MULTIPLATFORM Testbed: An Integration Platform for Multimodal Dialog Systems

Gerd Herzog, Heinz Kirchmann, Stefan Merten, Alassane Ndiaye, Peter Poller
German Research Center for Artificial Intelligence
Erwin-Schrödinger-Straße, D–67608 Kaiserslautern, Germany
`{herzog,kirchman,merten,ndiaye,poller}@dfki.de`

Abstract

Modern dialog and information systems are increasingly based on distributed component architectures to cope with all kinds of heterogeneity and to enable flexible re-use of existing software components. This contribution presents the MULTIPLATFORM testbed as a powerful framework for the development of integrated multimodal dialog systems. The paper provides a general overview of our approach and explicates its foundations. It describes advanced sample applications that have been realized using the integration platform and compares our approach to related works.

1 Motivation

One central element of research in the field of intelligent user interfaces is the construction of advanced natural language and multimodal dialog systems that demonstrate the high potential for more natural and much more powerful human-computer interaction. Although language technology already found its way into fundamental software products—as exemplified by the Microsoft Speech SDK (*software development kit*) for Windows and the Java Speech API (*application programming interface*)—the development of novel research prototypes still constitutes a demanding challenge. State-of-the-art dialog systems combine practical results from various research areas and tend to be rather complex software systems which can not simply be realized as a monolithic desktop computer application. More elaborate software designs are required in order to assemble heterogeneous components into an integrated and fully operational system (Klüter et al., 2000).

A typical research project involves several or even many work groups from different partners, leading to a broad spectrum of practices and preferences that govern the development of software components. In particular, a common software platform for the construction of an integrated dialog system often needs to support different programming languages and operating systems so that already existing software can be re-used and extended. Taking into account the potential costs it is usually not feasible to start an implementation from scratch. Another important aspect is the use of rapid prototyping for accelerated progress which leads to frequent changes in design and implementation as the project unfolds.

Over the last ten years we have been concerned with the realization of various complex distributed dialog systems. A practical result of our long-term work as a dedicated system integration group is the so-called MULTIPLATFORM testbed[1] (**Mu**ltiple **L**anguage **T**arget **I**ntegration **Plat**form **for M**odules) which provides a powerful and complete integration platform. In this contribution, we will report on the experience gained in the construction of integrated large-scale research prototypes. The results obtained so far will be presented and the underlying principles of our approach will be discussed.

2 Anatomy of the Testbed

The MULTIPLATFORM testbed in its diverse instantiations comprises the software infrastructure that is needed to integrate heterogeneous dialog components into a complete system. Built on top of open source software (Wu and Lin, 2001), the testbed SDK offers APIs as well as a large set of tools and utilities to support the whole development process, including installation and distribution. The following sections provide an overview of the testbed and describe the design principles that govern its realization.

2.1 Architecture Framework

A distributed system constitutes the natural choice to realize an open, flexible and scalable software architecture, able to integrate heterogeneous software modules implemented in diverse programming languages and running on different operating systems. In our project work, for example, we encountered modules for Sun Solaris, GNU Linux, and Microsoft Windows written in Prolog and Lisp, as classical AI languages, as well as in common conventional programming languages like C, C++, and Java.

The testbed framework is based on a component architecture (Hopkins, 2000) and our approach assumes a modularization of the dialog system into distinct and independent software modules to allow maximum decoupling. These large-grained components—ranging from more basic modules that encapsulate access to specific hardware devices to complex components which may include entire application-specific subsystems—constitute self-contained applications which are executed as separate processes, or even process groups. The principle behind this view is to consider software architecture on a higher-level of abstraction as modularization is not concerned with decomposition on the level of component libraries in a specific programming language. Continuous evolution is one of the driving forces behind the development of novel dialog systems. The creation of a componentized system makes the integrated system easier to maintain. In a well-designed system, the changes will be localized, and such changes can be made with little or no effect on the remaining components. Component integration and deployment are independent of the component development life cycle, and there is no need to recompile or relink the entire application when updating with a new implementation of a component.

The term middleware (Emmerich, 2000) denotes the specific software infrastructure that facilitates the interaction among distributed software modules, i.e. the software layer between the operating system—including the basic communication protocols—and the distributed components that interact via the network. The testbed as a component platform enables inter-process communication and provides means for configuring and deploying the individual parts of the complete dialog system.

Our middleware solution does not exclude to connect additional components during system execution. So far, however, the testbed does not offer specific support for dynamic system re-configuration at runtime. In our experience, it is acceptable and even beneficial to assume a stable, i.e. a static but configurable, architecture of the user interface components within a specific system instantiation. It is obvious that ad hoc activation and invocation of services constitutes an important issue in many application scenarios, in particular Internet-based applications. We propose to hide such dynamic aspects within the application-specific parts of the complete system so that they do not affect the basic configuration of the dialog system itself.

The details of the specific component architecture of different dialog systems vary significantly and an agreed-upon standard architecture which defines a definite modularization simply does not exist. Nevertheless, we found it helpful to use a well-defined naming scheme and distinguish the following categories of dialog system components when designing a concrete system architecture:

Recognizer: Modality-specific components that process input data on the signal level. Examples include speech recognition, determination of prosodic information, or gesture recognition.

Analyzer: Modules that further process recognized user input or intermediary results on a semantic level. Such components include in particular modality-specific analyzers and media fusion.

Modeller: Active knowledge sources that provide explicit models of relevant aspects of the dialog system, like for example discourse memory, lexicon, or a suitable model of the underlying application functionality.

Generator: Knowledge-based components which determine and control the reactions of the dialog system through the transformation of representation structures. This includes the planning of dialog contributions and application-centric activities as well as fission of multiple modalities and media-specific generators, e.g., for text and graphics.

Synthesizer: Media-specific realization components that transform generated structures into perceivable output. A typical example is a speech synthesis component.

Device: Connector modules that encapsulate access to a hardware component like, for example, microphone and sound card for audio input or a camera system that observes the user in order to identify facial expressions.

Service: Connector components that provide a well-defined link to some application-specific functionality. Service modules depend on the specific application scenario and often encapsulate complete and complex application-specific subsystems.

2.2 Inter-Process Communication

Nowadays, a very broad spectrum of practical technologies exists to realize communication between distributed software modules. Techniques like remote procedure call and remote method invocation, which follow the client-server paradigm, have long been the predominant abstraction for distributed processing. In this programming

model, each component has to specify and implement a specific API to make its encapsulated functionality transparently available for other system modules. Only recently, the need for scalability, flexibility, and decoupling in large-enterprise and Internet applications has resulted in a strong general trend toward asynchronous, message-based communication in middleware systems.

In accordance with the long-standing distinction being made in AI between procedural vs. declarative representations, we favor message-oriented middleware as it enables more declarative interfaces between the components of a dialog system. As illustrated by a hybrid technology like SOAP, the *simple object access protocol*, where remote calls of object methods are encoded in XML messages, the borderline between a procedural and a declarative approach is rather difficult to draw in general. Our own data-oriented interface specifications will be discussed in more detail in section 3.

For message-based communication, two main schemes can be distinguished:

- Basic *point-to-point* messaging employs unicast routing and realizes the notion of a direct connection between message sender and a known receiver. This is the typical interaction style used within multi-agent systems (Weiss, 2000).

- The more general publish/subscribe approach is based on multicast addressing. Instead of addressing one or several receivers directly, the sender publishes a notification on a named message queue, so that the message can be forwarded to a list of subscribers. This kind of distributed event notification makes the communication framework very flexible as it focuses on the data to be exchanged and it decouples data producers and data consumers. The well-known concept of a blackboard architecture, which has been developed in the field of AI (Erman et al., 1980), follows similar ideas.

Compared with point-to-point messaging, publish/subscribe can help to reduce the number and complexity of interfaces significantly (Klüter et al., 2000).

The MULTIPLATFORM testbed includes a message-oriented middleware. The implementation is based on PVM, which stands for *parallel virtual machine* (Geist et al., 1994). In order to provide publish/subscribe messaging on top of PVM, we have added another software layer called PCA (*pool communication architecture*). In the testbed context, the term *data pool* is used to refer to named message queues. Every single pool can be linked with a pool data format specification in order to define admissible message contents.

In the different dialog systems we designed so far, typical architecture patterns can be identified since the pool structure reflects our classification into different categories of dialog components. The pool names together with the module names define the backbone for the overall architecture of the dialog system.

The messaging system is able to transfer arbitrary data contents and provides excellent performance characteristics. To give a practical example, it is possible to perform a telephone conversation within a multimodal dialog system. Message throughput on standard PCs with Intel Pentium III 500 MHz CPU is off-hand sufficient to establish a reliable bi-directional audio connection, where uncompressed audio data are being transferred as XML messages in real-time. A typical multimodal user interaction of about 10 minutes duration can easily result in a message log that contains far more than 100 Megabytes of data.

The so-called *module manager* provides a thin API layer for module developers with language bindings for the programming languages that are used to implement specific dialog components. It includes the operations required to access the communication system and to realize an elementary component protocol needed for basic coordination of all participating distributed components.

2.3 Testbed Modules and Offline Tools

In addition to the functional components of the dialog system, the runtime environment includes also special testbed modules in support of system operation.

The *testbed manager* component, or TBM for short, is responsible for system initialization and activates all distributed components pertaining to a given dialog system configuration. It forms the counterpart for functional modules to carry out the elementary component protocol, which is needed for proper system start-up, controlled termination of processes and restart of single components, or a complete soft reset of the entire dialog system.

The freely configurable testbed GUI constitutes a separate component which provides a graphical user interface for the administration of a running system. In Figure 1 the specific testbed GUI of the SMARTKOM system (cf. Section 4.2) is shown as an example. The GUI basically provides means to monitor system activity, to interact with the testbed manager, and to manually modify configuration settings of individual components while testing the integrated system.

A further logging component is being employed to save a complete protocol of all exchanged messages for later inspection. Flexible replay of selected pool data provides a simple, yet elegant and powerful mechanism for the simulation of small or complex parts of the dialog system in order to test and debug components during the development process.

Another important development tool is a generic data viewer for the online and offline inspection of pool data.
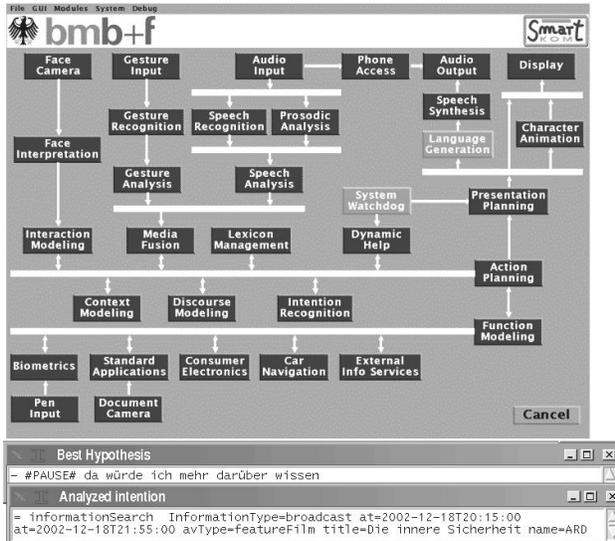
Figure 1: Testbed administration GUI and data viewer. Currently active components are highlighted using a different background color. The data viewer windows provide a compact display of selected pool data.

Further offline tools include a standardized build and installation procedure for components and utilities for the preparation of software distributions and incremental updates during system integration. Additional project-specific APIs and specifically adapted utilities are being developed and made avaliable as needed.

3 High-level Interfaces for Dialog System Components

Instead of using programming interfaces, the interaction between distributed components within the testbed framework is based on the exchange of structured data through messages. The communication platform is open to transfer arbitrary contents but careful design of information flow and accurate specification of content formats constitute essential elements of our approach.

Agent communication languages like KQML (Finin et al., 1994) and FIPA ACL (Pitt and Mamdani, 1999) are not a natural choice in our context. In general, large-scale dialog systems are a mixture of knowledge-based and conventional data-processing components. A further aspect relates to the pool architecture, which does not rely on unspecific point-to-point communication but on a clear modularization of data links. The specification of the content format for each pool defines the common *language* that dialog system components use to interoperate.

3.1 XML-based Data Interfaces

Over the last few years, the so-called *extensible markup language* has become the premier choice for the flexible definition of application-specific data formats for information exchange. XML technology, which is based on standardized specifications, progresses rapidly and offers an enormous spectrum of useful techniques and tools.

XML-based languages define an external notation for the representation of structured data and simplify the interchange of complex data between separate applications. All such languages share the basic XML syntax, which defines whether an arbitrary XML structure is well-formed, and they are build upon fundamental concepts like *elements* and *attributes*. A specific markup language needs to define the structure of the data by imposing constraints on the valid use of selected elements and attributes. This means that the language serves to encode semantic aspects of the data into syntactic restrictions.

Various approaches have been developed for the formal specification of XML-based languages. The most prominent formalism is called *document type definition*. A DTD basically defines for each allowed element all allowed attributes and possibly the acceptable attribute values as well as the nesting and occurrences of each element. The DTD approach, however, is more and more superseded by XML Schema. Compared with the older DTD mechanism, a schema definition (XSD) offers two main advantages: The schema itself is also specified in XML notation and the formalism is far more expressive as it enables more detailed restrictions on valid data structures. This includes in particular the description of element contents and not only the element structure. As a schema specification can provide a well-organized type structure it also helps to better document the details of the data format definition. A human friendly presentation of the communication interfaces is an important aid during system development.

It should be noted that the design of an XML language for the external representation of complex data constitutes a non-trivial task. Our experience is that design decisions have to be made carefully. For example, it is better to minimize the use of attributes. They are limited to unstructured data and may occur at most once within a single element. Preferring elements over attributes better supports the evolution of a specification since the content model of an element can easily be redefined to be structured and the maximum number of occurrences can simply be increased to more than one. A further principle for a well-designed XML language requires that the element structure reflects all details of the inherent structure of the represented data, i.e. textual content for an element should be restricted to well-defined elementary types. Another important guideline is to apply strict naming rules so that it becomes easier to grasp the intended meaning of specific XML structures.

From the point of view of component development, XML offers various techniques for the processing of

```
<intentionLattice>
[...]
  <hypothesisSequences>
    <hypothesisSequence>
      <score>
        <source>   acoustic   </source>
        <value>    0.96448    </value>
      </score>
      <score>
        <source>  understanding  </source>
        <value>    0.91667    </value>
      </score>
      <hypothesis>
        <discourseStatus>
          <discourseAction> set </discourseAction>
          <discourseTopic><goal> epg_info </goal></discourseTopic>
        [...]
        <event id="dim868">
          <informationSearch id="dim869">
            <pieceOfInformation>
              <broadcast id="dim863">
                <avMedium>
                  <avMedium id="dim866">
                    <avType> featureFilm </avType>
                    <title>  Die innere Sicherheit  </title>
                    [...]
    </hypothesisSequence>
[...]
  </hypothesisSequences>
</intentionLattice>
```

Confidence in the Speech Recognition Result

Confidence in the Speech Understanding Result

Planning Act

Object Reference
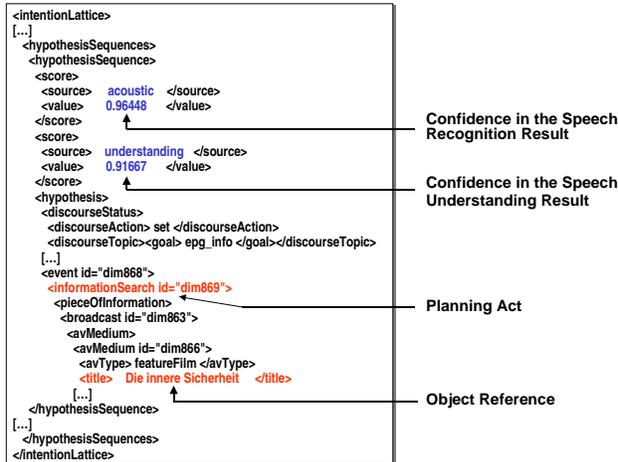
Figure 2: Partial M3L structure. The shown intention lattice represents the interpretation result for a multimodal user input that can be stated as: "I would like to know more about this [↗]."

transferred content structures. The DOM API makes the data available as a generic tree structure—the *document object model*—in terms of elements and attributes. Another interesting option is to employ XSLT stylesheets to flexibly transform between the external XML format used for communication and a given internal markup language of the specific component. The use of XSLT makes it easier to adapt a component to interface modifications and simplifies its re-use in another dialog system. Instead of working on basic XML structures like elements and attributes, XML data binding can be used for a direct mapping between program internal data structures and application-specific XML markup. In this approach, the language specification in form of a DTD or an XML Schema is exploited to automatically generate a corresponding object model in a given programming language.

3.2 Multimodal Markup Language

In the context of the SMARTKOM project (see section 4.2) we have developed M3L (**M**ulti**m**odal **M**arkup **L**anguage) as a complete XML language that covers all data interfaces within this complex multimodal dialog system. Instead of using several quite different XML languages for the various data pools, we aimed at an integrated and coherent language specification, which includes all sub-structures that may occur on the different pools. In order to make the specification process manageable and to provide a thematic organization, the M3L language definition has been decomposed into about 40 schema specifications.

Figure 2 shows an excerpt from a typical M3L expression. The basic data flow from user input to system output

continuously adds further processing results so that the representational structure will be refined step-by-step. Intentionally, M3L has not been devised as a generic knowledge representation language, which would require an inference engine in every single component so that the exchanged structures can be interpreted adequately. Instead, very specific element structures are used to convey meaning on the syntactic level. Obviously, not all relevant semantic aspects can be covered on the syntax level using a formalism like DTD or XSD. This means, that it is impossible to exclude all kinds of meaningless data from the language definition and the design of an interface specification will always be a sort of compromise.

Conceptual taxonomies provide the foundation for the representation of domain knowledge as it is required within a dialog system to enable a natural conversation in the given application scenario. In order to exchange instantiated knowledge structures between different system components they need to be encoded in M3L. Instead of relying on a manual reproduction of the underlying terminological knowledge within the M3L definition we decided to automate that task. Our tool OIL2XSD (Gurevych et al., 2003) transforms an ontology written in OIL (Fensel et al., 2001) into an M3L compatible XML Schema definition. The resulting schema specification captures the hierarchical structure and a significant part of the semantics of the ontology. For example in Figure 2, the representation of the event structure inside the intention lattice originates from the ontology. The main advantage of this approach is that the structural knowledge available on the semantic level is consistently mapped to the communication interfaces and M3L can easily be updated as the ontology evolves.

In addition to the language specification itself, a specific M3L API has been developed, which offers a lightweight programming interface to simplify the processing of such XML structures within the implementation of a component. Customized testbed utilities like tailored XSLT stylesheets for the generic data viewer as well as several other tools are provided for easier evaluation of M3L-based processing results.

4 Sample Applications

Our framework and the MULTIPLATFORM testbed have been employed to realize various natural language and multimodal dialog systems. In addition to the research prototypes mentioned here, MULTIPLATFORM has also been used as an integration platform for inhouse projects of industrial partners and for our own commercial projects.

The first incarnation of MULTIPLATFORM arose from the VERBMOBIL project where the initial system architecture, which relied on a multi-agent approach with point-to-point communication, did not prove to be scal-

able (Klüter et al., 2000). The testbed has been enhanced in the context of the SMARTKOM project and was recently adapted for the COMIC system. As described in the previous sections, the decisive improvement of the current MULTIPLATFORM testbed is, besides a more robust implementation, a generalized architecture framework for multimodal dialog systems and the use of XML-based data interfaces as examplified by the Multimodal Markup Language M3L.

4.1 VERBMOBIL

VERBMOBIL (Wahlster, 2000) is a speaker-independent and bidirectional speech-to-speech translation system that aims to provide users in mobile situations with simultaneous dialog interpretation services for restricted topics. The system handles dialogs in three business-oriented domains—including appointment scheduling, travel planning, and remote PC maintenance—and provides context-sensitive translations between three languages (German, English, Japanese).

VERBMOBIL follows a hybrid approach that incorporates both deep and shallow processing schemes. A peculiarity of the architecture is its multi-engine approach. Five concurrent translations engines, based on statistical translation, case-based translation, substring-based translation, dialog-act based translation, and semantic transfer, compete to provide complete or partial translation results. The final choice of the translation result is done by a statistical selection module on the basis of the confidence measures provided by the translation paths.

In addition to a stationary prototype for face-to-face dialogs, a another instance has been realized to offer translation services via telephone (Kirchmann et al., 2000).

The final VERBMOBIL demonstrator consists of about 70 distributed software components that work together to recognize spoken input, analyze and translate it, and finally utter the translation. These modules are embedded into an earlier version of the MULTIPLATFORM testbed using almost 200 data pools—replacing several thousand point-to-point connections—to interconnect the components.

4.2 SMARTKOM

SMARTKOM is a multimodal dialog system that combines speech, gesture, and facial expressions for both, user input and system output (Wahlster et al., 2001). The system aims to provide an anthropomorphic and affective user interface through its personification of an interface agent. The interaction metaphor is based on the so-called situated, delegation-oriented dialog paradigm. The basic idea is, that the user delegates a task to a virtual communication assistant which is visualized as a life-like character. The interface agent recognizes the user's intentions and goals, asks the user for feedback if necessary,
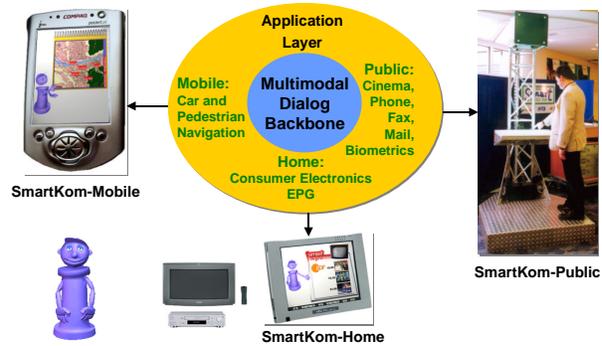
Figure 3: SMARTKOM kernel and application scenarios. Smartakus, the SMARTKOM life-like character is shown in the lower left corner.

accesses the various services on behalf of the user, and presents the results in an adequate manner.

The current version of the MULTIPLATFORM testbed, including M3L, is used as the integration platform for SMARTKOM. The overall system architecture includes about 40 different components. As shown in Figure 3, the SMARTKOM project addresses three different application scenarios.

SMARTKOM PUBLIC realizes an advanced multimodal information and communication kiosk for airports, train stations, or other public places. It supports users seeking for information concerning movie programs, offers reservation facilities, and provides personalized communication services using telephone, fax, or electronic mail.

SMARTKOM HOME serves as a multimodal portal to information services. Using a portable webpad, the user is able to utilize the system as an electronic program guide or to easily control consumer electronics devices like a TV set or a VCR. Similar to the kiosk application, the user may also use communication services at home. In the context of SMARTKOM HOME two different interaction modes are supported and the user is able to easily switch between them. In *lean-forward* mode coordinated speech and gesture input can be used for multimodal interaction with the system. *Lean-backward* mode instead is constrained to verbal communication.

SMARTKOM MOBILE uses a PDA as a front end, which can be added to a car navigation system or is carried by a pedestrian. This application scenario comprises services like integrated trip planning and incremental route guidance through a city via GPS and GSM, GPRS, or UMTS connectivity.
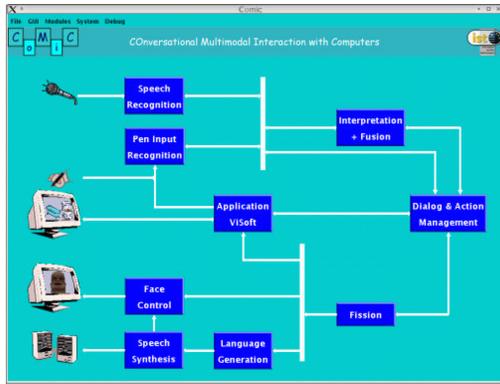
Figure 4: Adapted testbed GUI for the COMIC system.

4.3 COMIC

COMIC[2] (**C**onversational **M**ultimodal **I**nteraction with **C**omputers) is a recent research project that focuses on computer-based mechanisms of interaction in cooperative work. One specific sample application for COMIC is a design tool for bathrooms with an enhanced multimodal interface. The main goal of the experimental work is to show that advanced multimodal interaction can make such a tool usable for non-experts as well.

The realization of the integrated COMIC demonstrator is based on the MULTIPLATFORM testbed. Figure 4 displays the control interface of the multimodal dialog system. On the input side, speech and handwriting in combination with 3-dimensional pen-based gestures can be employed by the user. On the output side, a dynamic avatar with synthesized facial, head and eye movements is combined with task-related graphical and textual information. In addition to multiple input and output channels, there are components that combine the inputs—taking into account paralinguistic information like intonation and hesitations—and interpret them in the context of the dialog, plan the application-specific actions to be taken and finally split the output information over the available channels.

5 Related Work

GCSI, the **G**alaxy **C**ommunicator **s**oftware **i**nfrastructure (Seneff et al., 1999), is an open source architecture for the realization of natural language dialog systems. It can be described as a distributed, message-based, client-server architecture, which has been optimized for constructing spoken dialog systems. The key component in this framework is a central hub, which mediates the interaction among various servers that realize different dialog system components. The central hub does not only handle all communications among the server modules but is

also responsible to maintain the flow of control that determines the processing within the integrated dialog system. To achieve this, the hub is able to interpret scripts encoded in a special purpose, run-time executable programming language.

The GCSI architecture is fundamentally different from our approach. Within the MULTIPLATFORM testbed there exists no centralized controller component which could become a potential bottleneck for more complex dialog systems.

OAA, the **O**pen **A**gent **A**rchitecture (Martin et al., 1999), is a framework for integrating a community of heterogeneous software agents in a distributed environment. All communication and cooperation between the different is achieved via messages expressed in ICL, a logic-based declarative language capable of representing natural language expressions. Similar to the GCSI architecture, a sort of centralized processing unit is required to control the behavior of the integrated system. So-called facilitator agents reason about the agent interactions necessary for handling a given complex ICL expression, i.e. the facilitator coordinates the activities of agents for the purpose of achieving higher-level, complex problem-solving objectives. Sample applications built with the OAA framework also incorporated techniques to use multiple input modalities. The user can point, speak, draw, handwrite, or even use a standard graphical user interface in order to communicate with a collection of agents.

RAGS (Cahill et al., 2000) does not address the entire architecture of dialog systems and multimodal interaction. The RAGS approach, which stands for **R**eference **A**rchitecture for **G**eneration **S**ystems, focuses instead on natural language generation systems and aims to produce an architectural specification and model for the development of new applications in this area. RAGS is based on the well-known three-stage pipeline model for natural language generation which distinguishes between content determination, sentence planning, and linguistic realization. The main component of the RAGS architecture is a data model, in the form of a set of declarative linguistic representations which cover the various levels of representation that have to be taken into account within the generation process. XML-based notations for the data model can be used in order to exchange RAGS representations between distributed components. The reference architecture is open regarding the technical interconnection of the different components of a generation system. One specifically supported solution is the use of a single centralized data repository.

6 Conclusion

MULTIPLATFORM provides a practical framework for large-scale software integration that results from the re-

---

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

alization of various natural language and multimodal dialog systems. The MULTIPLATFORM testbed is based on an open component architecture which employs message-passing to interconnect distributed software modules. We propose to operationalize interface specifications in the form of an XML language as a viable approach to assemble knowledge-based as well as conventional components into an integrated dialog system. The testbed software is currently being refactored and we are planning to make it publicly available as open source software.

More than one hundred modules have already been used within the MULTIPLATFORM testbed. So far, however, these dialog system components are not freely available for public distribution. The availability of prefabricated modules as part of the testbed software would also enable third parties to develop complete dialog system applications through the reuse of provided standard components.

In addition to the software infrastructure, the practical organization of the project constitutes a key factor for the successful realization of an integrated multimodal dialog system. Stepwise improvement and implementation of the design of architecture details and interfaces necessitates an intensive discussion process that has to include all participants who are involved in the realization of system components in order to reach a common understanding of the intended system behavior. Independent integration experts that focus on the overall dialog system have proven to be helpful for the coordination of this kind of activities.

References

Lynne Cahill, Christy Doran, Roger Evans, Rodger Kibble, Chris Mellish, Daniel Paiva, Mike Reape, Donia Scott, and Neil Tipper. 2000. Enabling Resource Sharing in Language Generation: An Abstract Reference Architecture. In *Proc. of the 2nd Int. Conf. on Language Resources and Evaluation*, Athens, Greece.

Wolfgang Emmerich. 2000. Software Engineering and Middleware: A Roadmap. In *Proc. of the Conf. on the Future of Software Engineering*, pages 117–129. ACM Press.

Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. 1980. The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty. *ACM Computing Surveys*, 12(2):213–253.

Dieter Fensel, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, and Peter F. Patel-Schneider. 2001. OIL: An Ontology Infrastructure for the Semantic Web. *IEEE Intelligent Systems*, 16(2):38–45.

Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. 1994. KQML as an Agent Communication Language. In *Proc. of the 3rd Int. Conf. on Information and Knowledge Management*, pages 456–463. ACM Press.

Al Geist, Adam Beguelin, Jack Dongorra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderman. 1994. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press.

Iryna Gurevych, Stefan Merten, and Robert Porzel. 2003. Automatic creation of interface specifications from ontologies. In *Proc. of the HLT-NAACL'03 Workshop on the Software Engineering and Architecture of Language Technology Systems (SEALTS)*, Edmonton, Canada.

Jon Hopkins. 2000. Component Primer. *Communications of the ACM*, 43(10):27–30.

Heinz Kirchmann, Alassane Ndiaye, and Andreas Klüter. 2000. From a Stationary Prototype to Telephone Translation Services. In Wahlster (Wahlster, 2000), pages 659–669.

Andreas Klüter, Alassane Ndiaye, and Heinz Kirchmann. 2000. Verbmobil From a Software Engineering Point of View: System Design and Software Integration. In Wahlster (Wahlster, 2000), pages 635–658.

David L. Martin, Adam J. Cheyer, and Douglas B. Moran. 1999. The Open Agent Architecture: A Framework for Building Distributed Software Systems. *Applied Artificial Intelligence*, 13(1–2):91–128.

Jeremy Pitt and Abe Mamdani. 1999. Some Remarks on the Semantics of FIPA's Agent Communication Language. *Autonomous Agents and Multi-Agent Systems*, 2(4):333–356.

Stephanie Seneff, Raymond Lau, and Joseph Polifroni. 1999. Organization, Communication, and Control in the Galaxy-II Conversational System. In *Proc. of Eurospeech'99*, pages 1271–1274, Budapest, Hungary.

Wolfgang Wahlster, Norbert Reithinger, and Anselm Blocher. 2001. SmartKom: Multimodal Communication with a Life-Like Character. In *Proc. of Eurospeech'01*, pages 1547–1550, Aalborg, Denmark.

Wolfgang Wahlster, editor. 2000. *Verbmobil: Foundations of Speech-to-Speech Translation*. Springer, Berlin.

Gerhard Weiss, editor. 2000. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press.

Ming-Wei Wu and Ying-Dar Lin. 2001. Open Source Software Development: An Overview. *Computer*, 34(6):33–38.