectStore` implementation plugs unto the framework's `ObjectLocator` class to enable it to query the application for on-screen objects. This is used in the `AnaphorResolver` to find objects in the neighborhood of the point on the screen the user indicated when speaking a pronoun. It is used by the `ObjectResolver` to find objects matching a set of attributes. The elements of the `LocatableObjectStore` are instances of a concrete subclass of `LocatableObject`. `LocatableObject` uses the "wrapper" design pattern to provide a uniform interface to access an object's location, size, and attributes.

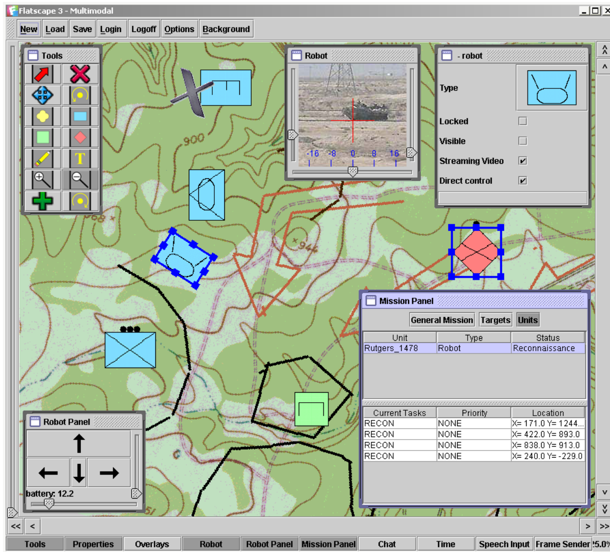

Figure 6: Flatscape

3.3 Dialog Management

Dialog management consists of maintaining dialog context and making application calls when frames are completed. The first task is implemented by having the dialog manager maintain a `DialogHistory` object, which implements the `ContextProvider` interface, so that it can provide data to resolving agents, notably the `DialogHistoryResolver`.

Application calls are made by the dialog manager through JavaScript. When the dialog manager receives a frame, it retrieves the associated script and executes it through the Bean Scripting Framework [10], a framework that provides scripting language support to Java applications. JavaScript provides a flexible way to interface between the dialog manager and the application. Since JavaScript is interpreted code, all that is needed to change the behavior of the multimodal interface is to modify the script and re-run the application; no code needs to be compiled.

4. CASE STUDY

4.1 Flatscape

Using the framework, a multimodal interface was created for Flatscape, our collaborative situation map application. With this application we can plan military missions by placing and moving icons representing military units on a map overlay. Additionally, we can track moving robotic vehicles on the same map and give them direct control commands or assign them higher level "missions". Camera feedback from the robots is available and can be viewed on screen. The camera can be rotated horizontally (panned) and vertically (tilted) and has a zoom function. Camera images are also used by the robot for target recognition.

The system runs on top of our DISCIPLE [3] collaborative middleware. Both Flatscape and the robots function as clients in this infrastructure, communicating over the collaboration bus (cBus) to exchange and synchronize state information. A change made by the user in Flatscape that is relevant to a robot will cause this robot to be notified of it, possibly making it perform some action, as the user indicated, such as moving to a different location.

The new multimodal interface allows the above things to be accomplished with multimodal commands:

- **Unit creation and manipulation**
  "create a friendly infantry squad *here*"
  "move *this* anti armor unit over *there*"
  "delete *that*"
  "move this to five fifty comma two hundred"
- **Robot control**
  "go forward fifty feet"
  "turn left"
  "back up slowly ... stop"
  "camera on ... look left ... zoom in"
- **Mission planning**
  "new recon mission"
  "home base is *here*"
  "new reconnaissance task *there*"
  "start mission"

Additionally, we are able to use the direct manipulation commands already present in the GUI alongside the multimodal commands. For each task, users can select whichever interaction paradigm is most suitable and use both interaction methods interchangably.

4.2 Evaluation

4.2.1 *Reusability*

Table 1 shows the implementation effort for this new multimodal interface expressed in kilobytes of code. As can be seen in the table, 92% of the code is framework code. This indicates that the framework is a good generalization for this type of application and that much effort is saved by using the framework.

Table 1: Implementation effort for the situation map tool

Application-specific code	**Size**
Grammar	28K
New Java code	46K
Fusion and Dialog Manager configuration (XML + Javascript)	33K
Natural language generation configuration	6K
Framework code	**Size**
Framework code	1152K
Parser code	148K
Code re-use	**92%**

4.2.2 *Response Times*

Figure 7 shows response times for five different types of speech acts, both speech-only and multimodal. Times were measured on a Pentium 4, 1.7 GHz with 512MB of RAM. As can be seen in the figure, speech recognizer delays can be significant; the reason for this is that present-day speech recognizers wait for a certain period of silence before assuming the user has finished speaking. These delays, due to their nature, can not be reduced by acquiring faster hardware.

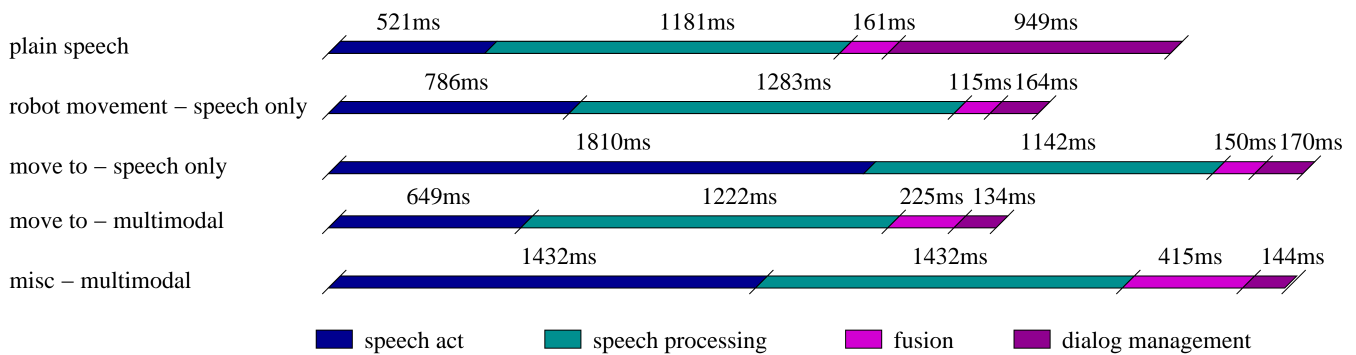

Figure 7: Response times for five types of speech acts

5. CONCLUSIONS AND FUTURE WORK

We have showed that the framework can be used to implement a multimodal interface with relatively little effort. The resulting interface has a reasonable response time, which can be improved by advances in speech recognition technology. The system currently uses mouse and gaze input, but other modalities can be added easily by creating new implementations of the `ContextProvider` interface. Direct manipulation and speech are used together to form a user interface that gives users the freedom to choose and combine interaction methods to create a more efficient and pleasant way of working.

One of the main thrusts for future work is implementing more complex direct manipulation examples and integrating them into the framework. The architecture presented in [12] gives a generic framework but there are also parts that are application-specific and will need to be implemented by the application developer.

We are investigating the use of fuzzy reasoning and Bayesian networks in fusion. Fuzzy values can be a better representation of uncertainty in multimodal systems than discrete probabilities and may enable a more natural way of configuring the fusion manager and resolving agents.

Bayesian belief networks trained with data captured from multimodal dialogs can be used to improve fusion. Statistical data on the way users use various modalities can be used to estimate a user's intent with a multimodal action.

Finally, adding a fission agent as proposed in the text would make a true multimodal system in which both user and system can communicate in a multimodal fashion, creating a more natural experience for the user.

6. ACKNOWLEDGMENTS

The research is supported by US Army CECOM Contract No. DAAB07-02-C-P301, a grant from New Jersey Commission on Science and Technology, and by the Center for Advanced Information Processing (CAIP) and its corporate affiliates.

7. REFERENCES

[1] R. A. Bolt. “Put-that-there”: Voice and gesture at the graphics interface. *Computer Graphics (SIGGRAPH ’80 Proceedings)*, 14(3):262–270, July 1980.

[2] L. Boves and E. den Os. Multimodal multilingual information services for small mobile terminals (MUST). Technical report, Eurescom, 2002. http://www.eurescom.de/~pub/deliverables/documents/P1100-series/P1104/p1104-d1.pdf.

[3] CAIP. DISCIPLE - mobile computing and collaboration. http://www.caip.rutgers.edu/disciple.

[4] P. Cohen, M. Johnston, D. McGee, S. Oviatt, J. Pittman, I. Smith, L. Chen, and J. Clow. Quickset: Multimodal interaction for distributed applications. *ACM International Multimedia Conference, New York: ACM*, pages 31–40, 1997.

[5] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[6] N. Krahnstoever, S. Kettebekov, M. Yeasin, and R. Sharma. A real-time framework for natural multimodal interaction with large screen displays. In *Proc. of Fourth Intl. Conference on Multimodal Interfaces (ICMI 2002), Pittsburgh, PA, USA*, October 2002.

[7] J. A. Larson and T. V. Raman. W3C multimodal interaction framework. http://www.w3.org/TR/mmi-framework, 2 December 2002. W3C Note.

[8] M. E. Markiewicz and C. J. Lucena. Object oriented framework development. *ACM Crossroads*, 2001.

[9] B. Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behaviour and Information Technology*, 1(3):237–256, 1982.

[10] The Apache Jakarta Project. Jakarta BSF – bean scripting framework. http://jakarta.apache.org/bsf.

[11] D. Toledano, S. Wang, S. Cyphers, and J. Glass. Extending the galaxy communicator architecture for multimodal interaction research. submitted to ACM Trans. on Human-Computer Interaction, Aug 2002.

[12] J. Vlissides and M. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, 1990.