InproTK$_S$: A Toolkit for Incremental Situated Processing

Casey Kennington
CITEC, Dialogue Systems
Group, Bielefeld University
ckennington[1]

Spyros Kousidis
Dialogue Systems Group
Bielefeld University
spyros.kousidis[2]

David Schlangen
Dialogue Systems Group
Bielefeld University
david.schlangen[2]

[1]@cit-ec.uni-bielefeld.de
[2]@uni-bielefeld.de

Abstract

In order to process incremental situated dialogue, it is necessary to accept information from various sensors, each tracking, in real-time, different aspects of the physical situation. We present extensions of the incremental processing toolkit INPROTK which make it possible to plug in such multimodal sensors and to achieve situated, real-time dialogue. We also describe a new module which enables the use in INPROTK of the Google Web Speech API, which offers speech recognition with a very large vocabulary and a wide choice of languages. We illustrate the use of these extensions with a description of two systems handling different situated settings.

1 Introduction

Realising incremental processing of speech in- and output – a prerequisite to interpretation and possibly production of speech concurrently with the other dialogue participant – requires some fundamental changes in the way that components of dialogue systems operate and communicate with each other (Schlangen and Skantze, 2011; Schlangen and Skantze, 2009). Processing situated communication, that is, communication that requires reference to the physical setting in which it occurs, makes it necessary to accept (and fuse) information from various different sensors, each tracking different aspects of the physical situation, making the system multimodal (Atrey et al., 2010; Dumas et al., 2009; Waibel et al., 1996).

Incremental situated processing brings together these requirements. In this paper, we present a collection of extensions to the incremental processing toolkit INPROTK (Baumann and Schlangen, 2012) that make it capable of processing situated communication in an incremental fashion: we have developed a general architecture for plugging in multimodal sensors whith we denote INPROTK$_S$, which includes instantiations for motion capture (via *e.g. via Microsoft Kinect* and *Leap Motion*) and eye tracking (*Seeingmachines FaceLAB*). We also describe a new module we built that makes it possible to perform (large vocabulary, open domain) speech recognition via the Google Web Speech API. We describe these components individually and give as use-cases in a driving simulation setup, as well as real-time gaze and gesture recognition.

In the next section, we will give some background on incremental processing, then describe the new methods of plugging in multimodal sensors, specifically using XML-RPC, the Robotics Service Bus, and the InstantReality framework. We then explain how we incorporated the Google Web Speech API into InproTK, offer some use cases for these new modules, and conclude.

2 Background: The IU model, INPROTK

As described in (Baumann and Schlangen, 2012), INPROTK realizes the *IU*-model of incremental processing (Schlangen and Skantze, 2011; Schlangen and Skantze, 2009), where incremental systems consist of a network of processing *modules*. A typical module takes input from its *left buffer*, performs some kind of processing on that data, and places the processed result onto its *right buffer*. The data are packaged as the payload of *incremental units* (IUs) which are passed between modules.

The IUs themselves are also interconnected via so-called *same level links* (SLL) and *grounded-in links* (GRIN), the former allowing the linking of IUs as a growing sequence, the latter allowing that sequence to convey what IUs directly affect it (see Figure 1 for an example). A complication particular to incremental processing is that modules can "change their mind" about what the best hypothe-

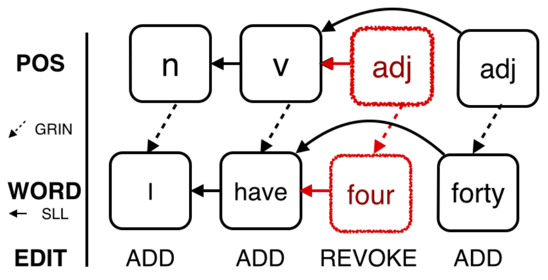

Figure 1: Example of IU network; part-of-speech tags are grounded into words, tags and words have same level links with left IU; *four* is revoked and replaced with *forty*.

sis is, in light of later information, thus IUs can be *added*, *revoked*, or *committed* to a network of IUs.

INPROTK determines how a module network is "connected" via an XML-formatted configuration file, which states module instantiations, including the connections between left buffers and right buffers of the various modules. Also part of the toolkit is a selection of "incremental processing-ready" modules, and so makes it possible to realise responsive speech-based systems.

3 InproTK and new I/O: InproTK$_S$

The new additions introduced here are realised as INPROTK$_S$ modules. The new modules that input information to an INPROTK$_S$ module network are called *listeners* in that they "listen" to their respective message passing systems, and modules that output information from the network are called *informers*. Listeners are specific to their method of receiving information, explained in each section below. Data received from listeners are packaged into an IU and put onto the module's right buffer. Listener module left buffers are not used in the standard way; left buffers receive data from their respective message passing protocols. An informer takes all IUs from its left buffer, and sends their payload via that module's specific output method, serving as a kind of right buffer. Figure 2 gives an example of how such listeners and informers can be used. At the moment, only strings can be read by listeners and sent by informers; future extensions could allow for more complicated data types.

Listener modules add new IUs to the network; correspondingly, further modules have to be designed in instatiated systems then can make use of these information types. These IUs created by the listeners are linked to each other via SLLs. As with audio inputs in previous version of INPROTK, these IUs are considered *basedata* and not explictly linked via GRINs in the sensor data. The modules defined so far also simply *add* IUs and do not revoke.

We will now explain the three new methods of getting data into and out of INPROTK$_S$.

3.1 XML-RPC

XML-RPC is a *remote procedure call* protocol which uses XML to encode its calls, and HTTP as a transport mechanism. This requires a server/client relationship where the listener is implemented as the server on a specified port.[1] Remote sensors (e.g., an eye tracker) are realised as clients and can send data (encoded as a string) to the server using a specific procedural call. The informer is also realised as an XML-RPC client, which sends data to a defined server. XML-RPC was introduced in 1998 and is widely implemented in many programming languages.

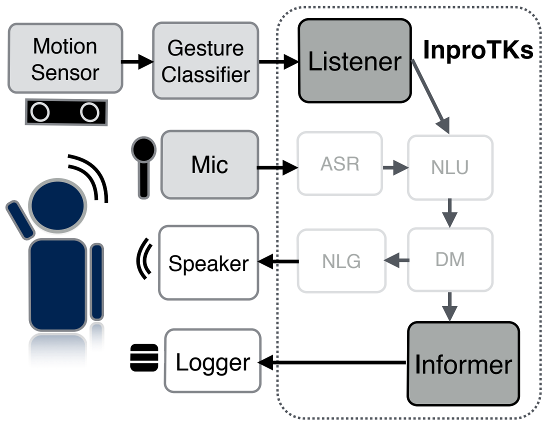

Figure 2: Example architecture using new modules: motion is captured and processed externally and class labels are sent to a listener, which adds them to the IU network. Arrows denote connections from right buffers to left buffers. Information from the DM is sent via an Informer to an external logger. External gray modules denote input, white modules denote output.

3.2 Robotics Service Bus

The *Robotics Service Bus* (RSB) is a middleware environment originally designed for message-passing in robotics systems (Wienke and Wrede, 2011).[2] As opposed to XML-RPC which requires

[1]The specification can be found at `http://xmlrpc.scripting.com/spec.html`

[2]`https://code.cor-lab.de/projects/rsb`

point-to-point connections, RSB serves as a *bus* across specified transport mechanisms. Simply, a network of communication nodes can either *inform* by sending events (with a payload), or *listen*, i.e., receive events. Informers can send information on a specific *scope* which establishes a visibility for listeners (e.g., a listener that receives events on scope /one/ will receive all events that fall under the /one/ scope, whereas a listener with added constants on the scope, e.g., /one/two/ will not receive events from different added constants /one/three/, but the scope /one/ can listen on all three of these scopes). A listener module is realised in INPROTK$_S$ by setting the desired scope in the configuration file, allowing INPROTK$_S$ seamless interconnectivity with communication on RSB.

There is no theoretical limit to the number of informers or listeners; events from a single informer can be received by multiple listeners. Events are typed and any new types can be added to the available set. RSB is under active development and is becoming more widely used. Java, Python, and C++ programming languages are currently supported. In our experience, RSB makes it particularly convenient for setting distributed sensor processing networks.

3.3 InstantReality

In (Kousidis et al., 2013), the *InstantReality* framework, a virtual reality environment, was used for monitoring and recording data in a real-time multimodal interaction.[3] Each information source (sensor) runs on its own dedicated workstation and transmits the sensor data across a network using the *InstantIO* interface. The data can be received by different components such as InstantPlayer (3D visualization engine; invaluable for monitoring of data integrity when recording experimental sessions) or a logger that saves all data to disk. Network communication is achieved via multicast, which makes it possible to have any number of listeners for a server and vice-versa.

The InstantIO API is currently available in C++ and Java. It comes with a non-extensible set of types (primitives, 2D and 3D vectors, rotations, images, sounds) which is however adequate for most tracking applications. InstantIO listeners and informers are easily configured in INPROTK$_S$ configuration file.

[3]http://www.instantreality.org/

3.4 Venice: Bridging the Interfaces

To make these different components/interfaces compatible with each other, we have developed a collection of bridging tools named *Venice*. Venice serves two distinct functions. First, *Venice.*HUB, which pushes data to/from any of the following interfaces: disk (logger/replayer), InstantIO, and RSB. This allows seamless setup of networks for logging, playback, real-time processing (or combinations; e.g, for simulations), minimizing the need for adaptations to handle different situations. Second, *Venice.*IPC allows interprocess communication and mainly serves as a quick and efficient way to create network components for new types of sensors, regardless of the platform or language. Venice.IPC acts as a server to which TCP clients (a common interface for sensors) can connect. It is highly configurable, readily accepting various sensor data outputs, and sends data in real-time to the InstantIO network.

Both Venice components operate on all three major platforms (Linux, Windows, Mac OS X), allowing great flexibility in software and sensors that can be plugged in the architecture, regardless of the vendor's native API programming language or supported platform. We discuss some use cases in section 5.

4 Google Web Speech

One barrier to dialogue system development is handling ASR. Open source toolkits are available, each supporting a handful of languages, with each language having a varying vocabulary size. A step in overcoming this barrier is "outsourcing" the problem by making use of the Google Web Speech API.[4] This interface supports many languages, in most cases with a large, open domain of vocabulary. We have been able to access the API directly using INPROTK$_S$, similar to (Henderson, 2014).[5] INPROTK$_S$ already supports an incremental variant of Sphinx4; a system designer can now choose from these two alternatives.

At the moment, only the Google Chrome browser implements the Web Speech API. When the INPROTK$_S$ Web Speech module is invoked, it creates a service which can be reached from

[4]The Web Speech API Specificiation: https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html

[5]Indeed, we used Matthew Henderson's *webdial* project as a basis: https://bitbucket.org/matthen/webdialog

the Chrome browser via an URL (and hence, microphone client, dialogue processor and speech recogniser can run on different machines). Navigating to that URL shows a simple web page where one can control the microphone. Figure 3 shows how the components fit together.

While this setup improves recognition as compared to the Sphinx4-based recognition previously only available in INPROTK, there are some areas of concern. First, there is a delay caused by the remote processing (on Google's servers), requiring alignment with data from other sensors. Second, the returned transcription results are only 'semi-incremental'; sometimes chunks of words are treated as single increments. Third, n-best lists can only be obtained when the API detects the end of the utterance (incrementally, only the top hypothesis is returned). Fourth, the results have a crude timestamp which signifies the end of the audio segment. We use this timestamp in our construction of word IUs, which in informal tests have been found to be acceptable for our needs; we defer more systematic testing to future work.

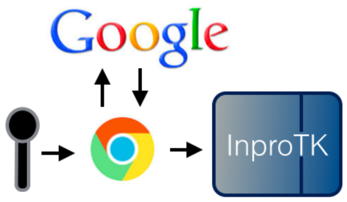

Figure 3: Data flow of Google Web Speech API: Chrome browser controls the microphone, sends audio to API and receives incremental hypotheses, which are directly sent to InproTK$_S$.

5 INPROTK$_S$ in Use

We exemplify the utility of INPROTK$_S$ in two experiments recently performed in our lab.

In-car situated communication We have tested a "pause and resume" strategy for adaptive information presentation in a driving simulation scenario (see Figure 4), using INPROTK$_S$ and OpenDS (Math et al., 2013). Our dialogue manager – implemented using OpenDial (Lison, 2012) – receives trigger events from OpenDS in order to update its state, while it verbalises calendar events and presents them via speech. This is achieved by means of InstantIO servers we integrated into OpenDS and respective listeners in INPROTK$_S$. In turn, InstantIO informers send data that is logged by Venice.HUB. The results of this study are published in (Kousidis et al., 2014). Having available the modules described here made it surprisingly straightforward to implement the interaction with the driving simulator (treated as a kind of sensor).

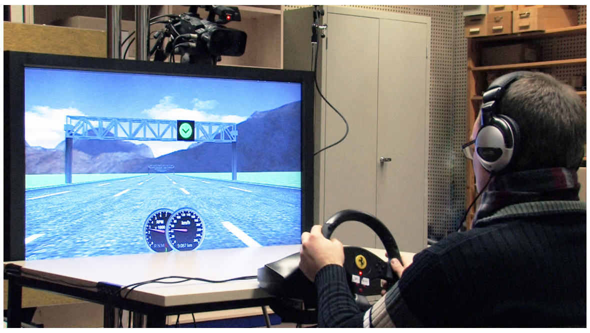

Figure 4: Participant performing driving test while listening to iNLG speech delivered by InProTK$_S$.

Real-time gaze fixation and pointing gesture detection Using the tools described here, we have recently tested a real-time situated communication environment that uses speech, gaze, and gesture simultaneously. Data from a *Microsoft Kinect* and a *Seeingmachines Facelab* eye tracker are logged in realtime to the InstantIO network. A Venice.HUB component receives this data and sends it over RSB to external components that perform detection of gaze fixation and pointing gestures, as described in (Kousidis et al., 2013). These class labels are sent in turn over RSB to INPROTK$_S$ listeners, aggregating these modalities with the ASR in a language understanding module. Again, this was only enabled by the framework described here.

6 Conclusion

We have developed methods of providing multimodal information to the incremental dialogue middleware INPROTK. We have tested these methods in real-time interaction and have found them to work well, simplifying the process of connecting external sensors necessary for multimodal, situated dialogue. We have further extended its options for ASR, connecting the Google Web Speech API. We have also discussed Venice, a tool for bridging RSB and InstantIO interfaces, which can log real-time data in a time-aligned manner, and replay that data. We also offered some use-cases for our extensions.

INPROTK$_S$ is freely available and accessible.[6]

[6]https://bitbucket.org/inpro/inprotk

Acknowledgements Thank you to the anonymous reviewers for their useful comments and to Oliver Eickmeyer for helping with InstantReality.

References

Pradeep K. Atrey, M. Anwar Hossain, Abdulmotaleb El Saddik, and Mohan S. Kankanhalli. 2010. *Multimodal fusion for multimedia analysis: a survey*, volume 16. April.

Timo Baumann and David Schlangen. 2012. The InproTK 2012 Release. In *NAACL*.

Bruno Dumas, Denis Lalanne, and Sharon Oviatt. 2009. Multimodal Interfaces : A Survey of Principles , Models and Frameworks. In *Human Machine Interaction*, pages 1–25.

Matthew Henderson. 2014. The webdialog Framework for Spoken Dialog in the Browser. Technical report, Cambridge Engineering Department.

Spyros Kousidis, Casey Kennington, and David Schlangen. 2013. Investigating speaker gaze and pointing behaviour in human-computer interaction with the mint.tools collection. In *SIGdial 2013*.

Spyros Kousidis, Casey Kennington, Timo Baumann, Hendrik Buschmeier, Stefan Kopp, and David Schlangen. 2014. Situationally Aware In-Car Information Presentation Using Incremental Speech Generation: Safer, and More Effective. In *Workshop on Dialog in Motion, EACL 2014*.

Pierre Lison. 2012. Probabilistic Dialogue Models with Prior Domain Knowledge. In *Proceedings of the 13th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 179–188, Seoul, South Korea, July. Association for Computational Linguistics.

Rafael Math, Angela Mahr, Mohammad M Moniri, and Christian Müller. 2013. OpenDS: A new open-source driving simulator for research. *GMM-Fachbericht-AmE 2013*.

David Schlangen and Gabriel Skantze. 2009. A General, Abstract Model of Incremental Dialogue Processing. In *Proceedings of the 10th EACL*, number April, pages 710–718, Athens, Greece. Association for Computational Linguistics.

David Schlangen and Gabriel Skantze. 2011. A General, Abstract Model of Incremental Dialogue Processing. *Dialoge & Discourse*, 2(1):83–111.

Alex Waibel, Minh Tue Vo, Paul Duchnowski, and Stefan Manke. 1996. Multimodal interfaces. *Artificial Intelligence Review*, 10(3-4):299–319.

Johannes Wienke and Sebastian Wrede. 2011. A middleware for collaborative research in experimental robotics. In *System Integration (SII), 2011 IEEE/SICE International Symposium on*, pages 1183–1190.