Dialogue and Discourse 2(1) (2011) 83–111 doi: 10.5087/dad.2011.105

A General, Abstract Model of Incremental Dialogue Processing

David Schlangen DAVID.SCHLANGEN@UNI-BIELEFELD.DE
Faculty of Linguistics and Literature
Bielefeld University
Bielefeld, Germany

Gabriel Skantze GABRIEL@SPEECH.KTH.SE
Department of Speech, Music and Hearing
KTH
Stockholm, Sweden

Editor: Hannes Rieser

Abstract

We present a general model and conceptual framework for specifying architectures for incremental processing in dialogue systems, in particular with respect to the topology of the network of modules that make up the system, the way information flows through this network, how information increments are 'packaged', and how these increments are processed by the modules. This model enables the precise specification of incremental systems and hence facilitates detailed comparisons between systems, as well as giving guidance on designing new systems. In particular, the model can serve as a framework for specifying module communication in such systems, as we illustrate with some examples.

Keywords: Incremental Processing, Dialogue Systems, Software Architecture

1. Introduction

Dialogue processing is, by its very nature, *incremental*. No dialogue agent (artificial or natural) processes whole dialogues, if only for the simple reason that dialogues are *created* incrementally, by participants taking turns at speaking (or typing, in computer-mediated, chat-like interaction). At this level, most current implemented dialogue systems are incremental: they process user utterances as a whole and produce their response utterances as a whole. Incremental processing, as the term is commonly used, means more than this, however, namely that processing starts before the input is complete (e.g., (Kilger and Finkler, 1995)). Incremental systems hence are those where "[e]ach processing component will be triggered into activity by a minimal amount of its characteristic input" (Levelt, 1989). If we assume that the characteristic input of a dialogue system is the utterance (see (Traum and Heeman, 1997) for an attempt to define this unit), we would expect an incremental system to work on units smaller than utterances. Doing this then brings into the reach of computational modelling a whole range of behaviours that cannot otherwise be captured, like concurrent feedback ("uh-huh", "yeah"), fast turn-taking, and collaborative utterance construction.

Our aim in the work presented here is to describe and give names to the options available to designers of incremental systems. The model that we specify is *general* in the sense that it describes elements that, as we believe, are essential to incremental processing, and hence can be expected to

©2011 David Schlangen and Gabriel Skantze Submitted 1/2010; Accepted 3/2011; Published online 5/2011

play a role in most, if not even all, systems performing such processing, and *abstract* in the sense that it only describes properties of these elements and relations between them, but not concrete ways to instantiate them in computer implementations.[1] Specifically, we define some abstract data types, some abstract methods that are applicable to them, and a range of possible constraints on processing modules. The notions introduced here allow the (abstract) specification of a wide range of different systems, from non-incremental pipelines to fully incremental, asynchronous, parallel, predictive systems, thus making it possible to be explicit about similarities and differences between systems. We believe that this will be of great use in the future development of such systems, in that it makes clear the choices and trade-offs one can make.

While we discuss some issues that arise when directly basing a module communication infrastructure for incremental systems on the model, our main focus here is on the conceptual framework, not on implementation details. What we are also *not* doing here is to argue for one particular 'best architecture'—what this is depends on the particular aims of an implementation/model and on more low-level technical considerations (e.g., availability of processing modules). As we are not trying to *prove* properties of the specified systems here, the formalisations we give are not supported by a formal semantics.

In the next section, we first motivate the use of incremental processing in dialogue systems, and then give some examples of the possible differences in system architectures that we want to capture. In Section 3, we present the abstract model that underlies the system and module specifications, of which we give some examples in Section 4.[2] In Section 5, we discuss how the model can be used when designing the communication infrastructure of new incremental dialogue systems. We close with a brief discussion of related work.

2. Motivation

2.1 Why Model Incremental Processing?

Before we go into the details of our model, we take a step back and discuss possible reasons for using incremental processing in models of dialogue processing. This discussion will be brief, however, as our main interest in this paper is not to convince anyone of choosing this style of processing, but is rather to describe some of the options that are available once that choice has been made.[3]

Depending on the goals one has when building a system, incremental processing may offer certain advantages over non-incremental processing (i.e., processing where only full utterances, or even longer units like turns, are considered by the system). For example, if one tries to optimize the *reactivity* of the system, then incrementality may be attractive for the straightforward engineering reason that starting to process an input before it is complete can lead to the processing being finished faster compared to starting processing only after the input is complete. This can be true even if the incremental processing takes somewhat longer than the non-incremental processing, if that difference is not too high, as illustrated in Figure 1. (Of course, the algorithms used in incre-

1. We do not claim, of course, that the list of issues discussed here is exhaustive. Moreover, even within the space that we sketch here, there are many detail questions that will still need much work to be answered fully.
2. These sections are based on Schlangen and Skantze (2009), but are revised and significantly extended.
3. See e.g. Allen et al. (2001); Aist et al. (2007); Skantze and Schlangen (2009); Buß et al. (2010) for more comprehensive discussions and some empirical results regarding the use of incremental processing in applied dialogue systems. Regarding models of human dialogue processing, there is an even longer thread of literature in Psycholinguistics, amassing evidence that the human language processor works incrementally; see eg. Marslen-Wilson (1973); Altmann and Steedman (1988); Tanenhaus et al. (1995); van Berkum et al. (2007).

mental processing should not have a systematically worse complexity than their non-incremental counterparts.)

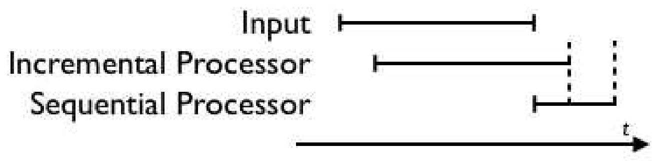


Figure 1: Input processed by Incremental and Sequential Processor. Even a much slower Incremental Processor can finish before a Sequential Processor.

In a similar vein, incremental processing can potentially promise better *quality* of processing, if the system has provisions for interactions between modules, where the results of higher-level processing of earlier bits of input can influence lower-level processing of later bits of input. (E.g., if the syntactic structure computed for the recognised speech so far constrains how subsequent speech is recognised—standard practice in speech recognition.)

If one takes, more generally, *naturalness* as a parameter to optimisation, then incremental processing can bring a range of phenomena into reach that cannot otherwise be modelled. Examples of such phenomena are, as mentioned above, concurrent feedback (“uh-huh”, “yeah”), fast turn-taking, collaborative utterance construction (Buß and Schlangen, 2010), and generation of hedges and self-corrections (Skantze and Hjalmarsson, 2010). (See Edlund et al. (2008) for a general discussion of the goal of naturalness for dialogue systems.)

Lastly, if one goes even further and chooses *realism* as a modelling goal, treating the dialogue system as a computational model of human cognition (Schlangen, 2009), then one would be well advised to make the model work incrementally, given the wealth of evidence that the human language processor works incrementally (see references above).

2.2 Invariants and Variants in Incremental Processing

2.2.1 MODULARITY

Implemented dialogue systems are typically modular systems, where separable processing tasks are encapsulated in separate software modules. Encapsulation of processing tasks can also be found in cognitively motivated theories (e.g., Levelt’s model of generation, Levelt (1989)).[4] Our first goal in the present work is to characterise such modular structure. This involves a) identifying the modules in a system (abstractly, without saying yet too much about what exactly they do) and b) specifying the connections between them.

Figure 2 shows three examples of *module networks*, representations of systems in terms of their component modules and the connections between them. Modules are represented by boxes, and connections by arrows indicating the path along which information flows between modules, the direction of this flow, and, through the use of parallel connections, the “bandwidth” of the

4. As an aside, note that modularity does not preclude extensive interaction between information sources, as postulated in current constraint-based theories of language processing (MacDonald, 1994), see discussion in Altmann and Steedman (1988).

connection (see below). Arrows not coming from or going to modules represent the global input(s) and output(s) to and from the system.

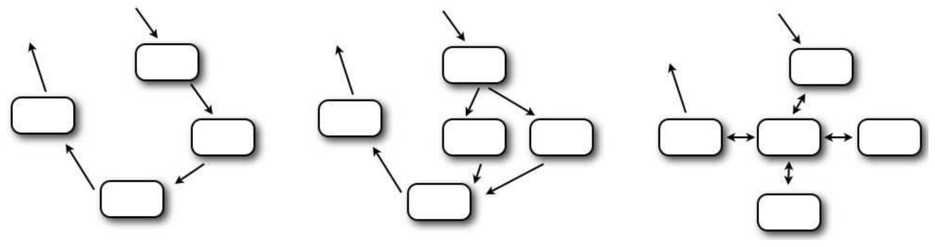


Figure 2: Module network topologies

Our goal now is to facilitate exact and concise description of the differences between module networks such as those in the example. Informally, the network on the left can be described as a simple pipeline with no parallel paths, the one in the middle as a pipeline enhanced with a parallel path, and the one on the right as a star-architecture; we want to be able to describe exactly the constraints that define each type of network.

An important element of such a specification is to state how information flows in the system and between the modules, again in an abstract way, without saying much about the information itself (as the nature of the information depends on details of the actual modules). As mentioned above, the Figure 2 indicates the direction of information flow (i.e., whose output is whose input) through the direction of edges and the “bandwidth” of a connection between modules. A connection between two modules may be single-directional, if information can only flow in one way (which typically will be from a ‘lower-level’ to a ‘higher-level’ module, e.g. from one that is closer to the input source to one that is further away, or from one that is closer to the generation source to one that is closer to the output channel). In a bi-directional setting, higher-level information may influence lower-level processing, for example through expressing expectations that the lower-level module should work towards.

The ‘bandwidth’ of the connection is represented via the number of edges between two nodes (the more edges, the higher the bandwidth), and this refers to whether parallel information can be sent or not. One possible source for such parallelism in an incremental dialogue system is illustrated in Figure 3 (right), where for some stretches of an input signal (a sound wave), alternative hypotheses are entertained (note that the boxes here do *not* represent modules, but rather bits of incremental information, namely words; flattened into strings, one alternative consists of four words, the other of three). We can view these alternative hypotheses about the same original signal as being parallel to each other (with respect to the input they are grounded in).

2.2.2 GRANULARITY

Even systems that modularise the dialogue processing task in roughly the same way can still differ along another dimension of incremental processing, namely in how they divide up larger units into increments. As an example, one system may decide on words as the minimal units—we will call such units simply the *incremental units* (IU) of a system—whereas another system may package incremental results triggered by time, e.g. every 500ms, and not by linguistic units. More formally,

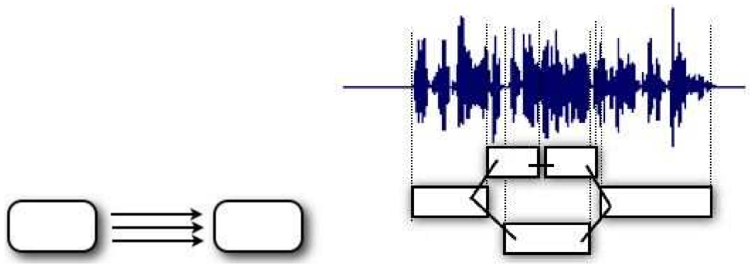


Figure 3: Parallel information streams (left) and alternative hypotheses (right)

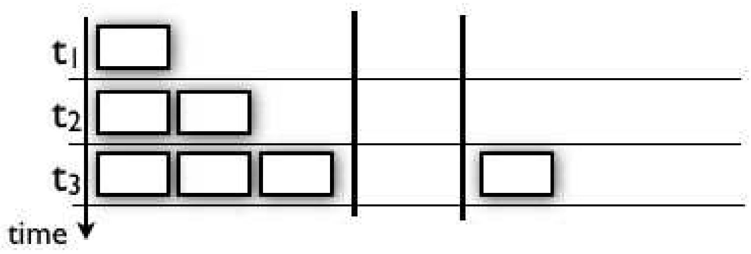


Figure 4: Incremental input mapped to (less) incremental output

we will express this difference as a difference in relation of incremental units to the larger unit of which they are increments.

2.2.3 INPUT/OUTPUT RELATIONS

We also want to be able to specify how incremental bits of input of a processing module can relate to incremental bits of its output. Figure 4 shows one possible configuration, where over time incremental bits of input (shown in the left column) accumulate before one bit of output (in the right column) is produced. (As for example in a parser that waits until it can compute a major phrase out of the words that are its input.) Describing the range of possible module behaviours with respect to such input/output relations is another important element of the abstract model presented here.

2.2.4 REVISABILITY

It is in the nature of incremental processing, where output is generated on the basis of incomplete input, that such output may have to be revised once more information becomes available (see (Baumann et al., 2009) for a detailed investigation of the behaviour of ASR in this respect). Figure 5 illustrates such a case. At time-step t_1, the available frames of acoustic features (shown in the left column) lead the processor, an automatic speech recogniser, to hypothesise that the word "four" has been spoken. This hypothesis is passed on from the internal state of the processor (middle column) to the output (right column). However, at time-point t_2, as additional acoustic frames have come in, it becomes clear that "forty" is a better hypothesis about the previous frames together with the new ones. It is now not enough to just output the new hypothesis: it is possible that later modules have

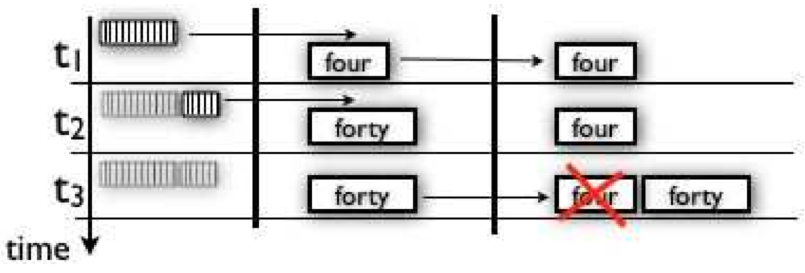


Figure 5: Example of hypothesis revision

already started to work with the hypothesis "four", so the changed status of this hypothesis has to be communicated as well. This is shown at time-step t_3.

Note that the situation just described is different from, and comes additional to, the uncertainty management in dialogue systems in general. It is often seen desirable to be able to represent a module's *confidence* in its hypotheses; indeed the increasingly popular approaches to dialogue management that make use of probablistic models for representing uncertain dialogue states (see e.g. Williams and Young (2007)) rely on such information. We want to distinguish provisions for representing such '*static* confidence' (confidence in a hypothesis at a given time) from the dynamic aspects of potentially having to revise this confidence in the light of later information and having to percolate this revision through the module network. It is the latter that Figure 5 illustrates, and it is another aspect of incremental systems that may be handled differently in different systems. Capturing these differences is the final aim of our model.

2.3 Related Work

The model described here is inspired partially by Young et al. (1989)'s token passing architecture; our model can be seen as a (substantial) generalisation of the idea of passing smaller information bits around, out of the domain of ASR and into the system as a whole. Some of the characterisations of the behaviour of incremental modules are inspired by (Kilger and Finkler, 1995), but adapted and extended from incremental generation to the general case of incremental processing.

While there recently have been a number of papers about incremental systems (e.g., (DeVault and Stone, 2003; Aist et al., 2006; Brick and Scheutz, 2007)), none of those offer general considerations about architectures, [5] which is what we are trying to offer in the following.

3. The Model

3.1 Overview

We model a dialogue processing system in an abstract way as a collection of connected processing modules, where information is passed between the modules along these connections. The third component beside the modules and their connections is the basic unit of information that is communicated between the modules, which we call the *incremental unit* (IU). We will only characterise those properties of IUs that are needed for our purpose of specifying different system types and ba-

5. Despite its title, (Aist et al., 2006) also only describes one particular setup.

sic operations needed for incremental processing; we will not say anything about the actual, module specific *payload*s of these units.

The processing module itself is modelled as consisting of a *Left Buffer* (LB), the *Processor* proper, and a *Right Buffer* (RB). When talking about operations of the Processor, we will sometimes use *Left Buffer-Incremental Unit* (LB-IU) for units in LB and *Right Buffer-Incremental Unit* (RB-IU) for units in RB.

This setup is illustrated in Figure 5 above. IUs in LB (here, acoustic frames as input to an ASR) are *consumed* by the processor (i.e., are processed) which creates an internal result; in the case shown here, this internal result is *posted* as an RB-IU only after a series of LB-IUs have accumulated. In reality, such processing will obviously take some time to complete. For the purposes of our abstract description here, however, we abstract away from such processing times and describe Processors as relations between (sets of) LBs and RBs.

We begin our description of the model with the specification of network topologies.

3.2 Network Topology

Connections between modules are expressed through *connectedness axioms* which simply state that IUs in one module's right buffer are also in another buffer's left buffer. (Again, in an implemented system communication between modules will take time, but we abstract away from this here.) This connection can also be partial or filtered. For example, $\forall x(x \in RB_1 \wedge NP(x) \leftrightarrow x \in LB_2)$ expresses that all and only NPs in module one's right buffer appear in module two's left buffer. If desired, a given RB can be connected to more than one LB, and more than one RB can feed into the same LB (see the middle example in Figure 2). Together, the set of these axioms defines the network topology of a concrete system. Different topology types can then be defined through constraints on module sets and their connections. I.e., a pipeline system is one in which it cannot happen that an IU is in more than one right buffer and more than one left buffer.

Note that for now we are assuming token identity and not for example copying of data structures. That is, we assume that it indeed is the *same* IU that is in the left and right buffers of connected modules, and hence any changes made to an IU are immediately known to all modules that contain it in a buffer. This allows us to abstract away from the actual "transportation" of IUs between modules (but see Section 5 below).

3.3 Incremental Units

So far, all we have said about IUs is that they are holding a 'minimal amount of characteristic input' (or, of course, a minimal amount of characteristic *output*, which is to become some other module's input). Communicating just these minimal information bits is enough only for the simplest kind of system that we consider, a pipeline with only a single stream of information and no revision. If more advanced features are desired, there needs to be more structure to the IUs. In this section we describe the kinds of information that the most capable systems need to represent, and that make possible operations like hypothesis revision, prediction, and parallel hypothesis processing. (These operations will be explained in the next section.) If in a particular system some of these operations aren't required, some of the structure on IUs can be simplified.

Informally, the representational desiderata are as follows. First, one needs to be able to express certain relations between IUs, which record how IUs within a module and between modules belong together. (Recall that IUs are meant to be incremental, minimal bits of what will at some point amount to one larger unit; e.g. words that form an utterance, or chunks of semantics that will eventually form the interpretation of an utterance, etc.) These relations come in two basic types; we discuss them in detail in the next two subsections. Apart from the relations, we want IUs to carry three other types of information as well: a confidence score representing the confidence its producer had in it being accurate; information about whether revisions of the IU are still to be expected or not; and information about whether the IU has already been processed by consumers, and if so, by whom. A full specification of all types of information about IUs one might want to track is given below in Section 3.3.4.

3.3.1 HORIZONTAL RELATIONS BETWEEN IUS

The first type of relation expresses what could be described as 'horizontal' relationships between IUs; horizontal, because these relationships hold only between IUs of the same level (produced by the same module), and typically reflect in some way the temporal order in which the IUs were created.

The most important instance of this type of relation is the *successor* relation, which links IUs in such a way that from a chain of IUs linked with this relation, the (possibly still partial) larger unit these IUs are increments of can be read off. To make this description concrete: this relation will link IUs representing words (for example, in the output of an ASR) in such a way that the (possibly still partial) utterance hypothesis can be produced by following the links. More formally, if two ASR word hypothesis-IUs IU_1 and IU_2 stand in this relation, this expresses that the ASR takes IU_2 to be the continuation of the utterance whose previous element is IU_1. If another IU, IU_3, were linked to IU_1 as well, this would express that the sequences IU_1-IU_2 and IU_1-IU_3 form alternatives. (See Figure 3, right; this relation is also illustrated in Figure 6, discussed below.)

This is the most basic kind of horizontal relation, which most incremental systems will have to realise at least implicitly (since it keeps track of how increments form larger units of information). For other purposes, one may also want to allow other relations of this type. For example, dependency relations between concepts would be of this type as well. Figure 6 illustrates this for a possible semantic representation for the two recognition alternatives "move the ball two fields" / "move the ball to field z". Here we have one IU holding the concept of a move action, and linked to it, as further specifying it, a concept representing the patient of the action (BALL-5) and, as alternatives corresponding to the alternative recognition results, two different directional concepts (DISTANCE:2 / "two fields" and TARGET:Z / "field z"). To recognise at this level that these two directional concepts are alternatives and do not both hold, one must follow the *successor* relation, where they are on different branches. This illustrates that the *specifies* relation as used here (and represented in the figure by dashed arrows) cannot be subsumed by the successor relation (solid arrows in the figure).

3.3.2 HIERARCHICAL RELATIONS

The other kind of relation orders IUs into an informational hierarchy, linking IUs to those bits of information—other IUs—on which they depend informationally, or, as we will call it, in which they are *grounded*. Examples for this are the links between IUs representing segments of audio material and IUs representing the corresponding word hypotheses (i.e., LB-IUs and RB-IUs of a speech recogniser); between word hypotheses and semantic representations (i.e., LB-IUs and RB-IUs of a semantic parser; in both these examples the links cut across module boundaries); between larger phrases in a parser and their constituents (e.g., an IU representing a sentence-level parse and

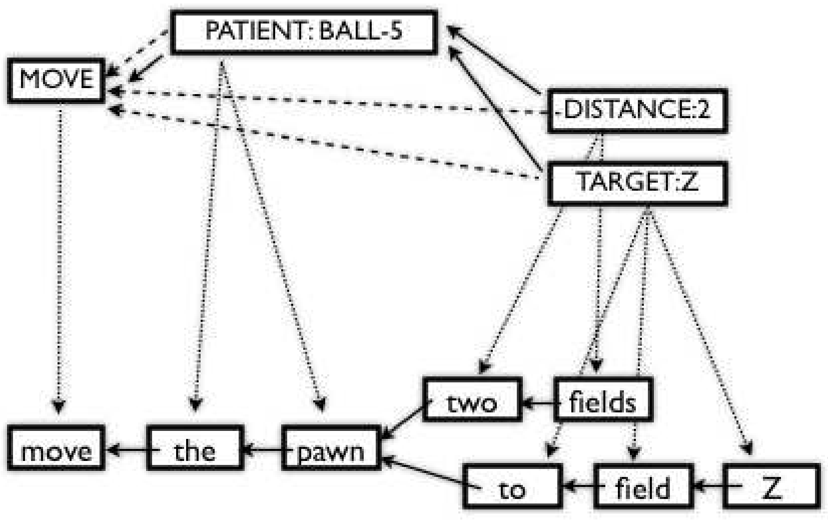


Figure 6: An example network, built by the relations *successor* (solid arrows), *specifies* (wide dashed arrows), *grounded in* (narrow dashed arrows)

IUs representing an NP and a VP, in which case the link holds between IUs from the same level); or between states of the dialogue manager and plans for system utterances. Figure 6 shows such *grounded in* links between word-IUs and semantic concept IUs, as narrow dashed lines.

This relation effectively tracks the flow of information through the system; following its transitive closure one can go back from the highest level IU, which is output by the system, to the input IU or set of input IUs on which it is ultimately grounded.[6] Going in the other direction, this is also the path along which doubts of a module about the quality of an IU, or ultimately its revocation, can percolate through the module network. For example, should an ASR decide to remove support for a word hypothesis, then all IUs that are linked to this IU via this relation will be thrown into doubt as well. (See the purge operation described below.)

3.3.3 META-INFORMATION

We also assume that it will be useful in some systems to be able to represent and communicate additional information about IUs. For example, in certain setups it may be useful for a module to learn whether its consumers have found a use for a hypothesis or not. This could be a parser that gets told whether an NP it has built has a denotation in the current dialogue context or not, and that could then base a decision on whether to extend the parse or not on that information. (See below in Section 4.1 the discussion of 'early interaction systems'.)

6. Of course, the output of a dialogue system is not fully determined just by its input. Other elements of the state figure as well; e.g., some parts of an answer to a query will depend on a database and not just the query, however, the intention to respond with a statement will still be grounded in the input signal.

The other type of information concerns the confidence of a module in its hypotheses. As noted above, this confidence can change during the processing of further input; in extreme cases, such changes in confidence may lead to revocation of hypotheses on which possibly already later hypotheses were built.

Finally, in building systems ourselves, we have found that it can be useful to add a binary notion of *commitment* that signals that a IU will not (or shall not) be modified again. A module can declare such commitment on its own IUs in cases where for technical reasons (e.g., to keep internal state to a manageable size) it decides that it will not touch them anymore. It can also demand commitment from the producers of its input IUs, if it has based unrevisable actions on them; should the producing module still have a need to revise them, then this would lead to an exceptional situation that should be handled specifically. (E.g., by performing an explicit self-correction.) Note that we intend a rather technical, house-keeping use for this. There is a more general sense in which modules are 'committed' to their IUs, but this we think is covered by providing confidence information and by agreeing on a system-internal strategy for putting out IUs in the first place (e.g., by either optimizing for speed, sending out even possibly very transient hypotheses, or by optimizing for quality, possibly holding back IUs until internal confidence has reached a certain threshold).

3.3.4 FORMAL SPECIFICATION

We define formally the universe $\mathcal{U}$ in which incremental units live as follows. It consists of a set of IU objects, $\mathcal{IU}$ (which includes a special IU $\top$), a set of module labels $\mathcal{M}$, and a collection of functions and relations defined on these:

- i is an identification function, which, for ease of reference, maps each IU object onto a unique ID (e.g., a natural number).
- A function c that maps each IU to a module label, its creator. (From this follows that each IU can only have one creator.)
- A family of relations between IUs created by the same module (i.e., IUs α, β, where $i(\alpha) \neq i(\beta)$ and $c(\alpha) = c(\beta)$); we will call these relations *same level links*. One instance of this type is the *successor* relation discussed above, which defines a (partial) order on IUs. We specify that by default, IUs are successor of a special IU $\top$. This guarantees that IUs linked via the *successor* relation form a connected graph rooted in the $\top$ element. The type of the graph will depend on the purposes of the sending and consuming module(s). For a one-best output of an ASR it might be enough for the graph to be a chain (and *successor* hence be a total order), whereas an n-best output might be better represented as a tree (with all first words linked to $\top$) or a lattice (as in Figure 3, right).
  There can be other kinds of such same level relations, as described above.
- $\mathcal{G}$ is the *grounded in* relation, connecting an IU to one or more IUs out of which it was built. For example, an IU holding a (partial) parse might be grounded in a set of word hypothesis IUs, and these in turn might be grounded in sets of IUs holding acoustic features.
  While the *same level link* always points to IUs on the same level, *grounded in* links can hold both between IUs from the same level as well as between IUs of connected modules; in both cases it expresses that one bit of information depends on other bits. The transitive closure of this relation hence links system output IUs to a set of system input IUs. For convenience, we may define a relation *supports(x,y)* for cases where y is grounded in x; and hence the closure of this relation links input-IUs to the output that is (eventually) built on them.

This is also the hook for the mechanism that realises the revision process described above with Figure 5: if a module decides to revoke one of its hypotheses, it sets its confidence value (see below) to 0; on noticing this event, all consuming modules can then check whether they have produced RB-IUs that link to this LB-IU, and do the same for them. In this way, information about revision will automatically percolate through the module network.
Finally, we can also define a special value (e.g., $n.n.$) for this relation and use it to trigger *prediction*: if an RB-IU is *grounded in*-related to $n.n.$, this can be understood as a directive to the processor to find evidence for this IU (i.e., to prove it), using the information in its left buffer.

- $\mathcal{T}$ is the *confidence* (or trust) function, which maps each IU to a numerical value (depending on the module's needs, either from $\mathbb{N}$ or $\mathbb{R}$) through which the generating processor can pass on its confidence in its hypothesis. This then can have an influence on decisions of the consuming processor. For example, if there are parallel hypotheses of different quality (confidence), a processor may decide to process (and produce output for) the best first.
  A special value (e.g., 0, or -1) can be defined to flag hypotheses that are being revoked by a processor, as described above.
- $\mathcal{C}$ is the *committed* property, which holds when a producing module has committed to the IU, i.e., it guarantees that it will never revoke the IU. See below for a discussion of how such a decision may be made, and how it travels through the module network.
- $\mathcal{S}$ is the *seen* relation, relating IUs and module labels. Using this relation, processors can record whether they have "looked at"—that is, attempted to process—the IU. In the simplest case, the positive fact can be represented simply by adding the processor ID to the list (assuming here that all processors in a system are identified by a unique ID); in more complicated setups one may want to offer status information like "is being processed by module ID" or "no use has been found for IU by module ID", or "I assign this IU a probability of n" via additional relations. This allows processors both to keep track of which LB-IUs they have already looked at (and hence, to more easily identify new material that may have entered their LB) and to recognise which of its RB-IUs have been of use to later modules, information which can then be used for example to make decisions on which hypothesis to expand next.
- $\mathcal{P}$ finally relates IUs to linguistic objects (like words or parses) which are their actual *payload*, i.e., the module-specific unit of 'characteristic input' (or output).

Systems can differ in which of these elements they realise, and even modules within a system can differ along this dimension. It seems plausible to assume that most incremental systems will have concepts that can be mapped to what may be called the core set of IU properties, $(i, c,$ *same level link, grounded in link*$, \mathcal{T}, \mathcal{P})$, while more sophisticated processing (e.g., using prediction and a high degree of interaction between modules) will make use of the other properties.

3.4 Modules

As explained above, modules consist of left and right buffers, and processors with internal state that operate on input IUs to create output IUs. Normally, the direction of this update operation is from the left to the right (meaning that LB-IUs are 'consumed' to produce new RB-IUs that are grounded in them), but in the case of expectation-guided modules can also be from right to left (i.e., attempting to find evidence for some predicted output). A central concept when looking at modules is that of the update step, which has three stages: 1) the left buffer of the module is updated from

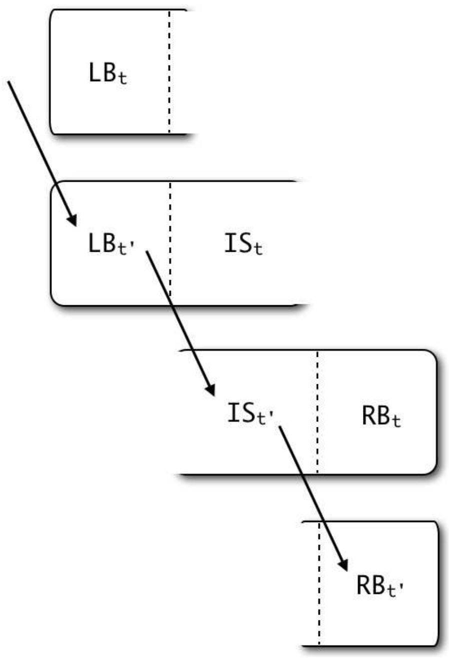


Figure 7: A schematic view of the update process

state LB$_t$ to state LB$_{t'}$; 2) the current module recognises what has changed, and performs its own update first of its internal state (IS$_t$ to IS$'_t$) and then 3) of its own right buffer (RB$_t$ to state RB$'_t$; this in turn will be the first update step for all consuming modules). Figure 7 illustrates this flow of information through a module, showing only those parts of the module that are relevant at each timestep.

We use these labels for the update stages when we now first discuss some more properties of buffers, then list operations that processors have to perform, and finally describe different possible module behaviours.

3.4.1 BUFFERS

For the purposes of the abstract model, we can conceptualise buffers simply as sets of IUs, for which the constraints hold that are specified by the connectedness axioms as described above. During the execution of a system, buffers change over time (through the updates described above). To denote the state of a buffer at a given time t, we define a function $state$ from time labels to IU sets. The changes made to a buffer from time t to t' (i.e., to get from $state(t)$ to $state(t')$) we can then denote by $\Delta_{t,t'}$. We will assume for now that the processor only receives this delta, and bases its computations on this. We discuss alternative set-ups below in Section 5. We also define a function are which returns the current *active right edge* of a buffer; this is the set of IUs that are a) currently active (we assume that IUs that belong to input that has been fully processed are marked as inactive); b) not revoked (see below); and c) maximal with respect to the *successor* relation (recall that this defines a partial order rooted in $\top$). This is the right edge insofar as this is where new increments will attach (e.g., where a new word will be added as more audio material comes in).

3.4.2 PROCESSOR OPERATIONS

At the most abstract level, the job of processors is to react to changes in their buffers (the $\Delta_{t,t'}$) by performing appropriate updates. The elementary task here is to react to the appearance of new LB-IUs and eventually build new RB-IUs out of them (or, in the case of a predictive system, to react to new expectations being entered as RB-IUs and to evaluate subsequent LB-IUs as to whether they provide evidence for these expectations). How exactly this is done is specific to the individual tasks of the module (e.g., ASR, parser, dialogue manager, etc.), and we won't have anything to say about this here; what we will describe here are the different *types* of updates that can be implemented in an incremental processing module. We also leave open *how* processors are triggered into action; we simply assume that on receiving new LB-IUs or RB-IUs or noticing changes to already known IUs—more generally, on being given $\Delta_{t,t'}$ of the appropriate buffer—they will eventually perform these operations. Again, we describe here the complete set of operations; systems may differ in which subset of the functions they implement, or even whether they implement these operations as recognisably separate steps at all.

purge LB-IUs that are revoked by their producer must be purged from the internal state of the processor (so that they will not be used in future updates) and all RB-IUs grounded in them must be revoked as well.

Some reasons for revoking hypotheses have already been mentioned. For example, a speech recogniser might decide that a previously output word hypothesis is not valid anymore (i.e., is not anymore among the n-best that are passed on). Or, a parser might decide in the light of new evidence that a certain structure it has built is a dead end, and withdraw support for it. In all these cases, *all* 'later' hypotheses that build on this IU (i.e., all hypotheses that are in the transitive closure of this IU's *support* relation) must be purged. If all modules implement the purge operation, this revision information will be guaranteed to travel through the network.

new IU update New LB-IUs have to be integrated into the internal state, and eventually new RB-IUs are built based on them (not necessarily in the same frequency as new LB-IUs are received; see Figure 4 above, and discussion below). The new RB-IUs have to be related appropriately to other IUs (e.g., via *same level links*, grounded in points, etc.). As mentioned above, this is the most basic operation of a processor, and can be expected to be implemented in all systems.

Processors can take *supports* information into account when deciding on the order in which they update. A processor might for example decide to first try to use the new information (in its LB) to extend structures that have already proven useful to later modules (that is, that support new IUs). For example, a parser might decide to follow an interpretation path that is deemed more likely by a contextual processing module (which has grounded hypotheses in the partial path). This may result in better use of resources—the downside of such a strategy of course is that modules can be garden-pathed.[7]

Update may also work towards a goal. As mentioned above, putting ungrounded IUs in a module's RB can be understood as a request to the module to try to find evidence for it. For example, the dialogue manager might decide based on the dialogue context that a certain type of dialogue act is likely to follow. By requesting the dialogue act recognition module to find evidence for this hypothesis, it can direct processing resources towards this task. (The dialogue recognition module then can in turn decide on which evidence it would like to see, and ask lower modules to prove this. Ideally,

7. It depends on the goals behind building the model whether this is considered a downside or desired behaviour.

this could filter down to the interface module, the ASR, and guide its hypothesis forming. Technically, something like this is probably easier to realise by other means; see (Schuler et al., 2009), briefly discussed below, for an example of an integrated approach, where semantics and reference resolution can directly bear on the speech recognition process.)

We finally note that in certain setups it may be necessary to consume different types of IUs in one module. As explained above, we allow more than one module to feed into another module LB. An example where something like this could be useful is in the processing of multi-modal information, where information about both words spoken and gestures performed may be needed to compute an interpretation.

commit There are three ways in which a processor may have to deal with commits. First, it can decide for itself to commit RB-IUs. For example, a parser may decide to commit to a previously built structure if it failed to integrate into it a certain number of new words, thus assuming that the previous structure is complete. Second, a processor may notice that a previous module has committed to IUs in its LB. This might be used by the processor to remove internal state kept for potential revisions. Eventually, this commitment of previous modules might lead the processor to also commit to its output, thus triggering a chain of commitments.

Interestingly, it can also make sense to let commits flow from right to left, as briefly discussed above. For example, if the system has committed to a certain interpretation by making a publicly observable action (e.g., an utterance, or an action in another modality), this can be represented as a commit on IUs. This information would then travel down the processing network; leading to the potential for a clash between a revoke message coming from the left and the commit directive from the right. In such a case, where the justification for an action is revoked when the action has already been performed, self-correction behaviours can be executed.[8]

other updates Finally, in some settings it may also be desirable to let modules change the confidence score of IUs after having put them into the RB (and so after they have already potentially been consumed by later modules); the consuming modules then might need to react to this change, perhaps by updating their internal state, by changing their future update strategy, or by changing their own confidence in something they have passed on into their own right buffer. It may also be useful in certain settings to allow other aspects of IUs to be changed later as well, such as *same level links*; again, this would be something that consuming modules need to notice and react to.

3.4.3 CHARACTERISING MODULE BEHAVIOUR

Modules can also be characterised through a description of the changes that updates yield to buffers and internal states, and the relations between changes to left buffers and those to right buffers. We list several dimensions along which such a characterisation can be made.

Updates to IU Sequences Using the notion of a *right edge* from Section 3.4.1, we can transfer some terms from (Wirén, 1992): a module is *left-to-right incremental* if it only produces *extensions* to the current right edge; within what Wirén (1992) calls *fully incremental*, we can make further distinctions, namely according to whether only revisions or also insertions and deletions are allowed. (Revisions are covered by the revoke operation described above; insertions and deletions can be expressed in our model by allowing *successor* links to be changed appropriately.) When we want

8. In future work, we will explore if and how (e.g. through the implementation of a self-monitoring cycle with commits and revokes) the various types of dysfluency described by Levelt (1989) and others can be modeled.

to make this further distinction, we call the former *right edge revision-incremental* and the latter *insertion/deletion incremental*.

Processor-Internal Incrementality We can also distinguish modules according to how they update their internal state. We call modules that keep their internal state between update steps and only enrich it according to the current Δ of their input buffer *internally incremental* (and the algorithms they use for doing so *fully incremental algorithms*). While this perhaps conforms best with an intuitive understanding of what incremental processing is, one can also imagine a different strategy (which has indeed been realised (DeVault et al., 2009), as briefly reviewed below). In this strategy, all internal state is thrown away between updates, and output is always computed from scratch using the full currently available input and not just the newest increments of it; we will call such modules *restart incremental*. This strategy can be used when one has available more conventional processing modules which happen to be robust against partial input, but are not built to handle incremental changes to their input.

Update Frequency This dimension concerns how the update frequency of LB-IUs relates to that of (connected) RB-IUs.

We write *f:in=out* for modules that guarantee that every new LB-IU will lead to a new RB-IU (that is grounded in the LB-IU). In such a setup, the consuming module lags behind the sending module only for exactly the time it needs to process the input. Following Nivre (2004), we can call this *strict incrementality*.

f:in≥out describes modules that potentially collect a certain amount of LB-IUs before producing an RB-IU based on them. This situation has been depicted in Figure 4 above.

f:out≥in characterises modules that update RB *more* often than their LB is updated. This could happen in modules that produce endogenous information like clock signals, or that produce continuously improving hypotheses over the same input (see below), or modules that 'expand' their input, like a TTS that produces audio frames.

Connectedness We may also want to distinguish between modules that produce 'island' hypotheses that are, at least when initially posted, not connected to previously generated output IUs via a common element that dominates them through *grounded in* links, and those that guarantee that this is not the case. For example, to achieve an *f:in=out* behaviour, a parser may output hypotheses that are not connected to previous hypotheses, in which case we may call the hypotheses 'unconnected'. Conversely, to guarantee connectedness, a parsing module might need to accumulate input, resulting in an *f:in≥out* behaviour, or may need to speculate on continuations, possibly resulting in *f:in≤out* behaviour.[9]

Completeness We define the *completeness* of a set of IUs which are connected via the successor relation informally as the relation of the sequence they form (e.g., a sequence of words understood as a prefix of an utterance) to (the type of) what would count as a maximal sequence. For example, for an ASR module, such a maximal sequence may be the transcription of a whole utterance and

9. The notion of *connectedness* is adapted from (Sturt and Lombardo, 2005), who provide evidence that the human parser strives for connectedness.

not just a prefix of one; for the parser maximal output may be a parse of type sentence (as opposed to one of type NP, for example), etc.[10]

Building on this notion, we can characterise modules according to completeness of their LB and RB. In a *c:in=out*-type module, the most complete set of RB-IUs is only as complete as the most complete set of LB-IUs. That is, the module does not speculate about completions, nor does it lag behind. (This may technically be difficult to realise, and practically not very relevant.)

More interesting is the difference between the following types: In a *c:in≥out*-type module, the most complete set of RB-IUs potentially lags behind the most complete set of LB-IUs. This will typically be the case in *f:in≥out* modules. *c:out≥in*-type modules finally potentially produce output that is *more* complete than their input, i.e., they *predict* continuations. An extreme case would be a module that always predicts complete output, given partial input. Such a module may be useful in cases where modules have to be used later in the processing chain that can only handle complete input (that is, are non-incremental); we may call such a system *prefix-based predictive, semi-incremental*. (Again, (DeVault et al., 2009) is an example of such a module; as is (Schlangen et al., 2009).)

With these categories in hand, we can make further distinctions within what Dean and Boddy (1988) call *anytime algorithms*. Such algorithms are defined as a) producing output at any time, which however b) improves in quality as the algorithm is given more time. Incremental modules by definition implement a reduced form of a): they may not produce an output at any time, but they do produce output at more times than non-incremental modules. This output then also improves over time, fulfilling condition b), since more input becomes available and either the guesses the module made (if it is a *c:out≥in* module) will improve or the completeness in general increases (as more complete RB-IUs are produced). Processing modules, however, can also be anytime algorithms in a more restricted sense, namely if they continuously produce new and improved output even for a constant set of LB-IUs, i.e. without changes on the input side. (Which would bring them towards the *f:out≥in* behaviour.)

As a final note, we can now see that a non-incremental system can be characterised as a special case of an incremental system, namely one where IUs are always maximally complete (with *c:in=out*) and where all modules update in one go (*f:in=out*). (Typically, in such systems IUs will also always be committed, but this need not necessarily be the case for a system to be non-incremental.)

3.5 System Specification

Combining all these elements, we can finally define a system specification as the following:

- A list of modules that are part of the system.
- For each of those a description in terms of which operations from Section 3.4.2 the module implements, and a characterisation of its behaviour in the terms of Section 3.4.3.
- A set of axioms describing the connections between module buffers (and hence the network topology), as explained in Section 3.2.

10. This definition is only used here for abstractly classifying modules. Practically, it is of course rarely possible to know how complete or incomplete an ongoing input is. Investigating how a dialogue system can better predict completion of an utterance is in fact one of the aims of the project in which this framework was developed.

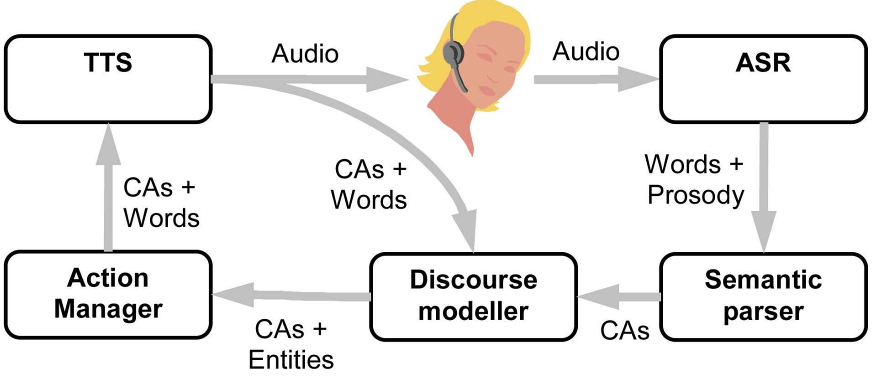


Figure 8: The NUMBERS system architecture (CA = communicative act)

- Specifications of the format of the IUs that are produced by each module, in terms of the definition of slots in Section 3.3. For technical reasons, one may also want to specify which information given about IUs may be changed later, and which can be considered immutable.

4. Some Example Specifications

4.1 'Early Interaction' Systems

Most previous work on incremental processing has focused on one of the many possible advantages of this processing style, namely on making available 'higher-level' information to 'lower-level' processes, where this information can then be used to guide processing. This typically has taken the form of letting a parser interact with extra-syntactic knowledge. (Despite considerable differences in the way this effect is achieved, (DeVault and Stone, 2003; Stoness et al., 2005; Aist et al., 2006, 2007; Brick et al., 2007; Brick and Scheutz, 2007) can all be subsumed under this description.)

The general approach can be described in IU terms as follows: the parser posts certain constituents (NPs and VPs) as RB-IUs, connected modules filter out the phrase type they care about and evaluate them in the domain (e.g., a domain ontology checks which operations are possible in the domain, and what likely frames are that express a certain action; or a module tests whether NPs have a denotation in the domain). This evaluation is attached to the IU (via the *seen* relation, for example); the parser then must be capable of noticing updates to an RB-IU and act accordingly (e.g., modify the chart that forms its internal state).

The cited papers all focus on this interaction and do not say much about the systems in which this interaction is realised, so we cannot give full system specifications here.

4.2 The Numbers System

The Numbers System (Skantze and Schlangen, 2009) has a special status here because it can not just be usefully described in the terms explained here, it actually directly instantiates some of the concepts and methods described in this paper.

The module network topology of the system is shown in Figure 8. This is pretty much a standard dialogue system layout, with the exception that prosodic analysis is done in the ASR and that dialogue management is divided into a discourse modelling module and an action manager. As can be seen in the figure, there is also a self-monitoring feedback loop—the system's actions are sent from the TTS (text-to-speech synthesizer) to the discourse modeller. The system has two modules that interface with the environment (i.e., are system boundaries): the ASR and the TTS.

A single hypothesis chain connects the modules (that is, no two same level links point to the same IU). Modules pass messages between them that can be seen as XML-encodings of IU-tokens. Information strictly flows from LB to RB. All IU slots except seen ($\mathcal{S}$) are realised. The purge and commit operations are fully implemented. In the ASR, revision occurs as already described above with Figure 4, and word-hypothesis IUs are committed (and the speech recognition search space is cleared) after 2 seconds of silence are detected. (Note that later modules work with all IUs from the moment that they are sent, and do not have to wait for them being committed.) The parser may revoke its hypotheses if the ASR revokes the words it produces, but also if it recovers from a "garden path", having built and closed off a larger structure too early. As a heuristic, the parser waits until a syntactic construct is followed by three words that are not part of it until it commits. For each new discourse model increment, the action manager may produce new communicative acts (CAs), and possibly revoke previous ones that have become obsolete. When the system has spoken a CA, this CA becomes committed, which is recorded by the discourse modeller.

No hypothesis testing is done (that is, no un-grounded information is put on RBs). All modules have a *f:in≥out*; *c:in≥out* characteristic; that is, they may collect information in the form of LB-IUs before they generate RB-IUs and hence potentially lag behind somewhat.

The system achieves a very high degree of responsiveness—by using incremental ASR and prosodic analysis for turn-taking decisions, it can react in around 200ms when suitable places for backchannels are detected, which should be compared to a typical minimum latency of 750ms in common systems where only a simple silence threshold is used.[11]

4.3 A Prefix-Based, Predictive System

The module described in (DeVault et al., 2009) has already been mentioned a couple of times above. The module is an NLU component that outputs full semantic frames, even if its input is only a partial utterance. In our terms, it is *prefix-based predictive*, with *c:out≤in*. The module is also 'internal-event based' in that updates are triggered by events of an internal clock (the ASR is polled every 200ms) and not by the event of receiving a new LB-IU (more on this distinction in the next section). No internal state is kept between update steps, so output is always computed on the basis of the latest, possibly still partial, full input and not on the newest increments only; the module is only *restart-incremental*.

4.4 Incremental Generation in the DEAL System

Skantze and Hjalmarsson (2010) describe an approach to incremental speech generation in dialogue systems that is based on the model presented here. The approach allows a dialogue system to incrementally interpret spoken input while simultaneously planning, realising and self-monitoring the system response. If the system detects that the user has stopped speaking and it is appropriate for the system to take the turn, the system may start to speak, even if it does not yet have a complete plan of what to say, or if the input IUs are not yet committed. As the input is processed, the action

11. A video showing an example interaction with the system can be found at `http://www.purl.org/net/Numbers-SDS-Video`.

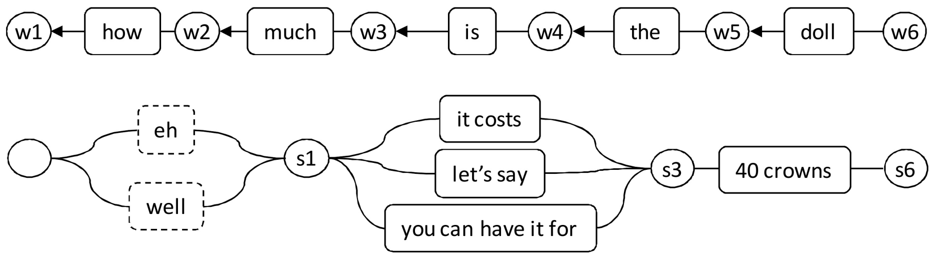


Figure 9: The right buffer of an ASR (top) and the speech plan that at is incrementally produced (bottom). Vertex s1 is associated with w1, s3 with w3, etc. Optional speech segments are marked with dashed outline.

manager in the system builds a tentative speech plan, which may then later be revised. If output IUs need to be revised, and they have already been spoken, the system will automatically perform an overt self-repair, using editing terms such as "sorry, I mean".

In order to facilitate incremental speech generation, system utterances are made up of smaller IUs. The action manager incrementally produces a *speech plan*, which is a graph that represents different ways of realising a message. Each edge in this graph is associated with *speech segment*. The speech output module may then traverse this graph in different ways, depending on a number of constraints, such as timing. Each speech segment is also made up of smaller *speech units*. These mark locations where an utterance may be aborted, or where self-corrections may occur. Figure 9 illustrates how a speech plan may be incrementally produced, as words are recognized by the speech recognizer.

The grounded-in links may then be followed all the way back from the speech units to the speech segments, to the speech plan, to the communicative acts in the discourse model that it was a response to, and finally to the phrases and the words in the user utterance. Thereby, a revision in the speech recogniser may trigger a revision in the spoken output.

This system has been evaluated in a Wizard-of-Oz setting, and achieved faster reaction times than a version of the system with incremental generation disabled, and was judged more polite, more effective, and better at indicating when to speak.

5. Using the IU Model as a Middle Layer in Incremental Dialogue Systems

As we said in the introduction, the model as laid out in the previous sections is meant to describe the design space for incremental systems, and one of its applications is the uniform description (and consequently, easier comparison) of extant models of incremental processing (including different models of human incremental language processing). However, the model has also proved useful for us in the design of new systems (as described above; other systems are currently under development). In this section, we discuss some additional conceptual issues that need to be addressed when basing a system on this model.[12] We present this in the form of a list of questions that a system designer must answer. Again, we do not give recommendations for one particular solution but rather

12. We will remain mostly on the conceptual level here. A lower level description of implementational problems and links to a collection of reference implementations of the framework can be found in Schlangen et al. (2010).

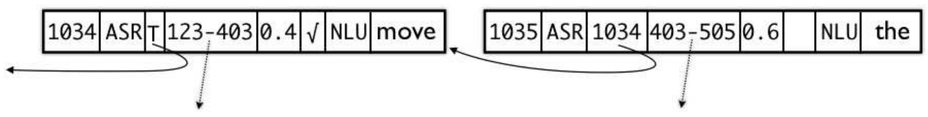


Figure 10: IUs as structured objects. The fields are: ID, producing module, target of *same-level*-link, target of *grounded-in*-link, confidence value, committed, seen-by, payload.

describe the available options, and the circumstances under which certain choices are advisable. In Subsection 5.2 we then present a more detailed description of two different ways to derive a working system infrastructure from these concepts.

5.1 Design Issues

How are IUs represented? The first question to decide is how to represent IUs. As we said above, besides the payload of an IU, which holds the actual incremental 'chunk' of information that is to be exchanged, there are a number of other properties and relations of IUs that one might need to keep track of in a system. A straightforward way of doing so is to realise IUs as a data structure with several fields that hold values (for which most programming languages have basic built-in datatypes), with Booleans for properties and values for relations; Figure 10 illustrates this approach (for two ASR-IUs that are linked via *successor*, of which the first is committed, and both have been used by an NLU component). However, a less direct approach may also be appropriate, where properties of IUs are indirectly represented via properties of the buffers.[13] Section 5.2 will give an example of such an approach, where the properties of being revoked or being committed are represented indirectly and must be inferred from the state of the buffer.

How are buffers synchronised between modules? The second, and more interesting challenge is to implement the flow of information—i.e., the flow of IUs—through the system. In the abstract model explained above, we have treated buffers as sets of entities, and have represented connections between modules (via their buffers) by relations between such sets. Doing this is enough from the point of view of *analysis* of a system, as there it is enough to know that information in one buffer is guaranteed to appear in another buffer as well. When designing a system, however, one has to actually make this happen.

What is to be achieved, then, is that IUs in a producer's right buffer must appear in all of its consumers' left buffers, and all changes of properties of the IUs must be synchronised; in other words, it must be guaranteed that after an update to a RB (step RB_t to $\mathrm{RB}_{t'}$ in Figure 7) all connected LBs must be updated as well (must perform their step LB_t to $\mathrm{LB}_{t'}$), so that the connectedness axioms hold for $\mathrm{RB}_{t'}$ and $\mathrm{LB}_{t'}$. (Or, respectively, for updates to LBs in case of right-to-left information flow.) There are two interrelated aspects to this: one is how the path along which the information travels is realised, the other is what actually travels along this path. Figure 11 sketches some of the available options.

13. This is one reason why we have taken care in Section 3.3.4 above to only specify that there are certain properties that need to be represented, and have not said anything about the concrete data structures to be used for representing them.

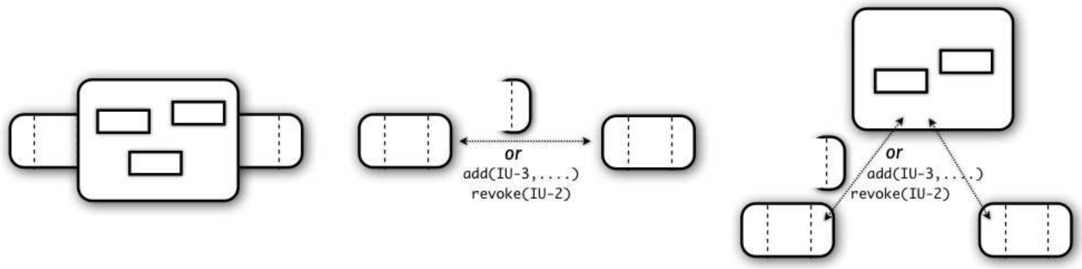


Figure 11: Options for realising paths and messages: shared memory (left); direct communication with either full copies of buffers or edit messages (middle); mediated communication via blackboard, again full copies or edit messages (right)

First, the path: In a set-up where the modules in question operate in a shared memory space (Figure 11, left), strictly speaking there is no path along which the information has to travel. In such a case IUs can simply be data objects in shared memory, and these objects and all changes made to them are immediately accessible to all modules; in this sense, this realisations corresponds most closely to the abstract conceptualisation explained above. (If modules work asynchronously—see discussion below—the usual precautions have to be taken to avoid inconsistencies when modifying shared memory.)

In set-ups where the modules need to be more independent from each other, perhaps running on different machines and/or being implemented in different programming languages, a ‘virtual shared memory’ can be created via message passing. Figure 11 in the middle shows a set-up where producing module and consuming module exchange instructions on how to achieve synchronisation, and on the right it shows a variant of this where the ‘virtual shared memory’ is managed by a dedicated module that implements a ‘blackboard’ (a common set-up in AI systems, provided for example by the Open Agent Architecture, (Cheyer and Martin, 2001)).

Second, the information that travels: The question here is whether the updated buffer is communicated as a whole ($\mathrm{RB}_{t'}$ is sent, to replace LB_t and yield $\mathrm{LB}_{t'}$) or whether only the changes that were made when updating it are communicated (as message like “this is how you can turn LB_t into $\mathrm{LB}_{t'}$”). In the former case, it is left to the consuming module to compute $\Delta_{t,t'}$, whereas in the latter case this information is provided by the producer; in most cases this may be the more efficient solution.

All these options realise the same functionality, namely that buffers are synchronised. Which option is the most appropriate in a given situation depends on other considerations, e.g. on how tightly the modules can be integrated, on whether existing resources have to be re-used, etc.

How are updates to modules triggered? Another question then is how updates in a module are triggered. There are two basic modes of operation here: One, updates can be triggered by the *reception* of new information. This requires that new information is automatically *pushed* to the consuming module; updates are hence driven by the external event of a producing module pushing new information into the buffer. (More accurately: this event triggers a *check* of whether an update to internal state is required; there may be cases where new information does not actually have an effect on the consuming module.)

The other option is to let a consuming module query the producing modules it is interested in for new information at self-determined intervals (or triggered by other, endogenous, internal events). This could then be called a ‘pull model’.

Mixtures of these approaches are of course possible, where some information is pushed and other pulled on demand (for example, new IUs are pushed, but later changes to the confidence slot may not be pushed but only be queried when that information is relevant).

How is control distributed between modules? On creation of an IU, are all further processing steps performed in sequence in all later modules, before control returns and the next IU can be created by this module? Or are modules running asynchronously (in threads, processes, agents)? The former case is more similar to normal dialogue systems, with the only change being that smaller increments travel through the network. The latter case requires more changes, to deal with possible concurrency problems. (Of course, asynchronicity is not only possible in incremental systems, see e.g. (Boye et al., 2000) for an early example of an asynchronous non-incremental system.)

What is preserved between updates? Is internal state cleared at the beginning of each update, or not? If it is cleared, then module needs to process the complete buffer containing all IUs that span the input so far (and in fact doesn’t really work fully incrementally; see above the definition of *restart incremental*.) This is an appropriate choice if a module is used that can handle partial input, but cannot incrementally update its internal state.

What is the relation between buffers and internal states? Lastly, one needs to think about how deeply the modules are encapsulated and protected from each other. The way we have described it so far, the buffers are distinct from the internal state of a module, and a module may not need to take everything from its left buffer into its internal state, and not every processing step that leads to changes in its internal state needs to lead to changes of its right buffer. However, one can also imagine cases where there is a more direct connection between modules, and output of one module is written directly into the internal state of another module (e.g., words from an ASR are put directly into a parse chart) or something is read directly out of an internal state.

5.2 Two Examples of IU-Architectures

We now discuss some more concrete details of how to plan the communication infrastructure of an incremental system. We do this in two variants, where different system goals and preconditions are taken into account. These descriptions are loosely based on our current work on two different systems.[14]

5.2.1 MODULES AS AGENTS; BI-DIRECTIONAL INFORMATION FLOW

Assume that we are in the following situation: we need the modules of the system we’re building to run on different machines (with different computer architectures), because we have to use legacy components (e.g., a vision system that only runs under Microsoft Windows® whereas other components run on UNIX machines). This means that we cannot rely on shared memory for implementing

14. Note that while the choices illustrated in these specifications do cluster together naturally, they by no means are completely dependent—it is possible to combine features of the model in other ways, e.g. have a distributed system with “right edge” encoding of changes (see below).

the module buffers. We also want the system to be predictive, expectation-driven; that is, we want 'right-to-left' information flow.

What we sketch here as a solution to these requirements is a pretty straightforward implementation of the model as detailed above. First, we decide to implement IUs as objects, with information fields as illustrated above in Figure 10; as mentioned there, support for such objects is present in many different programming languages, so we're not (very) restricted on this. Since we can't rely on (process specific) shared memory, we have to achieve synchronisation of buffers through explicit communication between modules. Details of how to set up such communication (registration of modules, routing and buffering of messages etc.) are beyond the scope of this paper, we just note that there is a variety of off-the-shelf solutions for this tasks (e.g., CORBA,[15] ICE,[16] OAA (Cheyer and Martin, 2001), and many more). What's more interesting is which messages we want to send. The following is a list of (informally specified) messages that realise the full spectrum of capabilities modelled in the IU approach. Keep in mind that the purpose of sending these messages is twofold: to achieve synchronisation between buffers, and at the same time to encapsulate what has changed; that is, the messages represent the Δ of the update.
First, from producer to consumer(s):

- *add this IU (full IU, all fields)*; this is, as noted above, the most basic type of information exchange, presenting a new increment for consumption. If parallel hypotheses are allowed in the system, the consumer has to look up to which of the active hypothesis chains the new IU is to be added. It can do this by checking the *successor* link on the new IU.

- *revoke this IU (IU ID)*; indicates that a producer withdraws support for a previously posted hypothesis.

- *reinstate this IU (IU ID)*; makes a previously revoked IU active again.

- *confidence change (IU ID, new value)*; notifying consumer(s) about a change of the confidence slot. (Note that if revocation is signalled via special values for confidence, then the previous two messages are only syntactic sugar for certain settings of this message.)

- *commit (IU ID)*; which is a guarantee to the consumer(s) that no update or revocation will be performed anymore on this IU.

- *no support (IU ID)*; this communicates to the consumer that an IU that the consumer wanted to be proven (that is, prediction; see below) could not be validated, and that the module has given up. (It is in some sense the 'negative' of a commitment.)

Consumers can send 'back' to producers the following messages:

- *used (IU ID)*; indicating that the consumer could make use of a given IU, and hence the larger hypothesis it is part of may be promising.

- *useless (IU ID)*; the consumer can show that no use can be made of the hypothesis path ending in this IU in the current context.

15. See e.g. `http://www.omg.org/gettingstarted/corbafaq.htm`.
16. `http://www.zeroc.com/`

- *commit (IU ID)*; the consumer has based an irreversible decision on this IU, e.g. by making an observable action, and so an exception should be raised if the producer wants to revoke it.
- *support (full IU)*; indicating that the consumer expects this to be the producer's output.

While it would also be possible for consumers to query producers for new messages, the more straightforward solution is to let the producers send such messages whenever they have made changes to their right buffers (i.e., on making the update step RB_t to RB'_t from Figure 7). Updates to a module can then be triggered by reception of these messages; i.e., the module is event-driven.

From the requirement stated at the beginning of this section that modules need to run on different machines it follows that they need to run in different processes. Again the straightforward solution then is to let the modules communicate asynchronously and run concurrently, with all the possible pitfalls that concurrent programming entails (see e.g. (van Roy and Haridi, 2004)).[17]

5.2.2 CLOSE COUPLING OF MODULES; SHARED MEMORY; EFFICIENT ENCODING OF UPDATES

Imagine a different scenario. We know that we are going to build all modules of the incremental system from scratch, and we know that we can do this within one programming language. Hence, we can make use of shared memory to realise buffers, the option sketched in Figure 11 (left). We also do not anticipate a need for prediction. In such a setting, we can use a compact alternative representation of IU networks, which makes communication more efficient.

In a shared memory setup, synchronisation of buffers is not an issue, as right buffers and left buffers are in fact the same memory space. Only the task of notifying consuming modules about what exactly changed remains, of what they have to look for when they access the shared memory. The idea here is now to reify positional information by superimposing a network of position nodes over the IU network, with the IUs being associated with edges in that network. These positional nodes then give us names for certain update stages, and so revisions can be efficiently encoded by reference to these nodes. An example can make this clearer. Figure 12 shows five update steps in the right buffer of an incremental ASR module. By reference to positional nodes, we can communicate easily a) what the newest (rightmost; as explained above in Section 3.4.1) committed IU is (we will call this NC for *newest committed*; indicated in the Figure as a shaded node) and b) what the newest non-revoked or active IU is (i.e., the right edge (RE); indicated in the Figure as a node with a dashed line). So, the change between the state at time t_1 and t_2 is signalled by RE taking on a different value. This value (w3) has not been seen before, and so the consuming module can infer that the network has been extended; it can find out which IUs have been added by going back from the new RE to the last previously seen position (in this case, w2; note that this would with no changes work for parallel hypothesis threads, if the positional network were a lattice). At t_3, a retraction of a hypothesis is signalled by a return to a previous state, w2. All consuming modules have to do now is to return to an internal state linked to this previous input state (more on this in a second). Finally, t_5 illustrates a commitment, where NC changes, and all IUs on the path from the new NC to the last committed IU now count as committed.

This representation style can be considered more parsimonious, as what in the approach from the previous section were three different messages (add, revoke, commit) is here implicitly expressed in

17. It would be interesting to formally analyse the potential for parallelisation in dialogue processing, with Petri-Nets or other modelling tools. We leave this to future work.

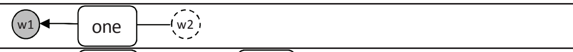
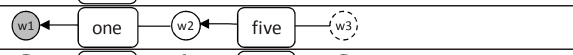
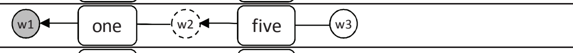
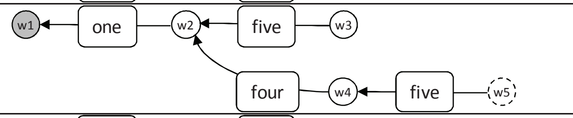
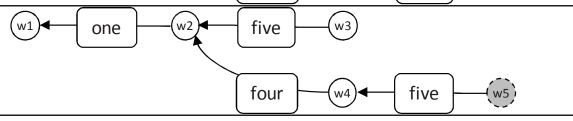
time	string	word buffer	update message
t_1	one		[w1, w2]
t_2	one five		[w1, w3]
t_3	one		[w1, w2]
t_4	one four five		[w1, w5]
t_5	[commit]		[w5,w5]

Figure 12: The Right Buffer of an ASR module, and update messages compactly representing revokes and commits

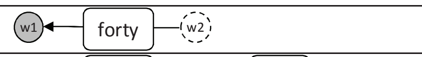
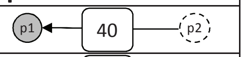
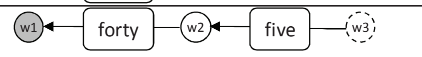
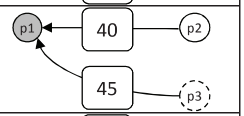
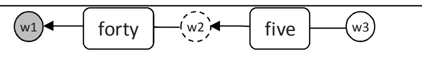
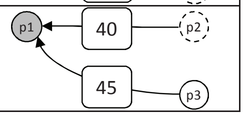
time	string	word buffer	parser buffer
t_1	forty		
t_2	forty five		
t_3	forty		

Figure 13: Connecting input states and output states of a buffer

changes to the pair (NC, RE). The status of an IU, and changes to it, are represented in the network and this pair, and so IUs themselves can be immutable (unchangeable) objects in this approach. This avoids the pitfalls of having concurrent modules altering and accessing the state of the IUs. Message size is always constant in this approach, even if comprehensive changes are made to a buffer. E.g., in t_4 in Figure 12, two IUs are added, but this still requires only one message, an update to what the label of the current right edge is. Also, coupling between modules can be tighter in this approach, as states of the consumer can be directly tied to states of the producer. This is illustrated in Figure 13. States of a parser (which is, for ease of presentation, not doing a very interesting job here, simply mapping number words to their logical representations as numbers) can be linked to the updates that created them, so that revision on the input state just requires going back to a previous output state. In the example, this happens in t_3, where the return of RE to w2 results in a return of the previously computed consumer state p2.

6. Conclusions and Future Work

We have presented a general, abstract model of incremental dialogue processing. The model is *general* in the sense that it describes elements that are essential to incremental processing, and hence can be expected to play a role in most, if not even all, systems performing such processing. It is *abstract* in the sense that it only describes properties of these elements and relations between them, but not concrete ways to instantiate them in computer implementations. We illustrated the notions developed here through the description of a number of existing systems in these terms, and we discussed some questions that arise when trying to build such systems.

In future work, we will attempt to describe more existing systems (such as (DeVault and Stone, 2003; Aist et al., 2006; Brick and Scheutz, 2007)) in the terms developed here, to more thoroughly investigate the coverage of our concepts. We are also currently exploring how more cognitively motivated models such as the model of speech generation by (Levelt, 1989) can be specified in our framework. A further direction for extension is the implementation of modality fusion as IU-processing. Lastly, we are now starting to work on connecting the model for incremental processing and grounding of interpretations in previous processing results described here with models of dialogue-level grounding in the information-state update tradition (Larsson and Traum, 2000). The first point of contact here will be the investigation of self-corrections, as a phenomenon that connects sub-utterance processing and discourse-level processing (Ginzburg et al., 2007).

Acknowledgements

This work was supported by a grant from DFG in the Emmy Noether Programme. Thanks to Timo Baumann and Okko Buß for discussions of the ideas reported here. We would also like to thank the anonymous reviewers of the precursor to this paper, (Schlangen and Skantze, 2009), and especially of this revised and extended version, for their very detailed and helpful comments.

References

Gregory Aist, James Allen, Ellen Campana, Lucian Galescu, Carlos A. Gomez Gallo, Scott Stoness, Mary Swift, and Michael K. Tanenhaus. Software architectures for incremental understanding of human speech. In *Proceedings of the International Conference on Spoken Language Processing*

(ICSLP), Pittsburgh, PA, USA, September 2006.

Gregory Aist, James Allen, Ellen Campana, Carlos Gomez Gallo, Scott Stoness, Mary Swift, and Michael K. Tanenhaus. Incremental understanding in human-computer dialogue and experimental evidence for advantages over nonincremental methods. In *Proceedings of Decalog 2007, the 11th International Workshop on the Semantics and Pragmatics of Dialogue*, Trento, Italy, 2007.

James Allen, George Ferguson, and Amanda Stent. An architecture for more realistic conversational systems. In *Proceedings of the conference on intelligent user interfaces*, Santa Fe, USA, June 2001.

Gerry Altmann and Mark Steedman. Interaction with context during human sentence processing. *Cognition*, 30:191–238, 1988.

Timo Baumann, Michaela Atterer, and David Schlangen. Assessing and improving the performance of speech recognition for incremental systems. In *Proceedings of the North American Chapter of the Association for Computational Linguistics - Human Language Technologies (NAACL HLT) 2009 Conference*, Boulder, Colorado, USA, May 2009.

Johan Boye, Beth Ann Hockey, and Manny Rayner. Asynchronous dialogue management: Two case-studies. In *Proceedings of the 4th Workshop on Semantics and Pragmatics of Dialogue (Götalog2000)*, pages 51–55, Gothenburg, Sweden, 2000.

Timothy Brick and Matthias Scheutz. Incremental natural language processing for HRI. In *Proceedings of the Second ACM IEEE International Conference on Human-Robot Interaction*, pages 263–270, Washington, DC, USA, 2007.

Timothy Brick, Paul Schermerhorn, and Matthias Scheutz. Speech and action: Integration of action and language for mobile robots. In *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '07)*, San Diego, CA, USA, October/November 2007.

Okko Buß and David Schlangen. Modelling sub-utterance phenomena in spoken dialogue systems. In *Proceedings of the 14th International Workshop on the Semantics and Pragmatics of Dialogue (Pozdial 2010)*, pages 33–41, Poznan, Poland, June 2010.

Okko Buß, Timo Baumann, and David Schlangen. Collaborating on utterances with a spoken dialogue system using an isu-based approach to incremental dialogue management. In *Proceedings of the SIGdial 2010 Conference*, pages 233–236, Tokyo, Japan, September 2010.

Adam Cheyer and David Martin. The open agent architecture. *Journal of Autonomous Agents and Multi-Agent Systems*, 4(1):143–148, March 2001. OAA.

Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of AAAI-88*, pages 49–54. AAAI, 1988.

David DeVault and Matthew Stone. Domain inference in incremental interpretation. In *Proceedings of ICOS 4: Workshop on Inference in Computational Semantics*, Nancy, France, September 2003. INRIA Lorraine.

David DeVault, Kenji Sagae, and David Traum. Can I finish? learning when to respond to incremental interpretation results in interactive dialogue. In *Proceedings of the 10th Annual SIGDIAL Meeting on Discourse and Dialogue (SIGDIAL'09)*, London, UK, September 2009.

Jens Edlund, Joakim Gustafson, Mattias Heldner, and Anna Hjalmarsson. Towards human-like spoken dialogue systems. *Speech Communication*, 50:630–645, 2008.

Jonathan Ginzburg, Raquel Fernández, and David Schlangen. Unifying self- and other-repair. In *Proceeding of DECALOG, the 11th International Workshop on the Semantics and Pragmatics of Dialogue (SemDial07)*, Trento, Italy, June 2007.

Anne Kilger and Wolfgang Finkler. Incremental generation for real-time applications. Technical Report RR-95-11, DFKI, Saarbrücken, Germany, 1995.

Staffan Larsson and David Traum. Information state and dialogue management in the TRINDI dialogue move engine toolkit. *Natural Language Engineering*, pages 323–340, 2000.

Willem J.M. Levelt. *Speaking*. MIT Press, Cambridge, USA, 1989.

Maryellen C. MacDonald. Probabilistic constraints and syntactic ambiguity resolution. *Language and Cognitive Processes*, 9(2):157–201, 1994.

William D. Marslen-Wilson. Linguistic structure and speech shadowing at very short latencies. *Nature*, 244:522–523, August 1973.

Joakim Nivre. Incrementality in deterministic dependency parsing. pages 50–57, Barcelona, Spain, July 2004.

David Schlangen. What we can learn from dialogue systems that don't work: On dialogue systems as cognitive models. In *Proceedings of DiaHolmia, the 13th International Workshop on the Semantics and Pragmatics of Dialogue (SEMDIAL 2009)*, pages 51–58, Stockholm, Sweden, June 2009.

David Schlangen and Gabriel Skantze. A general, abstract model of incremental dialogue processing. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2009)*, pages 710–718, Athens, Greece, March 2009.

David Schlangen, Timo Baumann, and Michaela Atterer. Incremental reference resolution: The task, metrics for evaluation, and a bayesian filtering model that is sensitive to disfluencies. In *Proceedings of SIGdial 2009, the 10th Annual SIGDIAL Meeting on Discourse and Dialogue*, London, UK, September 2009.

David Schlangen, Timo Baumann, Hendrik Buschmeier, Okko Buß, Stefan Kopp, Gabriel Skantze, and Ramin Yaghoubzadeh. Middleware for incremental processing in conversational agents. In *Proceedings of the SIGdial 2010 Conference*, pages 51–54, Tokyo, Japan, September 2010.

William Schuler, Stephen Wu, and Lane Schwartz. A framework for fast incremental interpretation during speech decoding. *Computational Linguistics*, 35(3), 2009.

Gabriel Skantze and Anna Hjalmarsson. Towards incremental speech generation in dialogue systems. In *Proceedings of the SIGdial 2010 Conference*, pages 1–8, Tokyo, Japan, September 2010.

Gabriel Skantze and David Schlangen. Incremental dialogue processing in a micro-domain. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2009)*, pages 745–753, Athens, Greece, March 2009.

Scott C. Stoness, James Allen, Greg Aist, and Mary Swift. Using real-world reference to improve spoken language understanding. In *Proceedings of Workshop on Spoken Language Understanding at AAAI05*, Pittsburgh, PA, USA, 2005.

Patrick Sturt and Vincenzo Lombardo. Processing coordinated structures: Incrementality and connectedness. *Cognitive Science*, 29:291–305, 2005.

Michael K. Tanenhaus, Michael J. Spivey-Knowlton, Kathleen M. Eberhard, and Julie C. Sedivy. Intergration of visual and linguistic information in spoken language comprehension. *Science*, 268, 1995.

D. Traum and P. Heeman. Utterance units in spoken dialogue. In E. Maier, M. Mast, and S. LuperFoy, editors, *Dialogue Processing in Spoken Language Systems*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1997.

Jos J.A. van Berkum, Arnout W. Koornneef, Marte Otten, and Mante S. Nieuwland. Establishing reference in language comprehension: An electrophysiological perspective. *Brain Research*, 1146:158–171, 2007.

Peter van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, Massachusetts, USA, 2004.

Jason Williams and Steve Young. Partially observable Markov decision processes for spoken dialog systems. *Computer Speech and Language*, 21(2):231–422, 2007.

Mats Wirén. *Studies in Incremental Natural Language Analysis*. PhD thesis, Linköping University, Linköping, Sweden, 1992.

S.J. Young, N.H. Russell, and J.H.S. Thornton. Token passing: a conceptual model for connected speech recognition systems. Technical report CUED/FINFENG/TR 38, Cambridge University Engineering Department, 1989.