Generic Sensor Fusion Package for ROS

Denise Ratasich, Bernhard Frömel, Oliver Höftberger and Radu Grosu
Institute of Computer Engineering
Vienna University of Technology, 1040 Vienna, Austria
Email: {denise.ratasich, bernhard.froemel, oliver.hoeftberger, radu.grosu}@tuwien.ac.at

***Abstract*— Sensor fusion combines multiple sensor measurements to improve a controller's knowledge about the internal state of an observed physical environment. Many such sensor fusion techniques exist and have been implemented for the Robot Operating System (ROS). However, they often have been developed for specific applications and cannot be easily reused for other applications. Reasons are the use of application-specific, partly undocumented interfaces, and the often limited reconfigurability caused by a tight coupling of the implementation to an application-specific purpose. Our approach is based on the concept of a *fusion node* which provides a configurable sensor fusion service with a generic interface. Fusion nodes can be interconnected to combine several sensor fusion techniques, can be attached to any single-dimension value sensor, can handle asynchronous multi-rate measurements and are robust regarding indeterministic, best-effort communication. This paper presents, to the best of our knowledge, the first generic sensor fusion package (GSFP) for ROS which collects various exemplary sensor fusion methods implemented as fusion nodes. We demonstrate the feasibility of our package in a small test application. Main benefits of our contribution are the developed ROS package's independence regarding specific sensors or applications, the easy integration of configurable fusion nodes in existing applications, and the composition of fusion nodes to realize complex sensor fusion scenarios.**

I. Introduction

Embedded systems interact with physical processes. For example, an embedded system may control a chemical process (e.g., to prevent overheating) or monitor the tire pressure of a vehicle (e.g., to ensure good handling of the vehicle and low fuel consumption). In order to govern a physical process, the controller must be aware of the internal state of the process, that is, the set of internal variables completely characterizing the past history of the process [1]. Sensors measure the outputs of a process, so-called observations, whereof the state can be estimated. But the conversion of a variable from the physical domain to a useful representation for a controller suffers from several problems. Observations are distorted by measurement noise, caused by fluctuations of the elements in the electrical circuit of the sensor and the limits of the sensing element (e.g., the finite resolution of the conversion from the physical property into a digital representation). Furthermore the whole range of a physical property may not be covered by a single sensor. But also an unpredictable environment, inaccurate models and limited computation rise the uncertainty of the observed state of the process [2], [3], [4]. An erroneous belief of the process' state can lead to fatal consequences. For example, a faulty sensor measuring the attitude of an autonomous aircraft leads to an incorrect belief of the aircraft's attitude. Hence the aircraft's computer is unable to maintain the desired flight path which may even result in a crash.

Sensor fusion is the most common way to overcome the addressed problems. Sensor fusion methods *"combine data from multiple sensors, and related information from associated databases, to achieve improved accuracies and more specific inferences than could be achieved by the use of a single sensor alone"* [5]. State estimation (also called filtering [6]) is a commonly used method to estimate the state of a process out of raw sensor measurements. If redundant data (e.g., measurements representing the same physical property) is processed by the state estimator, then a state estimator is also a sensor fusion technique increasing the certainty of the state. But there is a lack of libraries providing such filter algorithms or sensor fusion techniques in general to use them right out of the box. Especially at the beginning of the development of an embedded system it is useful to try out various methods and combinations to choose the optimal sensor fusion configuration for the application. A modular framework of sensor fusion methods is able to speed up this stage of development.

This paper presents, to the best of our knowledge, the first sensor fusion package to use within an application running on the Robot Operating System [7]. The implemented generic sensor fusion package (GSFP) is able to generate an instance performing sensor fusion (e.g., a state estimator). The sensor fusion instance is compiled out of a generic part and a single configuration file provided by the user. The fusion technique, inputs and other parameters can be exchanged simply by adapting the configuration file. GSFP is not bound to specific sensors, nor specific applications. The package is able to handle nondeterministic communication, i.e., the measurements for the fusion instance can arrive asynchronously and delayed. Furthermore GSFP supports multirate measurements, i.e., the sample rates of the sensors serving as inputs for the fusion instance may differ from each other. At the time of writing, GSFP collects several state estimators (linear Kalman filter (KF), extended Kalman filter (EKF) [4], unscented Kalman filter (UKF) [8], particle filter (PF) [4]), basic filters for noise reduction (moving average filter, moving median filter) and a filter to fuse values from redundant sensors (confidence-weighted averaging [2]). The source code is available at [9]. Further techniques may be easily added to the package. However, the main contribution of GSFP is the generic interface. It enables easy integration into existing applications, possibly improving their behavior by providing them a more accurate view of the environment.

The next section discusses state-of-the-art sensor fusion and state estimation libraries. Section III presents GSFP. In Section IV we demonstrate the benefits of the proposed ROS sensor fusion package. It shows the applicability of the developed package using an exemplary robotic application.

Finally an outlook and a summary is given in Section V and VI respectively.

II. Related Work

Sensor fusion targets lots of applications. The methods depend on the types of sensors (cf., image sensors to accelerometers) and the function to perform (cf., classification of objects to noise reduction) [3]. So a library or package collecting different sensor fusion methods is hard to find, rather there are packages for specific applications or libraries collecting similar types of methods (e.g., state estimators). For example, several concrete filters for position estimation are provided for ROS [10], [11], [12], [13]. However, these packages are limited to a specific application and very few specific sensors, in this case, pose estimation of a robot or a drone. Few libraries exist where variant types of, e.g., state estimators [14], [15], [16], are implemented for general usage. For example, the Bayesian Filtering Library of Orocos [15] is provided via a ROS package, however, it cannot be used right out of the box (see comparison to our package in Section IV). These libraries suffer of two main problems: nontrivial configuration and integration.

With the existing state estimation libraries several functions and code for initialization have to be programmed before the filter can be used. For example, to use a Kalman filter (a common state estimator for continuous systems) from the Bayes++ library [14] following parts have to be programmed: *1.)* A suitable C++ class for the model (a necessary parameter for the KF) has to be selected. From this C++ class one has to derive its own class to represent a concrete model for the filter. The code to apply the model has to be written into the constructor of this class. *2.)* The filter has to be instantiated and initialized with appropriate variables in the application. *3.)* Then sampling and filtering has to be initiated periodically. The point in time, when calling the filter algorithm, has to correspond to the model and to the sampling of the sensors. So the configuration is error-prone because the user of the library has to select the appropriate C++ classes and furthermore has to initialize the objects on its own. Furthermore, it is crucial to synchronize the execution of the state estimation and the sampling of the sensors. When performing state estimation three timing values are of interest: the sample rates of the sensors, the period of performing the estimation (the "resolution" of the estimate) and the time constants in the model (note that the model of a state estimator is based on difference equations and therefore usually contains time constants). When the time constants in the model deviate from the estimation period, the estimate is incorrect.

The possible sample rates of the sensors may differ. Hence deciding, when to sample the various sensors and to estimate, gets complicated. Possibly the sensors cannot be sampled at all, i.e., the state estimator is not able to request the sensor values at a specific point in time. The sensors' observations may then arrive asynchronously at the sensor fusion or filter application. These circumstances (e.g., different sample rates, asynchronous measurements) make the filters of the libraries hard to integrate. There are different approaches [17], [18], referred as *temporal alignment* [3], to overcome these issues. However the implementation of these methods require manual programming and needs thorough literature study on the approaches and state estimation itself. In the example above, sampling and estimation are in a tight application-specific relation. So the filter cannot be interchanged that easily. Adding other sensor fusion methods requires even more programming work. It may be the case that the code for sampling and sensor fusion has to be rewritten completely. The referenced libraries are versatile, maintained and tested - but only appropriate for experienced users, who have a deep understanding of the employed sensor fusion methods and are able to modify or extend their implementation. There are lots of points the inexperienced user has to struggle with, when setting up state estimation in one's control application. Although various examples are given for configuration within the documentation of the discussed libraries, embedded systems engineers may fail to integrate such a filter into their control application correctly and face undesirable results. Related state estimation libraries may be a good back-end for our sensor fusion package, but do not fulfill the needs on simple configuration and integration of state estimators on their own.

III. Generic Sensor Fusion Package

GSFP implements a component performing sensor fusion, henceforth called *fusion node*, within the Robot Operating System (ROS) for various applications and sensors. GSFP consists of three parts: *i) a C++ library* implementing the exemplary sensor fusion algorithms with a common interface for inputs, outputs and the estimation and a specific interface for individual configuration (e.g., model); *ii) a generic ROS node* receiving measurements, performing temporal alignment and calling the estimation periodically; *iii) the configuration* specifying the inputs, outputs, type of state estimator and the parameters of the ROS node.

The next section gives a short introduction to ROS, the platform of the implemented sensor fusion framework. Then, the library collecting the sensor fusion algorithms is proposed. It follows the configuration of a method and the temporal alignment of measurements to process. Finally the integration of a fusion node is described. Further information can be found at [9] (tutorials, trouble shooting, description of filters and its parameters, source code documentation).

A. Platform

The Robot Operating System is a framework consisting of libraries, tools and conventions which simplifies the development of robotic systems [7]. Basically it is a meta-operating system that provides services like hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. ROS follows the service-oriented approach, i.e., components can be invoked and its interface descriptions can be published and discovered. A component or so-called node may communicate over so-called topics, i.e., messages are published to the whole system through a topic A and all nodes subscribing to topic A receive these messages. In ROS, the communication is assumed to be unreliable and event-triggered, i.e., nodes may send and receive messages at arbitrary time instants. In other words, messages are sent asynchronously and might be lost. For receivers, it means, that messages arrive with arbitrary delay, may arrive out-of-order, or may arrive not at all.

TABLE I
SENSORS AND THEIR VARIANCES.

	SENSOR	VARIANCE
v_l	left wheel's velocity	0.002 m/s
v_r	right wheel's velocity	0.002 m/s
a	(linear) acceleration	0.25 m/s^2
ω	angular velocity	0.00017 rad/s

TABLE II
ROS TOPICS CONTAINING THE MEASUREMENTS.

	TOPIC	FIELD
v_l	velocity_wheel_left	data
v_r	velocity_wheel_right	data
a	acceleration	vector.x
ω	angular_velocity	vector.z

B. Library of Sensor Fusion Algorithms

Each sensor fusion algorithm is implemented in a separate C++ class. These classes are collected in a library. The algorithms have a common interface for inputs, outputs and the estimation, specified in a separate C++ abstract class. In particular all estimator classes have the very same function prototype for estimation passing the inputs through parameters and returning the outputs. Furthermore each method has its specific interface for configuration (which individually parametrize the sensor fusion method), i.e., a set of function prototypes located in the appropriate header file of the specific estimator class.

C. Configuration of a Fusion Node

The generic ROS node is specialized by a header file, such that during compilation the unique ROS fusion node, i.e., the executable program to run within ROS, is created. In this header file the configuration of the node and the sensor fusion method is located (inputs, outputs, type of estimation, parameters). The configuration is specified by several macros, as it is the simplest possibility in C++ to specify, e.g., the data types of ROS inputs without coding a separate interface for each application. The Boost Preprocessor library [19] is used to handle these macros.

As a first step of the compilation, the type of sensor fusion to perform is checked, which is specified within the configuration header file through a macro. For example:

```
#define METHOD   UNSCENTED_KALMAN_FILTER
```

According to this macro the right estimator is instantiated in the ROS node by selecting the appropriate C++ code for initialization. Next the estimator is initialized with the parameters given in the configuration header. The types of parameters range from single numeric values to matrices of arbitrary size. For example, the covariance matrix, of the measurements from the sensors given in Table I, is specified as follows (the measurements are assumed to be independent from each other):

```
#define MEASUREMENT_NOISE_COVARIANCE          \
  ( (0.002) (0)     (0)     (0)       )       \
  ( (0)     (0.002) (0)     (0)       )       \
  ( (0)     (0)     (0.25)  (0)       )       \
  ( (0)     (0)     (0)     (0.00017) )       \
  /**/
```

Even functions may be specified as parameters. For example, the model of an UKF is usually described by a mathematical expression, i.e., a formula containing several variables (e.g., the current state of the process to model). For example, the model to estimate the current state x of an object (e.g., a drone) every period T. The state x contains the velocity v and the acceleration a. The object is controlled by a signal u corresponding to the desired acceleration. Mathematically this model can be described by

$$x = \begin{pmatrix} v \\ a \end{pmatrix} \qquad \begin{aligned} v &= v + a \cdot T \\ a &= u \end{aligned} \tag{1}$$

So the current velocity is calculated by integrating the acceleration while the control signal u directly maps to the current acceleration. To evaluate the result of the formula, the C++ function implementing this expression must be compiled before execution (unlike languages using interpreters, e.g., Python, which lack performance compared to C++). In existing libraries (cf., Section II) this C++ function would have been implemented by the user. In this ROS package the function is specified in the configuration header by the right-hand side of the formula, similar to the original mathematical expression. Considering above example, the formulas in Equation 1 only need to be rewritten minimally to specify the model within the configuration header:

```
#define STATE_TRANSITION_MODEL          \
  ( x[0] + x[1]*FILTER_T )              \
  ( u[0]                 )              \
  /**/
```

The state variables v and a (or $x[0]$ and $x[1]$ respectively) will be updated by the expressions given in the macro, i.e., the right-hand side of the formulas in Equation 1. During compilation the C++ function that applies the model is automatically generated and passed to the filter. So no functions for models or for initialization of variables have to be coded. The parameters only have to be specified as macros. The user needs not to decide which C++ class suits the parameter, nor how the parameter has to be initialized.

Each individual parameter for a sensor fusion method has its own format and meaning. The developer of the sensor fusion algorithm decides about the macro's syntax (the type of the parameter, e.g., vector, and how it is specified) and semantics. Considering above example, the parameter `STATE_TRANSITION_MODEL` requires a sequence of functions $f_0..f_{n-1}$, calculating the current state (of size n) out of the previous state. So each state variable x_i (with $i = 0, 1, .., n-1$) is updated by the function f_i given in this parameter. For example, the current value of the state variable $v = x_0$ (semantics) represented by `x[0]` (syntax) is updated by $v + a \cdot T$ with $a = x_1$ (semantics) written as `x[0] + x[1]*FILTER_T` (syntax). However, the configuration file is a C++ header file, so one may use other definitions and functions included in the configuration header (e.g., `std::sin(x)` from the standard math library).

Once the instance of a class implementing a sensor fusion method is configured, the ROS interface is initialized. During compilation, callback methods for received messages (e.g., measurements) are specified, based on the information given in the header file (e.g., type of the message). The callbacks

are registered for the message topics specified - again - as macros in the configuration header. For example, to subscribe to the topics of Table II the user has to define:

```
#define TOPICS_IN                                    \
  ((velocity_wheel_left) (Float32) (data))           \
  ((velocity_wheel_right) (Float32) (data))          \
  ((acceleration) (Vector3Stamped) (vector.x))       \
  ((angular_velocity) (Vector3Stamped) (vector.z)) \
  /**/
```

One row of the above macro specifies one input. The first element defines the name of the topic to subscribe. Next, the type of the message has to be specified. In this example we use the ROS provided types `Float32` and `Vector3Stamped`. The last element determines the field of the message containing the value of the measurement (in ROS a message is represented by a C structure).

Finally, after creating and parametrizing the sensor fusion algorithm and subscribing to the inputs, the constructs to publish the estimate are initialized. The set of estimates (e.g., state variables) to publish, can also be specified in the configuration header.

To summarize the configuration process, the user provides a configuration header, which is processed during compilation to specialize the ROS fusion node. The macro defining the sensor fusion method is used to select the appropriate C++ code for the instantiation and initialization. The macros corresponding to the individual parameters of a fusion method are additionally validated, e.g., necessary parameters are checked for presence. The parameters for the sensor fusion method are initialized by appropriate macros in the configuration header. For each input of the sensor fusion node a callback method is generated. Finally for each output a publisher is instantiated and initialized.

D. Temporal Alignment of Measurements

In ROS measurements arrive asynchronously, i.e., *when* the inputs for the fusion node are delivered cannot be specified. Incoming messages are stored in a queue until the ROS node decides to process the received messages. Usually this is done as often as possible to keep the delay (from communication and receive queue) low. The same is true for a fusion node. To get an accurate estimate, the measurements should be processed, i.e., fused, with minimal delay. Outdated observations can lead to wrong estimates.

In Figure 1 the measurements of two sensors should be combined. The arrival time of measurements from sensor 1 (or sensor 2 respectively) are depicted as vertical lines of height 1 (or height 2 respectively). A minimal delay could be accomplished by performing estimation (represented by the shaded area in Figure 1) whenever a measurement is received. Hence each time a message arrives, a sensor's value is incorporated into the fusion algorithm on its own.

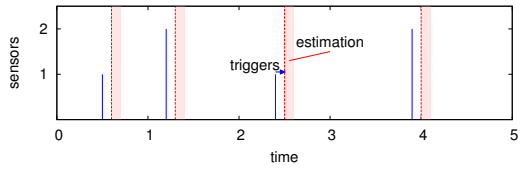

Fig. 1. Each measurement triggers an estimation.

This approach may not be suitable for several sensor fusion techniques. For example, it cannot be used when the fusion node should average the measurements of sensor 1 and sensor 2. In this case, the fusion node should await both sensor values, i.e., the measurements from sensor 1 and sensor 2. In some applications (especially when sensors have different sample rates) this may lead to high latencies of the state estimate depending on the sensors' sample rates and additionally decreases the accuracy of the estimate [3]. Performing estimation whenever a measurement is received is possible with some techniques, e.g., state estimators. But this increases the computational effort, because the estimation has to be performed more often. Note that commonly used state estimation algorithms have the same computational effort regardless how many measurements are available for fusion. With a very small number of sensors, very low sensor sample rates and a time efficient and appropriate fusion algorithm this may be the right approach. Problems occur, however, when the sample rate of a sensor is higher than the rate, the estimation can be performed with (see Figure 2 where the execution time of the estimation is depicted as shaded area). Some algorithms are more computationally expensive than

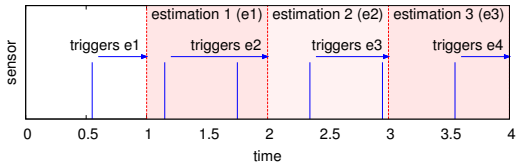

Fig. 2. Sensor with high sample rate triggers the estimation.

others and the execution time of estimation may also depend on the application, e.g., a Kalman filter has to perform matrix inversion (the size of the matrices depends on the application - the number of sensors, the size of the state, etc.).

A trade-off between reaching minimal delay, ensuring execution time of the estimation and awaiting measurements can be accomplished by periodic estimation. The estimation is then performed every T seconds, i.e., the estimation rate $f_e = 1/T$ is fixed, where $T \geq WCET_{estimation}$ (worst-case execution time of the estimation) is satisfied. Calling the estimation periodically requires temporal alignment of measurements. Regarding the sample rate of a sensor f_s and the estimation rate f_e two combinations may occur, which are handled as follows:

- Measurements of sensors which publish their value with a lower rate than the estimation rate, i.e., $f_s < f_e$. Hence some measurements are not available in every filter cycle (see sensor 2 in Figure 3), in other words, measurements appear to be missing. In state estimators of this ROS package, missing measurements are replaced by their expected values, i.e., the predictions of the measurements (this method is based on the recalculation method [17]). Using the same value for expected and actual measurement, causes no correction of the predicted state by this measurement.
- For measurements with a higher sample rate than the estimation rate, i.e., $f_s > f_e$ (see sensor 1 in Figure 3), the average of the measurements received in a period T is calculated. The average value represents the

observation for the next estimation.

The principle of temporal alignment is depicted in Figure 3. The dashed vertical lines feature the points in time when the estimation is triggered and started. All measurements arrived in the last period (time between two dashed lines) are aligned, i.e., averaged and forwarded to the state estimation algorithm. Note that instead of averaging other approaches

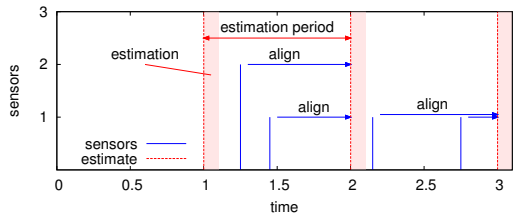

Fig. 3. Temporal alignment of measurements.

may be better suited (depending on the application), e.g., taking the last measurement or weighting the measurements. The estimated state is published periodically. The period T can be specified by the user within the configuration header. The estimation rate should be selected to match the highest sensor sample rate. In this case all measurements are used for sensor fusion, giving a more accurate estimate. But the estimation rate is bounded by the WCET of the used sensor fusion algorithm. For example, the execution time of the estimation is 5ms, whereas the sampling period of a sensor is 1ms. So the estimation period has to be selected to be greater than 5ms. The sensor's sample rate is higher than the (possible) estimation rate. So combining measurements of a sensor with a sample rate greater than the estimation rate may be necessary anyway.

ROS does not provide any real-time features. Hence it is not guaranteed that the estimation is called *exactly* every period T, because other tasks may run on the processor too. Furthermore ROS gives no possibility to particularly schedule services, e.g., simultaneous sampling of sensors by different services can not be specified. Finally, as already noted, the communication is nondeterministic. Considering these properties of ROS and the implementation of GSFP, the user must be aware, that the estimate may not be optimal. The measurements arrive at different time instants during the estimation period, i.e., the samples are not exactly synchronized. Furthermore the calculated estimate is delayed to the time instants the observations arrive at the fusion node, i.e., measurements arrive some time before the estimation is executed. Deterministic communication channels between sensors and fusion node and real-time computation as proposed in [2] could significantly improve the accuracy of the estimation. However, the error in accuracy of the estimate can be bounded by the estimation period.

E. Integration into Application

Due to the distributed service-oriented approach of ROS, integration of a ROS fusion node into (existing) applications is simple. The fusion node only has to be started and inputs and outputs redirected, as depicted in Figure 4. The consumer of, e.g., sensor measurements uses the estimated sensor values from the sensor fusion node, instead of the raw measurements.

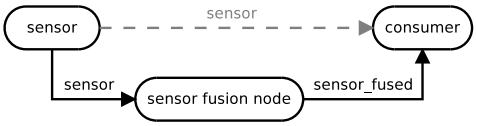

Fig. 4. Integration of the fusion node.

IV. Experiments

To test GSFP, the package has been installed on a robot running ROS. The applicability of GSFP is presented with a navigation application on a robotic platform (though it can be applied to other applications as well, e.g., estimation of a battery's state-of-charge). In particular the robot is controlled by the user, who sets the desired translation and rotation.

Starting from an initial position of the robot, the position can be calculated out of speed, heading and the elapsed time (this method is known as *dead-reckoning*). This kind of position estimation is often executed only with wheel encoders. The accuracy of the estimated position suffers from slipping of wheels, different kinds of driving surfaces, mechanical parameters, encoder resolution and radius variations of the wheels [20]. In addition to the wheel encoders, the robot is equipped with an Inertial Measurement Unit (IMU) containing an accelerometer and a gyroscope. The pose, i.e., the position and heading, of the robot can be calculated in three independent information sources (depicted by the three arrows towards the fusion node in Figure 5): *i)* the IMU, *ii)* the wheel encoders, and *iii)* the controls from the user via keyboard. With sensor fusion these sensors and controls (here: giving redundant data for translation and rotation) can be combined to provide a robust estimate of the robot's pose.

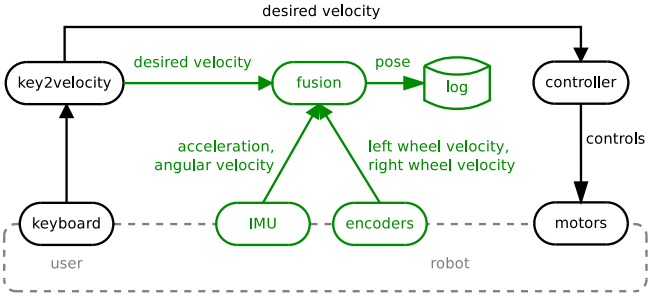

Fig. 5. Test application.

As depicted in Figure 5 the original application, i.e., controlling a robot over the keyboard, has remained untouched. Note that the number of sensor fusion nodes is not fixed to one. For example, to compare different configurations, several instances may simultaneously run in an application on ROS. We briefly compare GSFP for ROS to an implementation with BFL [15] (other similar libraries, see Section II, would lead to likewise results). With GSFP only one file has to be created or adapted, namely the configuration header. With BFL first a ROS package has to be created. The code for subscribing to the sensor topics and publishing the estimated result has to be written. We noticed that the documentation does not correspond to the current version of BFL. Therefore we spent a lot of time to select suitable classes and to initialize the filter. However, the time exposure depends on

the foreknowledge of the user. So we compare the lines of code (LOC) written to fuse the sensors in our application, see Table III (the statements are without comments and blank lines). The integration of state estimation into an application is the big difference between GSFP and general state estimation libraries. The effort for configuring the filter stays almost the same. We selected an appropriate estimation

TABLE III
COMPARISON OF LINES OF CODE.

	GSFP	BFL
LOC - configuration of the filter	36	50
LOC - integration	18	151
LOC - all together	54	201

rate to avoid handling of missing measurements with BFL. However, as noted in Section III-D the estimation rate should equal the highest sample rate to reach a good accuracy. As figure 6 shows, the accuracy of the final estimated positions increases when choosing a smaller estimation period. For this experiment three UKFs have been instantiated using the IMU, encoders and controls as inputs but different estimation periods (10ms, 50ms and 100ms). The sensors used (see also

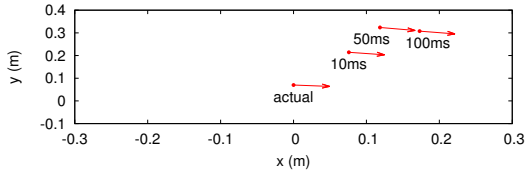

Fig. 6. Final estimated and actual poses using different estimation periods.

Table I), are sampled every 10ms (acceleration and angular velocity) and 100ms (wheel velocities). The UKF filtering every 10ms has the best accuracy (closest to actual position), because every measurement from accelerometer and gyroscope is used to correct the estimated state. The other UKFs (50ms and 100ms) incorporate only the average of several measurements and therefore "lose" some information.

V. FUTURE WORK

At the time of writing, several filters are implemented in GSFP. There are further state estimation methods and further types of fusion [3]. The exemplary sensor fusion techniques in our package may be replaced with an appropriate and maintained C++ library of fusion methods, e.g., BFL [15]. In a large distributed system messages may be received out-of-order. If components are time-synchronized, messages can be time-stamped at the sender. Then the receiver is able to resolve out-of-order messages by reordering or dropping messages that are older than the last processed message. The configuration could be even automated, e.g., the configuration header of GSFP may be generated based on an existing system model. Finally, we would like to confirm the usability with feedback gathered from ROS and GSFP users.

VI. CONCLUSION

Engineers inexperienced in sensor fusion struggle with various problems when setting up a mechanism for adequate state awareness. Insufficient knowledge about the theory of sensor fusion and nontrivial usage of existing libraries may slow down the development of a control application. In this paper we introduced, to the best of our knowledge, the first generic sensor fusion framework for the Robot Operating System. It allows rapid configuration and usage of various sensor fusion methods, like state estimation. The setup of fusion nodes is easy, because GSFP automatically instantiates sensor fusion nodes according to a user supplied configuration. The instantiated fusion nodes also perform temporal alignment of inputs to handle multirate and asynchronous measurements. Parameters are specified in a short and well readable form, e.g., a mathematical expression instead of a C++ function. Finally, the sensor fusion node is clearly separated from other application parts (e.g., a controller). Hence GSFP allows a simple integration of sensor fusion mechanism into (existing) applications.

ACKNOWLEDGMENT

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement n° 621429 and from the National Funding Agencies of Austria, Germany, Italy and France under the funding ID ARTEMIS-2009-1-100208.

REFERENCES

[1] F. Szidarovszky and A. T. Bahill, *Linear Systems Theory*. Boca Raton, Florida: CRC Press, 1992.
[2] W. Elmenreich, "Sensor Fusion in Time-Triggered Systems," Ph.D. dissertation, Vienna University of Technology, 2002.
[3] H. Mitchell, *Multi-Sensor Data Fusion - An Introduction*. Berlin, Heidelberg, New York: Springer, 2007.
[4] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. Cambridge: MIT Press, 2006.
[5] D. Hall and J. Llinas, "An Introduction to Multisensor Data Fusion," *Proceedings of the IEEE*, vol. 85, no. 1, pp. 6–23, 1997.
[6] S. Russel and P. Norvig, *Artificial Intelligence - A Modern Approach*, 3rd ed. Upper Saddle River, New Jersey: Pearson Education, 2010.
[7] Robot Operating System Wiki, "ROS Documentation - Introduction," http://wiki.ros.org/ROS/Introduction, May 2014.
[8] E. Wan and R. V. der Merwe, "The Unscented Kalman Filter for Nonlinear Estimation," in *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, 2000, pp. 153–158.
[9] Denise Ratasich, "Generic Sensor Fusion Package for ROS - Source Code," https://github.com/tuw-cpsg/sf-pkg, Jun. 2015.
[10] Robot Operating System Wiki, "robot_localization," http://wiki.ros.org/robot_localization, 2015.
[11] ——, "ethzasl_sensor_fusion," http://wiki.ros.org/ethzasl_sensor_fusion, 2012.
[12] ——, "robot_pose_ekf," http://wiki.ros.org/robot_pose_ekf, 2012.
[13] ——, "tum_ardrone/drone_stateestimation," http://wiki.ros.org/tum_ardrone/drone_stateestimation, 2012.
[14] M. Stevens, "Bayes++: Open Source Bayesian Filtering Classes," http://bayesclasses.sourceforge.net/Bayes++.html, 2005.
[15] Orocos, "The Bayesian Filtering Library," http://www.orocos.org/bfl, 2014.
[16] M. Laitl, "PyBayes," http://strohel.github.io/PyBayes-doc/, 2010.
[17] V. Fathabadi, M. Shahbazian, K. Salahshour, and L. Jargani, "Comparison of Adaptive Kalman Filter Methods in State Estimation of a Nonlinear System Using Asynchronous Measurements," in *Proceedings of the World Congress on Engineering and Computer Science 2009 Vol II*, October 2009.
[18] L. Yan, B. Xiao, Y. Xia, and M. Fu, "The Modeling and Estimation of Asynchronous Multirate Multisensor Dynamic Systems," in *Proceedings of the 32nd Chinese Control Conference*, July 2013.
[19] Boost, "The Boost Library Preprocessor Subset for C/C++," http://www.boost.org/doc/libs/1_46_1/libs/preprocessor/doc/index.html, 2014.
[20] A. C. J. Gu, M. Meng and P. Liu, "Sensor Fusion in Mobile Robot: Some Perspectives," in *Proceedings of the 4th World Congress on Intelligent Control and Automation*, 2002, pp. 1194–1199.