

ICE: A Communication Environment for Natural Language Processing

Jan W. Amtrup*

University of Hamburg, Computer Science Department,
Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany
email: amtrup@informatik.uni-hamburg.de

Abstract *Modern AI systems need theoretically sound, easy to use, communication models in order to be able to explore distributed computing and agent-oriented operation. ICE (Intarc Communication Environment) represents such a system. It grounds on the theoretical framework of CSP (Communicating Sequential Processes) and implements channels as bidirectional, asynchronous data streams that can be configured in various ways. On top of PVM (Parallel Virtual Machine), a de-facto standard of message passing systems, software layers have been built that implement the channel operation modes together with interfaces for programming languages most often used in the AI community. A separate layer supports the use of complex data types, as they often arise in speech processing. We describe the design of ICE with a focus on configuration and synchronization during the creation of channels.*

Keywords: Parallel/Distributed applications; Computer natural language processing; Heterogeneous software systems; Software tools

1 Introduction

Modern AI systems are almost always constructed with distributed computing in mind. Often, an agent-oriented design is pursued. For example, the system built by the Verbmobil project [1], a German joint research

project for machine interpreting, consists of a runtime configuration with approximately 100 Unix processes.

Such systems can only be successfully built if a sound, easy to use, yet effective communication model is provided to support information transport between system modules. The communication infrastructure should be high-level enough to allow researchers to concentrate on their modules and not on communication issues, but on the other hand there should be enough configuration options to tailor the behavior of the interconnection pattern.

In this paper, we present ICE (Intarc Communication Environment) [2], an implementation of a channel-oriented, multi-architecture, multi-language communication model for large AI systems. It has been developed to be used primarily for architectural experiments within Verbmobil [3] and has been adapted for the main Verbmobil research prototype [4], as well.

ICE follows the theoretical direction of the channel model for interaction between software modules. We adopted the CSP approach [5], starting from its actual realization in the computer hardware [6]. The functional model of channels was slightly modified due to the needs that became apparent with experiences from early research prototypes.

The base communication software was not implemented from scratch (e.g. starting from Unix sockets). Instead, we decided to use PVM, the *Parallel Virtual Machine* [7], a de-facto standard process communication soft-

*This research was funded by the Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the VERBMOBIL Project under Grant 01 IV 101 A/O.

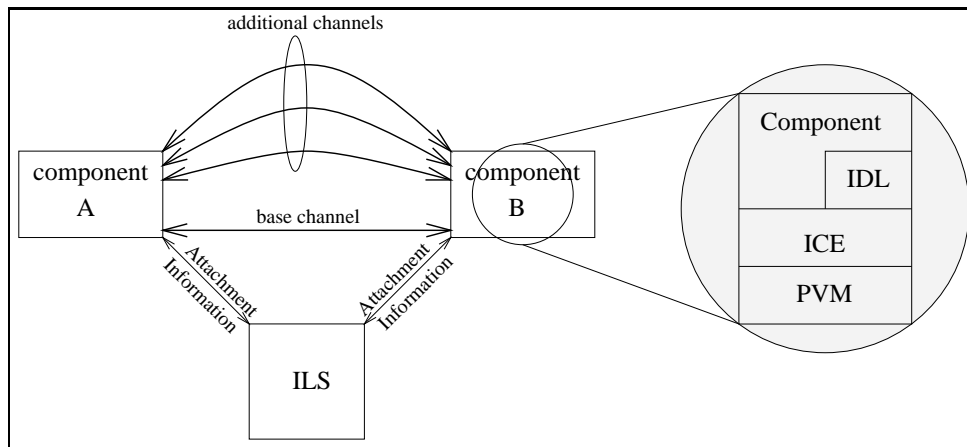


Figure 1: Principle component layout

ware, which proved extremely reliable.

The next section will describe the general architecture of a system using ICE as communication device. The internal structure of a typical component is outlined. Section 3 elaborates on the design and implementation of the channel model that governs the transportation of messages. The dedicated component acting as name and configuration server (ILS) will be described in section 4. Synchronization on startup will be explained in the following section. Finally, section 6 gives some information about applications built using ICE.

2 Application Architecture

ICE uses a channel abstraction to model the communication between components of a distributed system. The theoretical root of this approach lies with CSP. An implementation of that model can be found in the transputer hardware and the Occam programming language [8]. Here, channels are bidirectional data paths between Occam processes along which messages are exchanged. We estimated the remaining two widely available types of process communication, shared memory and remote procedure calls, as disadvantageous for our purposes. Using shared memory entails the risk of experiencing memory or bus contention when too many processors are attached to identical parts of memory. Additionally, write

operations have to be synchronized. RPC, on the other hand, uses a rendez-vous synchronization method, which may slow down a system due to network latencies¹.

Furthermore, the establishment of channels as explicit objects enables one to design numerous manipulations on them. For example, the splitting of channels (cf. section 3.1) has been introduced into ICE after the first release was published; this turned out to be not too much effort.

The overall architecture of a system using ICE is shown in fig. 1. An application may consist of any number of components realized in several different programming languages. Communication is performed through channels between the components, which come in two flavors for standard and bulk data exchange (cf. section 3). There is no central message handler, data exchange between modules is strictly bilateral, though it may involve some routing done by the pvm daemons.

A special component (ILS, the *Intarc License Server*) operates as name server for the application and stores configuration information. We designed the communication between individual components and the ILS to require as little synchronization as possible.

The software architecture of ICE uses a

¹The channels of CSP and Occam both use rendez-vous-synchronization. In this respect we deviated from the original model.

three-layered approach, PVM being the lowest level to carry out actual communication requests. The middle layer, made up from the core ICE routines, provides functions for sending, receiving and housekeeping. Furthermore, the middle layer contains interface functions for various programming languages. ICE currently supports C, C++, Fortran, Common Lisp (Allegro, Lucid, CLISP, Harlequin), Prolog (Sicstus, Quintus) and Tcl/Tk. The top-most, optional, layer supports the easy transmission of complex data types. Arbitrary data types can be transparently integrated into ICE by providing functions to encode and decode them. We provide compilers that receive an abstract data type description as input and generate the appropriate code needed for the different programming languages. As this part of the project is not yet completed (the compilers for C and C++ are already present), for the time being a user has to code the necessary functions by hand.²

3 Channel models

ICE distinguishes between two types of channels:

- Base channels work as standard, bidirectional, asynchronous, XDR-encoded channels in order to allow all components within the application to communicate with each other.
- Additional channels may be configured to provide synchronous message passing or bypass XDR encoding to speed up message delivery. They were added to fulfill some needs that frequently arise in the design of large AI systems. For example, they may be used to speed up the exchange of large data packages (e.g. speech samples) or to separate data and control streams between sophisticated modules.

²This has been done for the data types used by Verbmobil (e.g. speech signal data, word hypotheses, and semantic descriptions). The amount of work for coding is normally low.

Both types of channels are capable of transporting various scalar data types as well as complex, user-defined data types. This resembles structured messages that can be transported along Occam channels (cf. [8]).

3.1 Split channels

To allow for further flexibility in system design, both base and additional channels can be configured in a way modifying the topology of the application. Channels may be split up. That way, they do not only work as data pipelines between exactly two components, but other components may listen to traffic on the channel, as well. It is even possible for a component to inject data into the channel. All configuration is done in a transparent way, guaranteeing that the behavior of the modules does not change if the configuration is altered. Splitting of channels has been used to a large extent in Verbmobil:

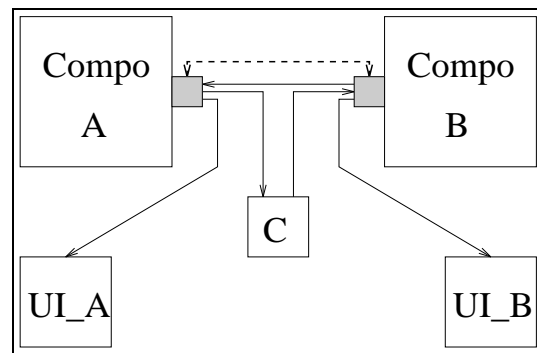


Figure 2: Split channel configuration.

- User interface components have been attached to channels to allow visualization of partial results during system operation. The visualization was simply added as additional listener on a channel and presented the data to the user.
- Interface modification components were inserted between two components. They were designed to cope with changing interface definitions during the development of the system. That way, interface specification modifications could be easily masked,

if two different versions of connected components had to be used.

Consider fig. 2 as an example for the configuration of split channels. Two components, **A** and **B**, are connected using a channel which is depicted by a dashed line. The channel endpoints are split up to allow visualization of message data sent by either component. The visualization is performed by two additional components labeled **UIA** and **UIB**. Furthermore, the data sent by component **A** must undergo some modification while being transported to component **B**. Thus, another component **C** is configured capable of transforming the data. It is spliced into the data path between **A** and **B**. Note that data sent by component **B** arrives at **A** unaffected from modification by component **C**.

4 Information service

The components of an application using ICE are distinctly named. In order to provide a name service that maps from component names to PVM task ids, we introduced a special component that works as information server.³

The ILS (*Intarc License Server*) stores information about the actual structure of the application. This includes names and locations of all components in the system. As a component wants to participate in the application, it sends an attachment message to the ILS. The ILS stores the registration and notifies other components of the presence. Likewise, a component detaches itself if it ceases to take part in the application. The detachment can also be handled autonomously by the ILS if a component crashes.

The second main function the ILS performs is the management of the channel configuration. It reads a configuration file on startup that describes the layout of split channels and

³This could have been achieved by using the group mechanism of PVM. As we needed further information, e.g. for channel configuration, we decided not to use that option.

issues messages to all involved components if a split channel is created. This ensures that a component, even if it does not issue a channel creation request, is able to receive messages.

A recently added feature is the possibility to broadcast messages to all components participating in an application. In order to perform this task, a component must know the identity of all other components. We restrict broadcasting to base channels. It consists of three stages:

- When a component wants to issue a broadcast message for the first time, it sends a request to the ILS in order to be informed about all other components it may not yet know. The ILS checks its configuration record and relates the necessary information to the requester.
- After receiving all configuration data, the component delivers the messages along all base channels it is aware of — or rather, of which the ICE layer in that component is aware. The application code does not have to know this.
- For the future, the ILS assumes that the component requesting a broadcast configuration is likely to issue further broadcasts. Consequently, upon attachment of a new component, a base channel is established between the new and the broadcasting component, regardless of whether it was preconfigured or not. That way, no further synchronization between the broadcasting component and the ILS is needed.

4.1 Graphical control panel

The setup of ICE requires at least three processes to be run. In order to make this task easier and to be able to supervise the configuration of an ICE application, we implemented a graphical control panel which is shown in fig. 3. Besides setting up and shutting down an ICE application, it is possible to use XPVM as graphical back-end. This is quite useful for

debugging purposes. Furthermore, the information the ILS provides about its actions are displayed in a window for supervision.

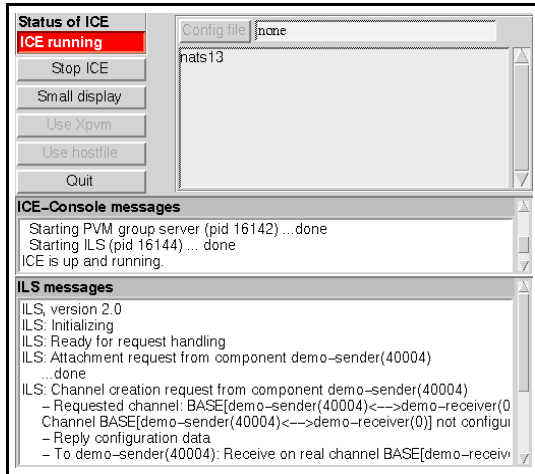


Figure 3: The ICE console.

5 Synchronization

The design of ICE included the goal to handle attachment of components and creation of channels as flexible as possible. A single Unix process may consist of any number of ICE components and may assign and delete channels at any given time. These principles required to restrict synchronization wherever possible. Only if a communication between two components actually happens, both partners should be aware of each other. We implemented an information profile that can handle under-specified component and channel data as long as it is not absolutely necessary to know about communication partners.

Figure 4 shows the sequence of messages exchanged between individual components and the ILS during attachment and channel configuration. First, component **A** calls `Ice_Attach()` to become part of the application. This is achieved by sending a message with a message tag of `ILS_ADD` to the ILS. **A** does not need to wait for an acknowledgment and continues processing. Next, **A** introduces a channel by calling `Ice_AddChan()`. A message (`ILS_CHC`) is sent to the ILS con-

taining the channel creation request. The ILS answers by sending several configuration messages back to **A**: Messages with tag `ILS_CHS` describe real channels to send on⁴. In this case, we assume two real channels to be used, thus two such messages are sent. There is only one real channel used for receiving, so one message with tag `ILS_CHR` suffices. To indicate the end of configuration, a final message with tag `ILS_CHE` is sent to the requesting component **A**.

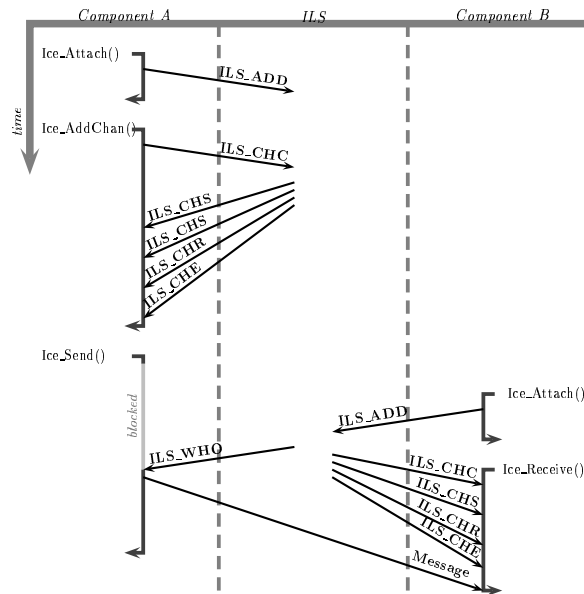


Figure 4: Time line of channel configuration with the ILS

After having completed the configuration of the desired channel, a message is sent on that channel by calling `Ice_Send()`. Since a target component (**B**) did not yet attach to the application, the call is instantly blocked. After a while, component **B**, the assumed target component for that message, attaches via `Ice_Attach`. As soon as the ILS receives the `ILS_ADD`-request, it checks whether there are components that should know the identity of **B**. It sends a message with tag `ILS_WHO` to component **A** to inform it about component **B**. Then the ILS initiates the configuration of the

⁴We use the term “real channel” to denote the data paths a message is actually delivered on. Due to splitting of channels, there may be more than one real channel per logical channel.

channel within component **B** by sending the appropriate messages (**ILS_CHC**, **ILS_CHS**, **ILS_CHR** and **ILS_CHE**).

Meanwhile, component **A**, informed about the location of **B** can actually send the message and returns from the `Ice_Send()` call. Component **B** receives the message, finally.

In essence, **A** does not need to know about **B** until it wants to send a message. Only at that time effective synchronization is needed and **A** has to wait for **B** to come up. In order to debug communication code and to avoid deadlocks, we installed a dynamically allocatable configuration option which simply returns an error code in case one of the receiving components is not present during a send request⁵.

6 Applications

ICE is currently used in a number of speech and virtual reality related applications. The largest so far is Verbmobil [1], the German joint project for machine interpretation. It aims at face-to-face dialogue interpreting in appointment scheduling situations. The research prototype consists of over 30 components, partly responsible for linguistic analysis, ranging from speech recognition, syntax and semantics to speech synthesis. Some components perform administrative tasks, such as overall system control and graphical user interfaces. If all components are activated, the runtime system requires 95 processes and 520 MB main memory. A total number of 76 channels is defined.

Viena [9] is an example of an application that is not restricted to speech processing. Here, multi-modal means of input (esp. gestures) are used to support intelligent communication with a technical system for the design and exploration of a virtual interior design environment.

These two examples may show that ICE represents a communication system which is neu-

⁵Due to the configuration of split channels, there may be any number of receiving components on a channel. We guarantee that each message is sent to either all or none of the components acting as receiver depending on whether all components are present or not.

tral w.r.t. the kind of application. Nevertheless, it was designed with speech processing in mind, where it is mainly used. In the next two sections, we are going to describe two applications in more detail.

6.1 INTARC

INTARC (*INTeractive ARChitecture*) is an incremental interactive speech interpreting system. It was implemented as part of the Verbmobil project for the purpose of investigation into innovative architectures for speech-language-systems. The approach was to strictly obey design principles originating in some of the assumed properties of human speech understanding. The main design rules were:

- **Incrementality.** Just as humans start to understand utterances right from their beginning, a system processing speech input should not wait until an utterance is completed before starting linguistic analysis. Incrementality enables exploration of parallelism on the inter-modular level, allowing several components of a system to run concurrently. Furthermore, some applications, such as simultaneous interpreting, can not be achieved in conventional ways.
- **Interactivity.** Operating incrementally allows the establishment of feedback-loops, which may be used to restrict the operation of individual components based on evidence from components that reside at a later stage of processing. Additionally, interactivity may lead to the generation of predictions that may steer the search behavior of certain modules.

These two principles have serious consequences for the design and implementation of components. Consider the interface between a word recognizer and a syntactic parser. The word recognizer delivers a *word graph* for each utterance, a suitable means to represent a large number of different utterance hypotheses in a compact manner [10]. Edges of the graph denote word hypotheses, vertices usually points

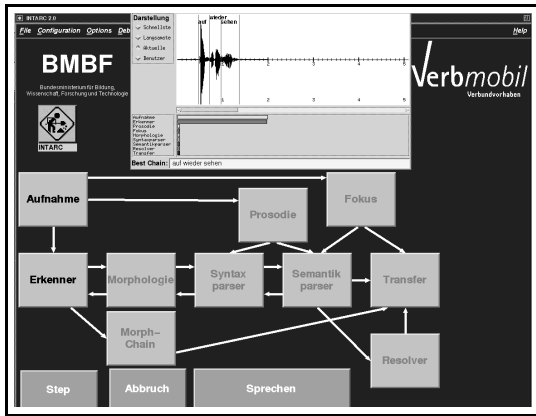


Figure 5: The application surface of INTARC 2.0.

in time. As the recognizer works incrementally, dead ends in the graph can not be suppressed: It is not known in advance if a prefix path through the graph may be further extended. Due to this property, the search space for the adjacent parser grows enormously: We noted that incremental word graphs contain as much as ten times more word hypotheses that non-incremental ones (resulting in more than 400 word hypotheses per second of speech signal given a vocabulary size of approximately 3000 word forms).

The architecture of INTARC is shown in fig. 5. Several modules are connected by a data path according to “standard” linguistic hierarchy (Recognition, Syntax, Semantics, Transfer, Synthesis). Additionally, there are a number of interaction paths that are used to explore different types of knowledge that may add to overall system performance (e.g. prosody).

At first glance, using an incremental approach only increases the complexity of the task at hand, due to larger input and the initial lack of global optimization over the input. Parallelism on the intermodular level reduces the impact of the overhead, while interactivity decreases the size of search spaces in a way that can not be pursued without incrementality.

Interactivity was explored between word recognition and syntax, for example (cf. [11]). The goal was to restrict the search space the recognizer has to traverse, on the basis of in-

formation gathered in the parser. To achieve this, we created a tight interaction schema between these two components, synchronizing at every 10ms interval of the speech signal (at each *frame*). For each frame, a set of word hypotheses is sent from the recognizer to the parser. The parser tries to incorporate the new input into the partial analyses already present. The feedback basically consists of a boolean decision about the applicability of words: If one could not be integrated at all, it is rejected. The word recognizer can adapt its behavior by cutting off subparts of the search space depending on that particular word.

By modifying this interaction pattern, a predictive mode of word recognition can be established. Again, every frame represents a synchronization point. The parser generates a set of words or syntactical categories that may be integrated into already existing partial analyses and transmits this set backwards to the word recognizer. This module is then able to restrict word recognition only to words that may be reasonable in a syntactic way. By this means, only the interesting part of the complete search space is constructed, which may reduce computing effort.

6.2 Layered Charts

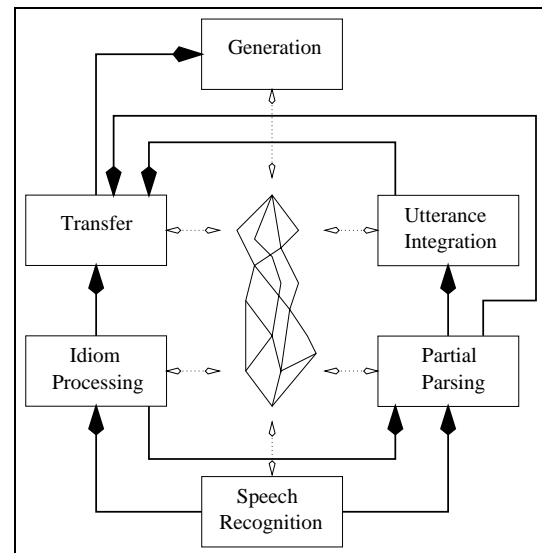


Figure 6: The architecture of MILC

A system currently under development [12] studies the impact of using a uniform data structure to represent linguistic information in a distributed speech interpreting model. The architecture of the system is shown in fig. 6. Central to the operation is the notion of a *layered chart* which is used to store linguistic data on all levels of interpretation. The chart we are using is a graph structure containing partial analyses that span certain intervals of a user's utterance. The starting point for this kind of data structure is given by the word graph being used as input. During the course of processing, additional edges are added which may span several word hypotheses and denote syntactical or other descriptions of the incorporated words.

One of the key features of the system is information separation and hiding. Data only pertaining to a certain level of processing remains in the module operating on that level. Analyses relevant for other components of the system are delivered along the channels outlined in fig. 6. Nevertheless, the union of all information present in every module forms a sound chart, thus we can speak of a distributed representation of that structure.

Mainly two types of information are passed between components:

- Graph edges. An edge of a layered chart represents a partial analysis of a sub-path within the underlying wordgraph. We use a typed feature formalism to describe linguistic contents, which was designed to be suitable for distributed systems, i.e. the objects of the formalism can be transferred without a heavy overhead for encoding and decoding. In essence, they are stored as relocatable arrays of feature-value nodes, which need no encoding at all to be interpreted in a different address space.
- Inhibition messages. They are used to reduce the probability of edges covered in a different component. The main application of this modification is to account for different depths of linguistic analysis. For

example, we integrated a component dedicated to the search for idiomatic expressions within an utterance. Once found, we assume that the word hypotheses contained by an idiom are unlikely to be part of any other syntactic description. To prevent them from being used elsewhere, we reduce their probability (and the probability of all their descendants) by a certain amount.

7 Conclusion

We have presented ICE, a channel-oriented model for communication in large AI systems, especially suited for speech processing applications. The theoretical orientation is given by the channel model of CSP (and its implementation in the transputer hardware), the software basis is provided by PVM.

ICE consists of three software layers: PVM, a layer of core routines and programming language interfaces, and a layer for the encoding of complex data types.

Channels work as bidirectional data streams, which can be configured in a variety of ways. The communication properties of channels can be altered as well as the topology of connections given by channels. It is possible to split channels in order to easily implement visualization components or interface managers. Using ICE, AI components implemented in a number of common programming languages can be integrated into a single application.

ICE has been successfully used for Verbmobil, a very large speech-to-speech translation project and its various architectural prototypes. Furthermore, it is used in a number of additional projects, e.g. for applications of virtual reality.

References

- [1] Wolfgang Wahlster. Translation of Face-to-Face-Dialogs. In *Proc. MT Summit IV*, pages 127–135, Kobe, Japan, 1993.

- [2] Jan W. Amtrup. ICE–Intarc Communication Environment: User’s Guide and Reference Manual. Version 1.4. Verbmobil Technical Document 14, Univ. of Hamburg, December 1995.
- [3] Walther v. Hahn and Jan W. Amtrup. Speech-to-Speech Translation: The Project Verbmobil. In *Proceedings of SPECOM 96*, pages 51–56, St. Petersburg, October 1996.
- [4] Jan W. Amtrup and Jörg Benra. Communication in large distributed AI Systems for Natural Language Processing. In *Proceedings of the 16th international Conference on Computational Linguistics*, pages 35–40, Copenhagen, Denmark, August 1996. Center for Sprogteknologi.
- [5] Charles A. Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.
- [6] Ian Graham and Tim King. *The Transputer Handbook*. Prentice Hall, New York, London et al., 1990.
- [7] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine. A Users’ Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, 1994.
- [8] Alan Burns. *Programming in OCCAM 2*. Addison-Wesley, Reading, Ma., 1988.
- [9] I. Wachsmuth and Y. Cao. Interactive Graphics Design with situated Agents. In W. Strasser and F. Wahl, editors, *Graphics and Robotics*, pages 73–85. Springer, 1995.
- [10] Martin Oerder and Hermann Ney. Word Graphs: An Efficient Interface Between Continuous-Speech Recognition and Language Understanding. In *ICASSP93*, 1993.
- [11] Andreas Hauenstein and Hans Weber. An Investigation of Tightly Coupled Speech Language Interfaces Using an Unification Grammar. In *Proceedings of the Workshop on Integration of Natural Language and Speech Processing at AAAI ’94*, pages 42–50, Seattle, WA, 1994.
- [12] Jan W. Amtrup. Layered Charts for Speech Translation. In *Proc. of the 7th Int. Conf. on Theoretical and Methodological Issues in Machine Translation*, Santa Fe, NM, July 1997.