View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Infoscience - École polytechnique fédérale de Lausanne



# Real-Time Control of Microgrids with Explicit Power Setpoints: an API for Resource Agents

Niek J. Bouman, Andrey Bernstein and Jean-Yves Le Boudec

***Abstract*—Renewable energy resources, such as photovoltaic panels, typically have very volatile power-injection characteristics, which poses a number of challenges to the real-time control of electrical grids that contain a significant fraction of these resources. Recently, a new paradigm for controlling such grids, termed COMMELEC, was proposed; it uses explicit power setpoints instead of droop-control. Central to this new paradigm is an abstract message format that enables resources to delegate the decisions related to their control actions to a grid controller. This is essential to the feasibility of the approach, as it makes the grid controller device-independent. However, it leaves to the resource agents the burden of translating device-specific information into this abstract format. In this paper, we present a solution to this problem; more specifically, we present a very simple Application Programming Interface (API) that can be used to design a COMMELEC-compliant resource agent. We present an easy-to-use High-Level API, which supports a pre-defined set of resources, such as a battery or a photovoltaic panel. We also describe a Low-Level API that provides full access to the underlying message format, and allows to design a resource agent that is not supported by the High-Level API. For message serialization, we use the Cap'n Proto framework, which allows for efficient manipulations of the mathematical objects used in COMMELEC.**

***Index Terms*—Smart-Grid Control, Commelec, Renewable Energy**

## I. Introduction

THE traditional approach for real-time control of electrical grids is voltage and frequency droop control. The method is simple yet very effective for controlling "classical" synchronous-generator-powered grids. The advent of using renewable energy sources gives rise to a new grid type, for which droop control is no longer the most suitable control method. This is most relevant in the low and medium voltage power distribution networks, where distributed renewable energy resources (DERs) enable the creation of *microgrids* with little or no mechanical inertia. The main problem with applying droop control to grids that encompass various non-inertial resources, like photovoltaic panels, is that the method cannot properly deal with the volatile power-injection characteristics of those devices. Indeed, these resources can cause an unpredictable reduction in the quality-of-supply (QoS) of the grid (e.g., undesired drop or raise in voltage magnitudes in the network). In this case, the droop controllers cannot utilize any information on the internal state of the different resources in the grid, and might be not able to assess the overall QoS correctly. Concretely, this means that DERs typically get curtailed by the grid's safety mechanisms to restore the QoS.

Recently, an alternative framework for the real-time control of power grids, and in particular microgrids with little or null inertia, was proposed [1], [2]. With the COMMELEC framework, electrical resources in the grid are under the control of one or several grid agents, which define explicit power setpoints in real-time (every $\sim 0.1$ sec). Contrary to classic droop control strategies, this mode of operation exposes the state of all resources, and in particular storage devices, to the local grid controller, which enables an efficient and stable operation *without* large rotating masses. The framework is designed to be robust (i.e., it avoids the problems inherently posed by software controllers) and scalable (i.e., it easily adapts to grids of any size and complexity). It uses a hierarchical system of software agents, each responsible for a single resource (loads, generators and storage devices) or an entire subsystem (including a grid and/or a number of resources). The hierarchy of control ("who controls who") in COMMELEC coincides with the electrical interconnection topology. This gives rise to the terminology of *leader* and *follower*: a leader always represents a grid agent that is the parent of all its followers, where each such follower can be either another grid agent or a resource agent.

One of the main features of COMMELEC is that it is an *abstract* framework in the sense that there is a simple *device-independent* protocol for message exchange between the agents that hides the specific details of the resources and exposes only the essential information needed for real-time control. In response to a *request* message from the leader, each follower replies with an *advertisement* message, which expresses (in an abstract way) the flexibility and constraints of that follower. In this way, COMMELEC essentially provides a *Grid Operating System* (Grid OS), similarly to a computer OS. Here, a grid agent can be viewed as an OS kernel, while the resource agents are the different applications running on the OS. The device-independent protocol thus provides a way for the applications (resource agents) to interact with the kernel. We note that, as in a regular OS, the kernel is *generic*. Namely, there is only one (generic) version of software for the grid agent that can manage any given grid.

The device-independence property is essential for the deployment of COMMELEC, however, it leaves to resource agents the burden of translating device specific information into the abstract format. In this paper, we provide a generic solution to this problem. Specifically, we present in detail how a resource agent can translate the internal objectives and constraints of the resource into COMMELEC advertisements and how to send these to the leader grid agent. To accomplish this, we propose a concrete (byte-level) representation for COMMELEC advertisements and requests, as well as a simple application programming interface (API) to use this novel message-format representation, with which one can build a COMMELEC-

compliant resource agent. (While the contents of an advertisement and a request had been defined *mathematically* already in [1], it had not yet been specified how these contents, certain mathematical objects, should be *represented* on a computer or inside a network packet, i.e., as a sequence of bytes.)

In fact, our API consists of two parts. The High-Level API is an easy-to-use API that allows to construct advertisements for a predefined set of resources, and to parse requests. It is aimed at those who want to merely use COMMELEC but have no interest in knowing the details of the actual message format. The Low-Level API, on the other hand, gives full flexibility for the user and allows to design a resource agent that is not supported by the High-Level API. However, it requires understanding the structure of the message format and hence is targeted at expert users. The High-Level API and Low-Level API are open-source and are freely available [3].

In order to illustrate the ideas involved, consider the following example. Assume that the task of controlling a *battery* resource is a part of a COMMELEC-controlled microgrid. Obviously, we want to ensure that the battery always stays within its operating constraints. In addition, we would like to achieve that the battery will be (almost) fully charged near the end of the day. During the day, the battery should provide buffer capacity to the microgrid, for example, to cope with the volatility introduced by a PV system that is also connected to the microgrid. Thus, in order to "plug" the battery into COMMELEC-controlled microgrid, we should design a *battery agent* that will translate the internal constraints and objectives of the battery into COMMELEC advertisements. We distinguish between *short-term (real-time) constraints*, related to the maximum operating conditions of the battery, and *long-term objectives*, i.e., achieving full state-of-charge at the end of each day. In Section III, we explain in detail how this translation can be done using our High-Level API, and give an example implementation in the LabVIEW platform.

The second main contribution of the paper is the definition of the actual message format, and the corresponding Low-Level API that allows to design a resource agent that is not supported by the High-Level API. Although our message format is in principle independent of the chosen serialization framework, in this paper we use the modern *Cap'n Proto* framework in order to represent a message as a sequence of bytes that is ready to be sent over the network. The choice of this framework is motivated by a number of benefits that are important for the real-time control system, such as low processing complexity and small message size. Further, it supports various programming environments and platforms. Finally, it is possible to extend the message format without breaking the compatibility with older versions of resource agents.

We evaluate the performance of our approach and compare it to other methods to encode and serialize mathematical objects. We also outline how our method can be easily made compatible with the IEC 61850 communication standard for substation automation [4].

The paper is structured as follows. Section II recalls the mathematical definition of a COMMELEC advertisement. Section III gives a step-by-step example of using the High-Level API for controlling a battery. Section IV discusses the High-Level API's support for a PV system. Section V covers details of the message format and Low-Level API. Section VI evaluates the performance of our message format representation by means of some experiments.

## II. Mathematical definition of an advertisement

In this section we recall the mathematical definition of an advertisement as given in [1]. A *power setpoint* is a tuple $u = (P, Q) \in \mathbb{R}^2$, where $P$ denotes real power and $Q$ denotes reactive power. Let $\mathcal{B}(\mathbb{R}^2)$ denote the collection of all bounded non-empty closed subsets of $\mathbb{R}^2$, and let $\mathcal{C}(\mathbb{R}^2)$ denote the collection of all convex sets in $\mathcal{B}(\mathbb{R}^2)$. A *$PQ$ Profile* $\mathcal{A} \in \mathcal{C}(\mathbb{R}^2)$ of a follower $\mathsf{F}$ represents the collection of power setpoints that $\mathsf{F}$ claims to be able to implement.

The convexity requirement on the $PQ$ profile can be viewed as a limitation that originates from the control algorithm that we currently use in the grid agent.[1] Nonetheless, we want to stress that convexity is not an intrinsic property of the $PQ$ profile, hence, there might be suitable alternative control algorithms (to be run in the grid agent) that do not require the $PQ$ profile to be convex.

A *Belief Function* is a function $\mathrm{BF} : \mathcal{A} \to \mathcal{B}(\mathbb{R}^2)$. For every setpoint $u \in \mathcal{A}$, the belief function represents the uncertainty in this setpoint implementation: when the follower is requested to implement a setpoint $u \in \mathcal{A}$, the follower states that the actually implemented setpoint (which could depend on external factors, for example on the weather in case of a PV) lies in the set $\mathrm{BF}(u)$.

A *Virtual Cost Function* is a continuously differentiable function $\mathrm{CF} : \mathcal{A} \to \mathbb{R}$. It represents the follower's aversion (corresponding to a high cost) or preference (respectively, low cost) towards a given setpoint. For example, a battery agent whose battery is fully charged will assign high cost to setpoints that correspond to further charging the battery. We have used the adjective *virtual* to make clear that we do not mean a monetary value. However, from now on, we will omit this adjective and simply write "cost function".

An *Advertisement* of a follower $\mathsf{F}$ is the quadruple $(\mathcal{A}, \mathrm{BF}, \mathrm{CF}, s)$, where $s = (P, Q)$ is the currently implemented power setpoint. We note the an advertisement is typically valid only for one COMMELEC cycle and is thus periodically sent by the follower to its leader.

## III. Designing a Battery Resource Agent using the High-Level API

To motivate the use of COMMELEC, and, in particular, to demonstrate the use of the High-Level API, we consider the task of controlling a battery that is part of a COMMELEC-controlled microgrid. Obviously, we want to ensure that the battery always stays within its operating constraints. In addition, we would like to achieve that the battery will be (almost) fully charged near the end of the day. During the day, the battery should provide buffer capacity to the microgrid, for example, to cope with the volatility introduced by a PV system

[1] This particular control algorithm computes orthogonal projections onto the $PQ$ profile, which are well-defined only if that set is convex.

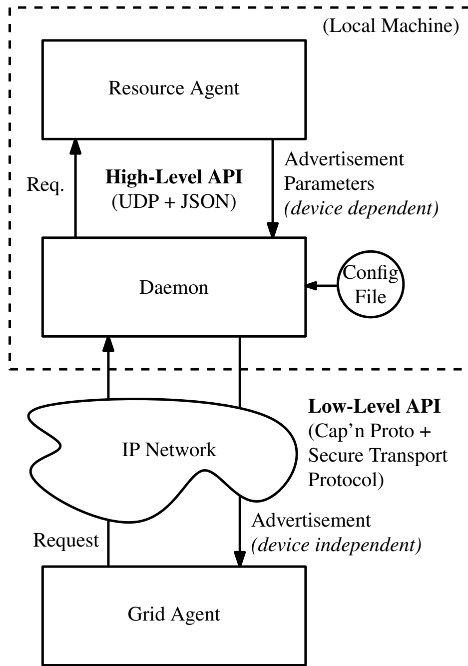


Fig. 1. The daemon acts as gateway to the grid agent and hides details of the device-independent message format as well as the transport protocol from the user.

that is also connected to the microgrid. In this section, we will see a step-by-step example that explains how we can turn this goal into a resource agent for the battery.

### *A. High-Level API*

The High-Level API is aimed at those who want to merely use COMMELEC but have no interest in knowing the details of the message format. The High-Level API currently supports receiving requests and sending advertisements for commonly used resources, including a battery, a fuel cell, and a PV.

Concretely, the High-Level API is provided by a *daemon* (a background process) that runs on the resource agent's machine as a middleman between the resource agent and the leader grid agent, see also Figure 1. The main benefits of this approach are a) that the transport protocol (between the daemon and the grid agent) is hidden from the user, and b) that the interface between the resource agent and daemon becomes very simple.

The daemon is configured by means of a small configuration file, through which the user can set static parameters like the resource type (battery, PV, etc.) and the (unique) ID number of his resource, as well as the relevant IP addresses and port numbers. Dynamic parameters (i.e., those that are repeatedly updated in real-time) are exchanged between the daemon and the resource agent over UDP, encoded as a JSON[2] object [5]. (Most programming environments have out-of-the-box support for reading and writing JSON objects.)

For requests, the daemon acts as a translator: it translates the requests as encoded in COMMELEC's Cap'n Proto-based wire format into JSON objects. For advertisements, however,

[2] *JavaScript Object Notation*.

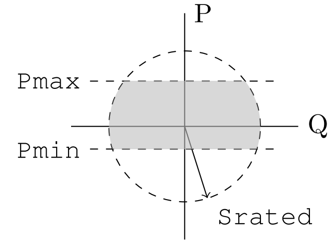


Fig. 2. The battery $PQ$ profile (capability curve).

the daemon acts as a *compiler*: the resource-dependent advertisement parameters, along with some static parameters (as set in the configuration file) are compiled into a COMMELEC advertisement (see Section II), which is *device-independent*.

### *B. Translating the Desired Behavior into an Advertisement*

The first step is to formalize the description of the desired behavior into a COMMELEC advertisement. Note that this desired behavior is two-fold: we distinguish between *short-term (real-time) constraints*, related to the maximum operating conditions of the battery, and *long-term objectives*, i.e., achieving full state-of-charge at the end of each day.

*1) Short-Term Constraints:* The real-time constraints of the battery are depicted in Figure 2, as proposed in [2]. They consist of an upper and lower bound on the active power and a "disk" constraint of a power converter. Observe that the power converter constraint is static, while the active power bounds change dynamically according to the battery state; see [2] for details. These constraints are to be encoded in the advertised $PQ$-profile.

*2) Long-Term Objectives:* For illustration purposes, we present a simplistic long-term objective that aims at bringing the battery to maximum state of charge (SoC) at the end of the day. This objective can be encoded in the advertised *cost function*. In particular, the cost function shape depends on the current SoC of the battery as well as on the time of the day. We adopt a simplified version of the cost function from [2] using a *quadratic* function in $P$:

$$(P, Q) \mapsto \alpha P^2 + \beta P, \quad \alpha, \beta \in \mathbb{R}, \tag{1}$$

In particular, the coefficients are set to

$$\alpha = |\text{SoC}_{\text{target}} - \text{SoC}| / S_{\text{rated}}^2,$$

$$\beta = 2(\text{SoC}_{\text{target}} - \text{SoC}) / S_{\text{rated}},$$

where $\text{SoC}_{\text{target}}$ is the *target* SoC (see below), SoC is the current SoC, and $S_{\text{rated}}$ is the rated power of the battery. The target SoC, $\text{SoC}_{\text{target}}$, is set according to the current time of the day. Specifically, in the afternoon, it set to $\text{SoC}_{\text{target}} = 0.9$ with the objective to fill the battery, while in the evening it is set to $\text{SoC}_{\text{target}} = 0.1$ with the objective to use the battery. Figure 3 shows different possible shapes of this cost function.

We have not yet discussed the choice of the belief function; however, like in [2], we will assume the battery to have an ideal belief function (the identity belief function): $(P, Q) \mapsto \{(P, Q)\}$.

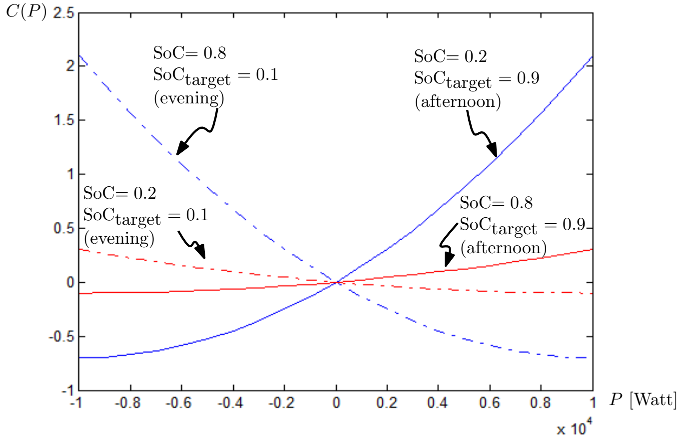


Fig. 3. Illustration of the battery agent cost function that encodes long-term objectives.

### C. Configuring and Running the Daemon

Next, we will setup the daemon. The daemon is a command-line program, invoked as

```
$ commelecd configfile.json
```

where `configfile.json` is a file with a single JSON object that contains several configuration parameters:

```
{"resource-type":"battery",
 "agent-id":1000,
 "remote-RA-ip-address":"127.0.0.1",
 "remote-RA-port":12345,
 "local-daemon-port":12346}
```

As can be deduced from the file above, we can set the resource type, the (resource) agent ID, the IP address (either and IPv4 or an IPv6 address) and port number of the resource agent, and on which ports the daemon is listening to packets from the resource agent. Instead of specifying the IP address of the resource agent, one can specify its hostname using the `remote-RA-hostname` field, for example: `"remote-RA-hostname":"batt-agent1.epfl.ch"`.

### D. Receiving Requests

Once the daemon is running, it listens for incoming requests from the grid agent, which it will translate into JSON objects and transmit to `remote-RA-ip-address` : `remote-RA-port` over UDP. A request in JSON format as transmitted by the daemon has the following fields:

| Name | Type |
|---|---|
| `senderId` | integer (32-bit unsigned) |
| `P` | double |
| `Q` | double |
| `setpointValid` | boolean |

The `senderId` contains the (unique) Agent Id number of the leader that sent the request. If `setpointValid` is true, the resource agent is supposed to implement the setpoint (`P,Q`) (specified in Watts and VAR, respectively) and reply by sending an advertisement. Otherwise (if `setpointValid` is false), the resource agent should ignore the setpoint and reply by sending an advertisement.

For example, a request in JSON format might look as follows:

```
{"senderId": 500, "P": 10.0, "Q": 20.0,
"setpointValid": true}
```

### E. Sending Battery Advertisements

Our High-Level API can be used to encode and send the $PQ$ profile and the cost function illustrated in Figures 2 and 3. Note that we have already configured the daemon to send battery advertisements to the grid agent by means of appropriately setting the `resource-type` field. Now, to trigger the transmission of a battery advertisement to the grid agent, we need to send the necessary parameters in JSON format over UDP to the daemon (to port `local-daemon-port`). These necessary parameters are listed below.

| Name | Type | Name | Type |
|---|---|---|---|
| `Pmin` | double | `coeffPsquared` | double |
| `Pmax` | double | `Pimp` | double |
| `Srated` | double | `Qimp` | double |
| `coeffP` | double | | |

First, note that $\alpha$ and $\beta$ of (1) correspond respectively to `coeffPsquared` and `coeffP`. It should be no surprise that `Pmin`, `Pmax` and `Srated` correspond to the same names in Figure 2. The parameters `Pimp` and `Qimp` should be set to the real and reactive power that the resource is currently producing (or consuming, if $P$ is negative). As noted before, the belief function of the battery agent is assumed to be *ideal*. Hence, there are no parameters in the High-Level API for the battery agent that are related to the belief function.

### F. Example: Partial Implementation in LabVIEW

Motivated by the popularity of graphical programming environments like Simulink and LabVIEW in the power-systems-research community, we outline in this section how the previously discussed steps could be implemented in LabVIEW. In Figure 4 we show a concrete example of a LabVIEW circuit that uses our High-Level API. Below, we describe different aspects of this example.

*1) Daemon: Configuration via LabVIEW GUI and Automatic Start/Stop:* On the left side of Figure 4, we demonstrate how the daemon can be configured via LabVIEW's graphical user interface, and how it can be started automatically (from LabVIEW). By making use of the *Flat Sequence Structure*, we ensure that the daemon is started before the Main Loop is executed.

The "Daemon Configuration" block, visible on the left side of Figure 4, is a *cluster* (a LabVIEW data structure) whose fields are shown on the right. Note that the names of the cluster elements must be identical to the names of the daemon configuration parameters discussed in Section III-C. The data

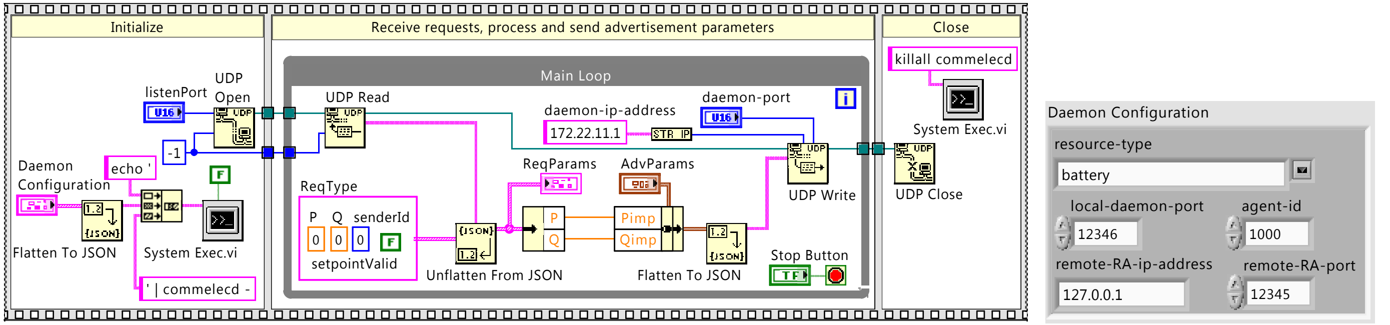


Fig. 4. Example of using the JSON-High-Level API from LabVIEW. In the *Initialize*-phase (left), the UDP socket is opened and the daemon program is started automatically (the daemon's configuration parameters can be altered via a graphical user interface). In the *Main Loop* (middle), the circuit receives requests, and simulates perfect implementation of the setpoint $(P, Q)$ by sending the same values back as the implemented setpoint $(P_{\text{imp}}, Q_{\text{imp}})$. (We omit setting the remaining advertisement parameters in this example.) In the *Close*-phase (right), the daemon process is stopped and the UDP socket is closed.

in the cluster is translated into JSON format via the built-in *Flatten to JSON* function, and this JSON data is provided to the daemon's standard input via a "pipeline": we set the "command line" parameter of the System Exec function to `echo '<JSON data>' | commelecd -` (using string concatenation). Note the single quotes around the JSON data, and the dash behind `commelecd` that lets the daemon obtain its configuration data from the standard input, instead of from a file. The Boolean constant (visible on the left side) is set to False to disable the "wait until completion" option from the System Exec function.

Note that we assume in the example that the `commelecd` executable is installed in a location that is included in the PATH environment variable of the target system. If this is not the case, then the full path should be specified.

When stopping the resource agent (using the Stop Button), the daemon process is automatically stopped by executing `killall commelecd` (here, we assume that the resource agent runs on Linux).

*2) Receiving and Parsing Requests and Sending Advertisement Parameters:* To parse incoming requests as sent by the daemon, data is read from a UDP socket using the LabVIEW's built-in *UDP Read* function, and converted into a *cluster* (a LabVIEW data structure) using the built-in *Unflatten from JSON* function (see Figure 4). The cluster "ReqType" specifies the fields and their types that should be extracted from the request; the field names (and their types) are identical to those listed in Section III-D. The extracted parameters are available to the user at the "value" output terminal of the *Unflatten from JSON* function; in Figure 4 we have connected the "ReqParams" indicator to this output.

To send the parameters for an advertisement to the daemon, all required parameters (for constructing an advertisement for a particular resource) should be bundled in a cluster, shown here as "AdvParams", whose structure is shown in full detail in Figure 5. Note that the names of the cluster elements should be identical to the names of the parameters. The data in the cluster is translated into JSON format via the built-in *Flatten to JSON* function, and sent over UDP via the built-in *UDP Write* function. Note that the *Main Loop* is an infinite loop, which can be terminated via the Stop Button. In this example, we only set a subset of the parameters ($P_{\text{imp}}$ and $Q_{\text{imp}}$); in reality one has to set all advertisement parameters.

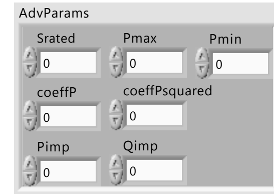


Fig. 5. Structure of the "AdvParams" cluster appearing in Figure 4.

## IV. Sending an Advertisement for a Photovoltaic System via the High-Level API

In the previous tutorial section we have already seen how the High-Level API can be used to receive a request, and to send an advertisement for a battery. In this section, we would like to highlight the High-Level API's capability to send an advertisement for a photovoltaic system.

The High-Level API allows to create an advertisement for a PV system with the following parts, as suggested in [2]:

- the $PQ$ profile as shown in Figure 6 (the grey area);
- the belief function

$$(P, Q) \mapsto \text{Rectangle}((P, Q), (p(P), q(P, Q))), \qquad (2)$$

with:

$$p(P) := \max\{0, P - \Delta\}$$
$$q(P, Q) := \text{sign}(Q) \cdot \min\{|Q|, p(P) \tan \varphi\}$$

where $\text{Rectangle}((x_1, y_1), (x_2, y_2))$ represents a rectangle in $\mathbb{R}^2$ with corner points $(x_1, y_1)$ and $(x_2, y_2)$, and $\Delta \in \mathbb{R}$ (in Watts) represents a possible drop in real power, which could be caused, for example, by a cloud.

- the cost function:

$$(P, Q) \mapsto -aP + bQ^2, \quad a, b \in \mathbb{R}.$$

At this point, we have not yet made clear why the particular belief function specified above should be useful or "the right one" in some sense. Further below, we will argue that this

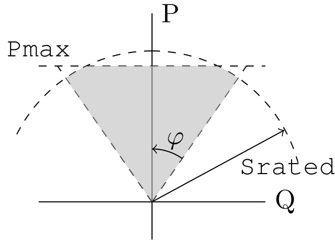


Fig. 6. The $PQ$ profile (capability curve) of the photovoltaic system that is constructed by the `makePVAdvertisement` function.

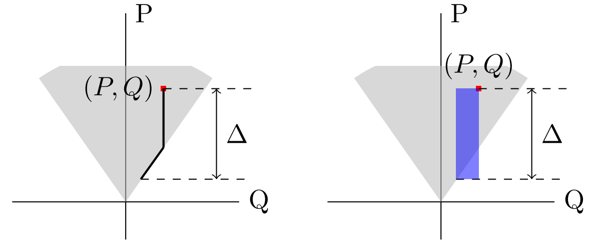


Fig. 7. General belief function (left) and its rectangular approximation (right), both evaluated at $(P, Q)$.

belief function is actually a rectangular approximation (or, in other words, a simplified version) of a more general belief function, and that by using this approximation we will not incur any loss, given the current implementation of the grid agent.

To instruct the daemon to send PV advertisements, we need to set the `resource-type` configuration parameter to `pv`. The following table lists the dynamic parameters for the PV advertisement.

| Name | Type | Name | Type |
|---|---|---|---|
| `Pmax` | double | `a_pv` | double |
| `Srated` | double | `b_pv` | double |
| `cosPhi` | double | `Pimp` | double |
| `Pdelta` | double | `Qimp` | double |

Note that we encountered most of the parameters before when discussing the battery advertisement. Those parameters have exactly the same meaning here. The parameter `Pdelta` corresponds to $\Delta$ in the definition of the belief function, and furthermore `a_pv` and `b_pv` correspond to $a$ and $b$ respectively. The parameter `cosPhi` corresponds to $\cos\varphi$ (the "power factor"), where $\varphi$ is the phase angle shown in Figure 6.

### *General Form of the Photovoltaic Belief Function and its Rectangular Approximation*

In [2], the belief function of a PV system is defined as follows:

$$f(p,q) := \bigcup_{\rho\in[\max\{0,p-\Delta\},p]} \big\{\big(\rho, \operatorname{sign}(q)\cdot\min\{|q|,\rho\tan\varphi\}\big)\big\}.$$

The definition of the belief function involves a parameterized union, which we currently do not support in the message format. Hence, we define a belief function that, for every input $u \in \mathcal{A}$, returns a superset of $f(u)$ that is representable in our message format. Note that this superset should be kept as small as possible, otherwise the belief function will lose its purpose and the grid agent will treat our resource too conservatively. For simplicity we take the axis-aligned rectangular hull around $f(u)$. As a matter of fact, this approximation does not influence the grid-control performance when using the current implementation of the grid agent, because this implementation also uses $\operatorname{rect\,hull}(\mathrm{BF}_i(u))$ internally as a proxy for $\mathrm{BF}_i(u)$, for every $u \in \mathcal{A}$, and for every $i \in \mathcal{I}$, where $\mathcal{I}$ denotes the set of all followers associated to that grid agent and $\mathrm{BF}_i$ represents the belief function of follower $i$.

In Figure 7 (left) we illustrate the shape of an evaluation of the belief function at a particular point in the $PQ$ plane, i.e., the set $BF((P,Q))$. Figure 7 (right) shows the rectangular approximation, as defined in Eqn. (2).

## V. Design and Structure of The Message Format and the Low-Level API

In this section, we describe the structure of our message format and the Low-Level API. The Low-Level API, which is targeted at expert users (i.e., it requires the user to understand the structure of the message format), exposes all features of our message format. For example, it can be used to define an advertisement for a device that is not supported in the High-Level API. In case the resource agent uses the Low-Level API, the `resource-type` field in the daemon configuration file should be set to `custom`; the daemon will then operate in a different mode in which it merely takes care of running the transport protocol.

We will start by listing the requirements that gave rise to our message-format design. Then, we explain that our message format is based on an existing serialization framework, and motivate the choice for a particular framework. We then proceed by giving the actual definition of the message format and the Low-Level API, and demonstrate the use of the latter by giving some concrete examples.

### *A. Requirements*

First and foremost, an obvious formal requirement is that the message format should be capable of representing the mathematical objects that appear in advertisements and requests. Concretely, this means that the format should support the description of $PQ$-setpoints, $PQ$ profiles, cost functions and belief functions.

Furthermore, we formulated a number of qualitative requirements.

- It should be possible to extend the message format, without breaking compatibility with resource agents that "speak" an older version of the format.
- One of the design goals of COMMELEC is to create a platform that makes it easy for third parties to implement their own resource agent. Hence, it should be straightforward to construct an advertisement (and to parse a

request) from multiple programming environments, on mainstream hardware architectures.
- Resource agents as well as grid agents will typically run on resource-constrained embedded platforms. Hence, the time and space complexity (i.e., CPU clock cycles and memory usage) of creating and interpreting messages should be low. This is especially important for the grid agent, which needs to process advertisements from several resource agents.
- The byte size of the encoded messages should be small, while respecting the low-processing-complexity constraint. Preferably, requests as well as most advertisements should fit in a single Ethernet packet ($< 1500$ bytes).
- The message format should reflect the capabilities of the interpreter (the grid agent), which enforces the resource-agent designer to express the advertisement in an "interpretable" way.

### B. Selection of a Serialization Framework

We have designed our message format on top of an existing *serialization framework*. A serialization framework provides a programming-language-independent and platform-independent way to convert structured data to bytes and *vice versa* and typically consists of three parts: 1) a *schema language*, being a strongly-typed language in which one can define the data structures, 2) a *schema compiler* which can generate source code (typically, for several mainstream programming languages) for easily reading and writing those data structures, and 3) an *encoding specification* that defines how the data structures will be encoded on the bit- or byte-level.

Many serialization frameworks exist, ranging from established standards like *ASN.1* [6] to newer frameworks like *Protocol Buffers*, *Thrift*, *Avro*, or yet newer frameworks featuring "zero-copy" like *Cap'n Proto* [7], *Simple Binary Encoding* and *FlatBuffers*. Here, "zero-copy" roughly means that the data format "on the wire" is the same as (or closely related to) the in-memory data format.

Out of these alternatives, we have selected Cap'n Proto, which satisfies our needs and provides several of our requirements as a feature: most notably, schema evolvability, a compact encoding through a very simple yet effective compression method called *packing*, and, because of its zero-copy design, a low encoding and decoding complexity (compared to non-zero-copy frameworks). Also, we have chosen to use this framework because its C++11 reference implementation and documentation are of high quality, the schema language is intuitive and convenient, and because the framework has a permissive software license (MIT). The latter implies in particular that Cap'n Proto can be used free of charge in commercial products. For an in-depth explanation of Cap'n Proto's schema language and encoding specification, we refer to Cap'n Proto's online documentation [7].

*IEC 61850 Interoperability:* IEC 61850 is a communication standard for substation automation [4] (or, as from Edition 2, for power utility automation). Although IEC61850 uses ASN.1 (with the *Basic Encoding Rules*, i.e., BER) to serialize data, we can easily achieve interoperability with IEC61850 by embedding Cap'n Proto-serialized COMMELEC advertisements and requests into ASN.1 data structures via ASN.1's OCTET STRING field type.

### C. Message Format Definition and Low-Level API

Our proposed message format is formally defined by the schema shown in Appendix A, which is expressed in Cap'n Proto's schema language [7]. It defines the data structures for representing (numerical approximations of) requests and advertisements, where the latter is defined mathematically in Section II. Some of these data structures can be linked to other data structures, by which one can build tree-like data structures.

The Low-Level API consists of the automatically generated API for C++11 (generated by Cap'n Proto's schema compiler) based on our schema (Appendix A), augmented with some (manually written) convenience functions, which should make it particularly easy to write a resource agent or to extend the High-Level API in C++11. Note, however, that Cap'n Proto is supported in several other programming languages as well.

Further details about the message format and the Low-Level API can be found in Appendix B.

## VI. Performance Evaluation

### A. Comparison of Format: Cap'n Proto/RealExpr vs. Content MathML

We start by comparing the byte size of encoding a cost function in our format, i.e., as a `RealExpr`, versus the same expression in *Content MathML*. Content MathML is a W3C standard for encoding the semantics of mathematical objects, and is supported by some computer algebra systems like Mathematica and Maple. Because our aim is merely to get a rough indication of the byte-size difference between these two approaches, we restrict here to real-valued expressions. To encode an entire advertisement in Content MathML, one probably has to use the Strict Content MathML profile (or, alternatively, OpenMath, which is compatible to the latter), and one would have to carefully design a custom *Content Dictionary* first, which is beyond the scope of this paper.

The fragment below was created with the help of Mathematica.

```
<math
 xmlns='http://www.w3.org/1998/Math/MathML'>
 <apply>
  <plus />
  <apply>
   <times />
   <cn type='real'>-10</cn>
   <ci>P</ci>
  </apply>
  <apply>
   <power />
   <ci>Q</ci>
   <cn type='integer'>2</cn>
  </apply>
 </apply>
</math>
```

TABLE I
COMPARING REALEXPR TO CONTENT MATHML: BYTE SIZES OF SERIALIZED COST FUNCTION $(-10p + q^2)$

| Method | Byte size |
|---|---|
| **RealExpr / Cap'n Proto (with packing)** | **82** |
| RealExpr / Cap'n Proto (without packing) | 280 |
| Content MathML (plain text XML) | 196 |
| Content MathML (gzip'ped XML) | 164 |

Here, we see quantitatively that Cap'n Proto's packing feature significantly influences the byte size of a message: without packing, the representation of a simple real-valued function in our format is larger than a representation of the same function in Content MathML (which is an XML-based format). With packing enabled, our message format yields the smallest encoding.

TABLE II
BYTE SIZE OF A REQUEST AND OF TYPICAL ADVERTISEMENTS

| Method | Request | Batt. Adv. | PV Adv. |
|---|---|---|---|
| **Cap'n Proto (with packing)** | **18** | **182** | **425** |
| Cap'n Proto (without packing) | 48 | 760 | 1544 |
| Cap'n Proto (JSON) | 48 | 514 | 1153 |

We conclude from these numbers that for our scenario, Cap'n Proto's packing algorithm achieves a compression ratio of around 3. Further, we see that by using the packed encoding, a typical advertisement fits easily in a single Ethernet packet.

The results of the size comparison are shown in Table I. For the Cap'n Proto-related entries, we compare the packed versus the unpacked encoding. Note that the exact byte size of the packed encoding depends on the coefficients of the cost function (for the unpacked encoding this is not the case). To obtain the Content-MathML-related entries, we have counted the byte size of the above fragment (where we omitted whitespace and newline characters around the XML tags). We have also applied standard gzip compression to this fragment, however, due to the small size of the fragment the compression factor achieved here (which is rather low) will not be representative for larger fragments.

### *B. Comparison of Encoding and Software Implementations: Cap'n Proto with and without packing vs. (Rapid-) JSON*

In Table II we compare the byte sizes of requests and advertisements when represented in our message format, but encoded in various ways. The JSON encodings were obtained by converting messages encoded in Cap'n Proto's encoding to JSON via Paryani's Python wrapper for Cap'n Proto [8]. Table III shows the decoding performance of Cap'n Proto's C++ reference implementation, versus an open-source C++ JSON library, *RapidJSON* [9], for our specific examples on an ARM-based platform.

TABLE III
UNPACKING/PARSING AND DATA-ACCESS SPEED COMPARISON ON AN ARM CORTEX-A9 EMBEDDED PLATFORM

| Method | Action | Batt. Adv. | PV Adv. |
|---|---|---|---|
| Cap'n Proto (without packing) | access data | 24.9 µs | 40.7 µs |
| **Cap'n Proto (with packing)** | unpack + access data | **58.1 µs** | **105 µs** |
| RapidJSON | parse JSON + access data | 83.0 µs | 164 µs |

In this experiment, we measure the time spent on accessing fields (of numerical data types) from advertisements, including parsing or unpacking times (if applicable), on an embedded platform (NI cRIO-9068) with a 667 MHz ARM Cortex-A9 CPU. To prevent the compiler from optimizing away critical parts of the simulation code, we sum all extracted values and output the result. The reported figures include the time spent on summing the (double-valued) numbers. By combining these results with Table II, we see a) that the processing speed scales roughly linearly in the message's byte size, as one would expect, and b) that Cap'n Proto (with packing) is superior to JSON/RapidJSON for our use case in terms of message size and processing speed.

## VII. CONCLUSION

In COMMELEC, the main task of a resource agent is to translate the resource-specific state into a device-independent description for the grid agent. In this work, we have proposed a practical and low-footprint solution for accomplishing this task. We hope that our API eases the adoption of our solution; in particular, we have added a separation between the easy-to-use High-Level API and the more general Low-Level API, to make the right trade-off between easily sending and receiving COMMELEC messages for common resources, while preserving the possibility of supporting non-standard resources. We have shown an example of how the High-Level API could be used in a simple way from LabVIEW to control a battery.

As our main technical contribution, we have proposed a concrete representation for COMMELEC's message format. Our performance benchmarks show that it has attractive quantitative characteristics in terms of message size and processing speed.


## ACKNOWLEDGMENT

We wish to thank Mario Paolone and Lorenzo Reyes, co-authors of the COMMELEC framework, for many useful discussions and for advice on the LabVIEW implementation.


## REFERENCES


[1] A. Bernstein, L. Reyes-Chamorro, J.-Y. L. Boudec, and M. Paolone, "A composable method for real-time control of active distribution networks with explicit power setpoints. part i: Framework," *Electr. Pow. Syst. Res.*, vol. 125, pp. 254–264, 2015.

[2] ——, "A composable method for real-time control of active distribution networks with explicit power setpoints. part ii: Implementation and validation," *Electr. Pow. Syst. Res.*, vol. 125, pp. 265–280, 2015.

[3] N. J. Bouman, "Commelec API description and link to source code," 2015. [Online]. Available: http://smartgrid.epfl.ch

[4] "IEC 61850 communication networks and systems in substations part 8-1: Specific communication service mapping (scsm) mappings to mms (iso 9506-1 and iso 9506-2) and to iso/iec 8802-3," IEC, 2004.

[5] "The JSON Data Interchange Format," Oct. 2013, ECMA-404.

[6] "Abstract Syntax Notation ONE." [Online]. Available: http://www.itu.int/en/ITU-T/asn1

[7] K. Varda, "Cap'n Proto — cerealization protocol," Sandstorm.io. [Online]. Available: http://capnproto.org

[8] J. Paryani, "pycapnp — Python wrapper library for Cap'n Proto." [Online]. Available: http://jparyani.github.io/pycapnp

[9] M. Yip, "RapidJSON — a fast JSON parser/generator for C++ with both SAX/DOM style API," THL A29. [Online]. Available: https://github.com/miloyip/rapidjson

[10] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. New York: Springer, 2006.

[11] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," 2010. [Online]. Available: http://eigen.tuxfamily.org

## APPENDIX A
## SCHEMA DEFINITION

```
using Cxx = import "/capnp/c++.capnp";
$Cxx.namespace("msg");

struct Message {
  agentId         @0 :UInt32;
  union {
    request       @1 :Request;
    advertisement @2 :Advertisement;
  }
}

struct Request {
  setpoint        @0 :List(Float64);
}

struct Advertisement {
  pQProfile           @0 :SetExpr;
  beliefFunction      @1 :SetExpr;
  costFunction        @2 :RealExpr;
  implementedSetpoint @3 :List(Float64);
}

struct RealExpr {
 name             @0 :Text;
 union {
  real            @1 :Float64;
  polynomial      @2 :Polynomial;
  unaryOperation  @3 :UnaryOperation;
  binaryOperation @4 :BinaryOperation;
  listOperation   @5 :ListOperation;
  caseDistinction @6 :CaseDistinction(RealExpr);
  reference       @7 :Text;
  variable        @8 :Text;
 }
}

struct UnaryOperation {
  arg       @0 :RealExpr;
  operation :union {
    negate  @1  :Void;
    abs     @2  :Void;
    sign    @3  :Void;
    multInv @4  :Void;
    square  @5  :Void;
    sqrt    @6  :Void;
    sin     @7  :Void;
    cos     @8  :Void;
    tan     @9  :Void;
    exp     @10 :Void;
    ln      @11 :Void;
    log10   @12 :Void;
    round   @13 :Void;
    floor   @14 :Void;
    ceil    @15 :Void;
  }
}

struct BinaryOperation {
  argA      @0 :RealExpr;
  argB      @1 :RealExpr;
  operation :union {
    sum         @2 :Void;
    prod        @3 :Void;
    pow         @4 :Void;
    min         @5 :Void;
    max         @6 :Void;
    lessEqThan  @7 :Void;
    # an indicator function
    greaterThan @8 :Void;
    # also an indicator function
  }
}

struct ListOperation {
  args      @0 :List(RealExpr);
  operation :union {
    sum     @1 :Void;
    prod    @2 :Void;
  }
}

struct Polynomial {
  variables    @0 :List(Text);
  maxVarDegree @1 :UInt8;
  coefficients @2 :List(SparseCoeff);
}

struct SparseCoeff {
  offset @0 :UInt32;
  value  @1 :Float64;
}

struct CaseDistinction(CaseType) {
  variables @0 :List(Text);
  cases     @1 :List(ExprCase(CaseType));
}

struct ExprCase(CaseType) {
  # Representation of a single case
  # for use in CaseDistinction
  # (the order of evaluation follows
  # List ordering)
  set        @0 :SetExpr;
  expression @1 :CaseType;
}

struct SetExpr {
 name              @0 :Text;
 union {
  singleton       @1 :List(RealExpr);
  ball            @2 :Ball;
  rectangle       @3 :List(BoundaryPair);
  convexPolytope  @4 :ConvexPolytope;
  intersection    @5 :List(SetExpr);
  caseDistinction @6 :CaseDistinction(SetExpr);
  reference       @7 :Text;
 }
}

struct Ball {
  center @0 :List(RealExpr);
  radius @1 :RealExpr;
}

struct BoundaryPair {
  boundA @0 :RealExpr;
  boundB @1 :RealExpr;
}

struct ConvexPolytope {
  a @0 :List(List(RealExpr));
  b @1 :List(RealExpr);
}
```

## APPENDIX B
## MESSAGE FORMAT AND LOW-LEVEL API – CONTINUED

### A. The Top-Level Message Type

Every COMMELEC message has a common base type, `Message`. In Cap'n Proto terminology, the `Message` struct is the root struct of every COMMELEC message. As we can see in Appendix A, a message contains, beyond an ID number that uniquely identifies an agent, either an `Advertisement` or a `Request`.

### B. Requests

In the current version of our schema, a `Request` may contain a setpoint, encoded as a list of double-precision floating point values. Currently, we only use lists of size 2, where the first entry corresponds to real power ($P$) and the second entry corresponds to reactive power ($Q$).

Whether a request actually contains a setpoint depends on the state of our leader (a grid agent). If a grid agent wants to receive an advertisement from a follower, to which it can not yet provide a meaningful setpoint for implementation, the grid agent will send a request without a setpoint. For example, when a grid agent has just rebooted, it should first gather advertisements from its followers, before it can compute new setpoints that the followers should implement.

We can easily check whether a request includes a setpoint, and if it does, extract the values for $P$ and $Q$:

```
// req is of type msg::Request::Reader
enum{P = 0, Q = 1};
if (req.hasSetpoint()){
  // request has a setpoint, let's extract it
  auto sp = req.getSetpoint();
  if(sp.size() < 2) throw; // error
  cout << "Setpoint: " << sp[P] << ", "
                       << sp[Q] << endl;
}else{
  // request does not include a setpoint
}
```

The comment on the first line indicates that `req` is a *Reader*, which is a Cap'n Proto concept (hence, see [7] for details). A Reader is a handle that provides read-only access to its associated object, which is a `Request` object in this example. In other upcoming code examples, we will also encounter *Builder* objects, i.e., objects of a type with the suffix `::Builder`. Builders are handles that provide read and write access to their associated object.

The availability of the predicate `hasSetpoint` follows directly from Cap'n Proto's C++ API: the field `setpoint` in the `Request` struct (Appendix A) is encoded as a `List`, which is a *pointer field* (see [7]). Every such pointer field has a "`has`[Fieldname]" predicate.

### C. Advertisements

Let us start by looking at the definition of the `Advertisement` struct in the schema, and printed below for convenience.

```
struct Advertisement {
  pQProfile            @0 :SetExpr;
  beliefFunction       @1 :SetExpr;
  costFunction         @2 :RealExpr;
  implementedSetpoint  @3 :List(Float64);
}
```

In case you are not familiar with Cap'n Proto's schema language, note that for our discussion here we can safely ignore the meaning of the `@`-prefixed numbers. Suffice it to say, each line that is indented in the above listing defines a field in the `Advertisement` data structure, where the leftmost word defines the name of that field, and the rightmost word (right from the colon) defines its type.

First, observe the correspondence to the mathematical definition of an Advertisement that we gave in Section II: we immediately recognize its quaternary structure. Second, note that both the $PQ$ profile and the belief function are of the same type, namely `SetExpr`. Recall that the $PQ$ profile is an explicit set, while the belief function is a set-valued function, from which we can obtain an explicit set by *evaluating* this function at a specific power setpoint. As we will see below, the definition of `SetExpr` is general enough to support both explicit and parameterized sets (where we view the latter as set-valued functions).

Third, the cost function is of type `RealExpr`. The `RealExpr` struct represents a real-valued expression, or, formally, the map $f : \mathbb{R}^n \to \mathbb{R}$, where $n \in \mathbb{N}$ typically depends on the context where the `RealExpr` is used. In the context of representing a cost function, $n = 2$. We will describe the `RealExpr` and `SetExpr` structs in more detail further below.

Fourth, note that the `implementedSetpoint` is defined in exactly the same way as `setpoint` in a `Request`, and also here we currently only use lists of length 2 where the first and second entry correspond to $P$ and $Q$ respectively.

### D. Defining Real-Valued Expressions

The `RealExpr` struct represents a real-valued expression $\mathbb{R}^n \to \mathbb{R}$, where $n \in \mathbb{N}$ is a parameter. From the schema (Appendix A), we see that a `RealExpr` contains a **`union`**, which has the property that only one of the fields defined inside that union can be active at a time. Hence, a `RealExpr` has a "subtype" that is determined by the active field in the union. At the time of this writing, a `RealExpr` can represent the following expression types (with in parentheses the associated fieldname(s) in the `RealExpr` struct):

- a constant (sometimes called *immediate value*), encoded as a IEEE 754 double-precision floating-point number (`real`) ;
- an operation on one or more `RealExpr` objects (`unaryOperation`, `binaryOperation`, `listOperation`);
- a symbolic variable (`variable`);
- a reference to another `RealExpr` that has been given an explicit name (`reference`);
- a case distinction, for selecting a particular `RealExpr` out of a list of `RealExpr` objects, depending on the value of the input (`caseDistinction`).

As some of those expressions are *self-referential* (they refer to yet another `RealExpr`), we can define real-valued ex-

pressions by means of building syntax trees with `RealExpr` objects as building blocks.

*1) Example: Encoding a Cost Function:* As a first example, let us demonstrate how to define the function

$$g(p, q) := -10p + q^2$$

as a cost function that is part of an advertisement. This example is in fact a specific instance of the PV-agent cost function described in Section IV; the parameters $a = 10$ and $b = 1$ have been chosen arbitrarily.

We will start by defining this function "by hand", that is, solely using Cap'n Proto's API, because defining the function via this route provides insight into the structure of our message format. Next, we will define the same function in a much simpler way, using our convenience function for defining real-valued expressions.

Using Cap'n Proto's API, we implement $g$ as follows,

```
// adv is of type msg::Advertisement::Builder
auto addOp = adv.initCostFunction()
                .initBinaryOperation();
addOp.initOperation().setSum();
auto multOp = addOp.initArgA()
                 .initBinaryOperation();
multOp.initOperation().setProd();
multOp.initArgA().setReal(-10);
multOp.initArgB().setVariable("P");
auto sqOp = addOp.initArgB()
               .initUnaryOperation();
sqOp.initOperation().setSquare();
sqOp.initArg().setVariable("Q");
```

*Remark* 1. The code above does not specify that `P` and `Q` are the input arguments to the function. Indeed, an important convention in our message format, which applies to the `beliefFunction` as well as to the `costFunction` in the `Advertisement` struct, is that the function arguments `P` and `Q` are defined *implicitly*, and can be referred to using symbolic variables, as shown above. More concretely, a grid agent that evaluates this cost function at the point $(p, q) \in \mathcal{A}$, will substitute the value $p$ for every occurrence of a "`P`" variable and similarly, the value $q$ for every "`Q`" variable, while recursively evaluating the expression tree.

Now, let us demonstrate how we define the same function using our convenience function:

```
// adv is of type msg::Advertisement::Builder
using namespace cv;
Var P("P");
Var Q("Q");
buildRealExpr(adv.initCostFunction(),
              Real(-10) * P + square(Q));
```

Under the hood, `buildRealExpr` uses *C++ Expression Templates* to parse symbolic expressions like `Real(-10)*P+square(Q)`. After the parsing step, `buildRealExpr` calls exactly the same Cap'n Proto API functions as we did in our "by-hand" approach.

Note that constant values should (currently, at least) be surrounded by the `Real(·)` function. Furthermore, the `buildRealExpr` function makes the unary operations (as found in the schema) accessible using functions, whose (lower-cased) name is identical to the corresponding field name in the `union` environment of the `UnaryOperation` struct, like the `square` function in the example above.

We will see another example of the use of `buildRealExpr` in the next subsection; for a fully detailed example of using `buildRealExpr` and the Low-Level API in general, we refer to the *implementation* of the High-Level API [3]. Based on experience gained while implementing the High-Level API, we claim that using `buildRealExpr` makes defining real-valued expressions easier and less error-prone, and results in more readable source code, as the latter more closely resembles the original mathematical expression that one wants to represent.

*2) Numerically Stable Derivatives:* In COMMELEC, grid agents currently employ a control strategy that is based on gradient descent. As a particular consequence of this, the grid agent very frequently computes the gradient of the cost function of each follower.

An important benefit of our proposed encoding of real-valued expressions, is that the syntax-tree structure naturally admits computing derivatives in a numerically stable way, by recursive application of the basic rules for differentiation: the sum rule, product rule, chain rule, etc. This technique is well known; in the literature it is commonly called *Automatic Differentiation* [10]. The implementation details of automatic differentiation in the context or our message format is beyond the scope of this paper.

### *E. Defining Sets and Set-Valued Functions*

Sets and set-valued functions are defined using the `SetExpr` struct. A `SetExpr` represents a, possibly parameterized, closed convex set of arbitrary dimension, however, we currently merely use it to encode subsets of $\mathbb{R}$ (intervals) and subsets of $\mathbb{R}^2$, like disks and 2-polytopes, as well as parameterized versions thereof. Similar to a `RealExpr`, the `SetExpr` definition (Appendix A) contains a **`union`**, whose active field determines whether the `SetExpr` represents a singleton set, a ball, an intersection between other `SetExpr`essions, etc. In the sequel, we will give some examples of how we can use `SetExpr` to define $PQ$ profiles and belief functions.

*Remark* 2. When defining objects using `SetExpr`, one must ensure that every $PQ$ profile, or, every evaluation of a belief function, is *bounded*. A `SetExpr` by itself does not necessarily represent a bounded set. A concrete example is the $PQ$ profile of the battery advertisement that we encountered in Section III-B1 (see also Figure 2). There, we use a polytope to represent the lower and upper bound on the real power (`Pmin` and `Pmax`) that the battery can produce. This polytope itself is unbounded in the dimension that corresponds to reactive power (the $Q$ coordinate). Only because the $PQ$ profile is defined as the intersection between this polytope and a disk (which is bounded in both dimensions), the boundedness constraint is satisfied.

*1) Example: Defining the $PQ$ profile of the PV System:* Let us demonstrate how we can specify the $PQ$ profile of the PV system, as shown in Figure 6, using the Low-Level API. This $PQ$ profile is an *intersection* between a disk and a polytope

(in $\mathbb{R}^2$).[3]

The disk is centered around the origin in the $PQ$ plane, and its radius is equal to `Srated`. As we will see below, defining a disk in our format is straightforward using the `ball` field.

The polytope is a isosceles triangle characterized by $\varphi$ and `Pmax`, as illustrated in Figure 6. To define this polytope in our format, we will use our convenience function for defining polytopes. We assume that we know the halfspace-representation of the polytope, i.e., the matrix $\mathbf{A} \in \mathbb{R}^{m\times 2}$ and the vector $\mathbf{b} \in \mathbb{R}^m$, where $m$ denotes the number of constraints, that define the polytope via

$$\mathbf{A}\mathbf{x} \leq \mathbf{b}, \quad \mathbf{x} \in \mathbb{R}^2.$$

In our example, $m = 3$. Note that we use matrix and vector classes from the *Eigen3* C++ library for linear algebra [11].

```
#include <Eigen/Core>
#include <commelec-api/polytope-convenience.hpp>
#include <commelec-api/schema.capnp.h>
// adv is of type msg::Advertisement::Builder
// Srated, Pmax, tanPhi are of type double
auto m = 3; // number of constraints
auto dimension = 2;
auto intsect = adv.initPQProfile()
                  .initIntersection(2);
auto disk = intsect[0].initBall();
auto diskCenter = disk.initCenter(dimension);
enum {P = 0, Q = 1};
diskCenter[P].setReal(0);
diskCenter[Q].setReal(0);
disk.initRadius().setReal(Srated);

Eigen::MatrixXd A(m, dimension);
A <<      1, 0,
    -tanPhi, 1;
    -tanPhi,-1;
Eigen::VectorXd b(m);
b << Pmax,
     0,
     0;
cv::buildConvexPolytope(A, b,
    intsect[1].initConvexPolytope());
```

*2) Example: Defining a Simple Belief Function:* As mentioned earlier, the `SetExpr` struct can also represent set-valued functions, in particular, a belief function. The simplest belief function is the identity belief function, which can be defined using a singleton set, which itself is defined as a `List` of `RealExpr` objects.

```
// 'adv' is of type msg::Advertisement::Builder
auto bf = adv.initBeliefFunction();
auto dimension = 2;
auto idBf = bf.initSingleton(dimension);
enum {P = 0, Q = 1};
idBf[P].setVariable("P");
idBf[Q].setVariable("Q");
```

Note that Remark 1 applies here.

*3) Example: Defining the Belief Function of the PV System:* The example below is taken from the implementation of the High-Level API, in which we define the belief function of the PV advertisement. As defined formally in Eqn. (2) (see page 5), the belief function is a rectangle in $\mathbb{R}^2$, whose boundaries in both dimensions are encoded as real-valued expressions that depend on $P$ and $Q$.

```
// adv is of type msg::Advertisement::Builder
using namespace cv;
auto bf = adv.initBeliefFunction();
auto rect = bf.initRectangle(2);

Ref foo("a");
Var P("P");
Var Q("Q");

buildRealExpr(rect[0].initBoundA(), P);
buildRealExpr(rect[0].initBoundB(),
 name(foo, max(Real(0), P + Real(-Pdelta))));
buildRealExpr(rect[1].initBoundA(), Q);
buildRealExpr(rect[1].initBoundB(),
 sign(Q) * min(abs(Q), foo * Real(tanPhi)));
```

First, note that the `buildRealExpr` function not only supports the unary operations, like `sign` and `abs`, but also binary operations like `max` and `min`. Then, note how, in the second occurrence of `buildRealExpr`, the expression is named *foo*, and how we refer back to *foo* on the last line. The name *foo* is internal to C++; the actual the name (which appears as `Text`[4] in the Advertisement) is “a” (as defined near the beginning of the fragment). It is no coincidence that we assigned a short name here: assigning short names is good practice as it will result in shorter message sizes.

[3] A well-known fact is that convexity is preserved under set intersection. In the context of defining a $PQ$ profile (recall that $PQ$ profiles are defined to be convex), this means that one can take arbitrary intersections of convex sets, while having the guarantee that the result will be convex.

[4] `Text` is Cap'n Proto's string type.