An open source process engine framework for realtime pattern recognition and information fusion tasks

Volker Fritzsch, Stefan Scherer, Friedhelm Schwenker

Institute of Neural Information Processing
Ulm University
firstname.lastname@uni-ulm.de

Abstract

The process engine for pattern recognition and information fusion tasks, the *pepr framework*, aims to empower the researcher to develop novel solutions in the field of pattern recognition and information fusion tasks in a timely manner, by supporting reuse and combination of well tested and established components in an environment, that eases the wiring of distinct algorithms and description of the control flow through graphical tooling. The framework, not only consisting of the runtime environment, comes with several highly useful components that can be leveraged as a starting point in creating new solutions, as well as a graphical process builder that allows for easy development of pattern recognition processes in a graphical, modeled manner. Additionally, numerous work has been invested in order to keep the entry barrier with regards to extending the framework as low as possible, enabling developers to add additional functionality to the framework in as less time as possible.

1. Introduction

We are happy to introduce a novel process engine for pattern recognition and information fusion tasks, the *pepr framework*. The process engine allows the development of applications in an abstraction level above that of common programming languages, therefore reducing the time to develop such solutions by allowing the reuse of established and well tested components. The framework is designed to be highly expandable in order to allow easy adaption to unique challenges and comes with a user interface that enables the creation of solutions in a graphical, modeled, manner. The framework is now freely available on the web[1]: `www.pepr-framework.org`, and is published under the Apache open source license allowing expansion and usage to everybody. Furthermore, some first applications have been published and demonstrated at a few selected renowned conferences: (Scherer et al., 2009a; Scherer et al., 2009b) showing only a few of the possibilities of this framework in the context of conversation analysis.

In short the main benefits of the engine can be summarized as follows: First of all the framework is entirely developed in Java and currently provides installation instructions for Mac OS X and Windows[2]. Therefore, it is as flexible as possible and personalized components are easy to develop. Further system specifications are described in Section 2. Additionally, the *pepr framework* enables the developer to create classification processes and prototypes in a timely manner due to the highly productive graphical process builder, which will be introduced in Section 3. Furthermore, there is already a set of well tested and established components available from previous work, comprising interfaces to large scale libraries such as *OpenCV*, or *libsvm* allowing access to even more established functionality in the area of computer vision and pattern recognition. Additionally, basic starter activities enabling input from different modalities such as cameras, microphones, and text files are already implemented to provide an easy entry to the framework. In Section 4. one example application is introduced in order to get a feeling of the basic capabilities. And last but not least, one of the main novel features of the framework is the possibility to scale processes over each available core in one machine as well as over multiple machines rendering the framework useful in large scale and performance critical realtime applications.

[1]The page is currently under construction and tutorials as well as video screen-casts on the usage of the framework will follow shortly.

[2]Linux will follow shortly.

2. System architecture

The conceptual context of the *pepr framework* is formed by **Processes**, **Process Instances**, **Context Tokens** and **Components**, such as **Activities** and **Starters**. While Activities and Starters are the building blocks that contain algorithms and interfaces to external sensors and data, the **Process** brings those components into a well defined order. It is the blueprint for the interaction of the aforementioned components. During runtime, whenever a Starter Component signals the *pepr framework* available output, it instantiates the Process (Process Instance) and creates a Context Token (Figure 1).

This Context Token, which acts as a container for data generated by the Components that form the Process, is then passed to the Process Instance and wanders from Component to Component, until the specific Process Instance is terminated when the Context Token leaves an Activity that has no succeeding Component. Each Activity in its path is activated by the incoming Context Token, assembles its necessary input data based on the output attached to the Token by preceding Activities and Starters, executes the encapsulated algorithms on this data and attaches the computed output to the Token (Figure 2).

While a specific Process can consist of more than one Starter, this does not lead to more than one Context Token in the resulting Process Instance. Instead, several Process

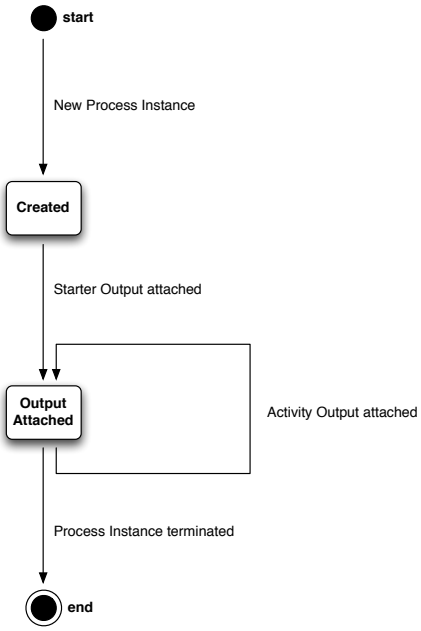

Figure 1: Process Instance and Context Token

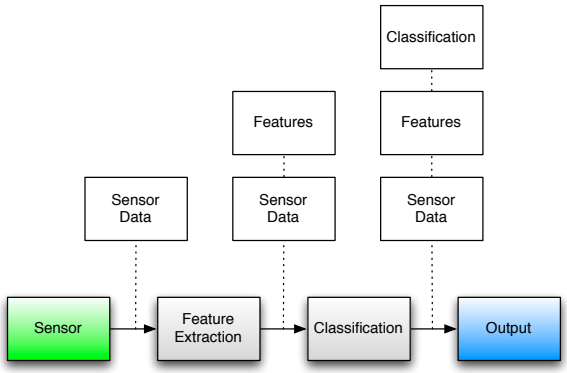

Figure 2: Output attached to the Context Token

Instances are created by the engine and those instances, described through their Context Token, are later merged by a Fusion Activity (Figure 3). This Fusion Activity can merge different Context Tokens based on, for example, the difference of the timestamp added to the Context Token during its creation or on other aspects, like both originating from the same Context Token, due to their creation being the result of a fork (Figure 4) in the control flow of the Process Instance.

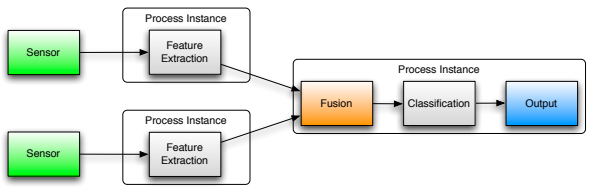

Figure 3: Fusion of Context Tokens

Several design goals influenced this approach. While modularity and scalability led to the architectural decision to base the framework on the Actor Model / Message Passing as described by (Agha, 1986), in order to hide the complexity of concurrent programming from the developer of novel components, the goal to keep the framework as open as possible in regards to future enhancements and applications drove the decision to forgo a predefined data model for information interchange by the involved Components. Instead, a solution was chosen that allows the designer of a Process to assemble the input for a given Component by mapping parts of the output of preceding Activities and Starters onto the input specified by this Component. This **Input Mapping** (Figure 5) is scripted by the Process Designer with help of the Graphical Process Builder and executed for each invocation of an Activity.

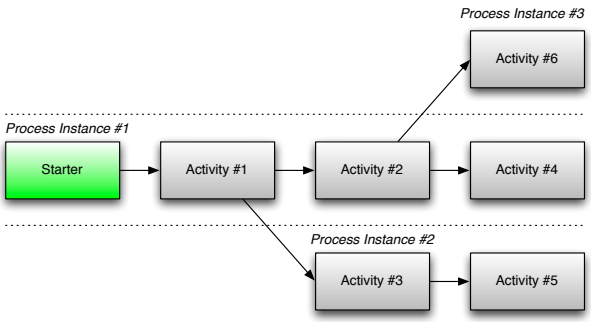

Figure 4: Forking of Process Instances

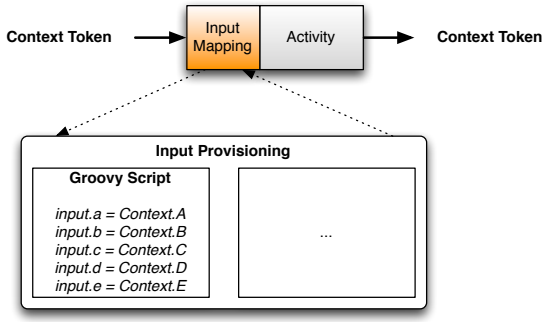

Figure 5: Input Mapping

3. Development of processes

3.1. Developing activities

The *pepr framework*, based on the OSGi Framework, is a highly expandable system. Even though there is already a useful collection of Activities and Starters in place, if choosing the *pepr framework* to solve a specific problem, the need might arise to extend it and create new activities and or starters to adapt it to a new problem.
It is now possible to create a new class that extends Activity – the base class for all activities. Activity is a generic class that takes three type parameters:

- `InputType`
- `OutputType`

- `ConfigurationType`

The `InputType` is an Object that is passed to the activity on each invocation, its member variables holding the data types necessary for the activities calculation.
The `OutputType` is the type of Object that the activity returns after a successful invocation to the engine and which is then added to the context.
The `ConfigurationType` contains information needed for initialization of the new activity, e.g. pointers to external resources like sockets, host names and other constants. It is configured by the user during the process design phase with the graphical process builder as described in Section 3.2.
With this classes implemented and added as generic type parameters to the new Activity class, solely the following method must be overwritten:

- `public OutputType handleMessage(InputType msg)`

Additionally, if special initialization (and termination) of the Activity is needed, the developer can choose to overwrite the following methods which are part of an Activities life-cycle:

- `public void onInitialisation()`
- `public void onTermination()`

As the source code of the *pepr framework* is already available, developers of novel components are advised to refer to the implementations of similar Activities and Starters for examples and reference.

3.2. Building and running a process

In order to be able to run the process engine one has to design a suitable process, this is enabled in the *pepr framework* with a bundled graphical process builder rendering the task easy to handle, and as transparent as possible. The graphical process builder (Figure 6) is based on the *Eclipse Platform* and the *Graphical Editing Framework*[3]. While

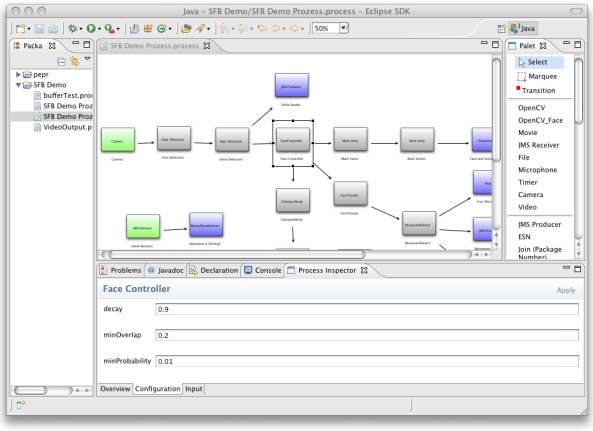

Figure 6: Screenshot of the Graphical Process Builder

Eclipse is probably most widely known for the Java Integrated Development Environment (IDE) by the same name, it provides a complete platform for Rich Client Applications with the Eclipse Java IDE being the most famous one. During the development of the *pepr framework*, the need for graphical aided process design arose quickly, as writing process descriptions by hand was time consuming, difficult, and error prone. By basing the Graphical Process Builder on the Eclipse Platform, it was possible to achieve a native look and feel on the three most important platforms today (Windows, Mac OS X and Linux GTK) and cut the overall development time down.
Using the bundled process builder the task to design a process is basically reduced to dragging and dropping activities, resembled by boxes of different colors[4] as shown in Figure 6. After having setup the proper activities one needs to combine their outputs and inputs using the so called transition tool resulting in visible arrows on the builder's editor frame. However, in order to enrich the possibilities of the combination of the activities we decided to introduce programmable inputs for each of the activities. Each of the activities can map the output of all the previous activities using the functionality of Groovy script[5] including all the datatypes and some functionality of Java including lists, or if- and for-clauses. In a last step the activities have to be configured adjusting for example the location of source files, boolean values, or other alphanumeric parameters.
Finally, the user interface allows the process developer to execute the process in development without leaving the application. Processes can be started either via the context menu (*run as pepr process*) or by using the toolbar.
This bootstraps a new instance of the *pepr framework*, which immediately loads and executes the process. Status information and output are captured and redirected to the console panel. The console panel also provides the ability to stop running instances.

4. Use case: SFB Demonstration

In June 2009, pepr was shown at the opening ceremony of the *SFB/Transregio 62 - Eine Companion-Technologie fr kognitive technische Systeme* at Ulm University. The scope of the SFB Demonstration was to show **Sensor Fusion** and communication between multiple instances of the *pepr framework*, running on different machines. The chosen setup consisted of two laptops, running both an instance of the process shown in Figure 8. Albeit the same process, each one was configured different to send and receive external data from the other machine.
Input was taken solely from the machines built-in camera, capturing the face and upper body of the user sitting in front of it. These signals were evaluated on the local machine by leveraging the OpenCV activity. The output was displayed (Screenshot 9) and compared to the movement tracked in earlier frames, therefore providing access to the movement of the head as shown in the succeeding diagram in Screenshot 10.
As the described process was executed on two separate machines, features extracted from the camera were distributed

[3] `www.eclipse.org/gef/`

[4] Colors indicate different types of activities: starters, outputs, joins, and generic activities. The differences and features of these will be introduced in much more detail in the full paper.

[5] http://groovy.codehaus.org/

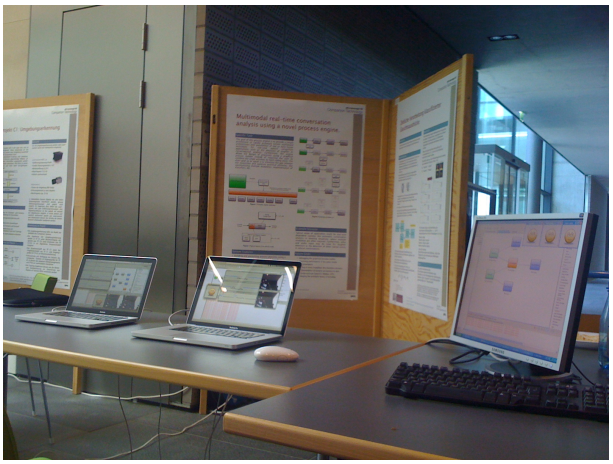

Figure 7: Photo of the pepr SFB Demo Booth

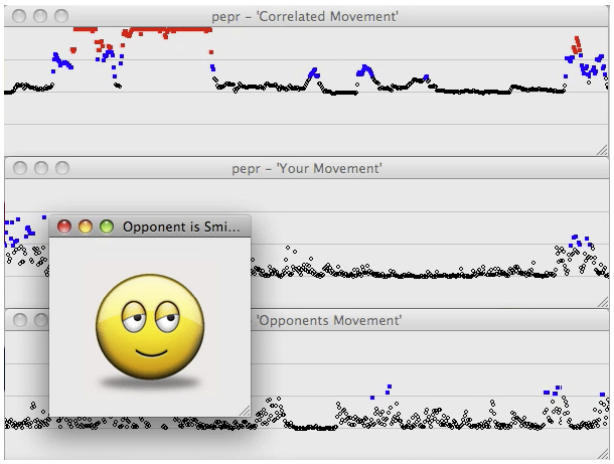

Figure 10: Screenshot of Graphical Plotters and Smiley

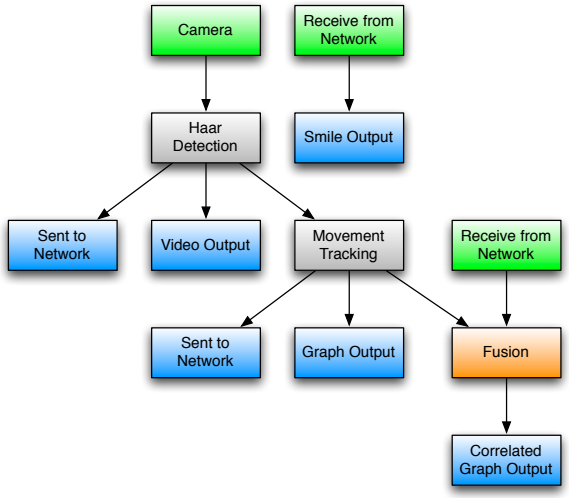

Figure 8: Simplified Diagram of the SFB Demo Process

over the network and could be used as additional input in each process. Thus, enabling correlation of movement and indication of emotions of the user in front of the other machine, e.g. a smiling user. In order to achieve this, two distinct feature sets were distributed. First, a sole bit to indicate if the process detected a smile in the users face and second the current movement of the user sitting in front of the local machine. The process running on the other machine received this data and displayed a smiling- / non-smiling icon based on the received features. Additionally, it

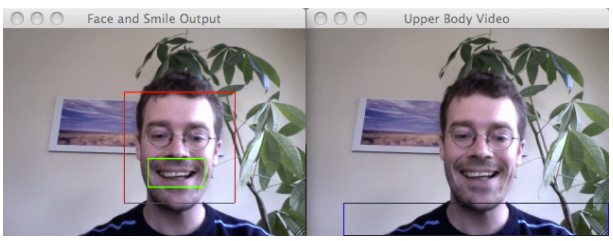

Figure 9: Screenshot of Face and Upper Body Tracking Video Output

correlated the received movement data with the local computed movement data to measure some kind of synchronous movement of both users, which is an indication for engagement of the participant in the conversation or for active listening. In order for this to work, input of two different starter nodes had to be merged. Therefore, time based fusion was selected and proved to be more then accurate, given the rather small network and the real-time environment.

5. Conclusions

The goal set for this work, to create a process engine framework suitable for pattern recognition and information fusion tasks has been reached, as the public demonstrations, and feedback from test users have shown. The *pepr framework* in combination with the graphical process builder form a robust and expandable foundation that helps to solve tasks in this field in a simple and timely manner. While being easy to use, the engine scales well over multiple cores and machines, thanks to the initial design. Therefore, even computationally intensive tasks can be solved with the framework.

The Graphical Process Builder, although not initially planned to be part of this framework, helps to further cut down process development time and is a great help to demonstrate and explain processes, thanks to its graphical modeling capabilities.

The flexibility in process design and adaption to different scenarios has been proven in the use-cases and public demonstrations. It is possible to adjust existing processes to changed guidelines and environments within minutes. Even the creation of working processes from scratch could be demonstrated live in a timely manner.

Over the course of the last months we decided to open up Activity development to other scientists and students from the Institute of Neural Information Processing of Ulm University. The Activities they created, although not yet publicly available, have reached a high quality and were fully usable within days. This level of flexibility and openness to new components is what sets the *pepr framework* apart from the monolithic solutions in the field of pattern recognition and information fusion tasks.

Acknowledgment

The presented work was developed within the Transregional Collaborative Research Centre SFB/TRR 62 "Companion-Technology for Cognitive Technical Systems" funded by the German Research Foundation (DFG).

6. References

Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA.

Stefan Scherer, Volker Fritzsch, and Friedhelm Schwenker. 2009a. Multimodal real-time conversation analysis using a novel process engine. In *in Proceedings of International Conference on Affective Computing and Intelligent Interaction 2009 (ACII '09)*, pages 253–255.

Stefan Scherer, Volker Fritzsch, Friedhelm Schwenker, and Nick Campbell. 2009b. Demonstrating laughter detection in natural discourses. In *Interdisciplinary Workshop on Laughter and other Interactional Vocalisations in Speech*.