A General, Abstract Model of Incremental Dialogue Processing

David Schlangen
Department of Linguistics
University of Potsdam, Germany
das@ling.uni-potsdam.de

Gabriel Skantze[*]
Dept. of Speech, Music and Hearing
KTH, Stockholm, Sweden
gabriel@speech.kth.se

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.

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 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 "Each 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.

Our aim in the work presented here is to describe and give names to the options available to designers of incremental systems. 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 sketch our work on one such system, our main focus here is on the conceptual framework. What we are *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).[1]

In the next section, we give some examples of differences in system architectures that we want to capture, with respect to the topology of the network of modules that make up the system, the way information flows through this network and how the modules process information, in particular how they deal with incrementality. In Section 3, we present the abstract model that underlies the system specifications, of which we give an example in Section 4. We close with a brief discussion of related work.

2 Motivating Examples

Figure 1 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 and the di-

[*] The work reported here was done while the second author was at the University of Potsdam.

[1] As we are also not trying to *prove* properties of the specified systems here, the formalisations we give are not supported by a formal semantics here.

rection of information flow. Arrows not coming from or going to modules represent the global input(s) and output(s) to and from the system.

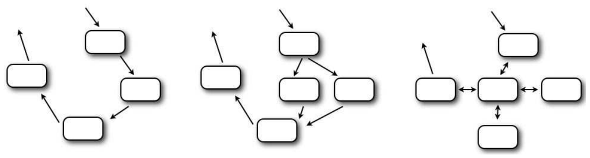
Figure 1: Module Network Topologies

One of our aims here is to facilitate exact and concise description of the differences between module networks such as 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.

A second desideratum is to be able to specify 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). The directed edges in Figure 1 indicate the direction of information flow (i.e., whose output is whose input); as an additional element, we can visualise *parallel* information streams between modules as in Figure 2 (left), where multiple hypotheses about the same input increments are passed on. (This isn't meant to imply that there are three actual communications channels active. As described below, we will encode the parallelism directly on the increments.)

One way such parallelism may occur in an incremental dialogue system is illustrated in Figure 2 (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). 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).

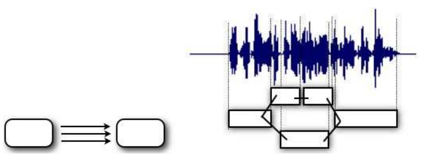
Figure 2: Parallel Information Streams (left) and Alternative Hypotheses (right)

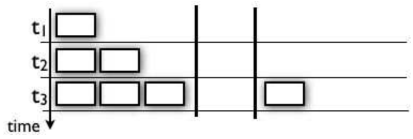

Figure 3: Incremental Input mapped to (less) incremental output

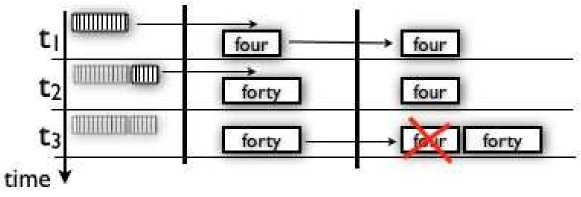

Figure 4: Example of Hypothesis Revision

We also want to be able to specify the ways incremental bits of input ("minimal amounts of characteristic input") can relate to incremental bits of output. Figure 3 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.

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. Figure 4 illustrates such a case. At time-step t_1, the available frames of acoustic features lead the processor, an automatic speech recogniser, to hypothesize that the word "four" has been spoken. This hypothesis is passed on as output. 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 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. Defining such operations and the conditions under which they are necessary is the final aim of our model.

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 basic operations needed for incremental processing; we will not say anything about the actual, module specific *payload* 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 4 above. IUs in LB (here, acoustic frames as input to an ASR) are *consumed* by the processor (i.e., is 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 our descriptions below, we will abstract away from the time processing takes 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 1). Together, the set of these axioms define 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 we are assuming token identity here, 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. This allows a special form of bi-directionality to be implemented, namely one where processors are allowed to make changes to IUs in their buffers, and where these changes automatically percolate through the network. This is different to and independent of the bi-directionality that can be expressed through connectedness axioms.

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 be 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 define what we see as the most complete version of IUs, which makes 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, we want to be able to represent relations between IUs produced by the same processor. For example, in the output of an ASR, two word-hypothesis IUs may stand in a *successor* relation, meaning that word 2 is what the ASR takes to be the continuation of the utterance begun with word 1. In a different situation, word 2 may be an alternative hypothesis about the same stretch of signal as word 1, and here a different relation would hold. The incremental outputs of a parser may be related in yet another way, through dominance: For example, a newly built IU_3, representing a VP, may want to express that it links via a dominance relation to IU$_1$, a V, and IU$_2$, an NP, which were both posted earlier. What is common to all relations of this type is that they relate IUs coming from the same processor(s); we will in this case say that the IUs are *on the same level*. Information about these *same level links* will be useful for the consumers of IUs. For example, a parsing module consuming ASR-output IUs will need to do different things depending on whether an incoming IU continues an utterance or forms an alternative hypothesis to a string that was already parsed.

The second relation between IUs that we want to capture cuts across levels, by linking RB-IUs to those LB-IUs that were used by the processor to produce them. For this we will say that the RB-IU is *grounded* in LB-IU(s). This relation then tracks the flow of information through the modules; 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. The network spanned by this relation will be useful in implementing the revision process mentioned above when discussing Figure 4, where the doubt about a hypothesis must spread to all hypotheses grounded in it.

Apart from these relations, we want IUs to carry three other types of information: a confidence score representing the confidence its producer had in it being accurate; a field recording whether revisions of the IU are still to be expected or not; and another field recording whether the IU has already been processed by consumers, and if so, by whom.

Formally, we define IUs as tuples $\mathcal{IU} = \langle \mathcal{I}, \mathcal{L}, \mathcal{G}, \mathcal{T}, \mathcal{C}, \mathcal{S}, \mathcal{P} \rangle$, where

- $\mathcal{I}$ is an identifier, which has to be unique for each IU over the lifetime of a system. (That is, at no point in the system's life can there be two or more IUs with the same ID.)
- $\mathcal{L}$ is the *same level link*, holding a statement about how, if at all, the given IU relates to other IUs at the same level, that is, to IUs produced by the same processor. If an IU is not linked to any other IU, this slot holds the special value $\top$.

  The definition demands that the same level links of all IUs belonging to the same larger unit form a graph; 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, whereas an n-best output might be better represented as a tree (with all first words linked to $\top$) or even a lattice (as in Figure 2 (right)); the output of a parser might require trees (possibly underspecified).
- $\mathcal{G}$ is the *grounded in* field, holding an ordered list of IDs pointing to those IUs out of which the current IU 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, the *grounded in* link always points to IUs from a previous level.[2] The transitive closure of this relation hence links system output IUs to a set of system input IUs. For convenience, we may define a predicate *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 4: 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, an empty *grounded in* field can also be used to trigger *prediction*: if an RB-IU has an empty *grounded in* field, 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) slot, 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

[2] The link to the previous level may be indirect. E.g., for an IU holding a phrase that is built out of previously built phrases (and not words), this link may be expressed by pointing to the same level link, meaning something like "I'm grounded in whatever the IUs are grounded in that I link to on the same level link, and also in the act of combination that is expressed in that same level link".

(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* field, holding a Boolean value that indicates whether the producing module has committed to the IU or not, that is, whether 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* field. In this field consuming 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; 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". 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 is the actual *payload*, the module-specific unit of 'characteristic input', which is what is processed by the processor in order to produce RB-IUs.

It will also be useful later to talk about the *completeness* of an IU (or of sets of IUs). This we define informally as its relation to (the type of) what would count as a *maximal* input or output of the module. For example, for an ASR module, such maximally complete input may be the recording of the whole utterance, for the parser maximal output may be a parse of type sentence (as opposed to one of type NP, for example).[3] This allows us to see non-incremental systems as a special case of incremental systems, namely those with only maximally complete IUs, which are always committed.

[3]This definition will only be used for abstractly classifying modules. Practically, it is of course rarely possible to know how complete or incomplete the already seen part of 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.

3.4 Modules

3.4.1 Operations

We describe in this section operations that the processors may perform on IUs. We leave open how processors are triggered into action, we simply assume that on receiving new LB-IUs or noticing changes to LB or RB-IUs, 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.

purge LB-IUs that are revoked by their producer (by having their confidence score set to the special value) 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.

update New LB-IUs are 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 3 above, and discussion below). The fields of the new RB-IUs (e.g., the *same level links* and the *grounded in* pointers) are filled appropriately. This is in some sense the basic operation of a processor, and must be implemented in all useful systems.

We can distinguish two implementation strategies for dealing with updates: a) all state is thrown away and results are computed again for the whole input set. The result must then be compared with the previous result to determine what the new output increment is. b) The new information is integrated into internal state, and only the new output increment is produced. For our purposes here, we can abstract away from these differences and assume that only actual increments are communicated. (Practically, it might be an advantage to keep using an existing processor and just wrap it into a module that computes increments by differences.)

We can also distinguish between modules along another dimension, namely based on which types of updates are allowed. To do so, we must first define the notion of a 'right edge' of a set of IUs. This is easiest to explain for strings, where the right edge simply is the end of the string, or for a lattice, where it is the (set of) smallest element(s). A similar notion may be defined for trees as well (compare the 'right frontier constraint' of Polanyi (1988)). If now a processor only expects IUs that extend the right frontier, we can follow Wirén (1992) in saying that it is only *left-to-right incremental*. Within what Wirén (1992) calls *fully incremental*, we can make more distinctions, namely according to whether revisions (as described above) and/or *insertions* are allowed. The latter can easily be integrated into our framework, by allowing *same-level links* to be changed to fit new IUs into existing graphs.

Processors can take *supports* information into account when deciding on their update order. 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.[4]

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, 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.)

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

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 modules 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. For example, if the system has committed to a certain interpretation by making a publicly observable action (e.g., an utterance, or a multi-modal action), 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.[5]

3.4.2 Characterising Module Behaviour

It is also useful to be able to abstractly describe the relation between LB-IUs and RB-IUs in a module or a collection of modules. We do this here along the dimensions *update frequency*, *connectedness* and *completeness*.

Update Frequency The first dimension we consider here is that of 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

[5]In future work, we will explore in more detail if and how through the implementation of a self-monitoring cycle and *commits* and *revokes* the various types of dysfluencies described for example by Levelt (1989) can be modelled.

(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$\geq$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 3 above.

f:in$\leq$out characterises modules that update RB *more* often than their LB is updated. This could happen in modules that produce endogenic 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 via *same level links* to previously output IUs, 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$\geq$out* behaviour.[6]

Completeness Building on the notion of completeness of (sets of) IUs introduced above, we can also characterise modules according to how the completeness of LB and RB relates.

In a *c:in=out*-type module, the most complete RB-IU (or set of RB-IUs) is only as complete as the most complete (set of) LB-IU(s). 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$\geq$out*-type module, the most complete RB-IU potentially lags behind the most complete LB-IU. This will typically be the case in *f:in$\geq$out* modules. *c:in$\leq$out*-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*.

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

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$\geq$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$\geq$in* behaviour.)

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 3.4.1 the module implements, and a characterisation of its behaviour in the terms of Section 3.4.2.
- A set of axioms describing the connections between module buffers (and hence the network topology), as explained in Section 3.2.
- Specifications of the format of the IUs that are produced by each module, in terms of the definition of slots in Section 3.3.

4 Example Specification

We have built a fully incremental dialogue system, called NUMBERS (for more details see Skantze and Schlangen (2009)), that can engage in dialogues in a simple domain, number dictation. The system can not only be described in the terms explained here, but it also directly instantiates some of the data types described here.

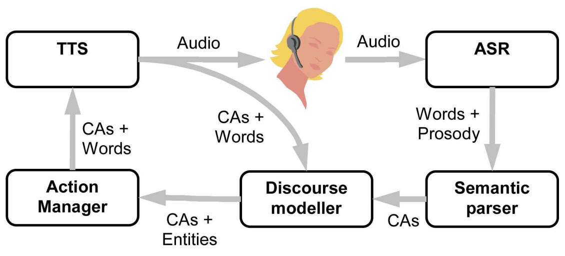

Figure 5: The NUMBERS System Architecture (CA = communicative act)

The module network topology of the system is shown in Figure 5. 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 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 ungrounded information is put on RBs). All modules have a *f:in≥out*; *c:in≥out* characteristic.

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.

5 Related Work, Future 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 were inspired by Kilger and Finkler (1995), but again we generalised the definitions to fit all kinds of incremental modules, not just generation.

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

In future work, we will give descriptions of these systems in the terms developed here. We are also currently exploring how more cognitively motivated models such as that of generation by Levelt (1989) can be specified in our model. 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).

Acknowledgments This work was funded by a grant in the DFG Emmy Noether Programme. Thanks to Timo Baumann and Michaela Atterer for discussion of the ideas reported here, and to the anonymous reviewers for their very detailed and helpful comments.

References

G.S. Aist, J. Allen, E. Campana, L. Galescu, C.A. Gomez Gallo, S. Stoness, M. Swift, and M Tanenhaus. 2006. Software architectures for incremental understanding of human speech. In *Proceedings of the International Conference on Spoken Language Processing (ICSLP)*, Pittsburgh, PA, USA, September.

Timothy Brick and Matthias Scheutz. 2007. 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.

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

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

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

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

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

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

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

Livia Polanyi. 1988. A formal model of the structure of discourse. *Journal of Pragmatics*, 12:601–638.

Gabriel Skantze and David Schlangen. 2009. 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)*, Athens, Greece, April.

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

D. Traum and P. Heeman. 1997. 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.

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

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