Synchronous Dataflow and Visual Programming for Prototyping Robotic Algorithms

Sebastian Buck, Richard Hanten, C. Robert Pech and Andreas Zell

University of Tübingen, Sand 1, 72076 Tübingen, Germany
{sebastian.buck, richard.hanten, robert.pech,
andreas.zell}@uni-tuebingen.de

Abstract. Robots perceive their environment by processing continuous streams of data, which can be very naturally modelled as a dataflow graph. The development of new perception algorithms is often an iterative process, involving the investigation of a set of parameters and their influence on the system. The amount of immediate feedback available to the developer can make these influences more obvious and can therefore speed up development. We present a framework based on synchronous dataflow and event-based message passing that forms the basis of a visual programming language for rapid prototyping of robotic perception systems. We explicitly model algorithmic parameters in the dataflow graph, which results in a more expressive feature set. We provide an open-source implementation, consisting of a user interface for immediate feedback and interactive manipulation of dataflow algorithms and an independent execution framework that can be directly used on any robot.

Keywords: perception, prototyping, visual programming, dataflow, robotics

1 Introduction

Robots interact with their environment and therefore need to solve a variety of perception tasks. Information has to be extracted from the raw data recorded by the robot's sensors, before any action can be performed. These tasks, which can be collectively seen as a part of a robot's cognitive functions, interpret streams of data for an indefinite amount of time.

Designing and implementing perception algorithms for a robotic system often begins with prototyping and experimentation, followed by iterative refinement. This is especially the case in research and education, where existing functionalities should be reused and improved. Reusability is an important characteristic of a well designed system. To create a reusable piece of software, it is important to maximize modularity, such that modules can be shared by different projects.

Many modern robotic systems and frameworks, such as ROS [1], are implemented using message passing. By only depending on the types of messages coming in and going out, this approach results in a very low coupling between modules, which simplifies team work. The ROS graph can be viewed as *dataflow* programming, where information flows from sources, like camera drivers, through

various other processes into sinks, for example the motor drivers. Different processes in such a system can read and produce messages at different frequencies and the overall *asynchronous* dataflow is thus hard to describe and control.

Synchronous data flow (SDF), on the other hand, models the flow of data explicitly and is widely used in signal processing. In SDF it is known a priori how many messages a node will consume and produce in each iteration. Processes that read one message on each incoming channel and produce one message on each outgoing channel are called homogeneous and allow an efficient execution without needing message queues, as shown by Lee and Messerschmitt in [2].

The asynchronous approach is well suited for modelling an entire robotic system, since different subsystems have different requirements: Low-level control has to be executed at very high frequencies, whereas high-level decision making might only be necessary relatively infrequently. Perception is a subsystem that is situated in between the two extremes, preferably running at frequencies equal to the data acquisition rate. The sequential nature of most perception algorithms is ideally implemented using SDF and derivations thereof. There already exist many approaches to model perception as a pipeline of processes, to which we will compare our approach in Section 5.

The main idea of this paper is to define a pragmatic dataflow model that can be directly used for prototyping perception algorithms with visual programming:

- We develop an implementation of coarse-grain synchronous data flow models without message queuing. Our model is based on SDF and is tailored to allow interactive dataflow graph manipulation. We explicitly model computation parameters in the data flow and introduce aspects of control flow. We show how the scheduling of such graphs can be controlled by the user (Section 2).
- We provide an open-source implementation of the presented dataflow framework, called the *Cognitive Systems Algorithm Prototyper and EXperimenter* (CS::APEX), which consists of a graphical user interface and an execution back end. The user interface allows direct interaction with the structure of the dataflow graph and introspection into the flow of data (Section 3).
- We share the lessons we learned using the framework in research projects, educational courses and robot competitions. Additionally we show how the introduction of control flow aspects makes this model more expressive for more complex applications than pure dataflow models (Section 4).

2 Graphical Model for Coarse-Grained Dataflow

The idea of this section is to give a useful and expressive model for the implementation of a dataflow-based visual programming framework with the following requirements:

- Dataflow graphs can be created and modified at run-time.
- Parameter values can be changed through the dataflow and by the user.
- Data in the flow can be inspected at any step in the process.
- The user can influence the scheduling policy of the processing nodes.
- Irregular events are made visible to the user and can be handled coherently.

2.1 Homogeneous synchronous data flow

Flow-based programming has been extensively investigated in literature. Our approach is derived from *homogeneous synchronous data flow* (SDF) [2], which is widely used for signal processing pipelines.

At first we will look at a general dataflow. Let $G = (V, E)$ be a directed graph of processing nodes $v_k \in V$ and message connections E. For each process v_k we define a set of inputs $\mathcal{I}_k$ and outputs $\mathcal{O}_k$

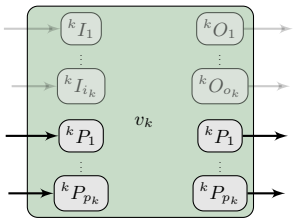
$$\mathcal{I}_k = \left\{ {}^kI_1, \ldots, {}^kI_{i_k} \right\}, \mathcal{O}_k = \left\{ {}^kO_1, \ldots, {}^kO_{o_k} \right\},$$

as visualized in Fig. 1(a). We add an edge $({}^kO, {}^lI)$ to E if an output ${}^kO$ of node v_k is sending messages to an input ${}^lI$ of node v_l. Inputs and outputs are typed and can be connected if their types are *compatible*. An output can be connected to arbitrarily many inputs, but inputs can only be connected to one output. We allow for an input to be *optional*, which means that it is ignored, if it is not connected to an output. It is treated as a normal input, otherwise.

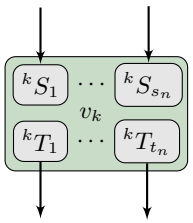
When the process v_k is executed, it will read the message from each $I \in \mathcal{I}_k$ and write a message to some of the $O \in \mathcal{O}_k$. After the execution of v_k, the messages for $O \in \mathcal{O}_k$ will be forwarded to all the connected inputs. If there are no messages to be sent, we propagate a special `Nothing` token instead.

2.2 Parameters

For each computational node $v_k \in V$ we define a set of parameters $\mathcal{P}_k$, which are treated both as inputs, as well as outputs of v_k by adding additional inputs $\mathcal{I}_k^P \subset \mathcal{I}_k$ and outputs $\mathcal{O}_k^P \subset \mathcal{O}_k$ (Fig. 1(a)). Parameters are declared for each node type and control the internal processing of each instance of that node type. A node v_k can read its parameters at any time and is allowed to change them.



(a) Inputs and outputs of node v_k. Parameters ${}^kP_i$ are both inputs and outputs, where messages are automatically forwarded.



(b) Triggers and Slots on a dual graph structure. Signals sent from Triggers represent *events* and do not behave like flowing data.

Fig. 1. A node consists of inputs and outputs (left), as well as slots and triggers (right). These entities implement data-driven and control-driven flow respectively.

A parameter's value can also be changed by other means: An incoming message at a parameter input port causes the value of that parameter to be updated. At every firing of v_k, all the parameters' values are sent as messages on the corresponding output ports. Both mechanisms together allow values of different parameters to be synchronized without their nodes knowing about each other.

Parameters behave the same as regular input or output ports and can be connected to any other port, making the parameter accessible to the network. Values computed by a node can be manipulated using further processing nodes and then assigned to a another node's parameter, for example. At the same time, this explicit modelling enables a user interface to present control panels to adjust the parameter values, giving the user a more direct control over the data flow.

2.3 Event-based message passing

Pure dataflow is ideal for processing indefinite streams of information. A robot's perception can be modelled using multiple subsystems that are based on dataflow. There are, however, stimuli the system has to respond to, which are more irregular and often not predictable. These can be both triggered by external means or detected within the data stream. We call these stimuli *events* and introduce means to handle them in a coherent framework with the dataflow itself.

To realize such non-dataflow communication between nodes in G, we define sets $\mathcal{S}_k$ and $\mathcal{T}_k$ representing *Slots* and *Triggers* of node v_k analogously to $\mathcal{I}_k$ and $\mathcal{O}_k$ (see Fig. 1(b)). Triggers can be used to signal *events* to another node by connecting them to slots, contributing an edge $({}^iT, {}^jS)$ to the edges E. Triggers can only be connected to slots and outputs only to inputs. Every node is therefore a composition of inputs, outputs, signals and slots, as visualized in Fig. 1.

In contrast to the dataflow, events are more irregular and should be handled asynchronously once they are triggered, so that not all triggers have to receive a message at the same time. This means that slots can be connected to multiple triggers and vice versa. By not using the dataflow to send events between nodes, we avoid sending special marker messages. Additionally, disjoint dataflow subgraphs can run at different frequencies but can still communicate via events.

A useful application of events is to control aspects of the execution and to send commands between nodes independently of the dataflow, for example disabling currently unneeded nodes, or resetting internal state of more complex, stateful nodes. As any subgraph H of G can be represented by a node v_H, this mechanism can be used to easily enable or disable large portions of the dataflow, depending on the global state.

2.4 Scheduling

Scheduling the execution of G requires a policy to decide, when to execute the different nodes. Node v_k becomes *enabled* once each of its inputs has received a message and each of its outputs can send a new message. Nodes without inputs, also called *sources*, are enabled whenever their outputs can send messages. *Sinks*, that is nodes without outputs, are enabled when all inputs have received

a message. Enabled nodes can be executed whenever processing resources are available. Once executed, v_k reads messages from the inputs, processes them and (possibly) generates outputs.

An output can send a new message, once all previously sent messages have been read down-stream. Messages are read before they are processed, so that earlier nodes in the network can already be executed again, even if their previously generated messages are still being processed. This allows the scheduler to perform pipeline execution of sequentially connected nodes.

There can be multiple enabled nodes at any time, which allows concurrent execution. The order in which these nodes are executed does not matter, they can run in a sequence or fully in parallel. To allow dynamic modification of the dataflow graph at run-time, a static scheduling scheme, as originally developed for SDF graphs [2], cannot be employed. Any dynamic scheduling algorithm can be implemented, however.

Events are also managed by the scheduler: If a slot has received an event, the node will be required to handle the event as soon as possible. The scheduler can execute event handling routines at any time, if the node is not currently being executed. If a node is enabled and also has pending events to handle, the scheduling policy can decide which will happen first.

3 Implementation in CS::APEX

The aim of CS::APEX[1] is to be a user-centred platform for developing and experimenting with flow-based algorithms for robots and other cognitive systems, encouraging modularity, extensibility and accessibility. We follow a pragmatic approach to dataflow programming, focused on providing a user-friendly interface, concentrating on speeding up the prototyping experience, providing useful user feedback and making parameters of the system more easily accessible. Resulting dataflow networks can be directly deployed on a robot (Section 4).

The framework consists of two components: A graphical user interface (see Fig. 2) based on Qt5 and a computation back end library for scheduling and maintenance[2]. We achieve modularity by implementing the flow-based graph structure presented in Section 2, encouraging users to implement component-based solutions that only depend on message types and can thus be easily reused. Extensibility is accomplished by a plug-in system, which makes modification of the main components unnecessary and simplifies the distribution of implemented computing nodes among collaborators.

The user interface allows the user to dynamically add and delete computation nodes at run-time. Nodes can also be disabled and enabled, moved and copied. Furthermore, the user can add and delete connections between nodes and inspect the transmitted values in these connections. No scripting or manual configuration file editing is required. The user interface is used to generate a network and to

[1] CS::APEX and its documentation are available for download at `http://www.ra.cs.uni-tuebingen.de/software/apex/`

[2] An overview video can be seen at `http://youtu.be/weFZZrQ1BeE`

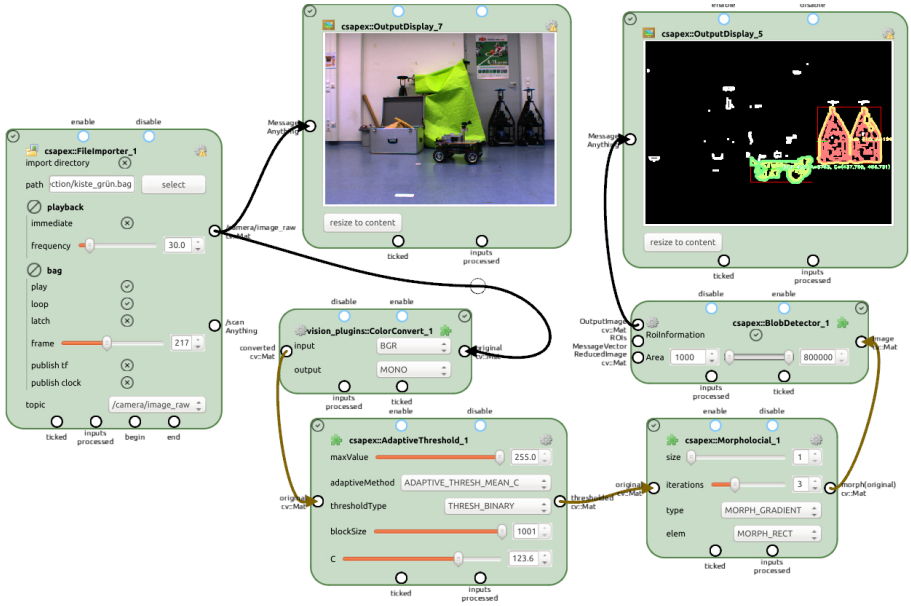


Fig. 2. Exemplary workflow that imports a ROS bag, converts the camera image to gray values, performs adaptive thresholding, morphological operations and blob detection.

provide feedback during the prototyping process. Once the configuration is done, the UI is no longer needed and the graph can be executed in a headless fashion. In this way, a prototype configuration can be used on a robot, without a screen attached.

3.1 Nodes and Messages

Adding custom functionality is possible by implementing new node types and providing parameters to allow fine tuning. Computation nodes are written in C++11 and dynamically linked once they are needed. We provide multiple ways to add new processing nodes: Nodes can be derived from a base class `Node`, or from specialized base classes, like image filters. Furthermore, we provide a utility class that can automatically generate nodes from a given C++ function by analyzing the function signature using template meta programming techniques.

When a new node is needed, three functions have to be implemented: `setup` tells the system about the required input and output ports, as well as event triggers and slots. The function `setupParameters` declares the parameters of the new node. Every parameter will automatically be wrapped into an UI widget so that the user can manipulate it easily (see Fig. 3). Finally, `process` implements the new functionality, i. e. messages from inputs are read and processed, before output messages are generated and published on the outputs.

Packages can also provide custom message types. Other plug-ins can use these messages without having to worry how to do I/O with them. Authors of a custom message can provide functions to serialize the message using YAML, to read them from a file, to visualize them and publish them via ROS. We have implemented messages for integral types, strings, images, laser scans and more. We have also implemented support for OpenCV[3] and the point cloud library (PCL)[4] via independent libraries.

3.2 Parameters

Parameters are central to our approach, since the direct feedback from changing parameters can speed up rapid prototyping. As described in Section 2, we create a pair of one input and one output per parameter. Since this would lead to a lot of ports for nodes with many parameters, we only add inputs and outputs for parameters that are marked as *interactive* by the user.

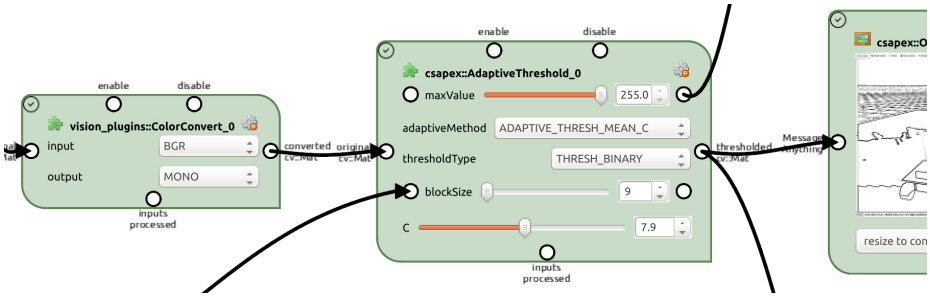


Fig. 3. A node that performs adaptive thresholding on an image with one image input and one output. There are five parameters: two floating point ranges (`maxValue`, `C`), one integer range (`blockSize`) and two sets (`adaptiveMethod`, `thresholdType`). `maxValue` and `blockSize` are connected to the data flow. The node has automatically generated slots to enable and one to disable it and a trigger that fires once the input is processed.

We provide different types of parameters: Boolean, integral and floating point values, ranges, intervals, pairs and more specialized variants like angles, file paths and color values. To enable fast changes to parameter values, we wrap each parameter into a specialized UI widget that allows the user to quickly modify the parameter's value. UI elements and the parameters themselves are strictly separated, so that a graph can be used on a robot completely without a graphical interface. We render these controls directly into the graphical representation of the node, as can be seen in Fig. 3. This way, the controls are physically located where they are used, which makes it easy to find the right parameter to change and allows us to provide helpful information for parameters via tool-tips.

[3] OpenCV is available at http://www.opencv.org/
[4] PCL is available at http://www.pointclouds.org/

3.3 Scheduling and concurrency

Executing a node does not have any side effects on other nodes in the graph. For this reason, concurrency can be achieved without any effort on the client side, merely the scheduler has to deal with the details of concurrent programming. This takes away the burden on inexperienced programmers, who can make full use of parallel architectures with thread-agnostic modules. Parallel execution is not necessary though, all nodes can be executed sequentially as well.

Even though users should not have to be concerned about matters of concurrency, we want to enable them to take control of the scheduling process. For this reason, we employ a distributed scheduling scheme using thread pools. Every node is assigned to a thread group and all the nodes in a thread group are managed by one scheduler. By default, every node is managed by a single scheduler, as to not fully utilize every CPU core, if that is not desired. The interface allows users to define their own thread groups and assign nodes to them.

4 Applications

Since the development of CS::APEX began in 2012, we have developed more than 300 plug-ins to solve a variety of perception problems for research projects and robotics competitions: We achieved the second place in the SICK Robot Day 2014 [3], which was an object-delivery competition. We also deployed the framework for the perceptions tasks at the SpaceBot Camp 2015[5], which was hosted by the national aeronautics and space research centre of Germany (DLR).

Additionally, we are using the framework in research projects: We developed a person-recognising autonomous transportation system in the BMBF-founded project PATSY[6], using dataflow graphs to detect obstacles and people in point clouds. In the project IZST IOC 104[7], founded by the state of Baden-Württemberg, we developed vision algorithms for laparoscopic surgery.

In the remainder of this section we present two use cases in more detail.

4.1 SICK Robot Day 2014

The SICK Robot Day 2014 [3] was an object-delivery competition, in which robots had to autonomously detect and approach filling stations where they received labelled objects. The objects then had to be delivered to delivery stations determined by the object label. Delivery stations were marked with large number signs, as shown in Fig. 4(a). We split the detection task into four separate dataflow graphs: Cross-hair detection, number sign detection, bar code reading, and environment map analysis. Here we focus on the specific task of number sign detection and demonstrate the benefits of using our framework.

[5] http://www.dlr.de/dlr/desktopdefault.aspx/tabid-10081/151_read-15747
[6] http://www.ra.cs.uni-tuebingen.de/forschung/patsy/
[7] http://www.ra.cs.uni-tuebingen.de/forschung/Chirurgische_Navigation

(a) The robot's camera image, showing an occluded bull's eye target and the number sign of station 1.

(b) Overlay of the detections of two dataflow graphs, one detecting numbers and one detecting bull's eye targets.

Fig. 4. SICK robot day 2014: Sensor input and visual feedback generated.

Many necessary processing steps had already been implemented as nodes before and could be reused without modification: First we had to rotate the image, because the camera was mounted on its side. Then we performed color conversion from YUV to BGR. Next we applied an adaptive thresholding step (see Fig. 3) to separate light from dark areas. After that we performed connected component analysis to find dark image areas.

The only missing component is defined by the following interface: A node that takes a vector of image components, classifies the components and outputs the classified result as vector of regions of interest. We implemented a plug-in to communicate with the Java-based neural network framework JANNLab developed by Otte et al. [4], using an MLP to perform the classification. By keeping a generic message passing interface, the resulting node is easily reusable. For training of the neural network we were able to reuse already existing nodes: We used the same preparation steps as described above and then employed an export node to write the training samples to a directory.

We were able to easily split up tasks and improve team work. The interfaces between the different nodes had to be changed multiple times, so that the initial interface specification was soon outdated. Being able to clearly see what kind of data a node consumes and produces, these changes had nearly no impact on productivity and could be accounted for with minimal or no changes to other nodes. We also benefited from fast parameter tuning and debugging due to the immediate visual feedback. This was especially helpful during the final moments of preparation for the competition.

We were able to use the resulting graphs on our robot without modification. Running the framework without a user interface entailed minimal overhead compared to a hand crafted program, since feedback and debug information (as seen in Fig. 4(b)) are not computed without a UI. Experiments show an average of $k\times$ 1.2 ms overhead, where k is the longest distance from a source to a sink.

4.2 Evolutionary optimization of a ROS node

The second use case we want to share is the application of evolutionary algorithms to optimize the parameters of a ROS node called `laser_scan_matcher` using the *Differential Evolution* algorithm (Fig. 5). Using a plug-in to communicate with the optimization framework Eva2 developed by Kronfeld et al. [5], we were able to find parameters that minimized the trajectory error on our dataset.

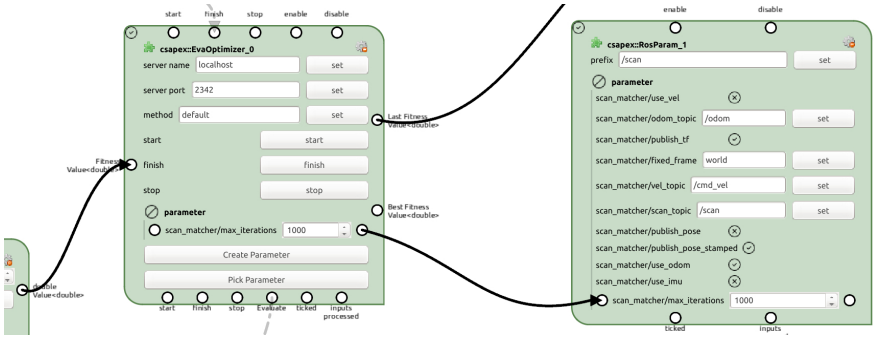


Fig. 5. Evolutionary optimization of a ROS node for localization. A node for calculating a fitness value has to be implemented. For each newly generated population, the `Evaluate` trigger is fired. A `finish` slot is triggered when the current population of parameters is fully evaluated and the fitness is finalized. Eva2's and ROS parameter setter's parameters are connected. (To simplify we only show one connection.)

The scan matching process was running in a separate ROS node, whose parameters were set using a generic ROS parameter interface, that can be seen in Fig. 5. The values of these parameters were determined by the optimization framework for each iteration and were propagated through the dataflow. Triggers and slots were used to control the optimization process: A trigger was executed every time a run of the scan matcher was complete. This trigger was connected to a slot in the optimizer, which caused a new parameter set to be generated.

The communication with the optimization framework is generic and can be applied to other problems. The user selects an arbitrary set of parameters using the user interface, which adds it to the optimization process. This is possible due to the generic integration of the parameters into the dataflow and can be implemented purely on a plug-in basis, since it does not require any modifications to the framework. The only additional functionality, that has to be implemented, is a node to calculate a fitness value for the current parameter set. This value is needed by the optimizer to assess different parameter combinations.

Furthermore, this example demonstrates the usefulness of the additional event-based mechanism. Events can be used to control the execution of different dataflow subgraphs. This way, processes like these can be automated and controlled based on the dataflow. To the best of our knowledge, such an approach to parameter optimization is not possible in related frameworks.

5 Comparison to Related Work

Many tools and frameworks utilizing flow-based programming have been published in related domains, such as Ptolemy II by Eker et al. [6] and the Ptolemy-based Kepler by Ludätscher et al. [7]. Special purpose frameworks include the Robot Task Commander by Hart et al. [8], the Konstanz Information Miner (Knime) [9] for data mining, the Waikato Environment for Knowledge Analysis (WEKA [10]) and Orange [11] for machine learning and MeVisLab [12] for medical image processing. There are also commercial products based on data flow processing, for example LabVIEW and MATLAB Simulink. A cognitive architecture for artificial vision is described by Chella et al. [13] using a more symbolic approach than the one presented here, whereas the approach presented by Hochgeschwender et al. in [14] is purely declarative.

Biggs et al. [15] provide a pipeline based approach, and show its potential with an example for point cloud processing. They impose requirements similar to the ones presented in Section 2, yet focus more on inter-node communication aspects and less on interaction. Their framework implements asynchronous dataflow, meaning that there is a need for message queues between components. They provide a user interface with which the graph can be modified at run-time and parameters can be adjusted, but they do not model parameters in the dataflow and don't seem to feature event-based functionality in their user interface.

Although our implementation can be used independently from ROS [1], we support ROS interaction, such as data import and export. We provide our user interface as a single ROS node, in which ROS topics can be subscribed and published to. ROS itself can be seen as a flow-based framework, where different nodes are separate processes and can run on different machines in a local network. ROS is implemented using the publish-subscribe pattern, where each node is publishing messages onto topics that are subscribed to by other nodes, which is another case of asynchronous dataflow. Our visual programming approach allows users to construct processing graphs on the fly via a graphical user interface instead of using text based configuration files. Our implementation can be compared to ROS nodelets, in that all processing nodes are running in the same process, yet ROS nodelets do not allow any interaction or scheduling control.

More relevant for our work is ecto [16], which grew out of the ROS scene and also represents computer vision and perception tasks as a directed acyclic graph. In contrast to ecto, our approach explicitly handles node parameters, allowing them to be used as data sources or sinks. Ecto also provides some form of event-handling, yet these are not part of the interface of a processing node as in our proposed solution. Graphs in ecto are meant to be specified and configured using python programs. There exists a web-based graphical user interface for ecto which allows users to create nodes and connect them. This approach has the advantage that the graph is accessible via the network, yet the implementation does not allow for a high level of interactivity. In our approach, the user interface is the key part of the framework and we focus on interactive graph manipulation and immediate feedback.

6 Discussion and Conclusions

We describe a graphical model based on synchronous dataflow and event-based message passing, as well as an implementation of this model in the form of a visual programming language framework called CS::APEX. Extensions to the well known SDF model are motivated by the application to prototyping perception algorithms in a scientific setting: We model program parameters directly in the dataflow, which we can use to perform automatic parameter optimization. Furthermore, we use signal-based mechanisms to model irregular events and allow users to manipulate both data flow and event handling in a coherent interface.

In other fields, graphical prototyping tools have become commonplace, yet in robotics there do not exist such standard tools. We think this is partly due to the fact, that robotics is a broad and multidisciplinary field. We aim to provide a user-friendly graphical interface that lowers the barrier to entry into robotics, especially robotic perception. The interface provides immediate feedback, allowing to visually construct new dataflow graphs at run-time, to get insight into the dataflow in real-time and to learn the effects of different parameters on the behaviour of the algorithm. Although we provide a library of reusable nodes, custom algorithms have to be implemented eventually. In contrast to fine-grained visual programming, where individual instructions are composed in a graphical interface, we rely on a plug-in based system which allows users to program custom processing nodes in C++ and then compose them visually. This approach simplifies using unknown modules, since inputs, outputs and parameters are directly displayed and can be connected visually.

On its own, homogeneous synchronous data flow seems to be a limiting factor, due to a uniform execution of the individual nodes, whereas robotic systems are composed of many subsystems requiring different update rates. Our approach, however, is meant to implement individual subsystems, such as perception modules. Many perception problems naturally show synchronous characteristics: If an online image classifying algorithm, for example, cannot operate in real-time, images have to be dropped, synchronizing the update rate of the pipeline.

Our framework can be fully applied to problems that can be separated into modules. Highly optimized algorithms that cannot be subdivided, however, have to be implemented as single nodes. This is not unusual in a coarse-grain data flow framework such as the presented approach. Even though large nodes are less reusable, they can still profit from the parameter system and other UI features such as execution profiling and data visualization.

Acknowledgement

This work is funded by the German Federal Ministry of Education and Research (BMBF Grant 01IM12005B). The authors would like to thank Sebastian Otte and Fabian Becker for providing their implementations of artificial neural networks and evolutionary optimization algorithms. Additionally, we want to thank all the students and colleagues, who have been using CS::APEX in their research, for providing constructive feedback.

References

1. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA workshop on open source software. Volume 3. (2009) 5
2. Lee, E., Messerschmitt, D.G., et al.: Synchronous data flow. Proceedings of the IEEE **75**(9) (1987) 1235–1245
3. Buck, S., Hanten, R., Huskić, G., Rauscher, G., Kloss, A., Leininger, J., Ruff, E., Widmaier, F., Zell, A.: Conclusions from an object-delivery robotic competition: Sick robot day 2014. In: Advanced Robotics (ICAR), The 17th International Conference on, Istanbul, TR (July 2015) 1–7
4. Otte, S., Krechel, D., Liwicki, M.: Jannlab neural network framework for java. In Perner, P., ed.: MLDM Posters, IBaI Publishing (2013) 39–46
5. Kronfeld, M., Planatscher, H., Zell, A.: The eva2 optimization framework. In: Learning and Intelligent Optimization. Springer (2010) 247–250
6. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity-the ptolemy approach. Proceedings of the IEEE **91**(1) (2003) 127–144
7. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E.A., Tao, J., Zhao, Y.: Scientific workflow management and the kepler system. Concurrency and Computation: Practice and Experience **18**(10) (2006) 1039–1065
8. Hart, S., Dinh, P., Yamokoski, J., Wightman, B., Radford, N.: Robot task commander: A framework and ide for robot application development. In: Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on. (Sept 2014) 1547–1554
9. Berthold, M.R., Cebron, N., Dill, F., Gabriel, T.R., Kötter, T., Meinl, T., Ohl, P., Sieb, C., Thiel, K., Wiswedel, B.: KNIME: The Konstanz Information Miner. In: Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007), Springer (2007)
10. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: An update. SIGKDD Explor. Newsl. **11**(1) (November 2009) 10–18
11. Demšar, J., Curk, T., Erjavec, A., Črt Gorup, Hočevar, T., Milutinovič, M., Možina, M., Polajnar, M., Toplak, M., Starič, A., Štajdohar, M., Umek, L., Žagar, L., Žbontar, J., Žitnik, M., Zupan, B.: Orange: Data mining toolbox in python. Journal of Machine Learning Research **14** (2013) 2349–2353
12. Bitter, I., Van Uitert, R., Wolf, I., Ibanez, L., Kuhnigk, J.M.: Comparison of four freely available frameworks for image processing and visualization that use itk. Visualization and Computer Graphics, IEEE Transactions on **13**(3) (2007) 483–493
13. Chella, A., Frixione, M., Gaglio, S.: A cognitive architecture for artificial vision. Artificial Intelligence **89**(1) (1997) 73–111
14. Hochgeschwender, N., Schneider, S., Voos, H., Kraetzschmar, G.K.: Declarative specification of robot perception architectures. In: Simulation, Modeling, and Programming for Autonomous Robots. Springer (2014) 291–302
15. Biggs, G., Ando, N., Kotoku, T.: Rapid data processing pipeline development using openrtm-aist. In: System Integration (SII), 2011 IEEE/SICE International Symposium on. (Dec 2011) 312–317
16. Ethan Rublee, V.R., et al.: Ecto - A C++/Python Computation Graph Framework (2 2015)