MutantT: A Modular and Generic Tool for Multi-Sensor Data Processing

Simon Hawe, Ulrich Kirchmaier, Klaus Diepold
Lehrstuhl für Datenverarbeitung
Technische Universität München
Munich, Germany
(simon.hawe, kirchmaier, kldi)@tum.de

Abstract – In this paper a novel data processing framework that is fast, easy to use, open source, and highly extendable is presented. Due to its modular design concept the framework is not limited to specific tasks like image- or speech-processing but can be used for any data processing application. Thus it is an ideal platform for multi-sensor data fusion. Furthermore developers have the opportunity to extend the framework with their own filters and data types in a simple fashion, as a filter is a plug-in dynamically loaded by the framework. The framework provides a graphical user interface, where filters can be combined by a drag and drop action and a user can adjust filters’ parameters during runtime which allows rapid prototyping. It will be publicly available on sourceforge.net under the BSD License and contributions from developers of diverse data processing areas are encouraged.

Keywords: fusion test-bed, algorithm development tool, rapid prototyping, modular software design.

1 Introduction

In the scope of signal processing, engineers are often confronted with problems, or parts of a problem they have already solved in the past. As a simple example consider the field of image processing. Grabbing an image from a camera, extracting edges from an image or smoothing an image for suppressing noise are tasks needed for nearly every image processing algorithm. An engineer has several possibilities to deal with those repetitive tasks. One naïve solution is to implement the functions every time anew from scratch, which is cumbersome and inefficient. Another approach is to use libraries like OpenCV [4] or Gandalf [7] that provide certain functions. However, you still have to invest a considerable amount of time in order to combine these functions and to setup a complete processing chain.

A much more efficient way is the concept of modular software design introduced in [10]. A module is an independent unit which fulfills a specific task and communicates with other modules over defined interfaces. A module’s interfaces express the elements that are both provided and required by a module, in other words its inputs and outputs. Once a module is created, you can use it in different applications, without paying attention to how it performs its task and how it was implemented. The only thing to be known, are the interfaces. In this way developers can avoid redundant code and it is not necessary to implement the same things every time anew.

Processing-systems that perform tasks like for instance fusing infrared video with LIDAR data are realized by combining interacting modules. One intuitive way to combine modules is via a Graphical User Interface (GUI) which provides a drag and drop method where a user can arrange and connect the appropriate modules. There exist several good module-based data processing tools like [1, 3, 6] but they are either not open source thus not extendable and customizable, only limited to specific tasks, or do not provide direct sensor access for real time data processing.

We present our framework MutantT\(^1\) (A Multi-Sensor Data Processing Tool) which is open source, platform independent, and allows rapid prototyping. It can directly acquire data from sensors and process it as well as data stored on a hard-disk. Its modular design offers the user the opportunity to develop own filters for extending the framework so that it meets his requirements. New filters are integrated easily as they are compiled as shared objects which are loaded dynamically. MutantT provides a communication mechanism and the interfaces for filters to interact with each other and to exchange data. For parallel and fast data processing every filter runs in its own thread. The filters are combined with an intuitive drag and drop mechanism. This allows engineers with even no programming knowledge or little experience in specific signal processing areas to create complex processing systems. Parameters of a filter can be adjusted during execution to get direct feedback on how a parameter influences the output.

The complete framework is not limited to any specific area of data processing but rather flexible and open for engineers from diverse areas. This makes it well suited as a

\(^1\)https://sourceforge.net/projects/mutantmulti/
development- and test-environment for multi-sensor data fusion.

2 The framework

Our framework MutanT is implemented in C++ and for designing a platform independent GUI we used the Trolltech QT API version 4.4 [12]. Figure 1 shows the developed GUI.

Figure 1: Graphical user interface of MutanT. The blue framed area shows the list of available filters and the red framed area is the debugging area that presents execution time and occurring errors. On the green framed area a user drops the filters he selects from the filter list. The small yellow framed area is an exemplary control panel for parameter adjustment.

The framework is multithreaded and its most important parts, the filters which process data, are compiled as shared objects. These filters are loaded dynamically and are not directly embedded into the framework, which gives a user the possibility to easily extend and customize his version with his own filters. Furthermore, the use of shared objects allows users to exchange filters, for instance to compare their filters’ performances, but without presenting the explicit code in order to keep their intellectual property. The framework provides the interfaces for connecting filters and allows an easy extension of these interfaces for exchanging any user data.

In our framework, an entire processing chain that performs for example the fusion of data from various sensors, consists of a set of communicating filters that all realize a specific task. In order to create such a processing chain in MutanT the user can choose from a list of filters and drag them onto an arbitrary position on the common processing area. Those filters are connected by pulling a connection line out of an output-port and dropping this onto the input-port of the filter it should be connected to. During the connection, a type check is performed between the output-port’s and input-port’s data type. If those are not conform, the user will be warned and the connection will be refused. One output-port can be connected to an arbitrary number of different filters’ input-ports to process the same data in parallel but in varying manners. In this way one is able to directly compare how divers filters perform on the same input data or to display intermediate results. Filters can be added or exchanged at any time without stopping the execution of the processing chain.

Parameters associated with a filter can be adjusted during the processing chain’s execution via a control panel, which is also provided by each filter. Depending on the programmers’ choice, the control panel contains slider rules, drop down menus or direct keyboard input for parameter adjustment. A developer can even add his own type of control panel if the provided ones are not sufficient. This is possible because we implemented the logic for extending the framework with new control panels in the same fashion as for extending it with new filters, which will be described in section 4. The entire configuration of a processing chain, including the adjusted parameters, can be stored in an XML file format, which later can be loaded in order to reproduce already gathered results.

If an error occurs while a filter is executed, an error message will be presented to the user via the debugging window. This message is directly connected to the filter that failed thus the user knows which filter could not be executed and the error can easily be traced back. If a filter fails to execute, only the filters which depend on the failed one’s output-data are effected but not the rest of the processing chain. If no error occurs, the filter’s execution time is shown to the user in the debugging window.

3 Modular software design

The concept of modular software design was first introduced in 1972 [10] and is also known as Information hiding. Various software systems have been designed regarding the modularity paradigm, as it offers several benefits in contrast to "hard wired" software design. Debugging your software becomes a lot easier as an error can be directly assigned to one specific module and has not to be allocated throughout the entire programm. The flexibility and maintainability of your system increases, as distinct changes can be made to one module without changing the rest at all, as long as the interfaces maintain constant. This further means that if desired, one can update only specific parts of the system and does not have to update the whole system if there are parts a developer likes to retain. Once it is created, a module can be reused in any other larger system in combination with other modules. In order to combine modules, the only thing that has to be available and what the modules have to have in common are the interface for communication. A module
hides its internal operations thus an operator does not need to understand or appreciate the internal complexities of a module, all that is required is an appropriate sequence of modules in a processing chain. Modules are atomic objects, meaning they reveal only as much of themselves as required to perform their functions. Because they are atomic objects, modules can be thought of and treated like puzzle pieces. The type of data a module processes (input data), and the type of data it produces (output data) determines its puzzle piece shape and also determines which other module it can be connected to. As long as the pieces match, they can be fitted together into a larger scheme, the processing chain. In order to realize such a modular design and to provide a user with the ability to extend his version of MutanT, we chose the concept of self-registering plug-ins which we will describe in the upcoming section 4. We also integrated a central device which manages the creation and execution of our modules and provide generic interfaces for data exchange and communication. In order to avoid confusions we want to mention, that in our framework and throughout the rest of this paper a module is the same as a filter.

4 The framework’s software architecture

To describe the software-design we used for MutanT in simple words, we would like to give an easy everyday example. Imagine you want to buy a specific car, and all you know is its name but you do not care about how this car is actually produced or how its engine works. Further imagine there is a single central company where you can order any car by telling them the car’s name. This company has a registry where they can look up whether they are able to provide you with this car. If the car can be found in the company’s registry, the company can force a respective factory, which can build the car, to produce it. Still the company does not know how to build a certain car, but it has an abstract way of forcing the factories to build it. Additionally, every time a new type of car is developed, it automatically inserts itself into the company’s registry so the company does not have to keep track of the car market but always has an up-to-date registry. In the end you get the ordered car from the company and you are able to drive it. Without knowing its technology, you can even drive any other type of car so the specific model does not affect the action of driving. A similar scheme like that has been applied to MutanT.

In our framework the cars are the filters which all have to follow the same basic design rule. Therefore we implemented an abstract base class from which all filters have to inherit called APFBase. This base class contains the registry, as the company in the car example, as a static data member. A static data member has two attributes: it is initialized at the start of an application and only one copy of it exists; it is shared with all objects of that class and with all objects that inherit from that class. In that way filters have direct access to the registry and can insert themselves into it when developed. Consequently, the registry is always up-to-date. To do this registration automatically, a filter holds a static prototype member of its own type. As pointed out before, a static member is always initialized while the application is started, which means that its standard constructor will be called. By putting the filter’s registration step into the standard-constructor, a new filter automatically inserts itself into the registry when this constructor is called. A minor drawback is that the standard-constructor can only be used for registering the filter and has to be made private in order to avoid that it is called from somewhere else.

For implementing the registry we used an associative map which takes a string, the filter’s name, as its key and a pointer to a filter’s prototype instance as its value. The thought behind that is, if the base class holds a pointer to an instance of a filter, it can use that instance to build new instances of that filter, which is known as the prototype design pattern [2]. The building is realized by adding a pure virtual protected method build() to the base class which all inheriting classes have to reimplement. This method returns a pointer to an instance of a filter, so in the reimplementation only the appropriate constructor of the filter has to be called to achieve the building. Deferring object instantiation to subclasses is known as the abstract factory pattern [2].

In the domain of software development the combination of the prototype-pattern with the abstract factory pattern and the automatic registration technique is called self-registering plug-in design pattern and has been introduced in [5]. With this design pattern a programmer is able to remove the dependency of the application, here the framework, on the concrete plug-ins, here the filters, as the framework is self executable without loading any filter. Our filters are compiled independently from the framework as shared objects, and register themselves as soon as they are loaded. We further created a central instance called FilterManager that also has access to the registry of available filters. The FilterManager can be seen as the company in the car example, which provides the user with the filters. A user just passes the name of the wanted filter to the FilterManager; when the FilterManager finds this name in the registry he calls the build method and the created instance of the filter is returned to the user. The FilterManager is also responsible to load and unload all available shared objects correctly.

In order to execute a filter and process data, the base class APFBase has a pure virtual method processData() which also has to be reimplemented by each filter that inherits from APFBase. To start the actual data processing, the method processData from the filter that has been delivered by the FilterManager has to be invoked. In this way, a user can process data by only knowing what the effect of a filter on the data is without knowing how it has been implemented, just like driving a car and not knowing how the engine works. A pseudo UML diagram that depicts the most important parts of the applied software-design concept can be seen in figure 2.
5 Filter interfaces and intercommunication

As the main purpose of MutanT is to allow the user to combine several filters which should be executed in parallel, adequate interfaces and an intercommunication mechanism between connected filters has to be established. For intercommunication, we identified two main criteria: request for and information of new available data, and a generic thread- and type-safe data exchanging procedure.

5.1 Intercommunication mechanism

A filter can have \( n \geq 0 \) input-ports and \( m \geq 0 \) output-ports. Input-ports can be defined to exchange data that is optional or indispensable for filter execution. Every filter has an input-registry with \( n \) entries and an output-registry with \( m \) entries. In the input-registry it is stored which input-port has been connected to an output-port and the status whether new data is available for an input-port, which is false by default. The status changes to true, when an input-port receives an inform-signal from the output-registry if it connected to. In the output-registry it is stored if and to how many input-ports an output-port is connected. Furthermore, a status is stored if all connected input-ports have requested new data from the respective output-port, which is true by default. The status changes to true, when the output-port has received a request-signal from all input-ports it is connected to.

Whenever an input-port is informed about new available data the respective input-registry status entry is set to true. In addition, the corresponding filter checks if all indispensable input-ports are connected and do already provide new available data. When those requirements are true, the filter retrieves data from the input-ports and processes it. Once the filter has finished processing data, it has to check whether all its connected output-ports have received a request signal. If this is true the processing result is copied to the output-ports memory, the input-registry and output-registry status is set to false, the filter requests for new data by forcing all input-ports to send the request signal, and informs all connected filters about new available data by forcing the output-ports to send the request signal. Figure 3 shows the described internal logic of the intercommunication process. In the above described fashion, the processing chain keeps working once it has been started, even if filters are removed, new ones are added or a failure occurs. The initial starting is performed manually by the user hitting the start button and can be stopped manually too. This scheme is independent from what a specific filter actually does, it is always regarded like retrieving data, an abstract processing scheme, and forwarding the processing result.

5.2 Generic data transfer

As quoted in the previous section, the data produced by a filter is written to its respective output-port’s memory once processing is done. When an output-port and an input-port of two filters are connected, the input-port gets read-only access to the output-port’s memory. Furthermore, several filters’ input-ports can be connected to the same filter’s output-port. This means that different threads have access to the same memory. In order to guarantee thread safety, meaning no simultaneous read/write-access to the same portion of memory, we employed a mutex type security mechanism. This means a filter’s output-port shares one mutex with all connected input-ports. Every time a filter has created new data, it locks the output-port’s mutex to prevent memory access from other filters, copies the new data to the output-port’s memory and unlocks the mutex. Each input-port acts in the same way but copies the data from the output-port’s memory to its own local memory which can not be accessed by any other filter. These copied data will then be processed by the filter.

A main aspect of MutanT is to provide a framework which can be extended to process any kind of data or objects. However, the data have to be exchanged between filters in the above described fashion, which means that it has to be possible to copy the data correctly. Additionally, it should be
checked that no two ports with different data types are connected. To achieve this, we introduced a pure virtual data container base class TData. This base class only consists of a pure virtual data copy method, a pure virtual data clone method, and a pure virtual type checking method. All objects that should be exchangeable in MutanT just have to inherit from TData and implement these three methods. The resulting object has to be compiled as a shared object. In that way this concrete data object has to be linked directly only to a filter that processes this type of data, but not to the framework. This only calls the three methods from TData, whose respective implementations are dynamically loaded when required. Thus MutanT can be easily extended to exchange any type of user data without changing the framework at all. Wrappers to exchange all standard C data types and STL-container are provided by MutanT.

6 Filter development

For creating a specific filter to be used within MutanT, all you have to do is to inherit from the base class, define the filter’s interfaces meaning the number and the type of input-data and output-data, and implement the actual task that should be performed by your filter. Furthermore, for parameter adjustment you have to define which control panels you want to use and the possible values that can be chosen. After compiling your filter, the created shared object just has to be placed in the correct folder and will then be loaded by the framework.

7 Case study

To show the capability of MutanT in the context of multisensor data fusion we combined a monochrom visible domain USB 2.0 camera from PixeLink with resolution 1280 × 1024, a focusable far-infrared IEEE 1394 camera from FLIR with resolution 320 × 240, and a 1D-LIDAR from Sick with a 180-range. The data of those sensors were all acquired and immediately processed by adequate filters within our framework. With this assembly we created an automatic image registration technique for complementary imaging devices working in the far-infrared and visible domain. We want to use the gathered information not only to create composite images but rather to build a pseudo stereo camera setup consisting of a far-infrared and an optical camera.

In order to register images or to create a stereo system it is crucial that both devices sample the environment at the same moment of time, which is usually solved using an external trigger signal. However, our infrared camera can not be triggered externally. To overcome this problem, we decided to generate a trigger signal for the visible domain camera, which is in phase with the exposure of the infrared camera. This trigger mechanism ensures nearly synchronous (±1 ms) capture of images and is also controlled by a filter within MutanT. The second problem we had to tackle, was to automatically adjust the focus of the infrared-camera to get sharp images. We found a characteristic line for this camera, which gives the relation of focus value to object distance. Therefore we used the LIDAR to measure the distance from our camera system to the scene and the acquired distance information was used for adjusting the infrared camera’s focus. Two synchronously taken images with adjusted focus can be seen in figures 4 (a) and (b).

(a) Image taken with visible domain camera

(b) Image taken with far-infrared camera

Figure 4: Synchronously taken pictures from different domains.

In order to register images, which means to transform one image to align with the other image, one has to establish correspondences between those images. As we deal with images from complementary domains there generally exists no relation between image intensities. Thus traditional approaches to establish correspondences like the sum of squared distances or the normalized cross correlation will inevitably fail. However, object boundaries are apparent in both infrared and visible spectrum as quoted in [9]. We used this fact and compared objects boundaries, to find an initial transformation that transforms the infrared image to align with the visible one. We then used the mutual information (MI) between the images to compare their similarity. Mutual information or transinformation is a similarity measurement from the field of information theory. It is a histogram-based technique and has been widely used for medical image registration (CT and MRT), see [11] for a detailed survey, as it is well suited for comparing images from different domains [8]. A high MI means a high similarity, thus we gradually change the transformation parameters, use the parameters to transform the infrared image, calculate the MI and stop this procedure when the MI exceeds a certain threshold [13]. In other words, we apply an optimization scheme that finds a
transformation, which maximizes the MI between the transformed infrared image and the image from the visible spectrum. A registration result showing overlapping images is shown in figure 5.

Figure 5: Registration result that shows overlapping images from infrared and visible spectrum.

8 Conclusion

In this paper we introduced our data-processing framework MutanT. This framework allows to directly acquire data from sensors as well as stored data and to process them. It is platform independent, open source, and has been designed in terms of the modular-software design paradigm. We showed that this design allows a user to easily extend the framework with custom filters and to rapidly create a processing chain. Furthermore, we introduced a generic threaded and type-safe data communication scheme to enable multithreaded data processing and to give a developer the ability to extend the framework's types of data that can be exchanged between filters. As the framework is not limited to any specific data type but can be arbitrarily extended, it is an ideal platform for developing and testing algorithms in the field of multi-sensor data fusion.

To emphasize the framework's applicability to fusion tasks, we presented a case study where we fused the signals of a visible domain camera, an infrared-camera, and a LIDAR. Currently we expand the presented software architecture to allow the implementation and integration of filters that use the graphic card for fast calculations. We want to encourage engineers from diverse areas of signal processing to use and contribute to our framework in order to create a powerful tool with a rich set of filters.

Acknowledgement

The authors gratefully acknowledge partial financial support by the German Research Foundation (DFG) within the Transregional Collaborative Research Centre 28 Cognitive Automobiles.

References


