Journal of Intelligent & Robotic Systems (2019) 94:371–387
https://doi.org/10.1007/s10846-018-0831-7

CS::APEX: A Framework for Algorithm Prototyping and Experimentation with Robotic Systems

Modeling Perception and High Level Robot Control with Activity Flow Graphs

Sebastian Buck[1] · **Andreas Zell**[1]

Received: 4 September 2017 / Accepted: 27 March 2018 / Published online: 24 April 2018

Abstract
Robotic systems differ drastically in their sensory capabilities, their computational power and their designated tasks. For efficient algorithm development, however, we need to have a common modeling framework that enables us to generalize and re-use existing solutions. A modular approach, which is coherent across different platforms, also allows faster prototyping of new systems, given that existing functionality can be reused from already implemented modules. In this paper we develop a modeling framework based on data flow graphs that achieves the following goal: We first merge synchronous data flow and reactive programming into hybrid flow graphs, where we explicitly model synchronous and asynchronous data flow. Then we transfer concepts from finite-state machines to achieve a coherent framework which we call *Activity Flow Graphs*. The flow of activity enables us to model high level states directly in the data flow graph. The result is a single computation graph that can express both perception and high level control aspects of any robotic system. This theoretical foundation is the core of our open-source software framework CS::APEX, which allows the creation, manipulation and evaluation of Activity Flow Graphs and enables rapid prototyping and experimentation and can be used with any robot supporting the *Robot Operating System* (ROS). We then demonstrate the framework with two high level models for a fetch-and-delivery robot and a person following robot.

Keywords Data flow programming · Activity flow graphs · Framework · Rapid prototyping · Robotics · Mobile robotics

1 Introduction

A common way to model any complex system is using the various forms of *Unified Modeling Language* (UML) diagrams. *Finite-state machines* (FSMs) are often used, both in the design and as the foundation of the implementation. They are, however, not really capable of modeling concurrent or data-driven processes. Data flow representations are commonly used to describe such data processing pipelines [21]. They are mostly stateless and therefore they nicely complement state machines.

Our work is inspired by UML activity diagrams [10], which can express computational and organizational processes. Instead of combining data flow and activity diagrams, we propose a novel model that unifies both domains:

- We propose a framework called *Activity Flow Graphs* (AFG). An AFG can represent perception and high level mission control of a robot in a common data flow graph.
- We show how to transform an existing state machine based system into an AFG.
- We present an open-source software framework called the *Algorithm Prototyper and Experimenter* (CS::APEX), which provides the necessary environment for implementing and executing activity flow graphs, as well as a graphical user interface that allows rapid prototyping with a visual programming or visual configuration approach.

The proposed system is based on *Synchronous Data Flow* (SDF) [19] and reactive programming. Both SDF and reactive programming are common ways to model data flow graphs and offer complimentary benefits: SDF graphs can be seen as *function evaluations*, where each

✉ Sebastian Buck
sebastian.buck@uni-tuebingen.de

[1] University of Tübingen, Sand 1, 72076 Tübingen, Germany

node in the graph processes all its incoming edges at the same time, thereby synchronizing the calculation. Reactive programming, on the other hand, is common in areas such as user interface design, where the data flow edges are seen as *events* and are processed asynchronously, whenever they arise. The combined framework, which was published in [6], is called SDF+ in the following and will be reviewed in Section 3. In Section 4 we unify the two distinct components of SDF+ to form a hybrid flow graph (HFG) model, which explicitly models synchronous and asynchronous data flow. This model can represent pure SDF and reactive programming graphs. In addition, many constructs resulting from the combination of the two components can be represented in a more concise way.

While HFGs can model synchronous, asynchronous and hybrid data flow graphs, high level states still have to be represented in an additional model, such as a *finite-state machine* (FSM). In many cases, this increases the complexity of a system, due to the usage of multiple, differing modeling paradigms. To eliminate the need for another system, we introduce the concept of *activity* in Section 5, which results in the *Activity Flow Graph* (AFG) model. The AFG model is capable of representing hybrid data flow graphs and high level state representation in a common flow graph. In Sections 6 and 7 we then model two different robotic systems using AFG.

Finally, in Section 8, we describe our open-source implementation CS::APEX, which provides a back-end for running and maintaining AFG graphs, as well as a front-end graphical user interface for the manipulation and evaluation of these graphs at runtime.

2 Related Work

In this paper we propose a novel way to model high level robot control based on data flow. Many existing approaches [2, 6, 12, 16] that model perception and mission control in a single framework use a combination of multiple models, such as data flow for perception and finite-state machines for high level mission control. Statemate [15], an earlier system, uses three separate model types: Module, state and activity charts. We, however, define a coherent computation graph model that can be used to model both perception and high level mission control in a single data flow framework. The proposed model is also designed to be adapted into a graphical user interface (cf. Fig. 1) to enable rapid prototyping and experimentation in the robotic scene.

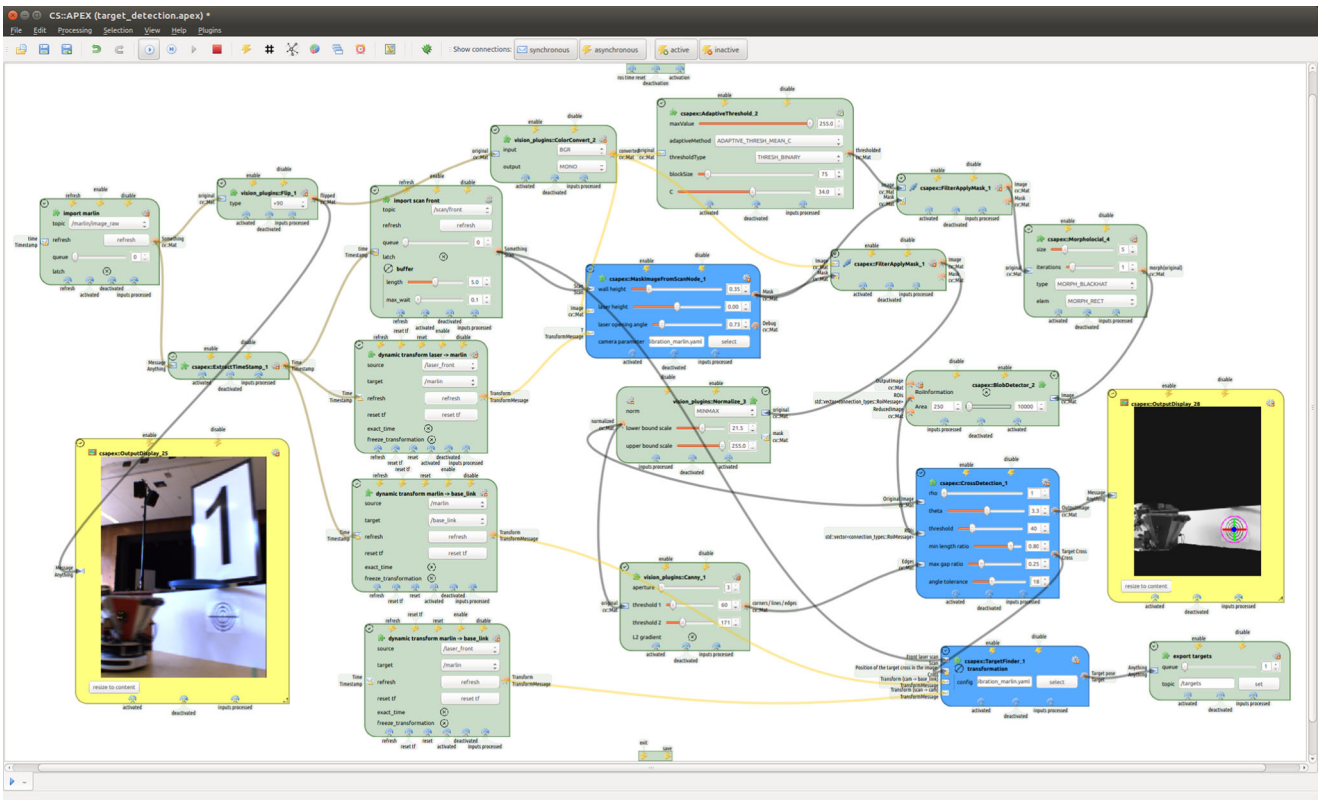


Fig. 1 Screen shot of a fully functional AFG model in CS::APEX. The graph performs several computer vision calculations to detect bull's eye signs for the SICK robot day 2014 competition [5], which is described in detail in Section 6. The graph was designed and evaluated using the CS::APEX GUI, which was turned off during the competition

One of the best developed, explicit data flow models is *Synchronous Data Flow* (SDF), which was first described by Lee et al. [19]. SDF is synchronous, which means that a processing node can only be executed, once all its inputs have received a message. This synchronizes parallel nodes, since the pipeline is blocked, until all nodes have finished processing. Additionally, SDF is also commonly constrained to be *homogeneous*, which means that a node always consumes and produces exactly one message on each input and output in each execution. A well-known example of a successful implementation of SDF is the LUSTRE language [13]. LUSTRE is an early SDF-based programming language that has been designed to implement reactive systems. Many other tools and frameworks utilizing flow-based programming have been published in related domains, such as Ptolemy II [11] and the Ptolemy-based Kepler [21]. Special purpose frameworks include the Robot Task Commander [16], the Konstanz Information Miner (Knime) [1] for data mining, the Waikato Environment for Knowledge Analysis (WEKA [14]) and Orange [9] for machine learning and MeVisLab [3] for medical image processing. There are also commercial products based on data flow processing, for example LabVIEW and MATLAB Simulink. More relevant for our work is ecto [12], which grew out of the ROS [22] 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. The ecto framework 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.

Other approaches base their unified model on finite-state machines and extend them by also modeling data flow. RAFCON by Brunner et al. [4], implements state machines that support hierarchies and concurrency. The data flow is used to represent parameters and return values for the states. The system needs a separate solution for the implementation of perception tasks. Sequentially constructive statecharts [25] are another way to represent synchronous computation based on state machines, which are designed for safety-critical applications.

To summarize, we compare our approach to the two closest related works in Table 1. The comparison is performed based on each model's capability to represent synchronous and asynchronous data flow graphs, as well as high level state transitions.

Flow diagrams are also common in workflow specification for businesses. Data flow and control flow are used in business process management to analyze and verify workflow processes [23, 24]. As with the similar sounding ActivityFlow [20], a workflow process schema typically specifies activities that constitute the workflow process and dependencies between these activities. Activities thus represent steps required to complete a business process. In our model, we view activity as an abstract property that flows through a graph, similar to data and control flow.

Table 1 Comparison to related work

Framework	Synchronous	Asynchronous	High level state
SDF	+	0	−
reactive	0	+	−
ecto	+	0	−
RAFCON	−	+	+
SDF+	+	0	−
HFG	+	+	−
AFG	+	+	+

Comparison of different approaches' capabilities to model synchronous and asynchronous data flow, as well as high level state transitions: (+) well-suited, (0) possible, but laborious, (−) not possible

3 Synchronous Data Flow and Event-Based Message Passing

The proposed approach is an extension of the SDF model and is hierarchically defined based on the SDF+ model published in [6], which we shortly summarize here. At first we look at the synchronous data flow model. We define both a formal and a graphical way to represent an SDF+ flow graph. Let $G = (V, E)$ be a directed graph of processing nodes V and edges E. Then G combines both aspects from SDF and event-based programming, where each $v_k \in V$ represents a processing node. All nodes exclusively communicate via message passing, which is realized using the connections E. This way, each node is a functional unit, solely described by its inputs and outputs.

3.1 Synchronous Data Flow

We represent each node v_k as a set of *ports* that can receive or send messages. Nodes can exclusively communicate using message passing via these ports, which defines a clear and precise interface between them. A node represents a single process in the computation graph. Each $v_k \in V$ is also assigned a *processing function* f_k that reads messages from incoming connections, then performs arbitrary computations and generates messages on the outgoing edges. In addition to the inputs, f_k can also read user-controlled parameters, which fully participate in the data flow.

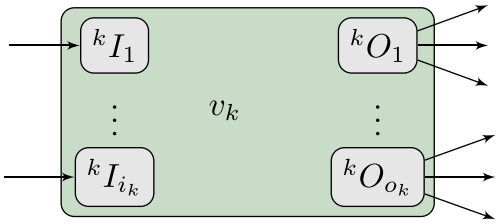


Fig. 2 A node v_k reads data from inputs ${}^{k}I$ and writes data to outputs ${}^{k}O$

For each node v_k we define *inputs* $\mathbf{I}_k$ and *outputs* $\mathbf{O}_k$

$$\mathbf{I}_k = \left\{ {}^{k}I_1, \ldots, {}^{k}I_{i_k} \right\}, \tag{1}$$

$$\mathbf{O}_k = \left\{ {}^{k}O_1, \ldots, {}^{k}O_{o_k} \right\}, \tag{2}$$

where node v_k has $|\mathbf{I}_k| = i_k$ inputs and $|\mathbf{O}_k| = o_k$ outputs. This is visualized in Fig. 2.

We add an edge $({}^{k}O, {}^{l}I)$ to E if an output ${}^{k}O$ of node v_k is sending messages to an input ${}^{l}I$ of node v_l (cf. Fig. 3). 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.

We call process nodes without inputs *sources* and nodes without outputs *sinks*. When the processing function f_k is executed, it will read the message from each $I \in \mathbf{I}_k$ and write a message to some of the $O \in \mathbf{O}_k$. After the execution of f_k, the messages for $O \in \mathbf{O}_k$ will be forwarded to all the connected inputs.

3.2 Event-Based Message Passing

Pure data flow is ideal for processing indefinite streams of information. A robot's perception can be modeled using multiple subsystems that are based on data flow. 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 data flow itself.

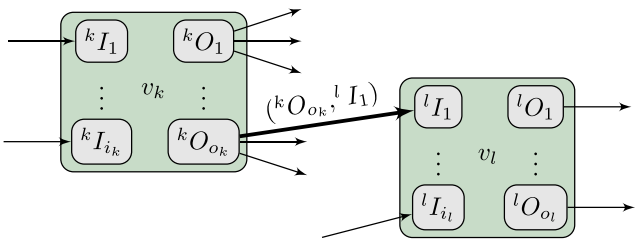


Fig. 3 Outputs can send messages to many inputs. Inputs can only be connected to one output

To realize such asynchronous data flow communication between nodes in G, we define sets $\mathbf{S}_k$ and $\mathbf{E}_k$

$$\mathbf{S}_k = \left\{ {}^{k}S_1, \ldots, {}^{k}S_{s_k} \right\}, \tag{3}$$

$$\mathbf{E}_k = \left\{ {}^{k}E_1, \ldots, {}^{k}E_{e_k} \right\}, \tag{4}$$

representing *slots* and *events* of node v_k analogously to $\mathbf{I}_k$ and $\mathbf{O}_k$ (cf. Fig. 4). Every node $v_k = (\mathbf{O}_k \cup \mathbf{I}_k \cup \mathbf{E}_k \cup \mathbf{S}_k)$ is therefore a composition of inputs, outputs, events and slots. In the graphical notation, we show slots on top of a node and events on the bottom.

Events can be used to send *signals* to another node by connecting them to slots, contributing an edge $({}^{i}E, {}^{j}S)$ to the edges E. Using SDF+, events can only be connected to slots and outputs only to inputs, which gives as a definition for any connection e between two nodes v_i and v_j as

$$e \in \left(\mathbf{O}_i \times \mathbf{I}_j\right) \cup \left(\mathbf{E}_i \times \mathbf{S}_j\right). \tag{5}$$

We will lift this constraint when we introduce hybrid data flow in Section 4.

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

4 Hybrid Flow Graphs

In Section 3 we have recapitulated the core of the SDF+ model published in [6]. Now we propose a unified approach that solves many of the limitations of SDF+. We begin with a hybrid flow graph (HFG) model. Let G be an SDF+ graph,

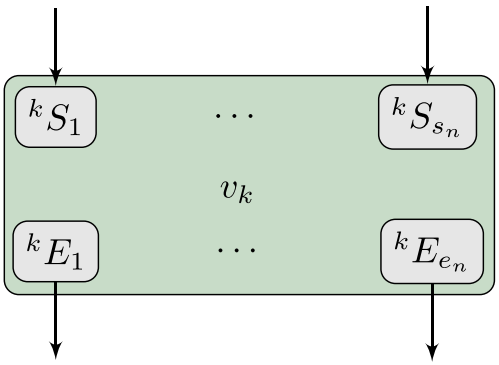


Fig. 4 Events and Slots on a dual graph structure

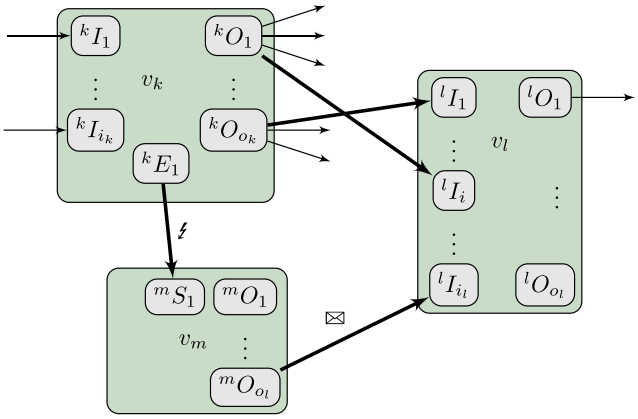


Fig. 5 An example graph in the detailed depiction. Optional ✉ tokens signal synchronous handling, and ↯ asynchronicity

then each node $v_k \in V$ again consists of four different types of ports, with synchronous inputs $\mathbf{I}_k$ and outputs $\mathbf{O}_k$, as well as asynchronous slots $\mathbf{S}_k$ and events $\mathbf{E}_k$. The SDF+ model requires a connection e between nodes v_i and v_j to satisfy Eq. 5, i.e. e can only connect inputs and outputs, or events to slots (cf. Fig. 5).

When all inputs of a node have received a token, the node processes them and generates output tokens. The inputs are thus handled synchronously. Furthermore, the node can only process another set of tokens on its inputs once all outgoing tokens have been processed down-stream.

Slots, on the other hand, are handled asynchronously. When an event sends a signal token to a slot, the token can be processed immediately. In addition, the event can immediately be triggered again and is not disabled until the tokens are processed down-stream.

Limitations of the SDF+ model are mostly linked to this distinction of the two flow types, which requires special nodes in the computation graph to translate between synchronous data flow and asynchronous event flow. The HFG model is an extension of SDF+ and represents a single, coherent graphical model. HFG also distinguishes synchronous from asynchronous data flow, however it allows for direct connections between the two, without explicit translation nodes. Inputs $\mathbf{I}_k$ and outputs $\mathbf{O}_k$ are still expected to be handled *synchronously*, whereas slots $\mathbf{S}_k$ and events $\mathbf{E}_k$ are *asynchronous*. We now generalize the edges E to allow hybrid connections, which means that a valid edge e between two nodes v_k and v_l is given by

$$e \in (\mathbf{O}_k \cup \mathbf{E}_k) \times (\mathbf{I}_l \cup \mathbf{S}_l)\,. \tag{6}$$

This definition allows the creation of hybrid connections, connecting outputs to slots and events to inputs. As we will shortly see, this allows more flexible graph constructions, such as synchronously reacting to an event. Furthermore, no more translation nodes are necessary to transition between asynchronous and synchronous data flow.

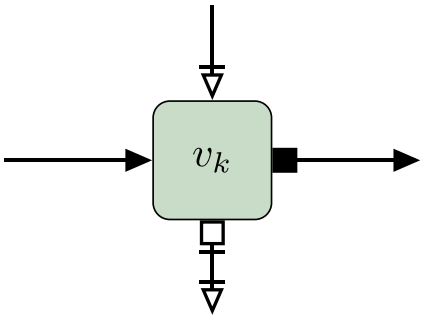


Fig. 6 Dark ports (inputs and outputs) are handled synchronously, light ones (slots and events) asynchronously

4.1 Simplified Graphical Representation

In most cases, nodes have multiple different message and event ports. This can quickly complicate the graphical notation, especially in large graphs (cf. Fig. 5). We therefore simplify the notation and do not show the individual ports where possible, as is demonstrated in Figs. 6 and 7. We omit multiple edges between nodes, if they are not essential in the current situation. Connections in the simplified notation are interpreted according to the style of the port. Dark ports are handled synchronously and light ones asynchronously.

The main difference between SDF+ and HFG is visualized in Figs. 8 and 9, where we compare the two ways to translate between asynchronous and synchronous data streams, once in Fig. 8 with SDF+ and once in Fig. 9 with HFG. Since HFG is a superset of SDF+, however, both are valid HFG graphs.

4.2 Exemplary Usage

4.2.1 Synchronous Reaction to Asynchronous Event

In SDF+ we use buffers to translate events to synchronous data flow. However, avoiding the use of an unbounded buffer is preferable in long lasting data flow graphs.

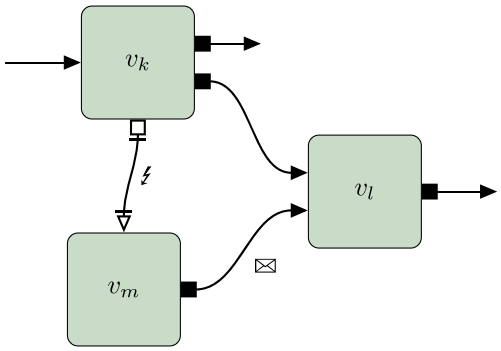


Fig. 7 The example from Fig. 5 in the simplified variant. We omit representing ports and multiple edges in contexts where they are not necessary

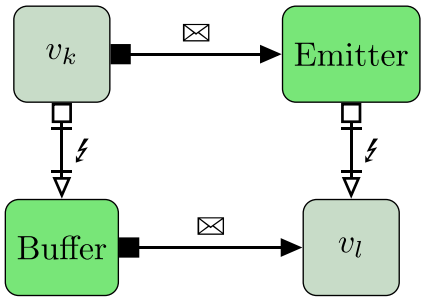


Fig. 8 In the SDF+ model, translation requires additional nodes and connections

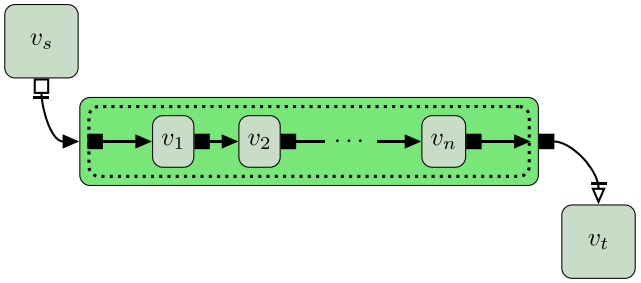


Fig. 10 A synchronous sub-graph is used to handle the event generated by v_s

Additionally, bounded buffers require tokens to be dropped, once the bound is reached. In contrast, the HFG model allows complete interaction between synchronous and asynchronous data streams. This makes it possible to implement a synchronous reaction to an event, guaranteeing that every event is handled. Figure 10 shows an example graph, where an event is synchronously handled and then converted back into an event.

4.2.2 Converging Data Streams

Besides a synchronous reaction to an event, where we use hybrid connections from events to inputs, we can also make use of hybrid connections from outputs to slots. This is demonstrated in Fig. 11, where a node v has several slots that each receive tokens from possibly different, synchronous data streams. All of these streams can operate at different frequencies, they can even be irregular. The node v handles all received tokens separately, in contrast to the synchronous data flow case, where all inputs have to hold a token for the node to become enabled.

5 Activity Flow Graphs

With HFG we have defined a coherent data flow model for synchronous data flow and reactive programming. However, we still need additional structures to model high level state that changes orders of magnitudes less frequently than the data flow. Usually, this is done using some form of an external finite-state machine. Instead, we propose to use data flow as a primary model by eliminating the need for an FSM with the introduction of the *activity* concept. We show that the expressiveness of FSMs can be translated into HFG with minimal changes to the model, resulting in *Activity Flow Graphs* (AFGs). Subsequently we show how to apply AFGs to model high level robotic mission control.

5.1 State Representation

In the example shown in Fig. 12, we demonstrate an HFG solution to a stateful problem, where the system has to wait for the arrival of a specific message, before it can continue. We use this in our examples to let our robots wait for a signal before the start of autonomous motion.

The example demonstrates one of the main reasons, why HFG alone is not sufficient to model more complex systems: Flow graphs are inherently stateless, with all components representing functional units. A sophisticated robotic system, on the other hand, always needs to consider the global state of the robot and its mission.

In the example we can also see that the vertices of the graph can be split into two disjoint sets, one representing data flow and the other representing the FSM, where interactions between the two sub-graphs are only possible via asynchronous message passing. Instead of giving a more precise definition for FSMs in the context of flow graphs,

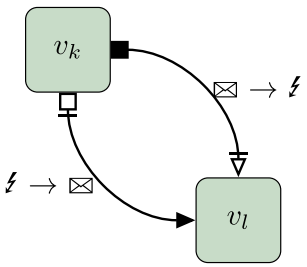


Fig. 9 With HFG the translation is directly possible and is therefore less verbose

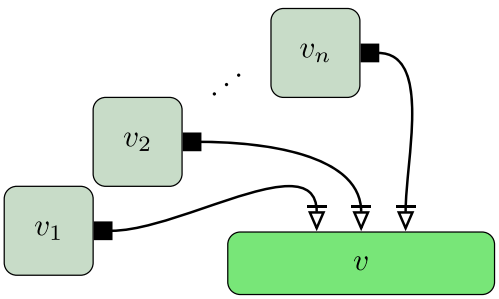


Fig. 11 Node v handles different synchronous data streams asynchronously

we directly integrate a generalization the concept of states and state machine graphs into the unified AFG model.

5.2 Activity Flow

Let us again consider the example from Fig. 12. If we only observe the state machine, we can interpret the FSM graph as a data flow graph: Each state is a data flow node and the transition edges between states are asynchronous connections. When the FSM transitions from state S_1 to S_2, we can imagine a message being sent via the transition δ_2. The FSM always has a single *active* state, which is why we can say that the message sent via δ_2 transfers the *activity* of the graph. This also holds in general, there is always exactly one active node in an FSM and the activity can only be transferred using a transition.

We now bring over the activity concept to HFG, with the aim to eliminate the need for an FSM altogether. Let G be an HFG with nodes V and edges E. For each $v_k \in V$ we define an *activated* attribute

$$A(v_k) \in \{0,\ 1\}\,, \tag{7}$$

where 1 indicates an activated node. We initially set $A(v_k) \leftarrow 0$ for all $v_k \in V$. The behavior of an activated node is the same as that of an active state in an FSM, i.e. it stays active until the activity is transferred to another node. Importantly, whether a node is activated or not does not necessarily influence the behavior of the node in the data flow. If no activity is injected into the graph, we have a pure HFG model. Otherwise, an activated node still participates normally in the data flow, except for specially implemented nodes that behave differently, depending on their activation.

For a node to become active, it has to receive the activity via a connection, i.e. via a message passed from another node. We therefore define an *activity modifier* attribute

$$a(\bullet) \in \{0,\ +1,\ -1\} \tag{8}$$

for tokens $\bullet$, which can take on one of three values: 0 representing no change, $+1$ representing an activation and -1 a deactivation. An inactive node receiving a token with $a(\bullet) = 1$ becomes activated. Likewise, if an activated node receives a token with $a(\bullet) = -1$, it will be deactivated.

In order to control the flow of activity, we need an equivalent structure to the transition in FSMs. We therefore define an *active* attribute for connections $e_k \in E$ as

$$A(e_k) \in \{0,\ 1\}\,, \tag{9}$$

such that only *active* connections can transfer the *activated* property of a node to another. Whereas the *activated* state of a node changes over time to reflect the state of a higher level system, the *activity* of an edge is a fixed part of the graph structure.

Sending an active token deactivates the node to achieve a similar behavior to an FSM. Therefore we set the activity of node v_k to 0 once it has sent an active token. This way, analogously to an FSM, activity is transferred between nodes via edges. However, we do not have to define additional node or port types. Instead, every output ${}^kO$ and every event ${}^kE$ of node v_k is able to relay activity and v_k can be implemented in an activity-agnostic way. This definition results in a more general model than FSM, since an active node can send activity to more than one succeeding node at a time. Additionally, nodes can become activated from other sources, e.g. any inactive node can send a signal $\lightning$ with a $a(\lightning) = +1$.

More than one node can be active concurrently, which means that not every AFG can be represented as an UML state diagram. Using the familiarly sounding UML *Activity Diagram*, we can, however, represent the active sub-graph, if we restrict the generation of activity to a single source.

5.3 Example Use Cases

5.3.1 Data Flow and High Level State Interaction

We can implement a solution to the problem shown in Fig. 12 using activity flow. The idea is to wait until a node v_k has produced a message and only afterwards continue

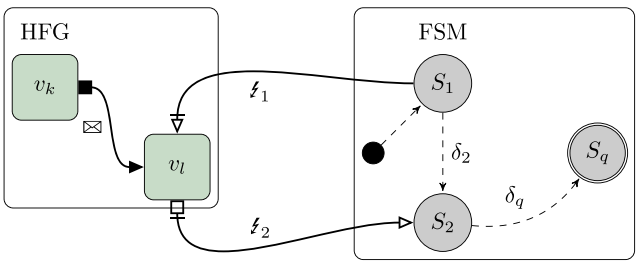


Fig. 12 Technical example of an interaction between an HFG and an FSM to wait for a message ✉: First state S_1 emits signal $\lightning_1$ and transitions via δ_2 to state S_2. $\lightning_1$ is handled by v_l, which sends $\lightning_2$, once ✉ is received. Only now will the FSM transition to S_q via δ_q

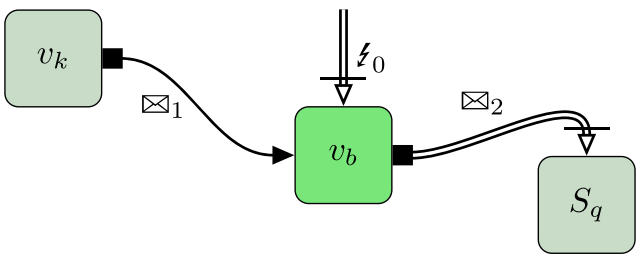


Fig. 13 A solution to the same problem as in Fig. 12, with active edges shown as a double line. The buffer node v_b relays incoming tokens ✉1 to ✉2. Once $\lightning_0$ activates node v_b, the buffer transfers the activity to S_q with the next ✉2

propagating the high level state. This can easily be modeled using AFG, as demonstrated in Fig. 13. We can use a buffering node v_b, as in Fig. 8. A node v_k is sending a message to v_b, which simply relays it in form of an event to S_q.

As long as v_b is inactive, the signals $\boxtimes_2$ will have a modifier of $a(\boxtimes_2) = 0$, so S_q will not be activated. However, once a $\lightning_0$ token with $a(\lightning_0) = +1$ arrives, v_b will be activated. The next relayed $\boxtimes_2$ will then have a modifier of $a(\boxtimes_2) = +1$, deactivating v_b and activating S_q. We therefore have constructed a graph where S_q will be activated only after v_k has produced a message.

Note that we do not have to consider activity in the implementation of v_k, v_b or S_q, all of which are completely activity-agnostic. Only when constructing the graph we have to consider which edges have to be active for the activity flow to be correct. We have a lower complexity than in the solution shown in Fig. 12, which needed two different graphs with completely different semantics, as well as many more nodes and edges.

5.3.2 Unified Data Flow and Robot Control

The previous example has shown how we can use the activity concept to control the current state of a robotic system from within the data flow. Additionally, we can model the reverse situation, where we control the data flow according to the high level state. This can be used to activate parts of a perception pipeline exactly when it is needed and thereby conserve computational power when it is not needed.

With SDF+ we have to use an FSM to deactivate parts of the system, where we needed additional complexity and where we did not have fine-grained control over the data flow. With AFG we can, for example, throttle a currently unimportant sub-graph to a lower frequency instead of completely deactivating it. All of this can be achieved without any of the involved nodes having to be specially implemented. This is especially useful for robotic systems with limited computational power, where not all pipelines can be executed at the same time.

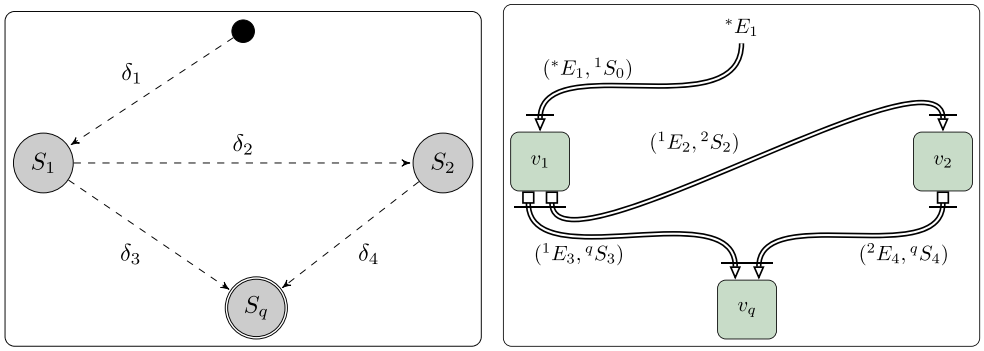
5.3.3 Finite-State Machine

Additionally, we show that activity flow graphs are capable of representing any finite-state machine, by translating a general FSM into AFG. Let $(\Sigma, S, s_0, \Delta, F)$ be a finite-state machine where Σ is the input alphabet, $S = \{s_0, \ldots, s_n\}$ is the set of all states, s_0 is the initial state, Δ is the set of all transitions and F is the set of all final states.

Fig. 14 shows an example of how we construct an AFG $G = (V, E)$:

(i) Create a global event ${}^*E_1$ that emits an active token to initialize the graph and represents the initial state s_0.
(ii) For each state $s_k \in S$, except for s_0, create a node v_k representing that state.
(iii) For each transition $\delta_t = (s_i, s_j) \in \Delta$ between s_i and s_j:
  (a) If $s_i \neq s_0$, create an event ${}^iE_t$ for node v_i and a slot ${}^jS_t$ for node v_j, and add an active connection $({}^iE_t, {}^jS_t)$ to E.
  (b) Otherwise create a slot ${}^jS_t$ for node v_j and add an active connection $({}^*E_1, {}^jS_t)$ to E.
(iv) Implement each node v_k analogously to the implementation of state v_k. Instead of invoking a transition $\delta_t = (s_i, s_j)$, publish an active signal token $\lightning$ with $a(\lightning) = +1$ on the event ${}^iS_t$.

This shows that activity flow graphs can represent any finite-state machine. We just have to ensure for each node, that only one active token is sent at a time, otherwise G implements a non-deterministic finite-state machine. AFGs are, however, more expressive than FSMs, because activity can be transferred to multiple nodes in parallel. This can, for example, be used to concurrently control the pose of an omnidirectional robot and determine its orientation.



(a) Finite-State Machine (b) Activity Flow Graph

Fig. 14 Translation of an FSM into the AFG framework. States are replaced by nodes, transitions are modeled as pairs of events and slots. Active edges are shown with double lines

Fig. 15 The robot receives an item

Fig. 16 The item is removed at a delivery

6 Modeling a Fetch-and-Delivery Robot

As a first example we describe an AFG-based solution used for a robot competition. SICK robot day 2014 was a competition held in an approximately circular arena with about 12 m diameter in which four robots competed at the same time. With a time limit of 10 minutes, each robot had to alternately collect labeled objects at *filling stations* and transport them to *delivery stations* based on the object label (cf. Fig. 15). The octagonal collection area, where the robots had to approach one of four filling stations, was located in the central part of the arena. Once a robot had reached the designated filling spot, it had to signal the human operator to provide an object to be delivered. These objects were known to be wooden cubes of side length 6 cm, which were labeled on each side with the designated delivery station using a bar code imprint. On the outer boundary there were four delivery stations, one for each possible object label. Whereas the filling stations could be selected freely, every delivery station could only be used to hand off one specific object type. Therefore robot-robot interactions were inevitable and had to be accounted for.

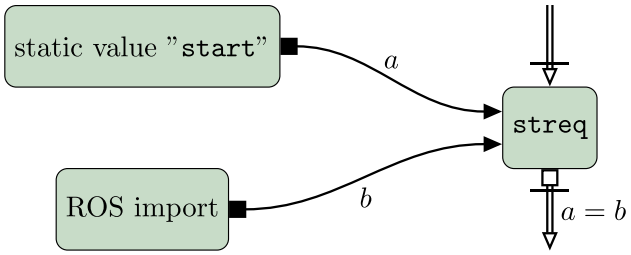
Both filling and delivery stations comprised of a ring of diameter 30 cm at a height of around 50 cm and a bull's eye target sign for precise positioning (cf. Fig. 16). Delivery stations were additionally marked with a large sign displaying one of the digits printed onto the wooden cubes. The robots therefore had to be able to detect both the bull's eye and the number signs. Once a robot had reached a station, it had to activate a green signal lamp to indicate its intention to collect or deliver an object. A human operator would then insert or remove a cube within at most 10 seconds.

6.1 Building Blocks

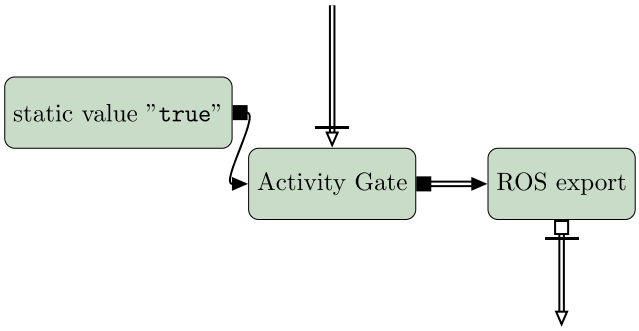
We reuse all the perception sub-graphs presented in [5]. These include the detection of cross-hair targets and number signs, the detection and evaluation of items in the robot's basket, as well as the analysis of the environmental map. Underlying systems, such as navigation and mapping, are also kept. This leaves the implementation of additional functional units to replace the finite-state machine needed for the high level control of the robot.

6.1.1 Waiting for a Start Signal and Controlling the Signaling Light

In the SDF+ model, we require an explicit state that remains active until a starting signal is sent to the robot. With AFG,



(a) The start signal is a string message received on a ROS topic. The *streq* node triggers a signal, when a and b are equal. Only then will activity be transferred.



(b) Setting the signal light is done via a boolean ROS topic. Since the activity gate only forwards messages when it is activated, it forwards exactly one (active) message.

Fig. 17 Two simple sub-graphs to replace explicit states in an FSM. Activity is only transferred via doubly drawn lines

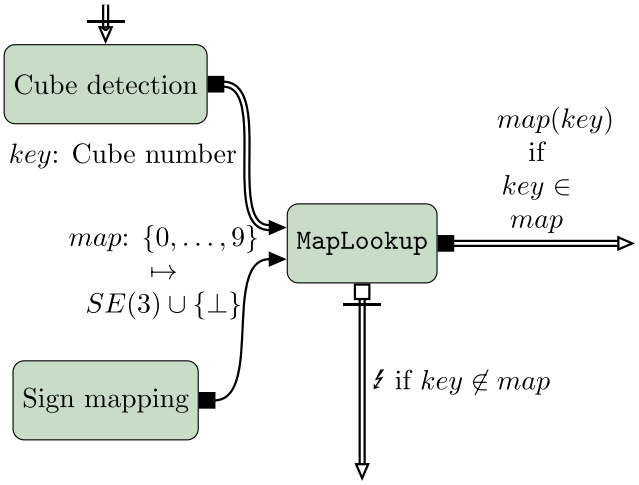


Fig. 18 The sign map node produces a map of known sign poses. Unknown signs are mapped to ⊥. The cube detection node sends the number printed on detected cubes. The activity flow is routed depending on whether the sign pose is known or not

we do not need an explicit state to wait for a message, as was already demonstrated in Fig. 13. Instead, we only need a source node that reads on a data channel. Every produced token is then compared to the predefined starting command, and activity is transferred once the incoming token matches. We do not need any problem specific implementation, everything can be achieved using the sub-graph shown in Fig. 17a.

Similarly, controlling the signaling lamp is achieved by sending a command via an outgoing ROS data channel. As shown in Fig. 17b, we need to employ an activity gate. The gate only forwards messages when it is active. This way we ensure that only one message is sent and with this message the activity will be transferred to the exporting node. Once the message is exported, the node triggers an event which will further forward the activity.

In both cases we do not need any application specific nodes. Rather, every node in the active sub-graphs shown in Fig. 17 can be interpreted as its own state.

6.1.2 Searching for a Delivery Station

To deliver an item, the robot has to know the location of the delivery stations. A Gaussian Mixture Model is used to track the positions of all known stations. The map may, however, not yet contain the required station, which needs to be handled robustly. This is done using the graph shown in Fig. 18: At first the cube detection node becomes active. Once a cube is detected, the imprinted number is sent to a node called *MapLookup*, which also takes the current state of the sign map. The output of the sign map node is a function that maps the number read from a cube to a pose in the world, whereas unknown stations are represented by ⊥. The *MapLookup* node emits a signal if the lookup fails, i.e. when the result is ⊥. Otherwise it forwards the result of applying the function to the key. This way, the activity flow branches into two different flows, depending on whether the delivery station is known or not.

In case the station is known, the pose resulting from the map lookup is used as the goal pose for the navigation system. Otherwise, the robot switches into an exploration mode, where it drives counter-clockwise around the arena, examining the outside wall for possible stations. This is again implemented similarly to the FSM case, where a problem specific node is implemented to generate goal poses for the navigation stack, based on the current pose of the robot and the results of map analysis.

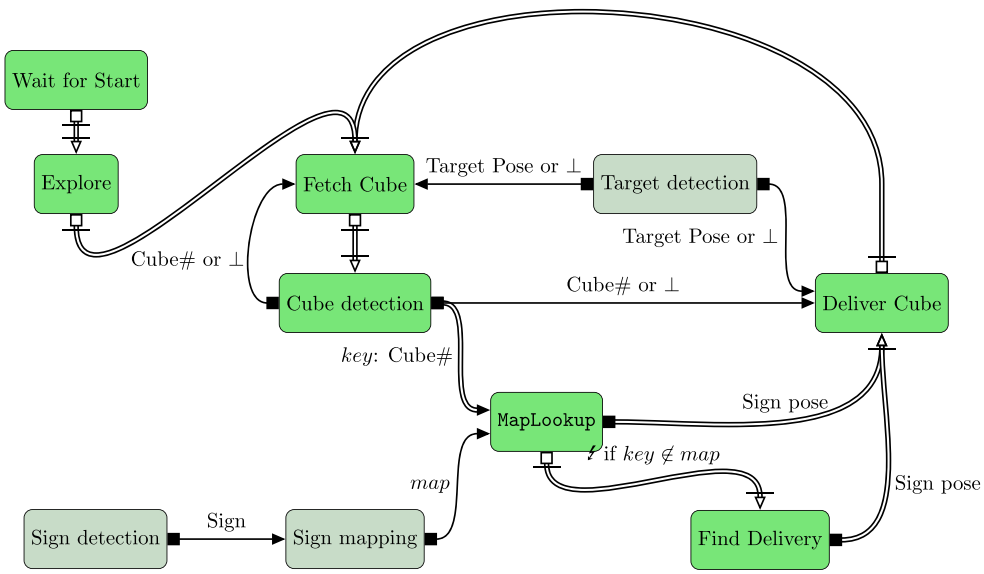


Fig. 19 A mostly complete activity flow graph that solves the SICK robot day 2014 challenge. The highlighted nodes form the active sub-graph, which also includes the cube detection, one of the perception graphs shown earlier. Sensor source nodes and command sink nodes are not shown to reduce complexity

6.1.3 Positioning the Robot Reactively

Positioning the robot is done analogously to the FSM implementation: To navigate the arena we employ a full navigation stack, including path planning and following. Additional movements are done reactively, i.e. not using global positioning but relative measurements. For this, the reactive programming approach achieved by the asynchronous data flow of AFG is a perfect match.

In the original design, the robot first quickly moved about 2 m diagonally forward to populate the occupancy grid map with enough measurements to extract the central filling stations. This center exploration is its own state in the SDF+ model and can be replaced with a more generalized AFG node *MoveRobot*. When activated, the node *MoveRobot* generates motion commands that move the robot for a given distance in a predefined direction. Additionally, the FSM has a state to back up the robot by about 1 m after collecting or delivering an item, for which we can immediately reuse the *MoveRobot* node.

In order to precisely place the robot below either a filling or a delivery station, another node called *PositionToTarget* is implemented. Whereas an FSM state would handle the complete control of the robot, the AFG node is implemented in a more functional way: As input it takes the pose of the robot, the detected target signs and the estimated pose of the target sign, and as output it generates a motion command. The AFG approach has the advantage that additional processing nodes can be used before and after the generation of the movement command. For example, a Bayesian filter can be used to improve the localization accuracy of the target without changing the implementation of *PositionToTarget*. Another possibility would be an obstacle avoidance node that post-processes the generated movement command to avoid collisions.

Fig. 20 The robot's task is to accompany a person at higher speeds in outdoor environments. High level control is needed due to unknown environments and obstacles

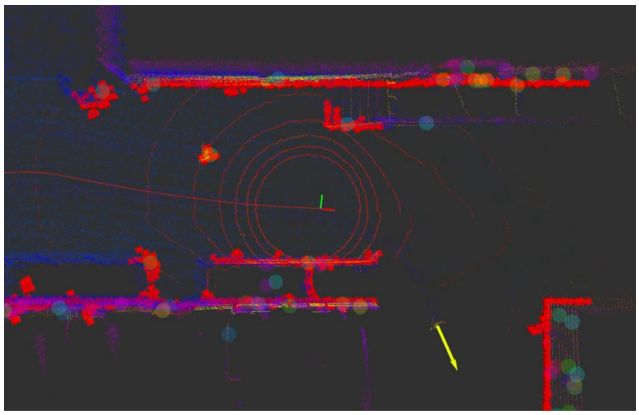


Fig. 21 Perceived obstacles in the environment (red) and the person's velocity (arrow)

After the robot is positioned relative to the target sign, it is commanded to drive towards the wall in front of it until a critical distance is reached. This is achieved using a general motion node *MoveToObstacle*, which has as input the current laser scanner measurement and as output a motion command. Again, the advantage of the AFG approach is that we can arbitrarily pre-process the input data using a data flow graph, e.g. filtering the laser scan. With a single framework it is also easier to find good parameters to tweak the performance of the robot, especially when using a graphical user interface.

6.1.4 Hierarchical Graph

We combine multiple nodes into a sub-graph node to reduce complexity and to make it easier to reuse existing solutions. We combine selecting a filling station, navigating to the station, positioning to the target, and evaluating the cube into a *Fetch Cube* node. Similarly, we group the sub-graphs shown in Fig. 17 into *Wait for Start* and *Explore*, respectively. Delivering a collected cube is also grouped

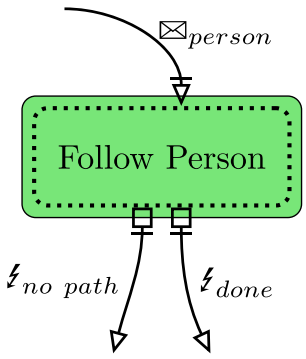


Fig. 22 The *Follow Person* graph takes a person and either emits $\lightning_{done}$ or $\lightning_{no\ path}$, depending on whether a path was found

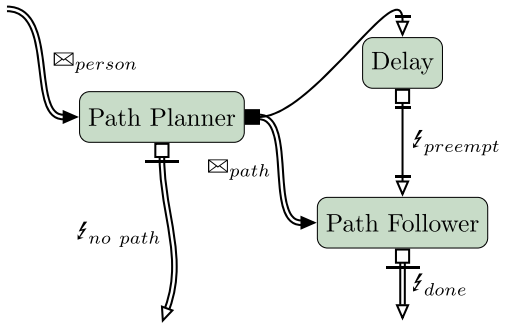


Fig. 23 Implementation of the nested graph: A path to the person is planned. If a path was found, the follower controls the robot until the $\lightning_{preempt}$ token is received

into *Deliver Cube*, which strongly resembles the *Fetch Cube* node. The only difference between the two is the determination of the target pose and whether to wait for a cube number or for a $\perp$ token at the output of the cube detection node. Combining the individual building blocks, we get the AFG shown in Fig. 19, which completely solves the challenge without the use of any additional high level control structure. Also, Fig. 1 shows a screen shot of the exact graph used for cross-hair target detection in the competition.

7 Modeling Human-Following Outdoor Robot

As another example, we model a high level control system for a Robotnik Summit XL robot with a stereo camera setup. The resulting robotic system is capable of following a person at higher speeds in outdoor environments, as is visualized in Fig. 20. Using the kinematic controller published by [17], the robot is able to follow a runner with a maximum velocity of 2.5 m s^{-1}.

7.1 Perception

We use a single AFG to model perception and high level control. We use a 3D Velodyne VLP-16 LIDAR to detect obstacles in the environment and to track the person. The data is first imported using a generic ROS importing node. Every scan is then segmented using a variant of the generic obstacle detection algorithm published in [7]. Detected obstacles are then clustered and tracked to distinguish dynamic from static obstacles. A person detection pipeline based on a Nerian SP1 stereo camera can optionally be used to further distinguish persons from other dynamic obstacles.

The result of the perception part of the graph is visualized in Fig. 21, where detected obstacles are shown overlaid on the LIDAR scan. The currently tracked person is visualized as an arrow, where the length of the arrow depicts the person's velocity.

7.2 High Level Control

Fig. 22 shows the central graph component needed to model this robot. The *Follow Person* graph takes a person's location and tries to navigate the robot there. The implementation of the graph (cf. Fig. 23) uses a predefined navigation strategy to try to reach the person. If a path can be found using this strategy, the path is sent to the *Path Follower* node, which steers the robot along the path. In addition, a *Delay* node receives the same path and emits a $\lightning_{preempt}$ token after a few seconds. This token stops the path following process and returns the activity to the outer graph

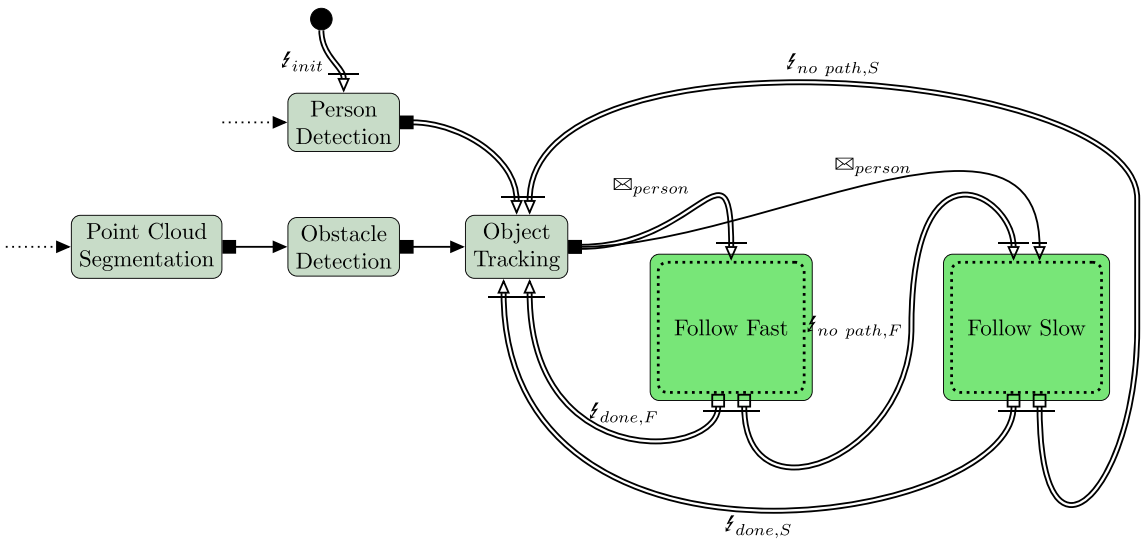


Fig. 24 Using two instances of *Follow Person* we achieve a high level controller, which can recover from navigation errors

Fig. 25 Trajectory of the robot following a jogger for ca. 1.7 km in varying environments

via $\lightning_{done}$. This way, *Follow Person* steers the robot for a predefined amount of time towards the person.

By combining differently parametrized instances of *Follow Person*, we can handle challenging situations. Figure 24 shows a graph using two instances: *Follow Fast* and *Follow Slow*.

Initially, the *Person Detection* node is activated. The activity is then transferred to the *Object Tracking* node with the first message produced by *Person Detection*. When a person is being tracked by the *Object Tracking* node, the fast following node gets activated via the $\boxtimes_{person}$ token. While it is activated, the robot follows a path towards the person that only contains forward motion. This way the robot can drive at its maximum speed without braking.

When *Follow Fast* is done, i.e. either the end of the planned path is reached or a time-out was signaled, the activity flows back to the object tracking via $\lightning_{done,F}$. This way we achieve a circular flow of control that regularly activates the *Follow Fast* node. In case no forward path can be found in *Follow Fast*, the $\lightning_{no\ path,F}$ token transfers the activity to *Follow Slow*, another instance of *Follow Person*. In this state the navigation is configured to slower velocities and the path planner is allowed to plan both forward and backward motion. This way, the robot can robustly follow a person at the maximum speed possible. It will only slow down if no forward path can be found, which rarely happens in real experiments. One such experiment in shown in Fig. 25, where the robot followed a person for ca. 1.7 km.

8 Implementation of Activity Flow Graphs in CS::APEX

In the previous sections we have described activity flow graphs in detail and we have shown multiple example applications. Both robotic systems have been developed using a common 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 (shown in Figs. 1 and 26) allows direct interaction with the structure of the data flow graph and introspection into the flow of data. This enables rapid prototyping of algorithms and also encourages reusable code and modular designs.

The aim of CS::APEX is to provide a user-centered platform for developing and experimenting with flow-based algorithms for robots and other cognitive systems, encouraging modularity, extensibility and accessibility. We are focused on providing a user-friendly interface, giving useful user feedback and making parameters of the system more easily accessible. Resulting data flow networks can be directly deployed on a robot.

The framework consists of two components: A graphical user interface (see Fig. 1) based on Qt5 and a computation back-end library for scheduling and maintenance.[1] We achieve modularity by implementing the flow-based graph structure defined by the AFG model hierarchy, presented in Sections 3, 4 and 5. Flow graphs automatically encourage 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 runtime. 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. No scripting or manual configuration file editing is required. The user interface is

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

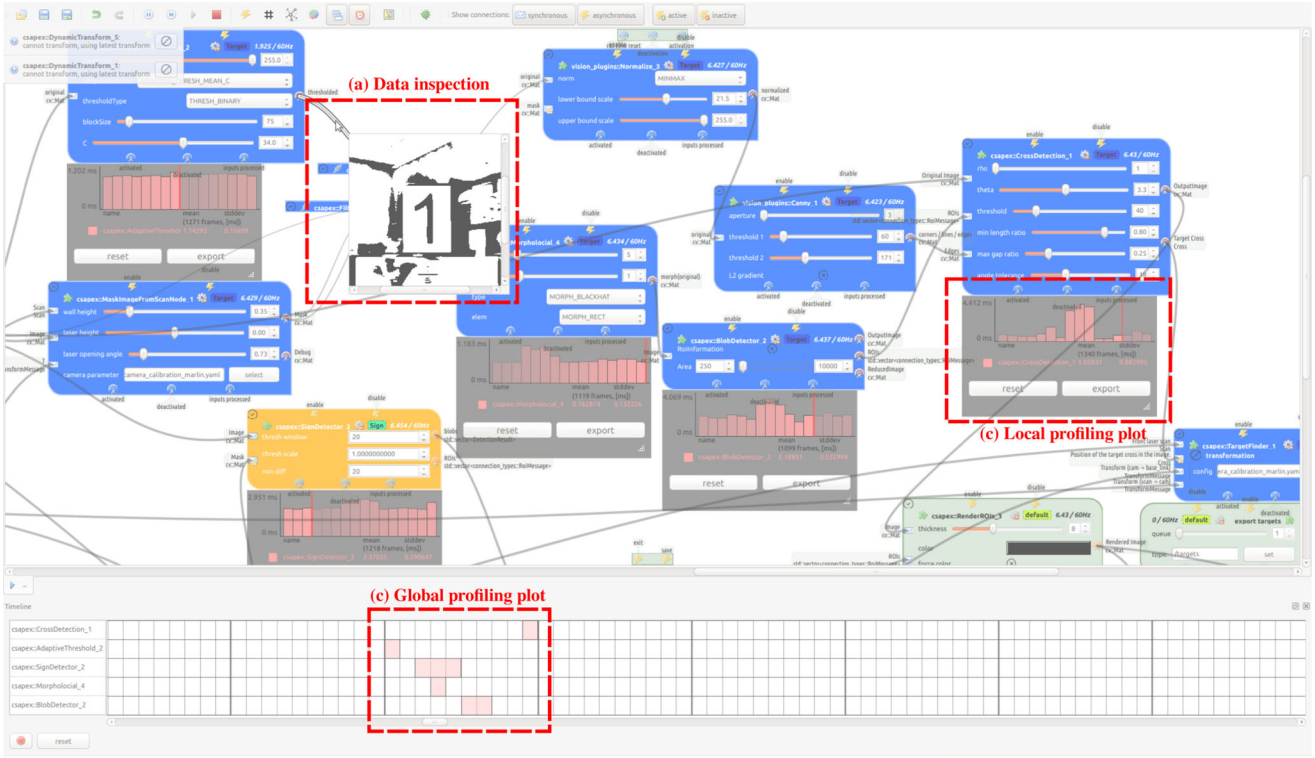


Fig. 26 Screen shot of some of the prototyping and profiling capabilities in CS::APEX. **a** The user can inspect the data flow at any point, here the output of a morphological operator node is shown. **b** Here five nodes are being profiled, next to each of them we can see a bar plot of the durations of the last executions. **c** A global profiling plot on the bottom of the screen gives more detailed information, accurate to 1 ms. In this example, two of the nodes are executed in parallel, indicated by overlapping red boxes

used to generate a network and to 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. This way, a prototype configuration can be used on a robot, without a screen attached.

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.

Since the development of CS::APEX began in 2012, we have developed more than 500 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, which was described in Section 6. We also deployed the framework for the perceptions tasks at the SpaceBot Camp 2015,[2] which was hosted by the national aeronautics and space research center of Germany (DLR). Additionally, we are using the framework in research projects: We developed a person-recognizing autonomous transportation system in the BMBF-founded project PATSY, using data flow graphs to detect objects and people in point clouds. In the project IZST IOC 104,[3] founded by the state of Baden-Württemberg, we developed vision algorithms for laparoscopic surgery. The framework is also used by many students to simplify the development of robotic prototypes.

The main motivation for the CS::APEX graphical user interface is to aid the user in rapid prototyping of new algorithms. As an example we show some of the available functionality in Fig. 26. The depicted graph is taken from another model used in the SICK robot day, which was described in Section 6. This graph, which was created after the competition, combines all other graphs presented before in a single instance.

First of all, the example demonstrates the profiling options available. Five nodes are profiled and for each of them a bar plot of the past execution durations is shown. On the bottom of the window there is an additional profiling

[2] http://www.dlr.de/dlr/desktopdefault.aspx/tabid-10081/151_read-15747

[3] http://www.ra.cs.uni-tuebingen.de/forschung/Chirurgische_Navigation

panel that shows the execution start and end time points for all profiled nodes. This enables rapid prototyping since the user can directly observe the effects of parameter changes on the performance of any node and on the complete system.

The second feature shown is data inspection. The user highlights a connection with the mouse cursor, which opens a second window that shows the data currently sent via this connection. Many message types that are available can be visualized this way.

The final feature shown is the assignment of nodes to thread groups, which is here additionally indicated by the color of the nodes. This way nodes from different groups can be executed at the same time, as can be observed in the profiling panel.

9 Discussion and Conclusions

An important aspect of any complex system is an architecture that is well defined and that allows all collaborating users to develop their required functionality as easily as possible. Developers in a team have different programming styles and different levels of experience, but their implemented functionality should nevertheless be as reusable and interchangeable as much as possible in order to maximize efficiency. This can best be achieved by a consensus on common interfaces between modules. The computation graph abstraction, which is the foundation of the AFG hierarchy, is very promising in this aspect. The interfaces of individual nodes in the graph are defined by the types of messages the nodes read and write. With the explicit introduction of synchronous and asynchronous input ports, this interface description is even more precise in our approach.

Conventional approaches employ disjoint structures to model complex robotic behavior: A form of a data flow graph and finite-state machines. FSMs are needed, because data flow graphs are inherently stateless, yet complex behavior requires a form of high level state management.

We have transferred the concept of activity from FSMs into the data flow. The resulting AFG model is a coherent way to model a complete robotic system without needing any additional frameworks. In many cases, AFG models require fewer states than equivalent, heterogeneous models. Additionally, the activity concept is unobtrusive to the implementation of individual nodes.

Another important aspect of large, complex systems is modularity. Individual modules should have as few dependencies on each other as possible. This means that only the modules needed to solve the task at hand have to be available. Using AFG we naturally achieve modularity, due to the foundation on pure message passing interfaces. Dependencies exist only on the types of messages sent between nodes and not on the nodes themselves. Up to this point we have implemented more than 500 individual processing node types and more than 20 messages types. Due to this modularity, the framework has successfully been used in many projects, competitions and student theses.

Another design goal for AFG is to provide a coherent approach, which means that every aspect of the computation graph should be the same for all types of nodes. This is where the lower level models (SDF+ and HFG) fall short, because they still require the use of a finite-state machine in order to implement higher level state. It is possible to use the lower level models to implement the perception part and a separate FSM to model high level control, where both graphs can communicate with each other. However, this requires the use of two completely different concepts.

Of course, we could implement a graphical user interface that combines the construction of hybrid flow graphs and finite-state machines, but then we would still need to precisely specify the ways of interaction between the two. The AFG approach instead generalizes the relevant aspects of finite-state machines and combines them with hybrid flow graphs. This combined approach is at first not as intuitive as the well-known finite-state machines, because the separation between state and data flow is lost.

However, the approach also has multiple advantages: First of all, there is no need to explicitly implement different states in the system. Every node can be directly used as a state that is active for the duration of the node's execution. This is a common case for FSMs, where we switch between states after specific computations are performed.

Second, the interactions between "states" and normal nodes are formally defined and behave deterministically. We can precisely understand the interaction between the collaborating nodes. This enables the implementation of algorithms for checking the graph structure and searching for flaws in the graph. Many flaws are detectable, such as missing connections that stall the execution of a graph or incompatible data types (e.g. image output connected to point cloud input). Since the AFG model is homogeneous, there is a guarantee that there are no errors in the amount of tokens sent to a node.

Third, since there is no distinction between different types of nodes, the model becomes simpler and easier to adapt into a graphical user interface with which users can manipulate these graphs.

Lastly, we can use a common scheduling algorithm for all nodes, which ensures that resources are properly managed. In our implementation we are using a thread pool, where each thread can implement any scheduling algorithm, such as completely fair scheduling (CFS) or stack resource policy (SRP). In our current implementation, each thread uses a simple task queue, which implements FIFO scheduling. The

user can then assign each node to any scheduler. Using the user interface, the schedulers can be managed, nodes can be assigned and the execution can be profiled.

We provide CS::APEX as an open-source implementation of a visual programming environment which implements AFG and which has already been used for various robotic systems. The core project is available at https://github.com/cogsys-tuebingen/csapex/ and various plug-in projects are also available as open-source projects. All of them are implemented in C++11, with user interfaces implemented in Qt5. Among other libraries, these offer support for ROS, OpenCV and PCL.

AFG has been used to model a freely navigating, person detecting autonomous transportation system [7, 8]. The framework has been used in two competitions: SICK robot day 2014 [6] and SpaceBotCamp.[4] Furthermore, we have used it for evolutionary parameter optimization to determine control parameters in [18].

For future work, we want to explicitly model time in the flow graph, allowing the formulation of constraints to guide the execution of more complex graphs. This would take AFGs closer to real-time systems, even on regular architectures and also allow reasoning about the execution over time using linear temporal logic. Additionally, time could be introduced into the control of the activity of nodes, to realize time-out behaviors on a model level, instead of in the implementation of individual nodes. Future releases of CS::APEX should include further features to increase the productivity of developers: The core implementation should be implemented using a client-server architecture, such that the user interface can be used on a different machine from the actual robot. This would greatly improve inspection and debugging tasks.

References

1. 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, Berlin (2007)
2. Biggs, G., Ando, N., Kotoku, T.: Rapid data processing pipeline development using openrtm-aist. In: 2011 IEEE/SICE International Symposium on System Integration (SII), pp. 312–317 (2011). https://doi.org/10.1109/SII.2011.6147466
3. 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. IEEE Trans. Vis. Comput. Graph. **13**(3), 483–493 (2007)
4. Brunner, S.G., Steinmetz, F., Belder, R., Dömel, A.: Rafcon: a graphical tool for engineering complex, robotic tasks. In: 2016 IEEE International Conference on Intelligent Robots and Systems(IROS). IEEE (2016)
5. 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: The 17th International Conference on Advanced Robotics (ICAR). Istanbul, pp. 137–143 (2015). https://doi.org/10.1109/ICAR.2015.7251446
6. Buck, S., Hanten, R., Pech, C.R., Zell, A.: Synchronous dataflow and visual programming for prototyping robotic algorithms. In: The 14th International Conference on Intelligent Autonomous Systems (IAS), pp. 911–923, Shanghai (2016). https://doi.org/10.1007/978-3-319-48036-7_66
7. Buck, S., Hanten, R., Bohlmann, K., Zell, A.: Generic 3d obstacle detection for agvs using time-of-flight cameras. In: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 4119–4124. Daejeon (2016). https://doi.org/10.1109/IROS.2016.7759606
8. Buck, S., Hanten, R., Bohlmann, K., Zell, A.: Multi-sensor payload detection and acquisition for truck-trailer agvs. In: 017 IEEE International Conference on Robotics and Automation (ICRA), 2. Singapore (2017)
9. Demšar, J., Curk, T., Erjavec, A., Gorup, Č., 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. J. Mach. Learn. Res. **14**, 2349–2353 (2013)
10. Dumas, M., Ter Hofstede, A.H.: Uml activity diagrams as a workflow specification language. In: International Conference on the Unified Modeling Language, pp. 76–90. Springer, Berlin (2001)
11. 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. Proc. IEEE **91**(1), 127–144 (2003)
12. Ethan Rublee, V.R., et al.: Ecto - A C++/Python computation graph framework. http://plasmodic.github.io/ecto/ (2015)
13. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language lustre. Proc. IEEE **79**(9), 1305–1320 (1991)
14. 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), 10–18 (2009). https://doi.org/10.1145/1656274.1656278
15. Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., Trakhtenbrot, M.: Statemate: a working environment for the development of complex reactive systems. IEEE Trans. Softw. Eng. **16**(4), 403–414 (1990)
16. Hart, S., Dinh, P., Yamokoski, J., Wightman, B., Radford, N.: Robot task commander: a framework and ide for robot application development. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014), pp. 1547–1554 (2014). https://doi.org/10.1109/IROS.2014.6942761
17. Huskić, G., Buck, S., Ibargüen González, L.A., Zell, A.: Outdoor person following at higher speeds using a skid-steered mobile robot. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). Vancouver (2017). (accepted for publication)
18. Huskić, G., Buck, S., Zell, A.: Path following control of skid-steered wheeled mobile robots at higher speeds on different terrain types. In: IEEE International Conference on Robotics and Automation (ICRA). Singapore (2017)
19. Lee, E., Messerschmitt, D.G., et al.: Synchronous data flow. Proc. IEEE **75**(9), 1235–1245 (1987)

[4] http://www.ra.cs.uni-tuebingen.de/forschung/DLR_SpaceBot_Cup_2015/

20. Liu, L., Pu, C.: Activity flow: towards incremental specification and flexible coordination of workflow activities. In: International Conference on Conceptual Modeling, pp. 169–182. Springer (1997)
21. 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. Concurr. Comput.: Pract. Exp. **18**(10), 1039–1065 (2006)
22. 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, vol. 3, p. 5 (2009)
23. Sadiq, S., Orlowska, M., Sadiq, W., Foulger, C.: Data flow and validation in workflow modelling. In: Proceedings of the 15th Australasian Database Conference, vol. 27, pp. 207–214. Australian Computer Society, Inc (2004)
24. Trcka, N., van der Aalst, W., Sidorova, N.: Analyzing control-flow and data-flow in workflow processes in a unified way. Computer science report (08-31) (2008)
25. von Hanxleden, R., Duderstadt, B., Motika, C., Smyth, S., Mendler, M., Aguado, J., Mercer, S., O'Brien, O.: Sccharts: sequentially constructive statecharts for safety-critical applications. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 372–383 (2014)

Sebastian Buck received his M.Sc. degree in computer science from the University of Tübingen, Germany in the year 2012. He is currently pursuing his Ph.D. in computer science and robotics under the supervision of Prof. Dr. Andreas Zell at the Chair of Cognitive Systems, Faculty of Science, University of Tübingen. His research is focused on environment perception, high level modeling of robotic systems and autonomous robots.

Andreas Zell received a University diploma in computer science in 1986 from the University of Kaiserslautern, Germany, an M.S. from Stanford University, USA, in 1987, a Ph.D. from the University of Stuttgart, Germany, in 1989, and the Habilitation (venia legendi) in 1994, all in computer science. Since 1995 he has been full professor for computer science at the University of Tübingen, Germany, specializing in machine learning, bioinformatics and mobile robotics. In the last five years his research has focused on deep neural networks, robot vision, navigation and SLAM for wheeled and aerial mobile robots.