Smart TSO-DSO interaction schemes, market architectures and ICT Solutions for the integration of ancillary services from demand side management and distributed generation

SmartNet simulation platform

Authors:

Giacomo Viganò, Marco Rossi (RSE), Peter Sels, Guillaume Leclercq, Thomas Gueuning, Marco Pavesi (N-SIDE), Yelena Vardanyan, Razgar Ebrahimy (DTU), Joseba Jimeno, Nerea Ruiz (TECNALIA), Gary Howorth (USTRATH), Juliano Camargo, Chris Hermans, Fred Spiessen (VITO), Harald Svendsen (SINTEF),

Distribution Level	Public
Responsible Partner	RSE
Checked by WP leader **Marco Rossi**	Date: 21/06/2019
Approved by Project Coordinator **Gianluigi Migliavacca**	Date: 25/06/2019

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 691405.

Issue Record

Planned delivery date	M36 (31/12/2018)
Actual date of delivery	M42 (30/06/2019)
Status and version	1.0 – Final

Version	Date	Author(s)	Notes
0.0	October 2018	RSE	First template with description of physical layer
0.5	November 2018	N-SIDE, DTU, TECNALIA, USTRATH, VITO, SINTEF	Contributions received by partners
0.7	May 2019	RSE	Harmonization of the content
0.8	June 2019	N-DISE, DTU, TECNALIA, USTRATH, VITO, SINTEF	Revision of the content
0.9	June 2019	RSE	Revision by WP leader
1.0	June 2019	RSE	Final version

About SmartNet

The project SmartNet (http://smartnet-project.eu) aims at providing architectures for optimized interaction between TSOs and DSOs in managing the exchange of information for monitoring, acquiring and operating ancillary services (frequency control, frequency restoration, congestion management and voltage regulation) both at local and national level, taking into account the European context. Local needs for ancillary services in distribution systems should be able to co-exist with system needs for balancing and congestion management. Resources located in distribution systems, like demand side management and distributed generation, are supposed to participate to the provision of ancillary services both locally and for the entire power system in the context of competitive ancillary services markets.

Within SmartNet, answers are sought for to the following questions:

- Which ancillary services could be provided from distribution grid level to the whole power system?
- How should the coordination between TSOs and DSOs be organized to optimize the processes of procurement and activation of flexibility by system operators?
- How should the architectures of the real time markets (in particular the markets for frequency restoration and congestion management) be consequently revised?
- What information has to be exchanged between system operators and how should the communication (ICT) be organized to guarantee observability and control of distributed generation, flexible demand and storage systems?

The objective is to develop an ad hoc simulation platform able to model physical network, market and ICT in order to analyse three national cases (Italy, Denmark, Spain). Different TSO-DSO coordination schemes are compared with reference to three selected national cases (Italian, Danish, Spanish).

The simulation platform is then scaled up to a full replica lab, where the performance of real controller devices is tested.

In addition, three physical pilots are developed for the same national cases testing specific technological solutions regarding:

- monitoring of generators in distribution networks while enabling them to participate to frequency and voltage regulation,
- capability of flexible demand to provide ancillary services for the system (thermal inertia of indoor swimming pools, distributed storage of base stations for telecommunication).

Partners

Table of Contents

List of Abbreviations and Acronyms

Acronym	Meaning
aFRR	automatic Frequency Restoration Reserve
AL	Atomic Load
AMPL	A Mathematical Programming Language (software)
CGCL	Curtailable Generation Curtailable Load
CHP	Combined Heat and Power
CON(V)	Conventional Generator
CPLEX	A mixed-integer linear programming solver offered by IBM (software)
CS	Coordination scheme
DB	Database
DSO	Distribution System Operator
EES	Electrical Energy Storage
EV	Electric Vehicle
mFRR	manual Frequency Restoration Reserve
OLTC	On Load Tap Changer
OPF	Optimal Power Flow
PF	Power Flow
PHYLAY	Physical Layer
PV	Photovoltaic
SEL	Sheddable Load
SQL	Structured Query Language
STATCOM	Static Compensator
STO	Storage
TCL	Thermostatically Controlled Loads
TSO	Transmission System Operator

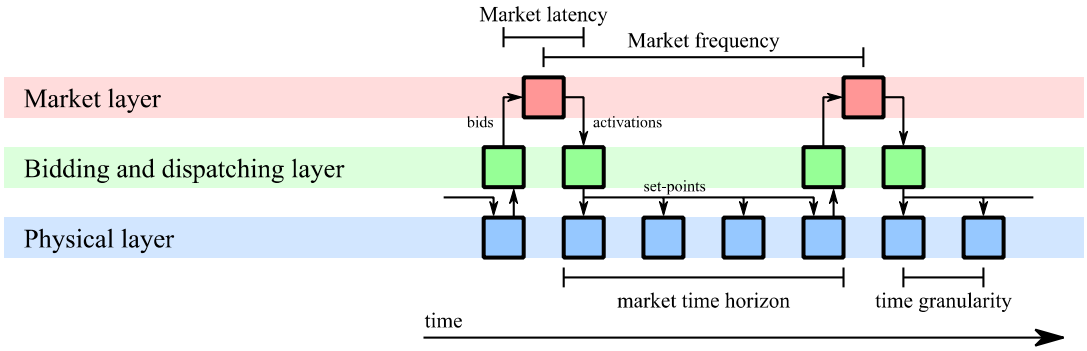
Executive Summary

The TSO-DSO coordination schemes proposed by SmartNet have been tested by means of dedicated simulations aimed at realistically reproducing the behavior of the electrical system and of the involved actors in hypothetical scenarios (expected 2030 situation of Italy, Denmark and Spain). The present report describes the software platform of the simulator which has been completely developed by the SmartNet team on the basis of the theoretical concepts in terms of aggregation/disaggregation, TSO-DSO interactions and responsibilities, market clearing strategies.

An exhaustive simulation of TSO-DSO coordination schemes requires the simulation of both transmission and distribution network, including all the power devices and source of flexibility which are connected to the electricity system. For this reason, all the software blocks managed by the simulation platform have been designed in order to manage large amount of data and the related algorithms have been designed by means of simplifications that define a trade-off between simplicity (low computational burden) and accuracy.

In order to test the main interactions among system actors, the simulator has been coded by defining three main layers:

- The **market layer**, which integrates the market clearing routines aimed at solving the forecasted conditions of network imbalance and congestions by optimally activating the flexibility bids. It can have different structures depending on the implemented TSO-DSO coordination scheme (central market, central+local markets, etc.) and integrates the possibility of accepting complex bids submitted by distribution resources too.
- The **bidding and dispatching layer**, which implements the aggregation/disaggregation routines that translate the flexibility of physical resources (both transmission and distribution ones) into profitable bids to be submitted to the market. It also convert the market directives into individual set-points to be sent to the activated resources.
- The **physical layer**, which simulates the physics behind each considered flexible power unit and the effects of its produced/consumed power on the electricity system. In addition to the network behavior, it also simulates the autonomous actions taken by network operators on grid asset, including curtailment in case of unforeseen congestions.



These layers are composed by different blocks that can be called in sequence by a dedicated scheduler. Numerous settings can be set in order to simulate arbitrary market and bidding dynamics (market frequency, latency, time horizon).

The document describes in detail the simulation blocks that have been developed by the SmartNet team, illustrating the role of the main functions, the variables used to make interaction among different layers, as well as the necessary settings for the setup of simulation platform. In spite the simulation platform will not publicly available, the level of details adopted for this report allows the reader to understand the main dynamics behind each different software block, as well as the overall information flow between the simulated system actors (market operator(s), network operators, aggregators, users/producers).

1 The SmartNet Simulator

In order to allow distribution system resources to provide ancillary services to the power system, the project SmartNet proposed different TSO-DSO coordination schemes [1]. These interaction models, in addition to enable an efficient procurement and activation of reserves located ad distribution level, new ancillary services aimed at guaranteeing a better management of the distribution network (i.e. congestion management and balancing).

Each TSO-DSO coordination scheme has its own peculiarities and the performance is expected to be dependent on the application scenario. The objective of the project SmartNet consists of testing the proposed TSO-DSO interactions on the 2030 scenarios expected for Italy, Denmark and Spain. For this reason, a dedicated simulation platform has been realized in order to implement the main concepts developed within the project aimed at supporting the coordination between network operators. The structure of the simulator can be described as a sequence of three main blocks:

- **The bidding and aggregation layer**

  It consists in the processes aimed at converting the flexibility of physical resources (located bot at transmission and distribution levels) into bids that guarantee a profit to their owner. In case numerous small resources are threated (such as the case of small distribution power units), these processes aggregates them in order to constructs completive flexibility bids.

  Once the ancillary services market has selected the optimal activations, the accepted bids are processed by this layer in order to convert market directives into power set-points for the controlled physical resources (disaggregation).

  This layer, described in detail in section 3, implements the algorithms proposed by [2].

- **The market layer**

  It implements the market clearing algorithms aimed at optimizing the activations of the submitted bids for the simultaneous management of two ancillary services: balancing and congestion management. Depending on the TSO-DSO coordination scheme, these services are limited to the transmission network (the TSO is a buyer of flexibility) or extended to distribution network too (both TSO and DSO are buyer of flexibility).

  In order to manage distribution resources and increase the competition, the implemented algorithms integrate the possibility of submitting bids with complex constraints, particularly useful for the management of (distribution) resources with rebound effects.

  This layer, described in detail in section 4, implements the algorithms described in [3].

- **The physical layer**

  It simulates the behavior of the physical devices connected to distribution and transmission grids on the basis of their internal states and set-points received by the bidding and dispatching layer. Their power exchanges are then computed in order to calculate the state of the electrical network.

  While the other two layers work on forecasted network situations, the physical blocks works on the actual state of the system. This means that residual imbalance (due to forecasting error) and congestions (due to unpredicted overloading and voltage issues) are managed separately by means

of secondary frequency regulation and unwanted measures.

The functions implemented within this layer are described in section 5.

The following sub-sections describes the main concepts behind the developed simulation environment, illustrating how the time dimension, the sequence of the blocks are managed by the software environment and their interactions. In addition, from section 2 to section 6, each layer is described in detail by listing all the coded functions and database variables called during the simulation process.

1.1 Time axis of the simulation

The SmartNet simulation is structured in order to return, on a discrete-time axis, the situation of the network as well as the status of the flexible devices. These time steps, represented in Figure 1, correspond to the time instants in which relevant updates are experienced by the physical layer: they can be represented by the state variation of devices and/or network assets, as well as the instants in which information are communicated from/to the other simulation layers.

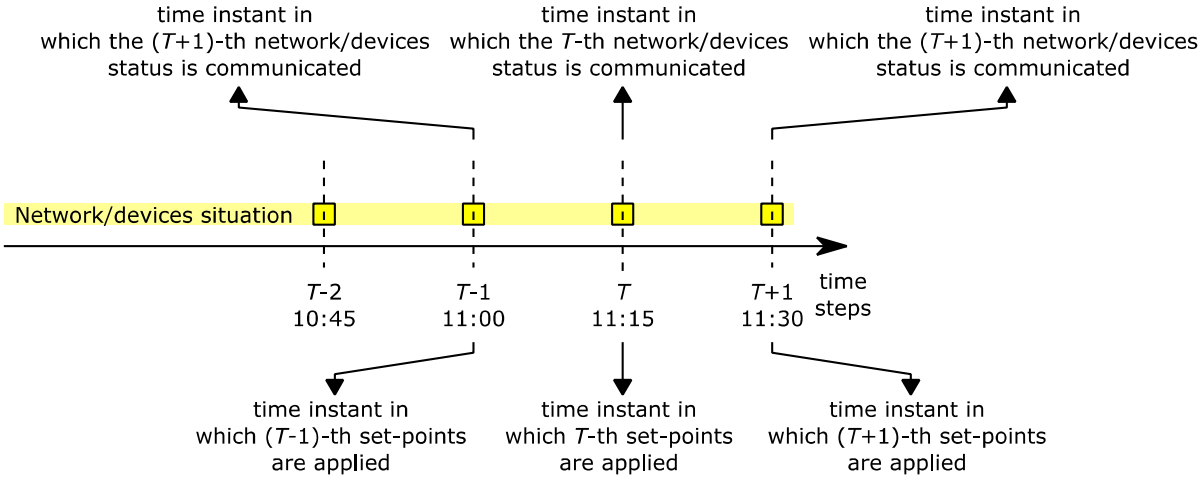


Figure 1 – Discrete-time axis for which the SmartNet simulator returns the network conditions and status of devices. A time resolution of 15 minutes is hypothesized

According to the diagram proposed in Figure 1, the application of the set-points (returned by the bidding and dispatching layer) and the communication of the network/devices status (again to the bidding and dispatching layer) are simultaneous. However, since the application of the set-points of the devices (as well as the natural evolution of the non-controlled power units) produces effects on the network physical quantities that have to be kept monitored until the reception of new set-points. In fact, between two generic T-th and $(T+1)$-th steps, the power exchange of non-controlled devices can change, failures occurs, network operators take action (congestion management, aFRR), etc. For this reason, an additional time step $(T+1)^-$ is added in order to provide information on the events occurred within two consecutive set-points reception (T and $T+1$). In practice (Figure 2), having assumed that the starting network situation is the one in T^-:

- the set-points returned by the other layers are applied in *T* and, consequently, the device and network status are updated;
- the updated status of the devices and of the network are communicated back to the other layer in the same time instant;
- the application of the set-points and the evolution of non-flexible devices determine transients in the network/devices status and their final values (steady state condition) are returned at $(T+1)^-$.
- During the time elapsed from *T* to $(T+1)^-$, the network/devices situation has evolved creating also new imbalance and activating the aFRR.

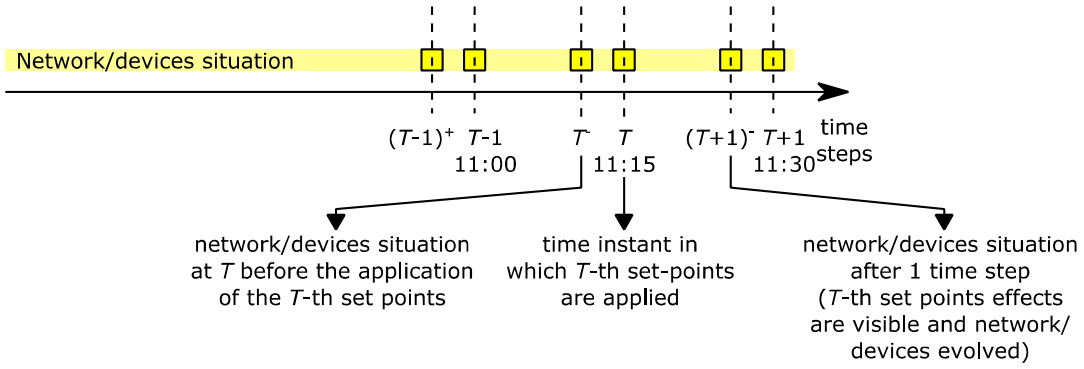


Figure 2 – SmartNet simulator events reported on the discrete-time axis.

A time resolution of 15 minutes is hypothesized

The set-points communicated at time *T* to the physical layer and the successive network/device status evolution correspond to the result of a structured process that involves all the SmartNet simulation layers. A graphical representation of this process (corresponding to one time step iteration of the SmartNet simulator) is reported in Figure 3 and it shows the following main steps:

1. The past network/devices situation is collected for:
    a. Physical layer – return a forecasting of the situation of the network/devices when the set-points will be applied (and beyond).
    b. Bidding and dispatching layer – Calculating bids of resources that directly access to the market.
    c. Bidding and dispatching layer – Aggregating small units to access to the market.
2. The outputs of these three sub-processes are managed by the market layer in order to report them in the optimization functions of the market clearing algorithm. The market returns the optimal set-points of the modelled flexible resources aimed at solving energy balance and network congestions.
3. Once set-points have been disaggregated/managed by the bidding and dispatching layer, this last layer sends them to the physical one, as well as the set-points for the network asset controlled by the operators (e.g. tap changing transformers).
4. At this step, set-points are applied (*T*-th time step) and begin to produce effects on the devices and on network status.
5. However, having considered that the forecasted situation at step 1.a. is subject to errors, unforeseen situations (deviations from ideal set-points, congestions, residual imbalance, etc.) may happen and:

a. resources subjected to forecasting error activate their assigned set-points only if their current flexibility margins allow them;
b. network operators perform corrective actions (curtailment of resources in case of congestions, aFRR activation in case of residual imbalance, etc.).

All these measures take time to be activated and, in order to evaluate their impact on the network state, their impact is reported at $(T+1)^-$ (immediately before the application of the new set-points, simultaneously computed by the other layers).

6. Therefore, in correspondence of $(T+1)^-$ time step, the network is assumed to be:
   a. Without congestions (solved by the market with mFRR activations and congestion management strategies performed by network operators)
   b. Balanced (thanks to the activation of mFRR and aFRR)
   c. Ready to receive new set-points by the other layers (in order to replace the activated reserves)
7. The process is repeated from step 1 for the computation of the set-points to be activated at $T+1$.

The diagram reported in Figure 3 already indicates some concepts related to the time dynamics of the system. The following sub-sections are explaining more in details the aspects of latency L and market clearing period T_s.

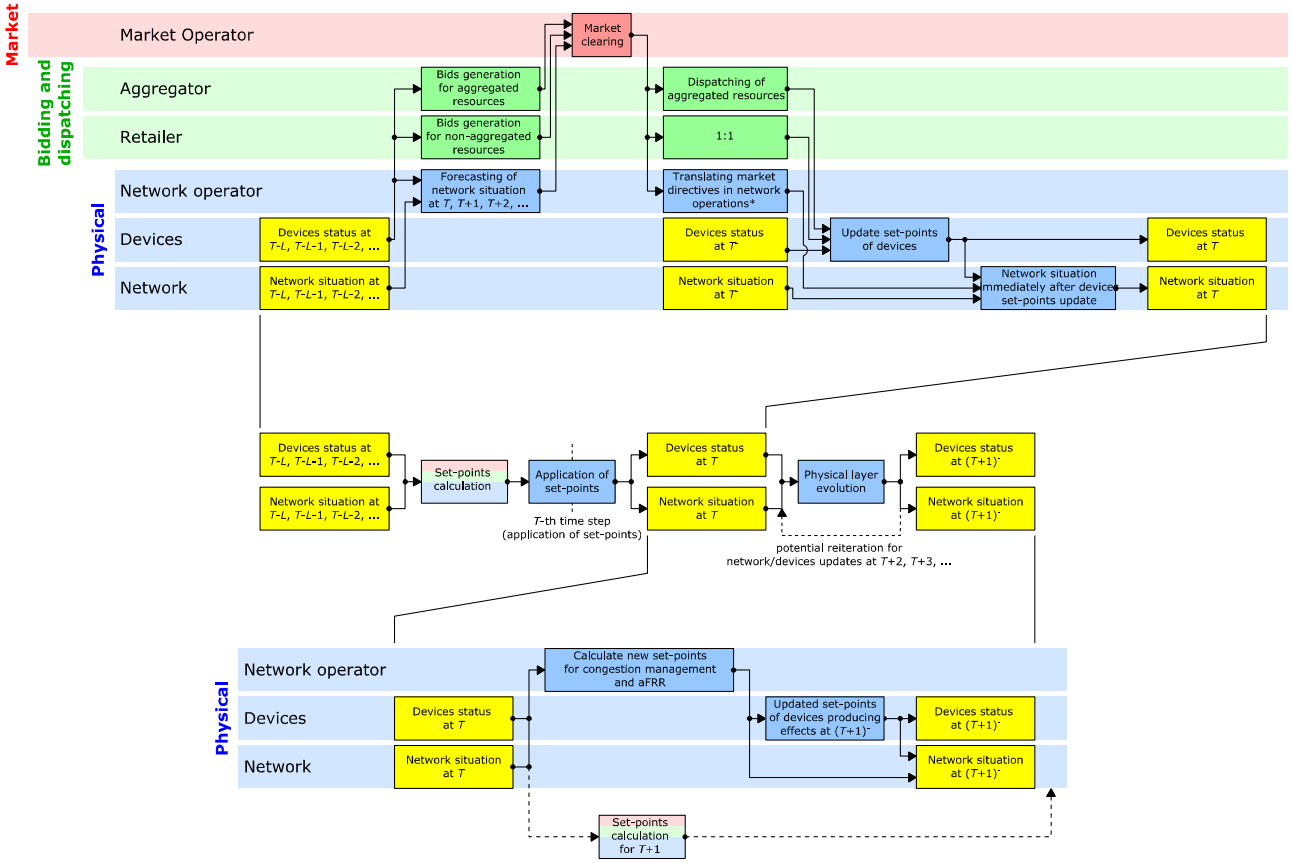


Figure 3 – Details of a generic SmartNet simulation process

1.1.1 Aggregation/Market/Disaggregation Latency (from devices data collection to application of set-points)

In real life, the execution of the bidding/market clearing/dispatching processes (also reported in Figure 3) isrequires computational burden and time. This means that, from the measured/estimated imbalance and congestion to the application of corrective set-points, there is a time latency. This latency may depend on several factors:

- size of the network;
- aggregation/disaggregation timing;
- amount of bids to be processed by market clearing algorithms;
- amount/complexity of constraints to be included in the optimization algorithms;
- etc.

Taking into account the discrete-time axis defined at the beginning of the document, this latency L can be measured with the amount of time steps elapsed from the data collection (instant in which bidding/dispatching and market layers are querying the physical one) to the set-points application on resources. Figure 4 graphically reports the latency concept:

- when L=1 the network/devices are monitored till T-1, the set-points are communicated at T^- and start to produce effects on the physical layer starting from T;
- when L=2 the network/devices are monitored till T-2, the set-points are communicated at T^- and start to produce effects on the physical layer starting from T;
- when L=3 the network/devices are monitored till T-3, the set-points are communicated at T^- and start to produce effects on the physical layer starting from T.

In all the cases, after the application of the set-points, the physical layer autonomously evolves until the reception of new set-points at T+1. Therefore, as anticipated above, at $(T+1)^-$ the simulation results will report the network/device status after their evolution according to the forecasting error and the possible interventions of network operators in order to contain residual imbalance and congestions.

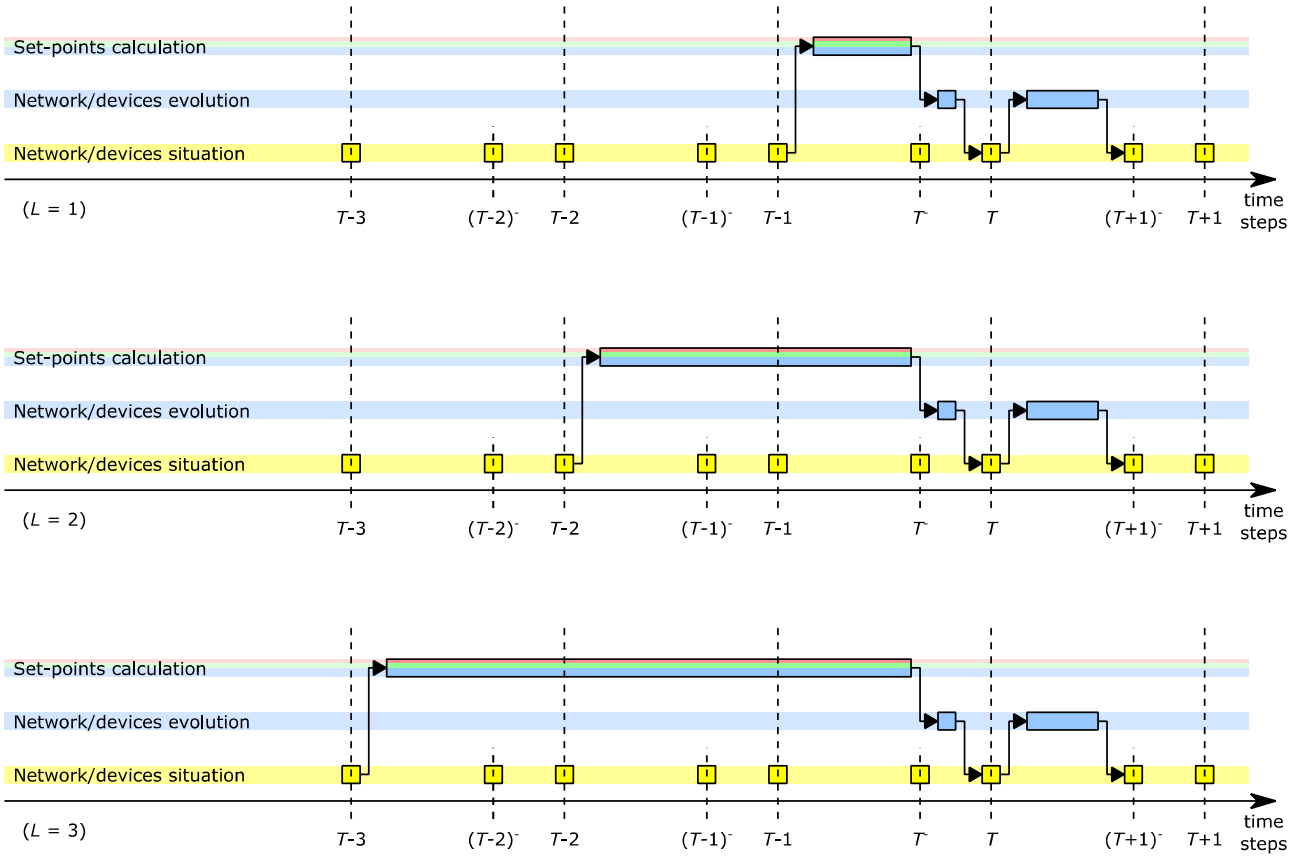


Figure 4 – SmartNet simulation process: latency (the set-point calculation process consists of a combination of aggregation, market and disaggregation routines)

1.1.2 Market clearing frequency

Previously, two important time quantities have been defined:

- the resolution (… T-1, T, T+1 …) of the discrete-time axis, on which the simulation results are reported;
- the latency L of the set-points calculation, starting from the network/devices measurements.

In addition to these, another important parameter to be defined is represented by the market clearing frequency. In fact, it is not necessary to have a cleared market with the same time resolution adopted for the provision of the simulation results, but it can be arbitrarily selected on the basis of standard balancing/congestion management needs.

According to this, the frequency (period) T_S is defined as the amount of time steps (referred to the simulation discrete-time axis) between two consecutive market clearing processes. Even in this case, the concept can be represented graphically (Figure 5). From the reported diagrams, it is possible to notice that:

- in order to provide the set-points at the generic time step T, aggregators and market can access only to network/devices situations before (T-L) and only making forecast for the consecutive time steps ($T > T$-L);
- the market clearing frequency cannot be lower than 1;
- when T_S>1, the intermediate network/devices situations have to be calculated by means of a dedicated processes which are separated from the market one (modelling only the network/devices independent evolution and the possible interventions of network operators);
- in this last case, the market clearing and disaggregation processes return a series of set-points to be applied in the next time instants (not only at T, but also at T+1, T+2, … T+T_S).

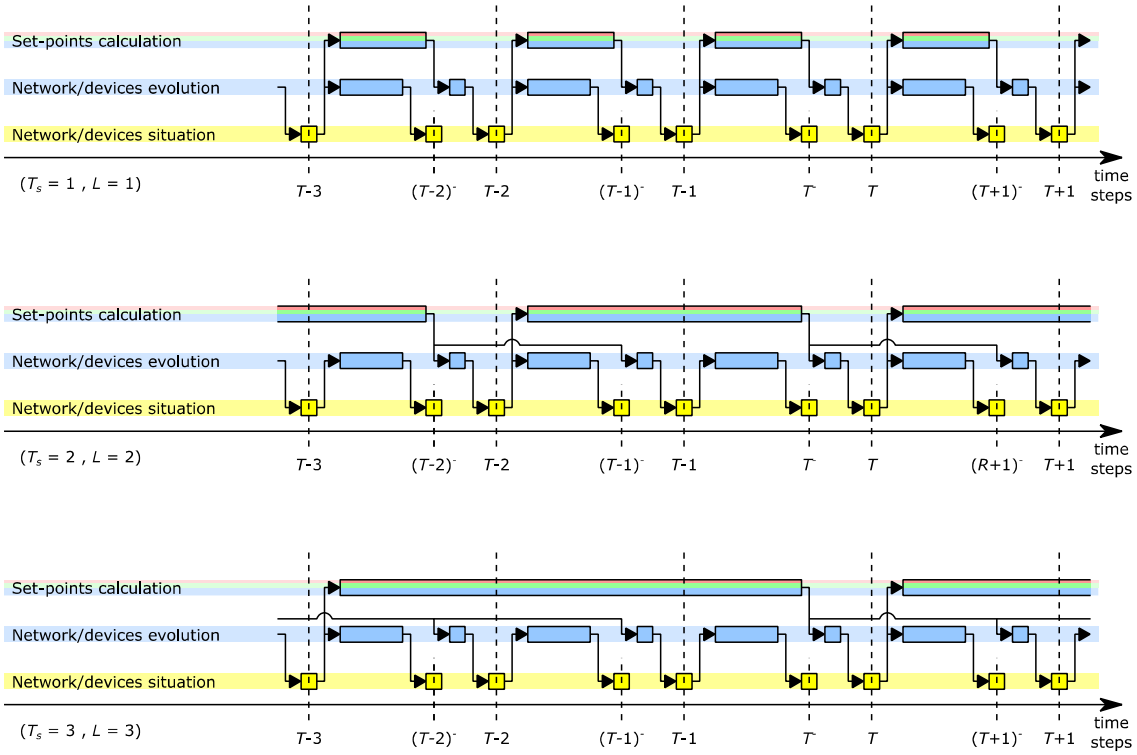


Figure 5 – SmartNet simulation process: market clearing frequency

1.1.3 Pipelining

The previous section reports scenarios in which the latency corresponds to the market clearing period ($L = T_S$). However the possibilities are not limited to this situation.

In principle, the provision of set-points happens with a fixed timing (e.g. every 15 minutes a new series of set-points is provided to the devices/network). However, the process of generating these values can also be faster, with a latency $L < T_S$ (Figure 6). In this case the market clearing algorithms generally process the set-points for more time intervals (T_s sets of set-points).

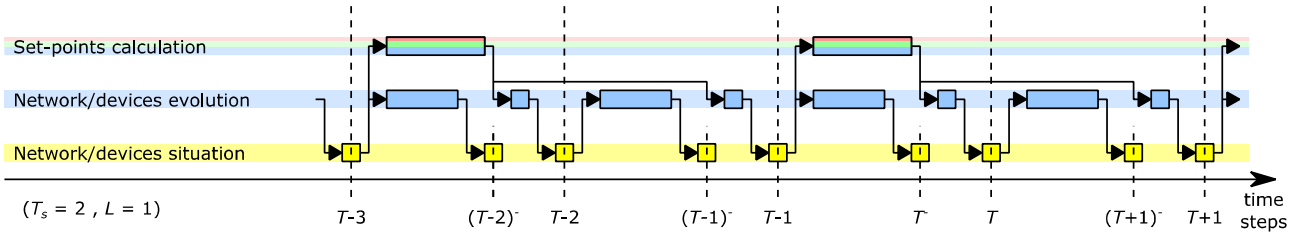


Figure 6 – SmartNet simulation process: case for which $L < T_S$

When $L > T_S$ the situation is more complex. Looking at Figure 7 it can be noticed that separate processes have to be run in parallel in order to provide the required set-points at every T_S. In this case, one of the processes aimed at generating the set-points (layer (1)) has not visibility of the ones generated by the parallel processes (layer (2)) and vice versa. This aspect is, of course, a significant issue and this decoupling should be faced with appropriate techniques.

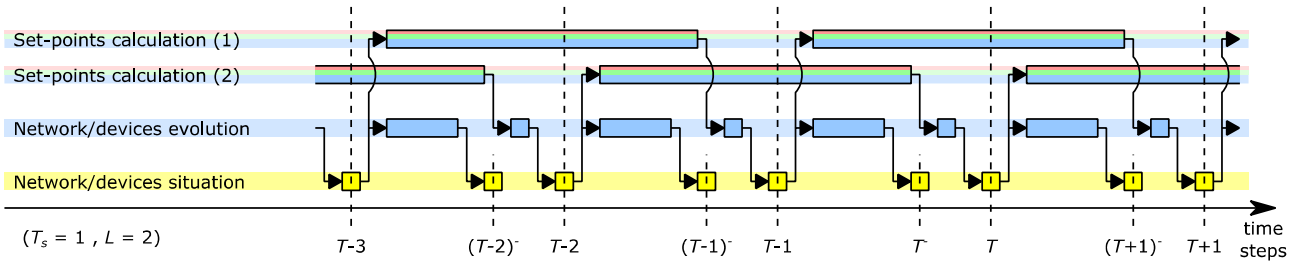


Figure 7 – SmartNet simulation process: case for which $L > T_S$

1.2 Simulation of the SmartNet TSO-DSO coordination schemes

The investigations performed by SmartNet have highlighted five different coordination schemes that describe the interactions between TSO and DSO for the management of flexible resources for the provision of ancillary services. These schemes are described in [1] and [3] and are represented by:

- A. Centralized ancillary services market model
- B. Local ancillary services market model
- C. Shared balancing responsibility model
- D. Common TSO-DSO ancillary services market model
- E. Integrated flexibility market model

Having considered the concepts described within the previous section, a single iteration of the SmartNet simulator (detailed in Figure 3) is reported in the sequence diagram depicted in Figure 8. In this scheme (representing two hypothetical cases of L and T_S) the processes responsible of generating and applying set-points are highlighted with a red path, while the processes describing the network/devices independent evolution are highlighted with a green path.

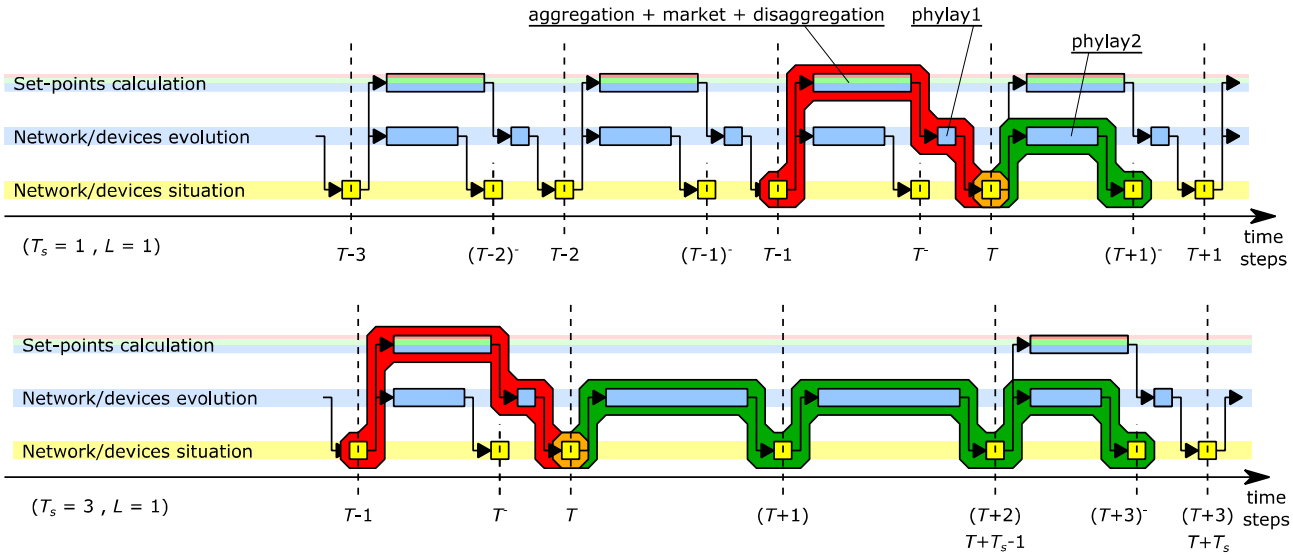


Figure 8 – Path followed by a single iteration of SmartNet simulator for two different selections of L and T_S

The following sub-sections report the detailed sequence diagrams adopted by the simulator in order to simulate the different TSO-DSO coordination schemes. It will be noticeable how TSO-DSO interactions will mainly impact on the market structure, rather than the other layer, except for TSO-DSO coordination scheme C, where the balancing responsibility assigned to the DSO is significantly affecting the simulation of the distribution network.

1.2.1 TSO-DSO coordination scheme A – Centralized ancillary services market model

In this scheme, the process describing the generation of individual set-points (one for each flexible resource) is reported in Figure 9. It can be noticed that, since the market includes only the transmission network model, the forecasted network status is required only at transmission level. According to [1], distribution grid constraints can be potentially modelled within the central market clearing algorithm. This last case, however, would be extremely similar to coordination scheme D1 (in terms of simulation results) and, for this reason, this variant is not investigated. The sequence diagram showing the simulation steps of TSO-DSO coordination scheme A is reported in Figure 9.

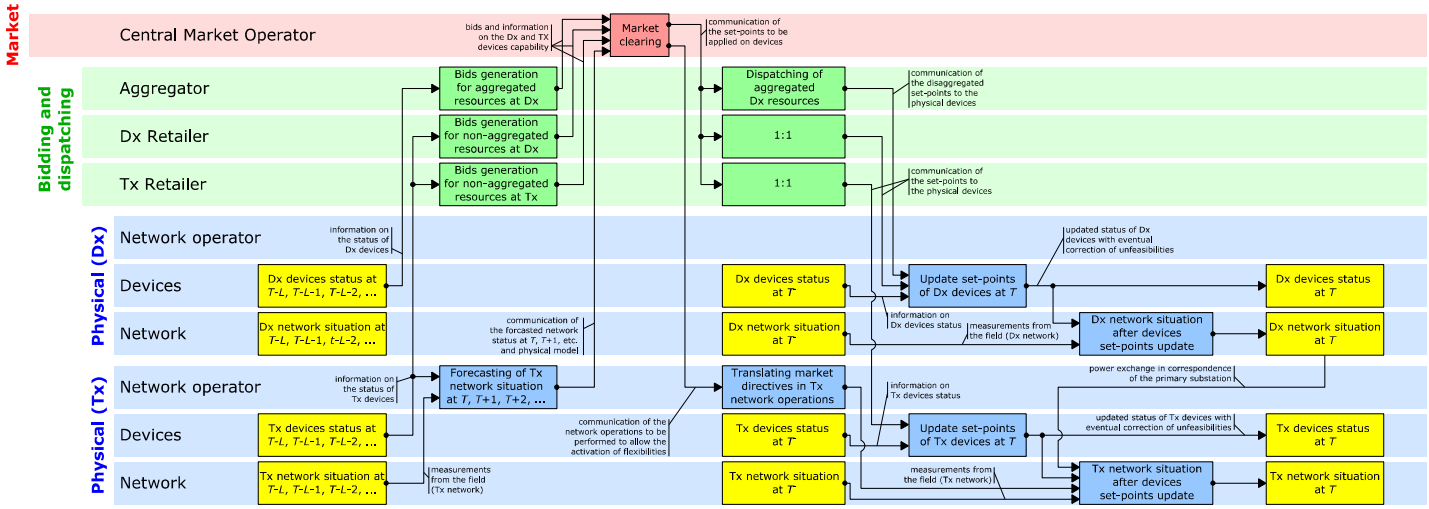


Figure 9 – TSO-DSO coordination scheme A – generation of individual set points for energy balancing

1.2.2 TSO-DSO coordination scheme B – Local ancillary services market model

In this coordination scheme, in addition to the centralized market, a local market is cleared at distribution level. The DSO has priority to procure the necessary flexibilities in the local market (i.e. for local congestion management and rebalancing of taken actions) and, after that, the remaining resources are forwarded to the centralized market (which also includes transmission elements). The sequence diagram showing these steps is reported in Figure 10.

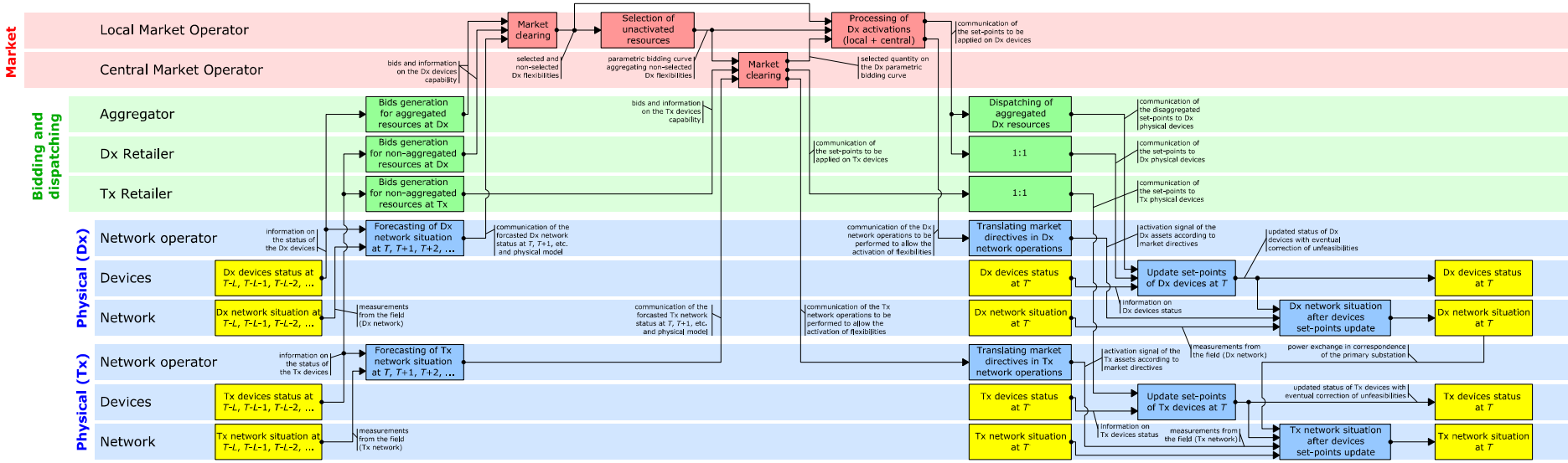


Figure 10 – TSO-DSO coordination scheme B – generation of individual set points for energy balancing

1.2.3 TSO-DSO coordination scheme C – Shared balancing responsibility model

This coordination scheme is fairly different with respect to the other ones, since the DSO is called to act with balancing responsibility and the management of transmission and distribution system is completely decoupled. The same basic sequence diagram can be considered representative of the two distinct processes (one for distribution and one for transmission) and they are schematized in Figure 11.

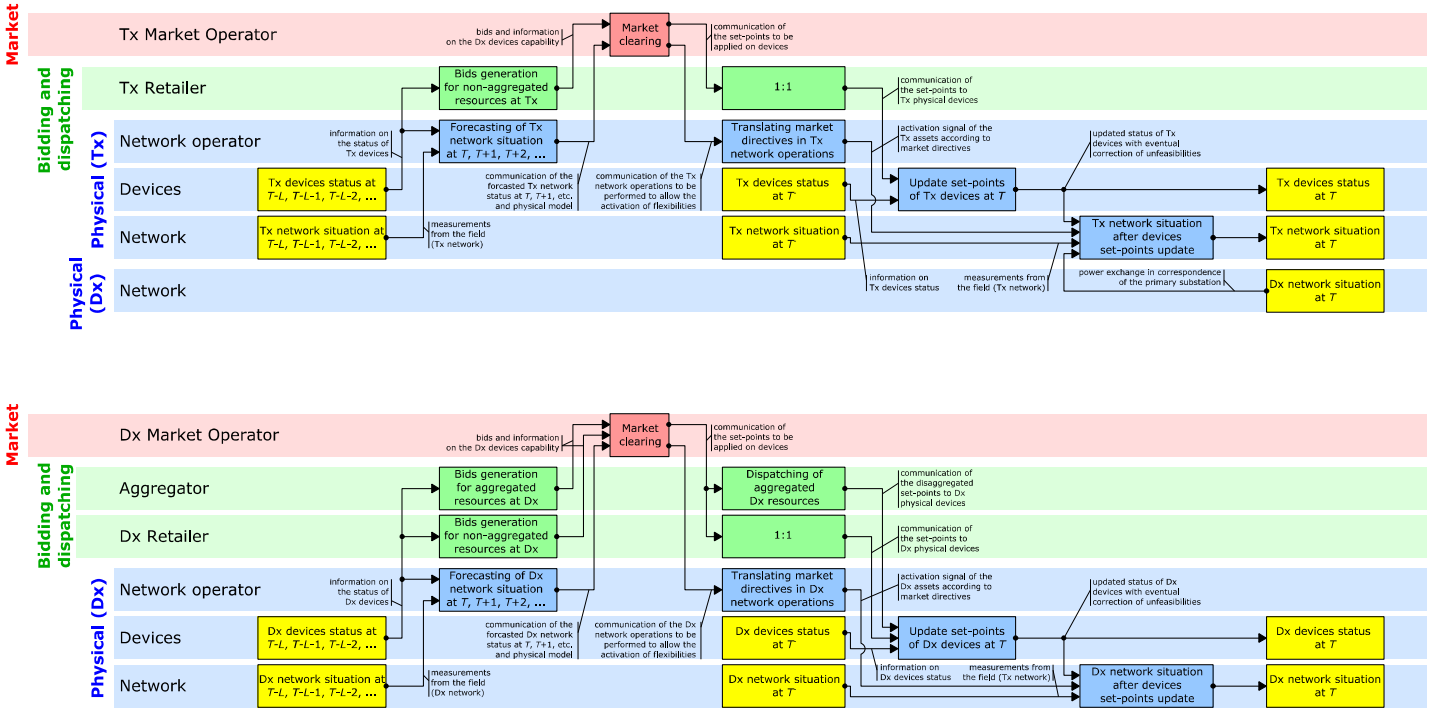


Figure 11 – TSO-DSO coordination scheme C – generation of individual set points for energy balancing

1.2.4 TSO-DSO Coordination scheme D – Common TSO-DSO ancillary services market model

From the simulation perspective, this coordination scheme is identical to coordination scheme A, except for the integration of the distribution network model in the centralized market clearing. This means that, also the forecasting of the distribution network state has to be calculated and sent to the common market. The process of generating individual set points is schematized in Figure 12.

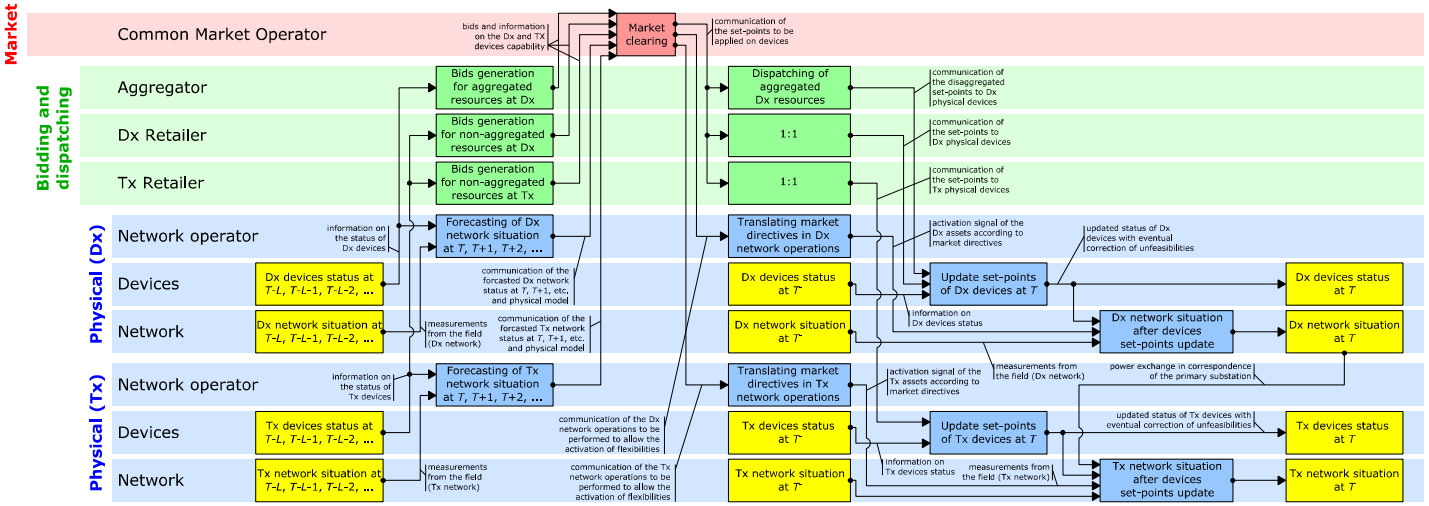


Figure 12 – TSO-DSO coordination scheme D – generation of individual set points for energy balancing

1.2.5 Process describing the independent evolution of device status and network

In real world, during the processing of aggregation/market/disaggregation routines, the controllable devices and the network states are subjected to independent evolution, which occurs because of the forecasting error that determines:

- deviations of resources from the set-points requested by the disaggregators;
- unforeseen network congestions that requires re-dispatching of critical resources;
- residual network imbalance (consequence of the previous two points) that has to be managed by means of aFRR activations.

The simulation environment takes into account the effects of the forecasting error on the application of set-points. The three considered points are processed by means of the sequence diagram reported in Figure 13, which describes how network/devices are evolving from the application of market/disaggregators set-points in *T*-1 to the time instant immediately before the application of the market/disaggregators set-points in *T*.

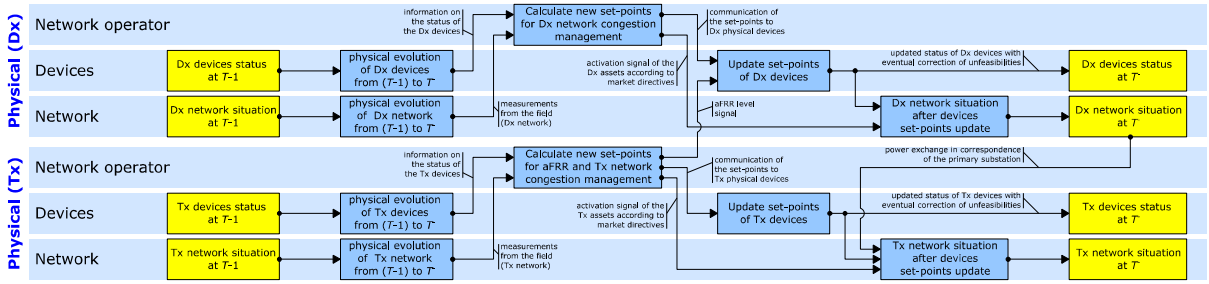


Figure 13 – Independent network evolution and operations during the bidding and market processes

This process can be applied to any TSO-DSO coordination scheme. However, a different approach has been selected for coordination scheme C. In this case, having assumed that the DSO has balancing responsibility for the distribution network, it is called to provide an independent aFRR regulation in order to manage the residual imbalance of its system. According to this, Figure 14 report the sequence diagram aimed at describing the network/devices status evolution for coordination scheme C.

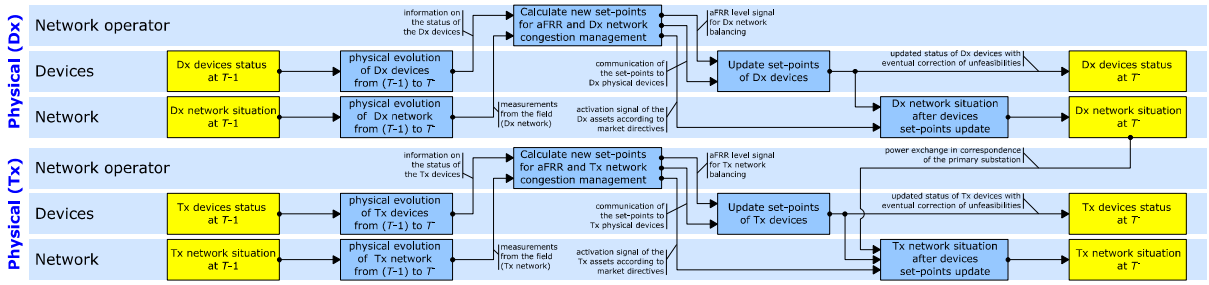


Figure 14 – Application of individual set points for energy balancing and consequent network evolution (TSO-DSO coordination scheme C)

2 Scheduler

2.1 Brief description of the module

The SmartNet architecture has been designed in a way as to allow different companies to develop and run each their own blocks with as minimal necessary interactions as possible. For this reason, an architecture based on sequentially executable blocks has been chosen. This sequential execution order leads to a scheduler that just corresponds to a simple for loop over the blocks and just executes the `block.execute()` method for each block. The scheduler defines the order in which blocks have to be executed. In particular, according to architecture described in section 1, three main blocks can be identified:

- **Bidding and dispatching layer** (described in section 3)
  It simulates the processes that translate the status of the flexible units in bids to be transmitted to the market operator. These processes also include aggregation of small unit flexibilities in a single one (big enough to participate to the market). Secondly, once the market has returned a solution, this layer dispatches the set-points to the controllable units.
- **Market layer** (described in section 4)
  It runs the optimization algorithms (market clearing routines) that return the optimal set-points that have to be applied in order to balance the considered electricity system. These set-points are communicated to the bidding and dispatching layer (some of them are aggregated) in order to be adequately dispatched to the resources of the physical layer.
- **Physical layer** (described in section 5)
  It simulates the electrical behavior of the network and devices for a given set of set-points provided by the other layers (bidding and dispatching layer). This layer also includes the low-level controls of network operators on grid assets and devices for the management of network congestions and borders balancing (secondary frequency control).

2.2 The high level architecture

The high level architecture and corresponding directory organisation of code is as given in Figure 15. It includes orange blocks (process blocks) and blue blocks (database storage blocks). In the same picture, one can see that some blocks operate at node level (green area) and some at device level (yellow area). Different modules do not send messages between them directly, but rather each block writes all its outputs to the global database and then finish execution. Then the next block knows exactly where to pick up that (and other) information from the database.

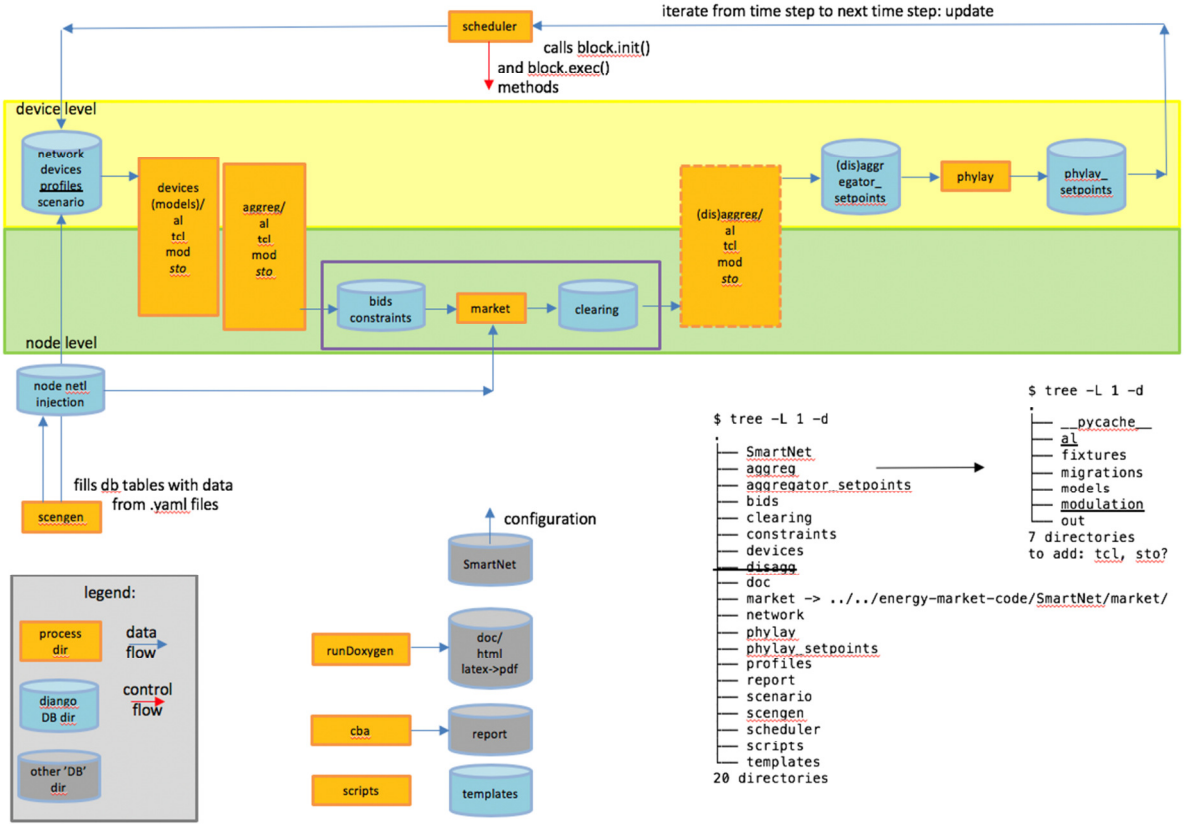


Figure 15 – Block sequence diagram: implementation in process, tables and directories

A SmartNet simulation starts by reading the prepared scenario. A scenario consists of human prepared data and then some of it is also still deduced/expanded initially in the simulation. Then, the aggregation blocks (one per specific technology) aggregates its flexibility capabilities per node. These are then written into bids and constraints table within the database. The scheduler then gives control to the market. The market reads these bids and constraints table and sets up a market clearing optimisation table (in AMPL). The market then solves it and writes its outputs (accepted bid quantities and nodal prices and the state of the network nodes and edges) to the database in what it is called the *clearing block*. This data is then read by the disaggregation blocks. These blocks disaggregate the accepted quantities at the node level into accepted quantities at the device level and as such reach set-points for their devices. The physical layer then checks whether these activations are feasible for the network, and performs a sequence of tuning algorithms to resolve any problems. The resulting set-points and state of the network is written to the database and serves as the starting point for the iteration corresponding to the next time step. The scheduler repeats this process for every time step defined in the simulation. The flow of operation depicted has to be repeated in order to simulate the desired time horizon.

2.3 Input from database

This section lists and describes the tables where the data are read. It is important is to underline the control parameters that affect the behaviour of the module and that can be changed in order to simulate different scenario configurations.

- `L_m`: This is the **L**atency of the **m**arket. Latency is defined as the (integer) time (index) difference by the time step at which the market calculates its outputs (`atT`) and the first tim step it calculates the outputs for (`forT`). This latency also implies that bidders should submit their bids by at least so much time in advance for the bids to be considered in the market.
- `H_m`: Length of the **H**orizon of the **m**arket. This is the [difference between the last and the first `forT` that the market considers in its bids] + 1.
- `rollingHorizonAggregatorAndMarket`: When `True`, this indicates that the market is run every time step. If `False`, the market will only be run every `H_m` time steps, so all decisions are final, since outputs for any single time step are only calculated once.
- `disaggregateOnlyOncePerMarketClearing`: If `True`, it disaggregators are called only once for the entire time horizon. If `False` disaggregation is done every time step instead, in which case disaggregators have to produce only values for one iteration (at a time).
- `disaggregateWithOnlyMostRecentPhylayToOutput`: Determines which `phylay` output is used to disaggregate. If `False`, disaggregators work with information from block `phylay2` (from older `atT` for `forT` where `atT<forT-L_m` already. If `True`, disaggregation happens only with `phylay2` info produced at `atT=forT–L_m`.

2.4 List of functions of the module

`run(scenarioId, overwrite_CS_code, marketSolver='cplex')`: This is the main scheduler function. One can specify the simulation to be carried out, the coordination scheme that has to be tested for this scenario and the solver that the market should use (cplex/gurobi). This function performs a loop that just calls the execute function of each block in a specific order. Different schedules are possible. For example it can be that the aggregator and market module are called for every loop iteration, or not. Some cases like these are described in the previous section where the meaning of the called parameters is described.

The scheduler has been designed with care that all configurations (parameter value combinations) generate a consistent data flow. This means that all data that is produced (written to an (`atT`, `forT`)-*location*) is read by subsequent modules (read after write guarantee) as well as that all data consumed (read from an (`atT`, `forT`)-*location*), is indeed actually produced by previous modules (write before read guarantee).

2.5 Output to database

The scheduler just corresponds to a loop calling all `block.execute` functions from the process blocks. The only table it also writes to is the timing logging table, where it logs the name of the function and starting time in seconds.

3 Bidding and dispatching layer

The bidding and dispatching layer is responsible of translating the physical flexibility of power devices in bids (bidding section) to be processed by the market blocks. Once the market clearing algorithm has returned its results, the accepted activations are again processed by this layer (dispatching section) in order to convert them into individual set-points for the physical resources (managed by the physical layer).

Since the majority of the connected resources are located at distribution level and are characterized by small power flexibility, most of the time the bidding and dispatching processed are actually represented by aggregation and disaggregation functions, which merge together small resources in order to produce bids with larger quantities.

According to [2], the aggregation and disaggregation functions are carried out for each technology independently. The following sections analyze in details how these algorithms have been implemented within the SmartNet simulator code in order to interact with the rest of the simulation environment (Scheduler, database, market and physical layers).

3.1 Atomic Loads Aggregation/Disaggregation module

3.1.1 Brief description of the module

This class performs the aggregation of atomic loads and produces the bids for the market block. An atomic load consists of a device for which the activation can be postponed for a while, but once started cannot be paused or interrupted. The flexibility is produced by checking what loads are available in the system, and anticipating or postponing some of them in a coordinated manner by solving a discrete optimization problem. In this simulation the `AggregatorAL` is used to provide demand response bids from wet appliances, ie. dishwashers, tumble-driers, washing machines. The system makes use of mixed-integer linear optimization and requires CPLEX.

The detailed description of the atomic loads aggregation/disaggregation model is found in [2]. What this model represents, in short, is the controlled postponed activation of wet appliance loads with a fixed power profile.

As with other aggregator modules, the atomic loads aggregator is created in the database by the `Scheduler` and controlled by calling three functions:

- `AggregatorAL.initialize`
  - Obtains scenario, nodes and networks data from the database, finds the bidding node of each associated device node (the nodal resolution of the system varies with the simulated TSO-DSO coordination scheme). The flow diagram is represented in Figure 16.
- `AggregatorAL.execute` in mode `"aggreg"`
  - In case of available devices for a specific network node, it creates a flexibility bid and places it into the database. The flow diagram is represented in Figure 17.
- `AggregatorAL.execute` in mode `"disagg"`
  - It activates the corresponding devices of a previously formulated bid that was accepted by the market clearing module. The flow diagram is represented in Figure 18.

As with the other aggregator modules, the `AggregatorAL` is integrated to the `Scheduler` (see section 2) and retrieves any other necessary data from the database.

The database is initially populated with characteristics of each aggregated device (stored in `devices_WetConstants`). Each stored record represents the activation of one particular appliance assumed to happen at a given time of the simulation. These `device_WetConstants` instances are generated from a probability distribution, but, for simplicity, instead of working with the probability distribution we assume complete information from the `device_WetConstants` records that will be available during the day and at what time.

The database records associated with the `AggregatorAL`, as with other aggregators, are constantly modified by the physical layer. The physical layer will be updating these appliances through the use of `devices_WetVariables` and the functions `WetAggToVar` and `WetVarToDevOut`. More information is available in section 5.

When the `Scheduler` calls the `AggregatorAL.execute("aggreg")`, it updates its internal state of the time of activation of the available loads, possibly altered by the physical layer module.

As with other bidding modules, the results of the market clearing module also affects the `AggregatorAL`. When calling the `AggregatorAL.execute("disagg")`, the accepted bids involving this module are retrieved from the database. The module recovers the individual appliances involved in the bid formulation and activates them.

3.1.2 Input from other modules

The resources controlled by the `AggregatorAL` are inserted in the database by the scenario creation module.

The `Scheduler` can set up the aggressiveness of the `AggregatorAL` by adjusting the tail threshold. This represents unwanted rebound caused by the flexibility activation that will be carried beyond the current bidding window. If this value is too low, the algorithm is too conservative and no flexibility bids will be produced by the `AggregatorAL`. If it is too high, it means that possible large imbalances are being carried out of the simulation horizon. There is also the risk that the aggregator exhaust all its loads in the first moment and keeps none for future market iterations.

When calling `AggregatorAL.execute` the only arguments are related to the time window characteristics. There is `atT`, the time at which the aggregator is called and the latency *L*, which represents a future time step where the flexibility is actually needed. For instance, when called on aggregation mode, if `atT=10` and `L=3`, the aggregator will look for loads that are still available at the time step 10 (they have not started yet) and will modify their activation in order to change the aggregated consumption profile as much as possible to the time slot `forT=13`.

In disaggregation mode these arguments are just used as filters to find any corresponding accepted bids in the database.

3.1.3 Flow chart of the module

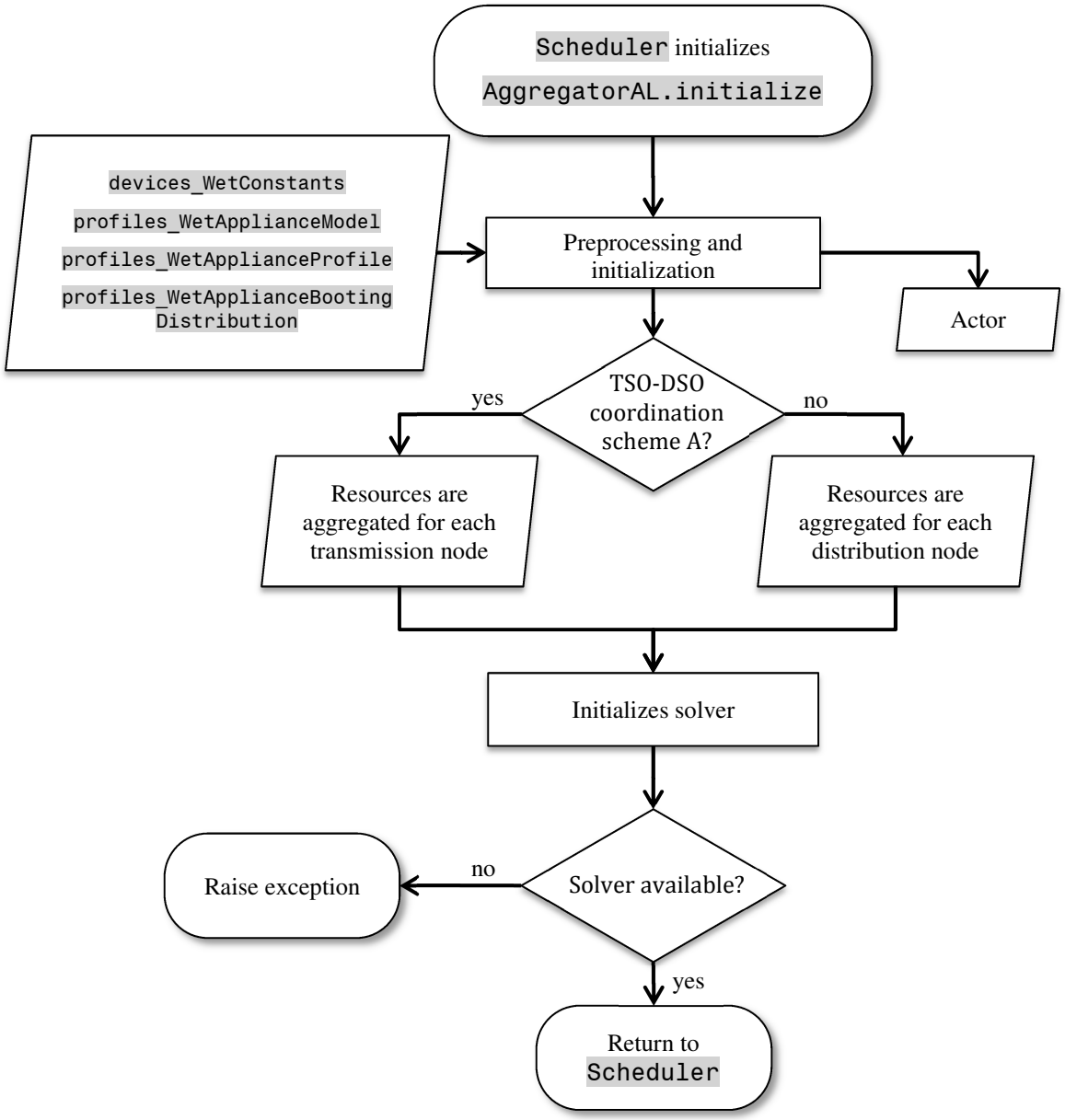


Figure 16 – Flow diagram of `AggregatorAL.initialize`

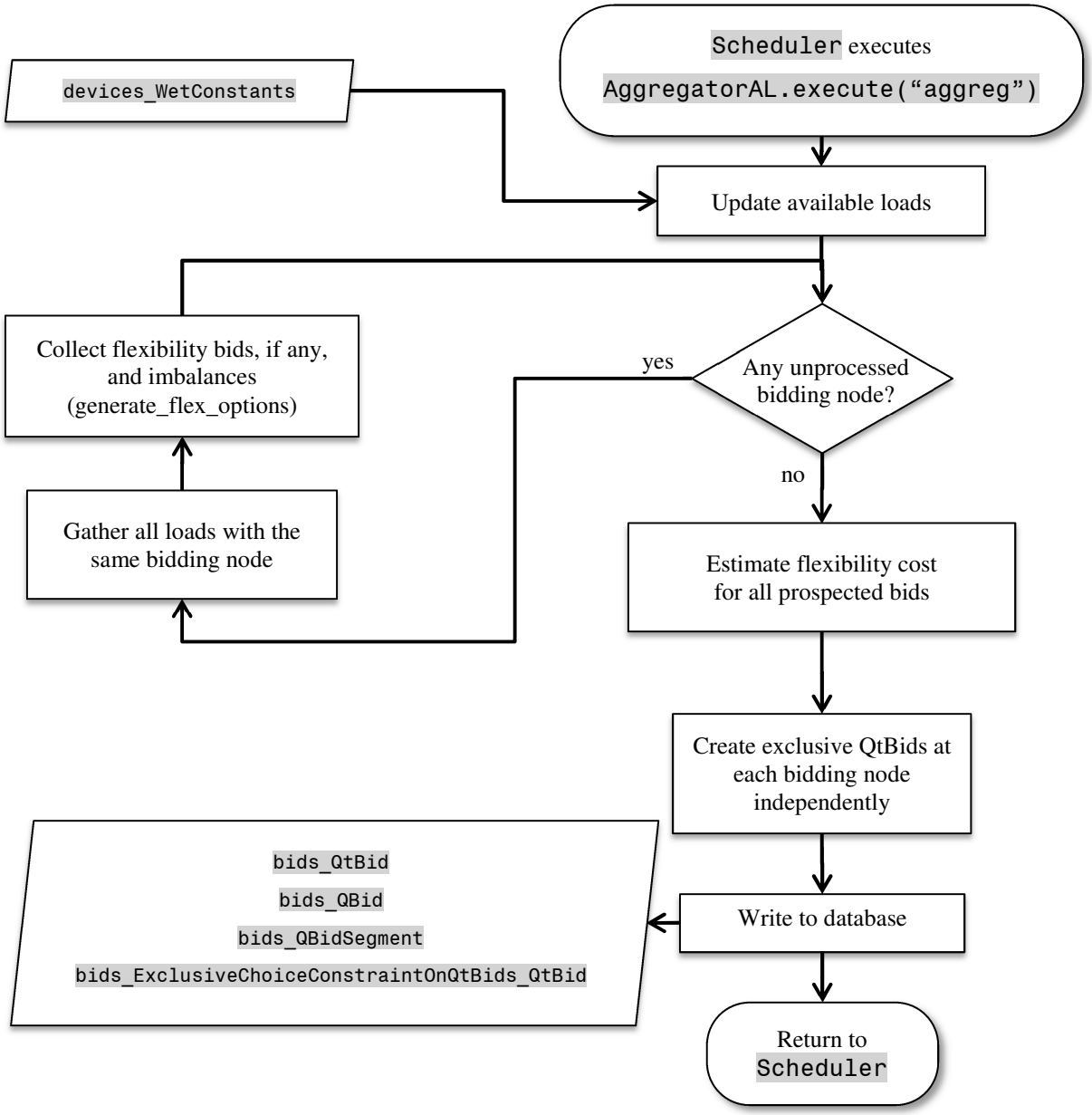


Figure 17 – Flow diagram of `AggregatorAL.execute("aggreg")`

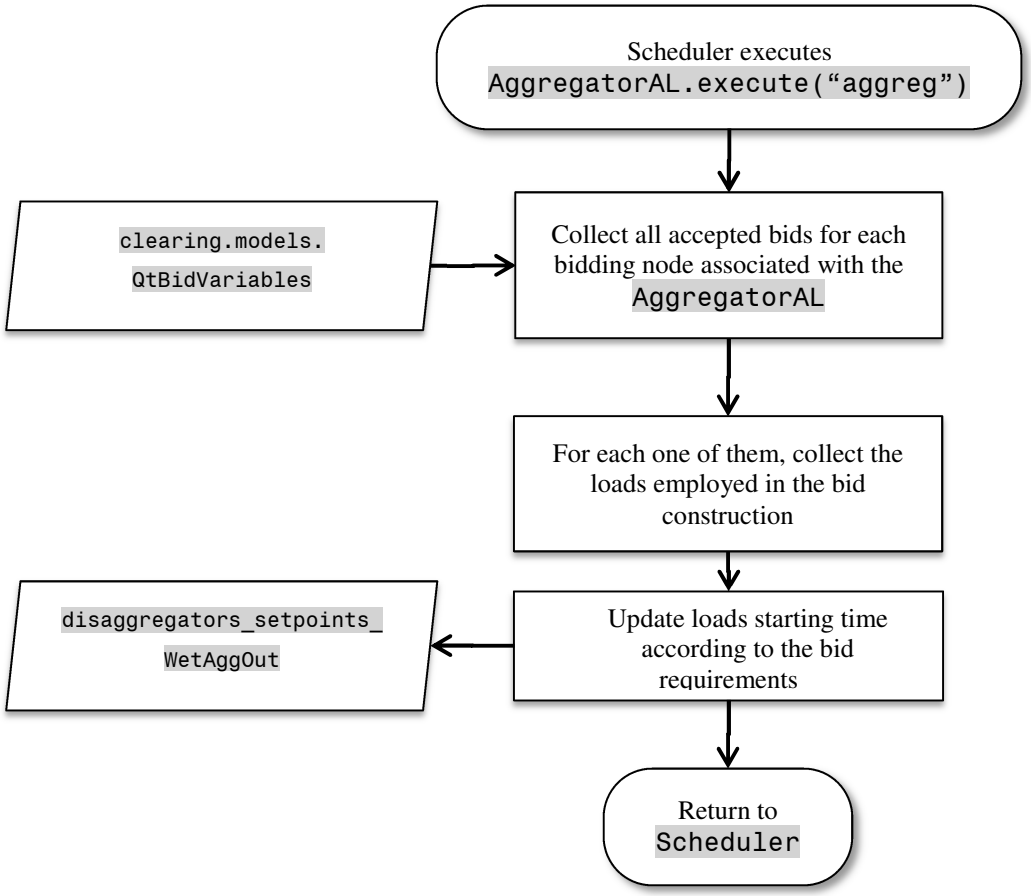


Figure 18 – Flow diagram of `AggregatorAL.execute("disagg")`

3.1.4 Output to database

The `AggregatorAL.initialize` adds only an instance of the **Actor** model representing the `AggregatorAL`.

`AggregatorAL.execute("aggreg")` possibly creates a bid for the corresponding market iteration. Creating a bid requires the creation of linked records on the following database tables:

- `QtBid`,
- `QBid`,
- `QBidSegment`.

If more than one single possible activation is returned by `AggregatorAL.execute("aggreg")`, multiple records are added to `QtBid`, `QBid` and `QBidSegment` database tables. In this case, the instance of `ExclusiveChoiceConstraintOnQtBids_QtBid` is also created, in order to specify the mutually exclusive nature of the submitted flexibilities.

`AggregatorAL.execute("disagg")` stores the activations to be sent to physical devices on `disaggregators_setpoints_WetAggOut` database table. These represent the activation of devices that were actually anticipated to form the flexibility promised by an accepted bid.

3.2 TCL Aggregation module

3.2.1 Brief description of the module

The objective of the Thermostatically Controllable Loads (TCLs) Aggregation module is to combine the flexibilities provided by a portfolio of TCLs defining, as output, a set of flexibility bids to be delivered to a certain market session. The aggregation model assumes a direct load control scheme over the TCLs, where the control variable consists of the temperature set-point that can be deviated from the baseline temperature (in general, the comfort temperature set-point) between the upper and lower limits (previously agreed between the end-users and the Aggregator). End-users received an economic compensation in exchange for the loss of comfort. Based on this cost, the Aggregator defines the bidding price. The detailed model of TCLs aggregation is described in [2].

A preliminary step to the algorithm is to define all possible control strategies than can be applied to each TCL within the portfolio. For this purpose, a set of possible temperature set-points is defined by splitting the control (and end-user comfort) margins of each TCL into a set of equal-sized temperature levels. In addition, it is assumed that the durations of the control actions are variable varying from one time-step to a maximum duration equal to the market horizon. As a result, the total number of control actions that can be applied to each TCL is the multiplication of the total number of temperature levels by the total number of possible durations. Each possible control action will lead to obtain a different (mutually exclusive) flexibility bid.

The TCL Aggregator module implements two main steps, one for each control action:

1. Simulation of the individual flexibility profile of each TCL that is defined as the difference between the baseline power profile and the controlled one. For this purpose, a second-order thermal model describing the dynamics of the TCL is implemented. Discomfort costs are calculated based on the internal temperature deviation from the baseline temperatures.
2. Aggregation of the individual flexibility profiles and discomfort costs to build the aggregated flexibility bid to be delivered to the market.

This process is repeated for all control actions, obtaining as a result a set of flexibility bids that are provided to the Market module. As anticipated above, these are sent with *exclusive choice constraints* to indicate that only a single bid can be accepted among a set of bids as they correspond to different type of control actions that cannot be supplied at the same time. As the flexibility that a TCL can provide at each time-step depends on the past states of the device, the bids include two additional constraints: they have to be "*non-curtailable bid*" and "*Accept all time steps or none".* In this way, it is ensured that a bid is accepted for all time-steps or it is not accepted at all (that is, that the whole power profile defined by the bid is accepted).

3.2.2 Input from database

The inputs from the database required by the TCL Aggregation module are listed next:

TCL Aggregator internal tables

- `aggreg_AggregatorTcl`
- `aggreg_AvailabilityProfile`
- `aggreg_AvailabilityStep`
- `aggreg_BidConfig`
- `aggreg_ComfTempProfile`
- `aggreg_ComfTempStep`
- `aggreg_Device`
- `aggreg_Envelope`
- `aggreg_ExtTempProfile`
- `aggreg_ExtTempStep`
- `aggreg_ExtTGProfile`
- `aggreg_ExtTGStep`
- `aggreg_IntTGProfile`
- `aggreg_IntTGStep`
- `aggreg_MaxTempProfile`
- `aggreg_MaxTempStep`
- `aggreg_MinTempProfile`
- `aggreg_MinTempStep`
- `aggreg_Tcl`
- `aggreg_TclStatus`
- `aggreg_TimeStep`
- `tcls_FlexProfSet`
- `tcls_TempSetPointSet`

Network parameters

- `network_Node`
- `network_SubNetwork`
- `network_SubNetworkType`
- `network_Network`

Price profiles

- `profiles_NodeDeltaCost`
- `profiles_NodeDeltaCostProfile`
- `profiles_NodeHasNodeDeltaCostProfile`
- `profiles_NodePrice`
- `profiles_NodePriceProfile`
- `profiles_NodeHasNodePriceProfile`

Scenario data

- `scenario_Scenario`

Devices set-points

- `devices_setpoints_TclDevOut`

3.2.3 Input from other modules

The only direct input of the TCL aggregation module is from the `Scheduler`, that provides at every market session the following parameters:

- Aggregation mode (`"aggreg"`)
- Scenario identifier
- Starting time-step of the simulation window
- Market latency
- Market horizon

3.2.4 List of functions of the module

The TCL Aggregation module is based on an iterative process aimed at generating the flexibility bids. This process comprises a set of loops that iterate, for each TCL in the portfolio, all the possible control actions that can be applied to each TCL unit, i.e. temperature set-points set and control durations.

The module has been implemented following an object-oriented programming model. Each main function corresponds to a different Class implemented in a separated python module (`*.py`) and includes all required functions to implement a certain loop. In the next lines, the main functions of each Class are described.

- `Aggregator.py` – Class `AggregatorTCL`

  It implements the interface between the Scheduler and the TCL Aggregator/Disaggregator modules. Gets all the required data for the simulation (scenario, coordination scheme, simulation window, list of nodes, etc.) and calls the function that starts the aggregation process:

  - `AggregatorTCL.doAggregation` starts the aggregation process.
  - `AggregatorTCL.updateStatusFromPhylay` reads the latest status of TCL units from the physical layer and updates the status within the aggregation model.

- `node_aggregator.py` – Class `NodeAggregator`

  It contains the main processes for simulating the aggregation process, especially the logic for generating flexibility bids for all network nodes (that have TCL devices connected), and sends the generated bids to the market.

  - `NodeAggregator.doAggregation` performs the practical aggregation. It includes a loop for carrying out the aggregation process for each network node.
  - `NodeAggregator.sendBidsMarket` saves the generated flexibility bids, including bid constraints, into the database.

- `tcl_aggregator.py` – Class `TCLAggregator`

  It contains the logic for simulating the aggregation process for all TCLs connected to a single network node. The main parameters of the simulation are initialised: list of available TCLs for control within the TCL set, the set of temperature set-points and the set of control durations to be simulated for each TCL connected to the specified node.

  - `TCLAggregator.doAggregation` performs aggregation for all available TCLs, all temperature set-points within the set-point set and all control durations within the control duration set.

- `portfolio_flex.py` – Class `PortfolioFlexSimulator`

  It contains the logic for simulating the individual flexibility profiles for all TCLs in the TCL set.

  - `PortfolioFlexSimulator.simulateTcls`: contains a loop for simulating each TCL for all the possible temperature control set-points and control durations within the control duration set.

- `set_point_flex.py` – Class `SetPointFlexSimulator`

  It contains the logic for simulating all temperature control set-points within the set-point set for a given TCL unit.

  - `SetPointFlexSimulator.simulateControlTemp`: for a given TCL, contains a loop for simulating each temperature control set-point within the set-point set for all control durations within the control duration set.

- `profile_flex.py` – Class `ProfilesFlexSimulator`:
  It contains the logic for simulating all control durations within the control duration set for a given TCL unit and temperature control set-point.
  - `ProfilesFlexSimulator.simulateProfiles`: for a given TCL and temperature control set-point, contains a loop for simulating each control duration within the control duration set.
- `multi_step_flex.py` – Class `MultiStepFlexSimulator`:
  It contains the logic for simulating all time-steps within the simulation window for a given TCL unit, temperature control set-point and control duration.
  - `MultiStepFlexSimulator.simulatePeriod`: for a given TCL, temperature control set-point and control duration, contains a loop for simulating each time-step of the simulation window
- `single_step_flex.py` – Class `SingleStepFlexSimulator`
  It contains the logic for simulating the behaviour of a TCL during a single time step.
  - `SingleStepFlexSimulator.simulateStep`: implements the required calculations to simulate individual flexibility and discomfort cost for a single time step when a particular temperature set-point and control duration is applied to a TCL unit. The calculation is based on a second-order thermal model. The evolution of the internal and envelope temperatures is also calculated as they are used as the initial values for the next time-step calculations.

3.2.5 Flow chart of the module

The general structure of the TCL Aggregation module is described in the following three flowcharts. The one illustrated in Figure 19 represents the main aggregation process. The second one (illustrated in Figure 20), which is included within the first chart, represents the steps for the generation of the aggregated bids. Finally, the third one (illustrated in Figure 21), which is included within the second one, represents the steps for the generation of the individual bids. The algorithm takes as inputs scenario and simulation information provided by the Scheduler, together with detailed data of the TCLs in the portfolio, including constants and profiles information from the internal database tables of the TCL aggregator. In addition, network information is also required because, as a function of the coordination scheme, the generated flexibility bids will be grouped per transmission or distribution nodes (CS-A: transmission nodes, other CSs distribution nodes). The results consist of a set of aggregated flexibility bids per node that are stored in the bids and constraints tables of the market.

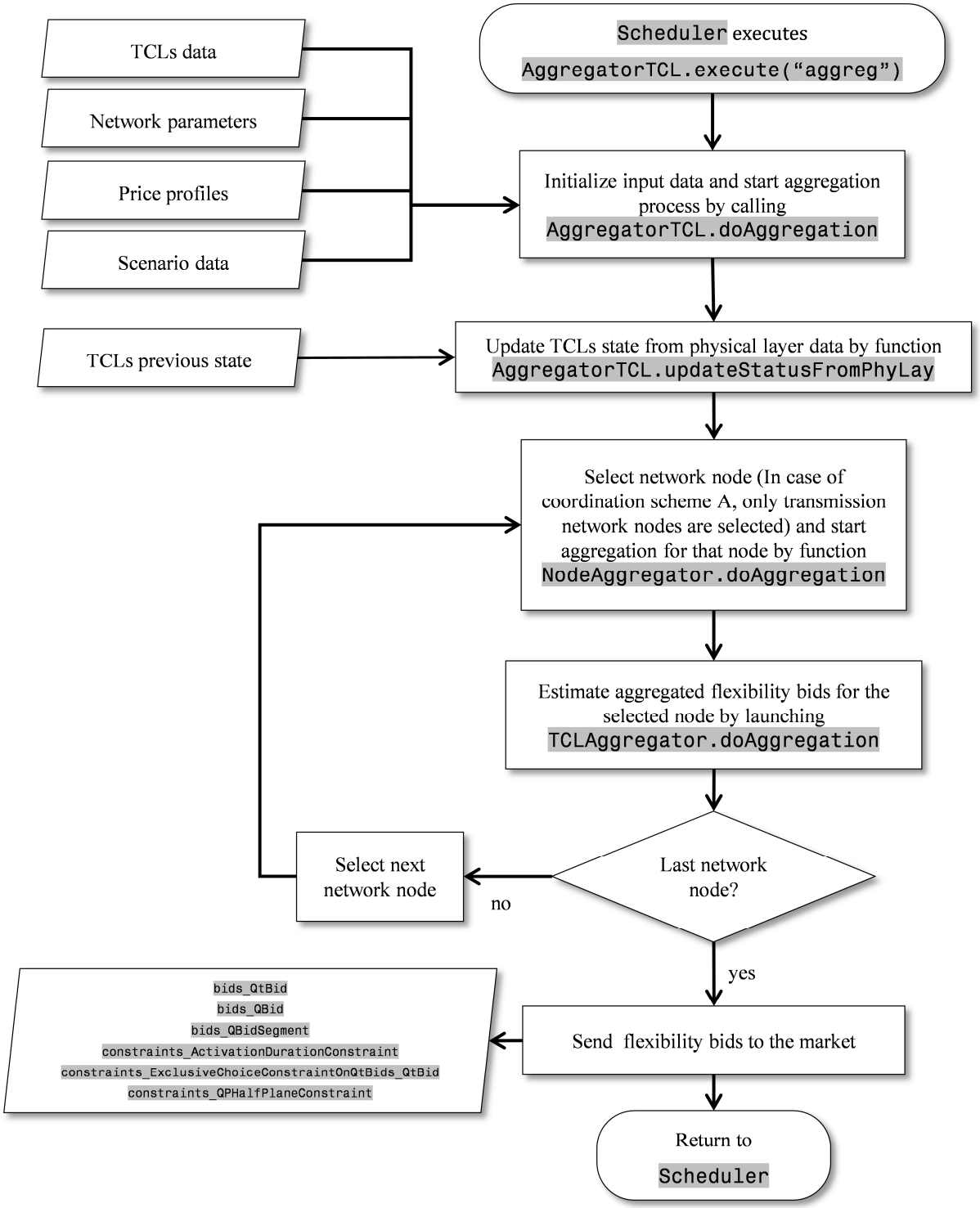


Figure 19 – Flow diagram of `AggregatorTCL.execute("aggreg")`

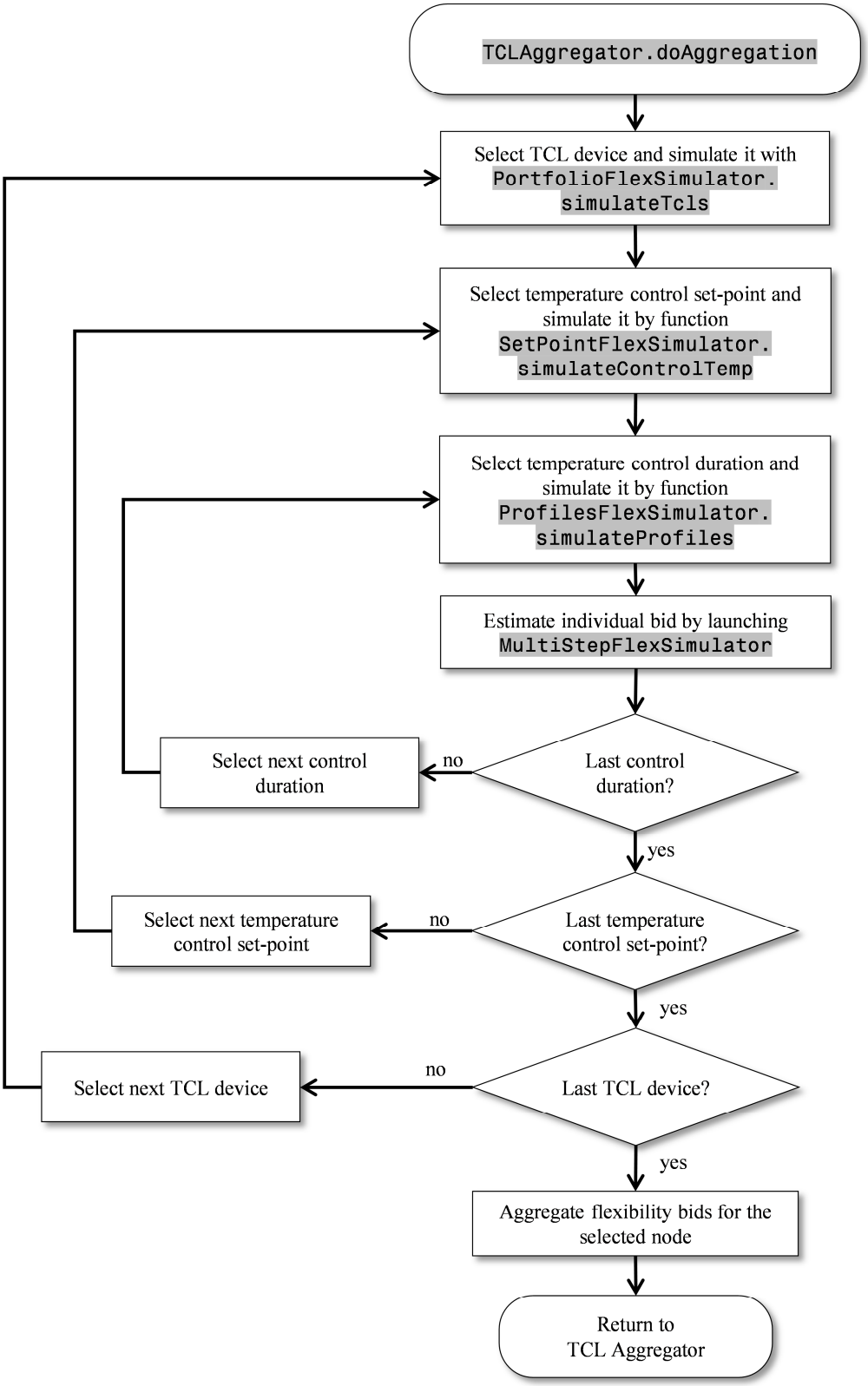


Figure 20 – Flow diagram of `TCLAggregator.doAggregation`

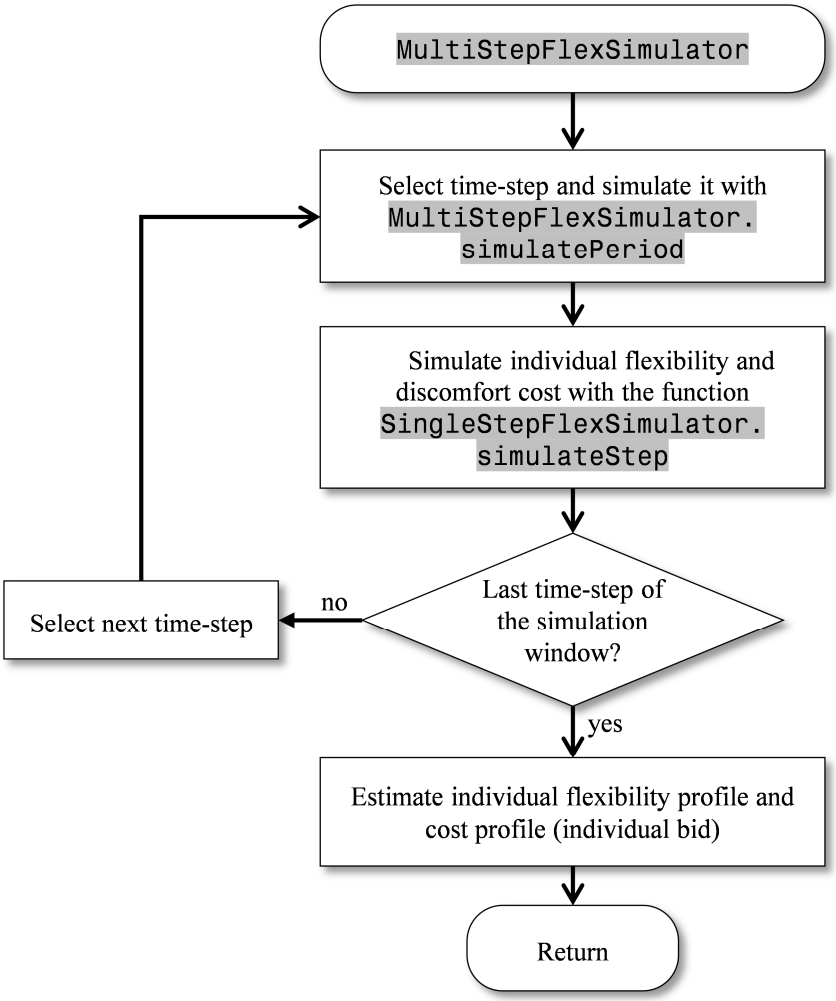


Figure 21 – Flow diagram of `MultiStepFlexSimulator`

3.2.6 Output to database

The TCL Aggregation module communicates with the other simulation blocks by writing in the database tables reported below:

TCL Aggregator internal tables

- `aggreg_FlexCalculation`
- `aggreg_BidProfile`
- `aggreg_BidCalculation`

Bids

- `bids_QBid`
- `bids_QBbidSegment`
- `bids_QtBid`

Constraints

- `constraints_ActivationDurationConstraint`
- `constraints_ExclusiveChoiceConstraintOnQtBids`
- `constraints_QPHalfPlaneConstraint`

3.3 TCL Disaggregation module

3.3.1 Brief description of the module

The objective of the TCL Disaggregation module is to translate the results of the market clearing process into control temperature set-points for the TCLs to attain the committed flexibility.

The implemented disaggregation process is straightforward since there is a direct link between the flexibility bids sent to the market and the individual control actions applied to the TCLs used to obtain them. So, when the market clearing results are known, the TCL Disaggregation module maps the accepted bids with the individual control set-points and sends them to the Physical Layer module.

3.3.2 Input from database

The inputs from the database required by the TCL Disaggregation module are listed next:

Market clearing

- `clearing_QBidSegmentVariables`
- `clearing_QBidVariables`
- `clearing_QtBidVariables`
- `clearing_SolveResult`

Bids

- `bids_QBid`
- `bids_QBidSegment`
- `bids_QtBbid`

TCL Aggregator internal tables

- `aggreg_FlexCalculation`
- `aggreg_BidProfile`
- `aggreg_BidCalculation`

3.3.3 Input from other modules

As for the aggregation module, the only direct input is from the `Scheduler` that provides the following parameters each time the TCL Disaggregation module is called:

- Aggregation mode (`"disagg"`)
- Scenario identifier
- Starting time-step of the simulation window
- Market latency
- Market horizon

3.3.4 List of functions of the module

The main functions of the TCL Disaggregation module are described below. In a similar way as the TCL Aggregation module, it has been implemented following an object-oriented programming model in which each main function has been implemented in a different Class, and each Class in a different python module (`*.py`).

- `Aggregator.py` – Class `AggregatorTCL`
  It implements the interface between the Scheduler and the TCL Aggregator/Disaggregator module.
  - `AggregatorTCL.doDisaggregation`: starts the disaggregation process
- `bid_disaggregator.py` – Class `BidDisaggregator`
  It contains the main entry point for simulating the disaggregation processes.
  - `BidDisaggregator.doDisaggregation`: performs the disaggregation based on the clearing results provided by the market module. As output, the temperature set-points of the TCLs for the Physical Layer module are generated.

3.3.5 Flow chart of the module

The general structure of the TCL Disaggregation module is described in the flowchart of Figure 22. The algorithm takes as inputs the clearing results of the market as well as information about the flexibility bids previously sent to the market and the simulation results stored in the internal database tables of the Aggregator. Afterwards, it carries out a straightforward disaggregation process to determine the set-points to be sent to the TCLs to attain the committed flexibility.

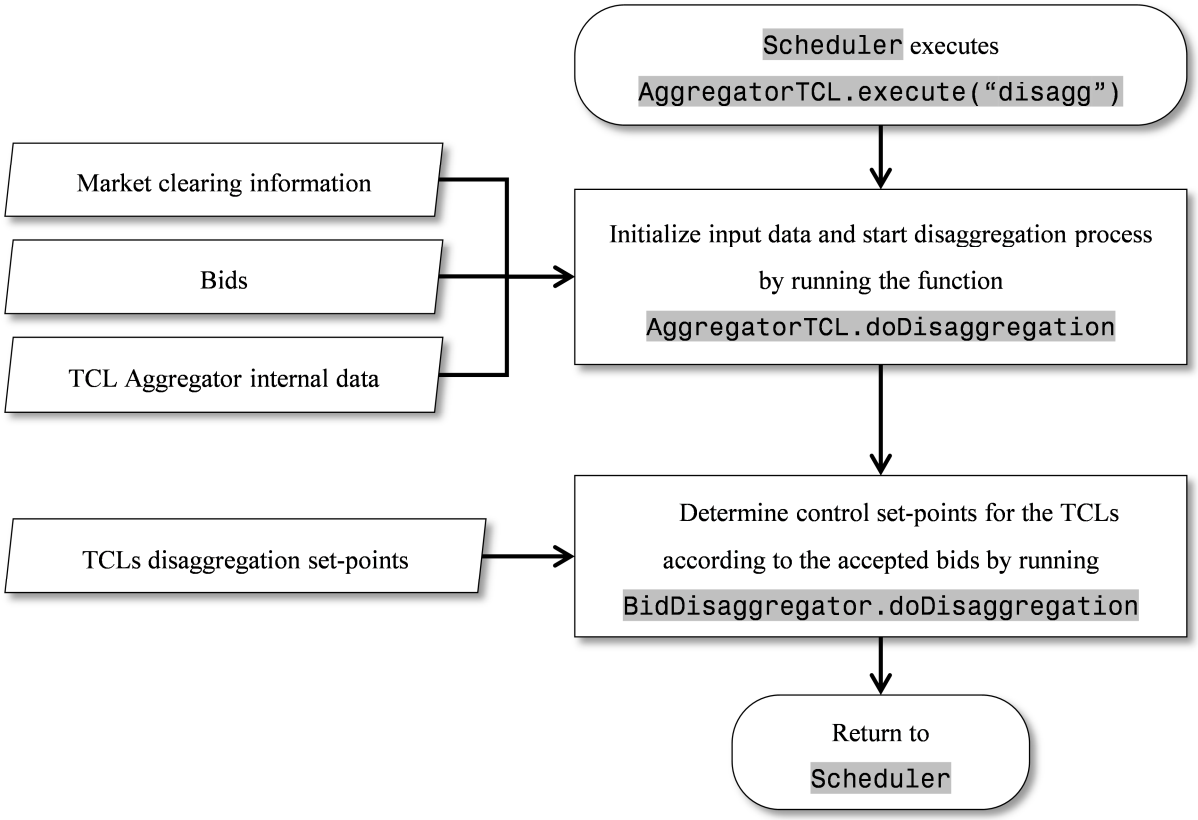


Figure 22 – Flow diagram of `AggregatorTCL.execute("disagg")`

3.3.6 Output to database

Outputs from the TCL Disaggregator module consist of individual control temperature set-points for each TCL that are written in the next table of the database:

Disaggregator set points

- `disaggregator_setpoints_TclAggOut`

3.4 Conventional Generators Aggregation module

3.4.1 Brief description of the module

The objective of this module consists of defining flexibility bids for the conventional generators. A simplified model is implemented assuming that the flexibility of each generator corresponds to the difference between the maximum/minimum power and the baseline. According to this a bid is created for each time step (a single `Qtbid` composed by multiple `Qbids`) containing two segments: one for positive flexibility (baseline production to maximum capacity) and the other for negative flexibility (baseline production to technical minimum). Ramp constraints (`RampConstraints`) and reactive power capability (`QPDiscConstraint`) are also forwarded to the market module.

3.4.2 Input from database

The inputs from the database required by the Conventional Generators Aggregation module are listed next:

Device Constants:

- `device_ConConstants`

Device Profiles

- `profiles_ConPowerProfile`
- `profiles_ConPower`

Price profiles

- `profiles_NodeDeltaCost`
- `profiles_NodeDeltaCostProfile`
- `profiles_NodeHasNodeDeltaCostProfile`
- `profiles_NodePrice`
- `profiles_NodePriceProfile`
- `profiles_NodeHasNodePriceProfile`

Network parameters

- `network_Node`
- `network_SubNetwork`

- `network_SubNetworkType`
- `network_Network`

Device variables

- `devices_ConVariables`

Scenario data

- `scenario_Scenario`

3.4.3 Input from other modules

The only direct input is from the Scheduler that provides the following parameters each time the Conventional Generators Aggregation module is called::

- Aggregation mode (`"aggreg"`)
- Scenario identifier
- Starting time-step of the simulation window
- Market horizon
- Market latency

3.4.4 List of functions of the module

The main functions of the Conventional Generators Aggregation module are described below:

- `Aggregator.py` – Class `AggregatorCONV`

  It implements the interface between the Scheduler and the Conventional Generators Aggregation/Disaggregation module. It initializes all the required data for the simulation (scenario, coordination scheme, simulation window, list of nodes, etc.) and calculates the flexibility bids.
  - `AggregatorCONV.execute("aggreg")`: contains the main entry point for simulating the aggregation process.
  - `AggregatorCONV.aggreg_bid`: creates the flexibility bids of the conventional generators

3.4.5 Flow chart of the module

The general structure of the Conventional Generators Aggregation module is described in the flowchart of Figure 23. The algorithm takes as inputs scenario and simulation information provided by

the Scheduler, together with detailed data of the conventional generators including constants, variables and profiles. Network information is also required to know the generators connection nodes. The results consist of a set of flexibility bids per node that are stored in the bids and constraints tables.

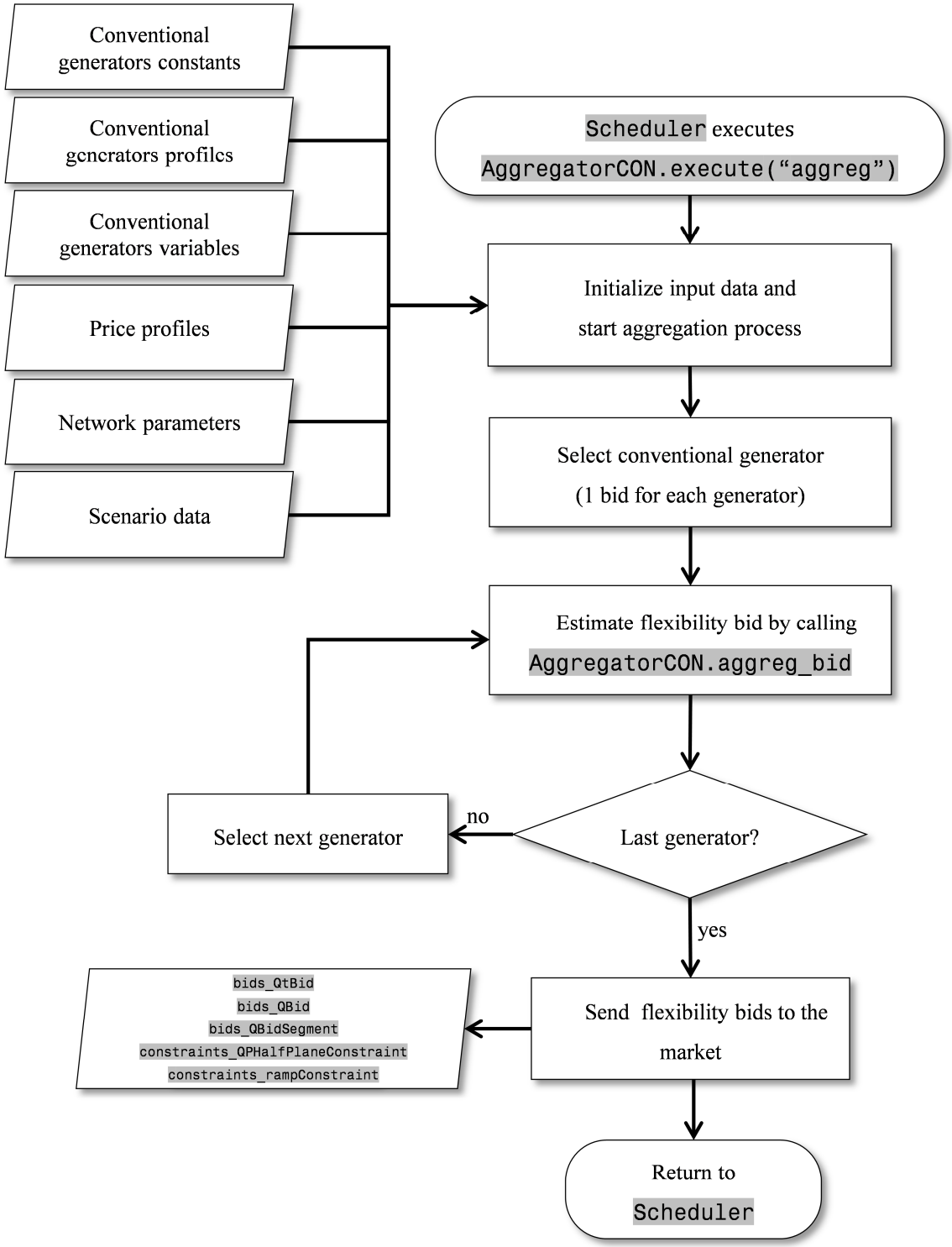


Figure 23 – Flow diagram of `AggregatorCON.execute("aggreg")`

3.4.6 Output to database

Outputs of the Conventional Generators Aggregation module are written in the following database tables:

Bids

- `bids_QBid`
- `bids_QBidSegment`
- `bids_QtBid`

Constraints

- `constraints_QPDiscConstraint`
- `constraints_RampConstraint`

3.5 Conventional Generators Disaggregation module

3.5.1 Brief description of the module

The objective of the Conventional Generators Disaggregation module is to translate the results of the market clearing process into power set-points for the conventional generators to attain the committed flexibility. The implemented disaggregation process is straightforward as each bid sent to the market is directly accepted or rejected.

3.5.2 Input from database

The inputs from the database required by the Conventional Generators Disaggregation module are listed as follows:

Market clearing

- `clearing_QBidSegmentVariables`
- `clearing_QBbidVariables`
- `clearing_QtBidVariables`
- `clearing_SolveResult`

Bids

- `bids_QBid`
- `bids_QBidSegment`
- `bids_QtBid`

3.5.3 Input from other modules

The disaggregation module takes as additional inputs the following parameters, which are provided by the `Scheduler` routine:

- Aggregation mode (`"disagg"`)
- Scenario identifier
- Starting time-step of the simulation window
- Market latency
- Market horizon

3.5.4 List of functions of the module

The main functions implemented in the Conventional Generators Disaggregation module are described below:

- `Aggregator.py` – Class `AggregatorCONV`
  It implements the interface between the Scheduler and the Conventional Generators Aggregator/Disaggregation module.
  - `AggregatorCONV.execute("disagg")`
    It contains the main entry point for simulating the disaggregation process.
  - `AggregatorCONV.disaggreg_bid`
    It performs disaggregation based on the clearing results provided by the market module. As output, power set-points for the conventional generators for the physical layer module are generated.

3.5.5 Flow chart of the module

The general structure of the Conventional Generators Disaggregator module is described in the flowchart of Figure 24. The algorithm takes as inputs the clearing results of the market as well as information about the flexibility bids previously sent to the market and carries out a straightforward disaggregation process to determine the set-points to be sent to the units to attain the committed flexibility.

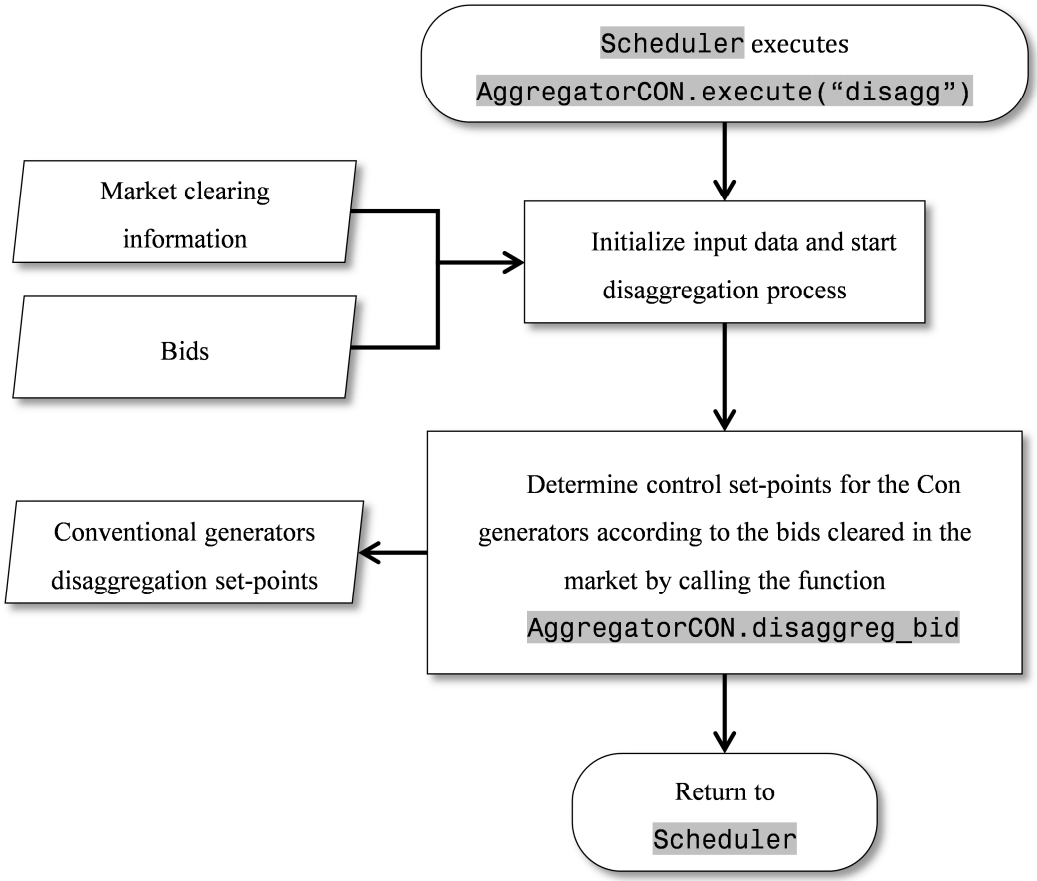


Figure 24 – Flow diagram of `AggregatorCON.execute("disagg")`

3.5.6 Output to database

Outputs from the Conventional Generators Disaggregation module consist of individual power set-points attributed to each conventional generator. These set-points are stored within the following database table:

Disaggregator set points

- `disaggregator_setpoints_ConAggOut`

3.6 CHP Aggregation module

3.6.1 Brief description of the module

The objective of this module consists of generating flexibility bids for the Combined Heat and Power (CHP) units. A simplified algorithm is implemented: it assumes that the flexibility that can be provided by each unit corresponds to a portion (defined by a power availability factor) of the power bandwidth between the maximum/minimum power and the baseline. The availability factor represents a fraction of the maximum flexibility which limits the possible baseline deviations in order to do not significantly deviate from the nominal thermal demand.

According to this a bid is created for each time step (a single `Qtbid` composed by multiple `Qbids`) containing two segments: one for upward power flexibility and the other for downward negative flexibility. The reactive power capability (`QPDiscConstraint`) is also forwarded to the market module.

3.6.2 Input from database

The inputs from the database required by the CHP Aggregation module are listed next:

Device Constants:

- `device_ChpConstants`

Profiles

- `profiles_ChpPowerProfile`
- `profiles_ChpPower`
- `profiles_NodeDeltaCost`
- `profiles_NodeDeltaCostProfile`
- `profiles_NodeHasNodeDeltaCostProfile`
- `profiles_NodePrice`
- `profiles_NodePriceProfile`
- `profiles_NodeHasNodePriceProfile`

Network parameters

- `network_Node`
- `network_SubNetwork`
- `network_SubNetworkType`
- `network_Network`

Device and Network Variables

- `devices_ChpVariables`

Scenario data

- `scenario_Scenario`

3.6.3 Input from other modules

The only direct input is from the `Scheduler` that provides the following parameters each time the CHP Aggregation module is called:

- Aggregation mode (`"aggreg"`)
- Scenario identifier
- Starting time-step of the simulation window
- Market latency
- Market horizon

3.6.4 List of functions of the module

The main functions of the CHP Aggregation module are described below:

- `Aggregator.py` – Class `AggregatorCHP`
  It implements the interface between the Scheduler and the CHP Aggregation/Disaggregation module. Initializes all the required data for the simulation (scenario, coordination scheme, simulation window, list of nodes, etc.) and calculates the flexibility bids.
  - `AggregatorCHP.execute("aggreg")`: contains the main entry point for simulating the aggregation process.
  - `AggregatorCHP.aggreg_bid`: creates the flexibility bids of the CHPs

3.6.5 Flow chart of the module

The general structure of the CHP Aggregation module is described in the flowchart of Figure 25. The algorithm takes as inputs scenario and simulation information provided by the Scheduler, together with detailed data of the CHPs including constants, variables and profiles. Network information is also required to know the connection nodes of the CHPs. The results consist of a set of flexibility bids per node that are stored in the bids and constraints tables of the market.

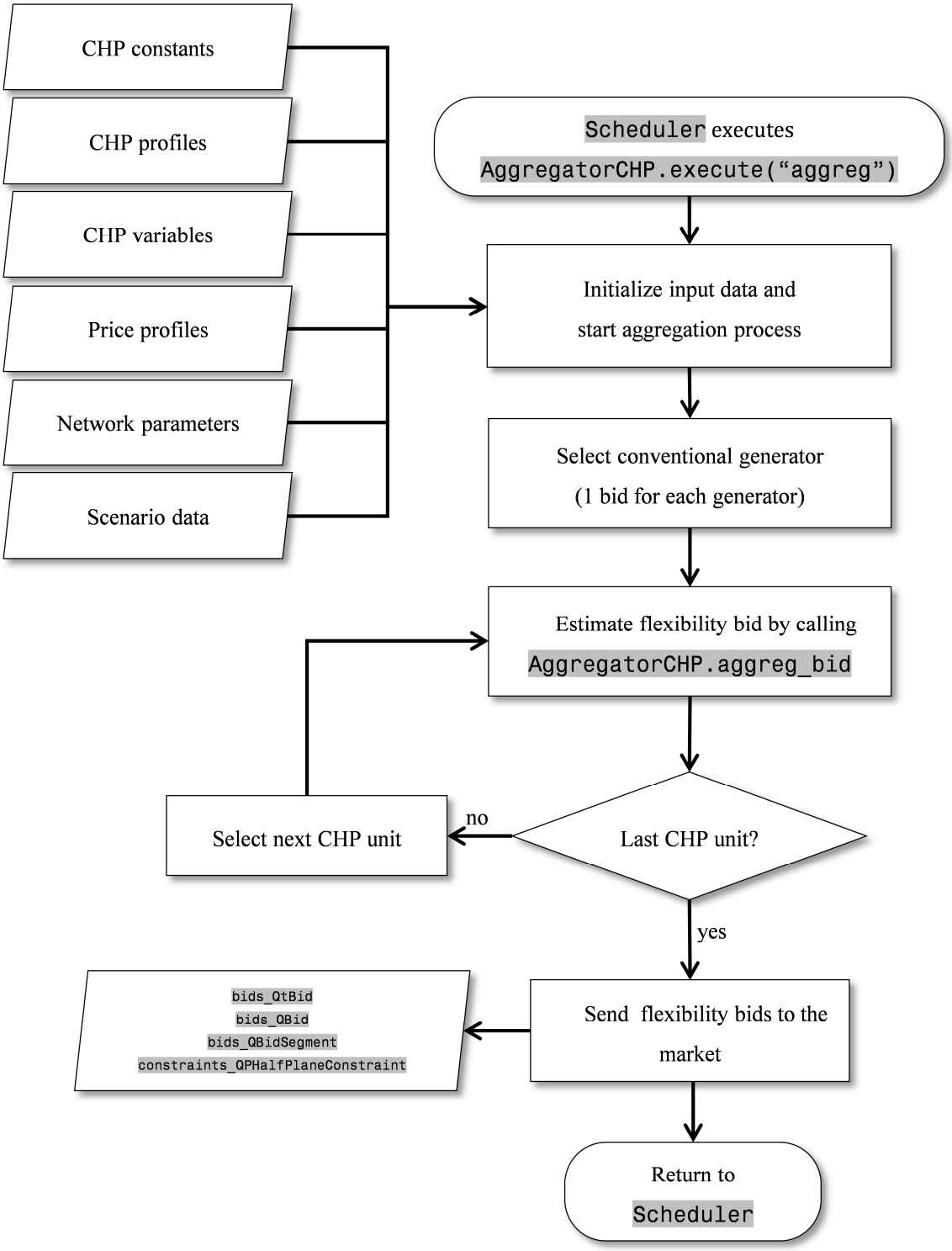


Figure 25 – Flow diagram of `AggregatorCHP.execute("aggreg")`

3.6.6 Output to database

Outputs of the CHP Aggregation module are stored within the following tables of the database:

Bids

- `bids_QBid`
- `bids_QBidSegment`
- `bids_QtBbid`

Constraints

- `constraints_QPDiscConstraint`

3.7 CHP Disaggregation module

3.7.1 Brief description of the module

The objective of the CHP Disaggregation module consists of translating the results of the market clearing process into power set-points for each CHP units to attain the committed flexibility. The implemented disaggregation process is straightforward as each bid sent to the market is directly accepted or rejected and does not need extra processing.

3.7.2 Input from database

The inputs from the database required by the CHP Disaggregation module can be listed as follows:

Market clearing

- `clearing_QBidSegmentVariables`
- `clearing_QBidVariables`
- `clearing_QtBidVariables`
- `clearing_SolveResult`

Bids

- `bids_QBid`
- `bids_QBidSegment`
- `bids_QtBbid`

3.7.3 Input from other modules

The only direct input is from the `Scheduler` that provides the following parameters each time the CHP Disaggregation module is called:

- Aggregation mode(`"disagg"`)
- Scenario identifier
- Starting time-step of the simulation window
- Market latency
- Market horizon

3.7.4 List of functions of the module

The main functions implemented in the CHP Disaggregation module are described below:

- `Aggregator.py` – Class `AggregatorCHP`

  It implements the interface between the Scheduler and the CHP Aggregation/Disaggregation module.

  - `AggregatorCHP.execute("disagg")`: contains the main entry point for simulating the disaggregation process.
  - `AggregatorCHP.disaggreg_bid`: performs disaggregation based on the clearing results provided by the market module. As output, power set-points of each CHP (to be processed by the physical layer module) are generated.

3.7.5 Flow chart of the module

The general structure of the CHP Disaggregator module is described in the flowchart of Figure 26. The algorithm takes as inputs the clearing results of the market as well as information about the flexibility bids previously sent to the market and carries out a straightforward disaggregation process to determine the set-points to be sent to the CHPs to attain the committed flexibility.

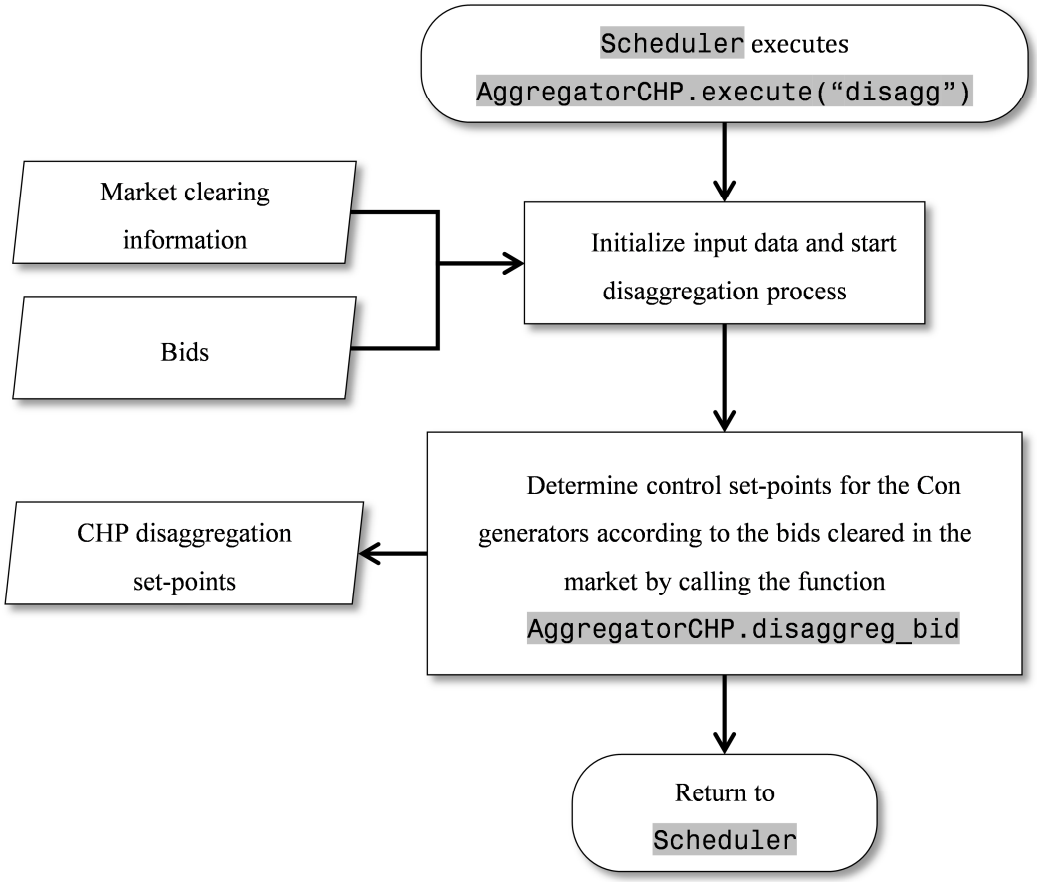


Figure 26 – Flow diagram of `AggregatorCHP.execute("disagg")`

3.7.6 Output to database

Outputs from the CHP Disaggregation module consist of individual power set-points for each CHP unit, which are stored within the following table of the database:

Disaggregator set points

- `disaggregator_setpoints_ChpAggOut`

3.8 Curtailable generation and curtailable load Aggregration/Disaggregation module

3.8.1 Brief description of the module

The Curtailable Generation Curtailable Load (CGCL) aggregation/disaggregation module simulates the aggregation/disaggregation of data and bids from hundreds of thousands of devices attached to a particular node/ or a set of nodes[1] on a physical power network. In this case, four different types of devices are collated:

- Run-of-the-river hydroelectric
- Photovoltaic – PV (Solar)
- Wind
- Sheddable Loads (SEL) – e.g. Street Lamps

For each time step, bids from these different types of devices are combined into price *buckets* to produce up to 20 price volume bids per time step[2] (10 up bids and 10 down bids). Market rules determine when aggregators will bid i.e. the time step and for how many future intervals (e.g. 8 bids of 15 minutes for the next hour).

Overall Control of the bidding process, including time steps, the number of periods bid, aggregation and disaggregation start signals is driven by the Market Scheduler and scenario inputs. This data is sent to a CGCL aggregator by the `Scheduler`.

The CGCL aggregator code initialises and creates all the aggregators for the scenario, and collects all the device data associated with each aggregator. In effect, each aggregator is an agent (a software object), who stores the data from all the devices connected to the aggregator's node or nodes, in an in-memory[3] three dimensional matrix.

An agent based object orientated design was used to construct the CGCL aggregator using a buckets or tranche system to aggregate bids from up to 300,000 devices of four different types (PV Solar, Hydro Wind and Sheddable Loads) across 10÷20,000 power nodes. In the current version, buckets are clustered by cost, but the concept can be extended to a more general clustering algorithm.

The CGCL aggregator code makes extensive use of Pythons `Numpy` Array data manipulation routines for calculation speed, which is ideally suited to n dimensional matrix manipulation

[1] Under certain TSO-DSO coordination schemes the aggregator my collate data from several nodes
[2] Parameter driven. User can change.
[3] For speed – RAM is thousands of times faster than SSD

The CGCL aggregator code is split into two main parts:

1. A CGCL Aggregator Factory (Figure 27)
   This creates the appropriate number of aggregator agent objects, creates a list (a python dictionary – an agent Directory) of those objects , so that we can take control of them later and populates them with data from a relational database. On receipt of an aggregation or disaggregation signal from the scheduler module it also triggers the individual agents to perform their calculations. Currently this is performed sequentially, but this could be potentially multi-tasked.
2. An CGCL Implementation module
   It contains the logic of the individual aggregator agents.

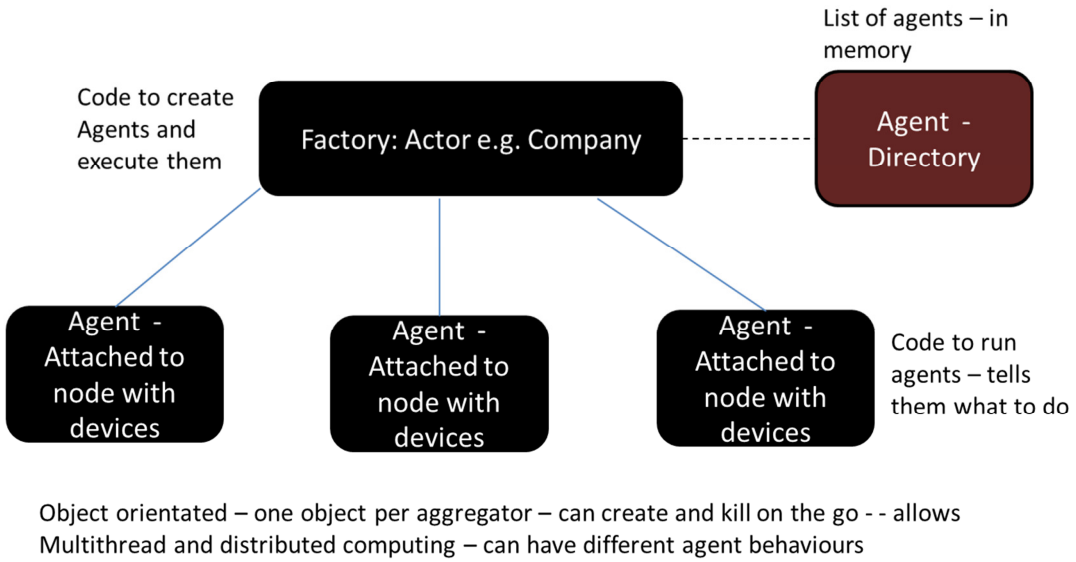


Figure 27 – Overall approach and design of CGCL aggregation

3.8.1.1 CGCL Aggregator Factory

The aggregator factory module creates an actor object that is used to create aggregator agents at nodes. A list (or an agent directory) of the agents is created and is used to cycle through each of the agents. The code in this module is essentially split into four and orchestrates the actions of CGCL aggregator agents, each at a different node. This module or python package carries out the following steps:

1. Initialisation of data – Pulling profile data once from the database at the very beginning of the simulation and storing such data in `Numpy` matrices and lists. This data is transferred to Agents so that they can create specific data (eg specific device profiles). This data is not specific to any particular agent – e.g. Standardized Profiles and types
2. Creates and triggers the initialisation of aggregator agents. A dictionary of agent objects is made, so that the code can address different agents at a later date. Control logic within this module checks to see what TSO-DSO Coordination Scheme is being used. In the case of Coordination Scheme A, aggregators are assigned to transmission nodes only, but a list/dictionary of the distribution nodes associated with the Transmission node is made and passed to the aggregator agent. So that it can collate all devices associated with the distribution nodes connected to the transmission node. In the case of all other coordination schemes, aggregators are placed at each and every node (Transmission and Distribution).
3. Sends an Executes aggregation signal to each of the aggregator agents – this triggers the agents to aggregate and send bids to a global list. At the end of this process the module bulk writes the bid data contained in this list. This is much quicker than having each individual agent do this.
4. Sends an Executes disaggregation signal to each agents – this triggers the agents to disaggregate cleared bids

Signalling is essentially controlled by an external scheduler. The overall flow and interactions of the CGCL aggregator with other modules is shown in Figure 28. Note black blocks in the diagram are modules that are provided by others simulation layers/blocks (e.g. Market Clearing or Scheduler modules). Dark brown blocks are associated with the CGCL Aggregator Factory module and the other colours are associated with the CGCL implementation module.

3.8.1.2 CGCL Aggregator Implementation Module (Agent Logic)

The aggregator implementation module, contains the logic of the aggregator agents, and performs four main tasks:

1. Initialisation of agent Object – When an agent object is created this simple function creates internal storage of variables and sets up certain parameters
2. Initialisation of Agent data – This function pulls device specific data from the database and creates device profiles for the devices attached to the aggregator.
3. Aggregation – Creates bids for the aggregator and sends these bids to a database, so that the market layers can clear the market.

4. Disaggregation – Recovers cleared bid data associated with the specific aggregator agent and disaggregates cleared bids. This results in an agent sending new set-points to all of the devices on the particular node. These set-points are stored in a dedicated table.

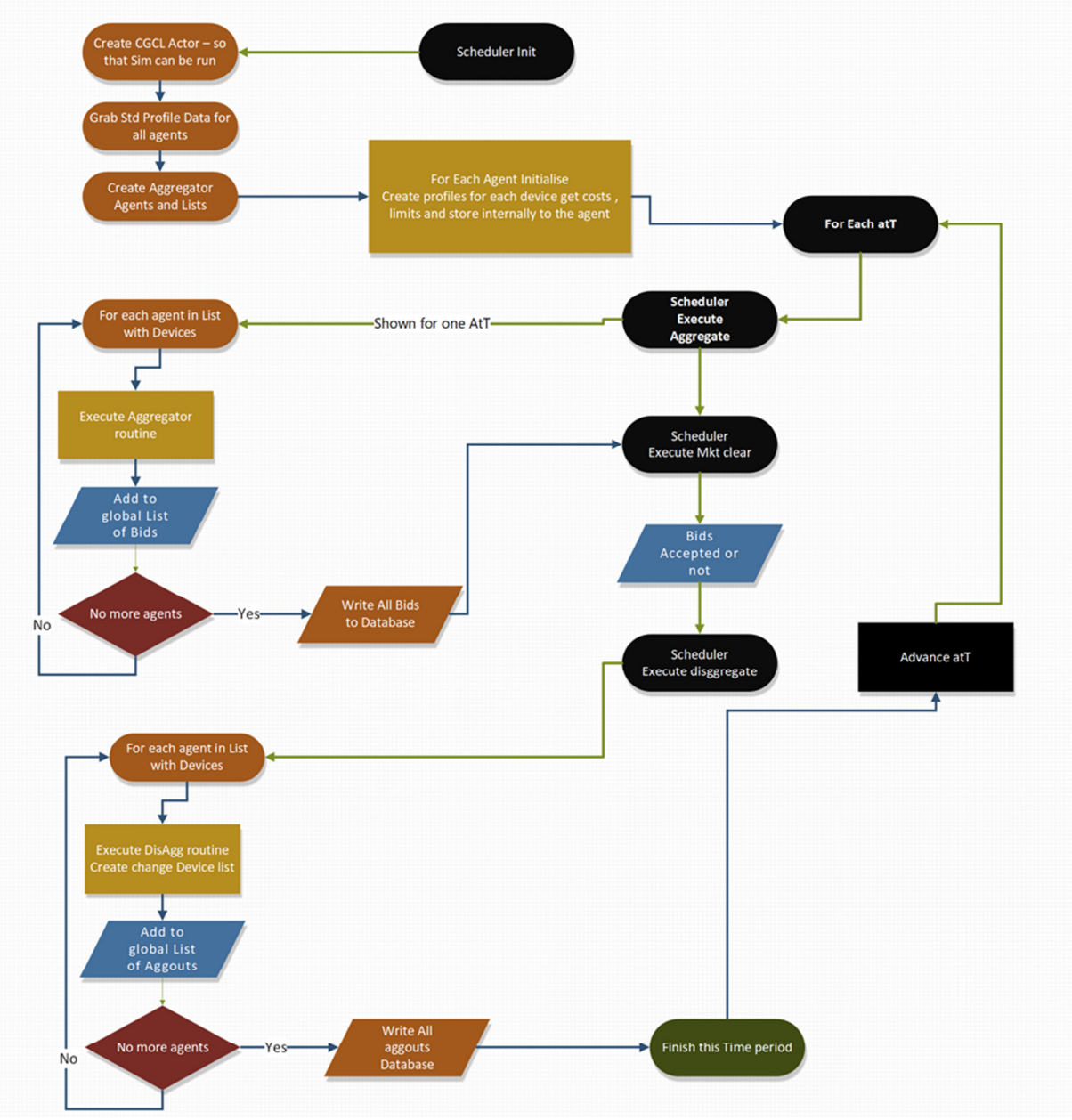


Figure 28 – Overall flow and interactions of the CGCL aggregator with other modules

3.8.2 Input from database

The inputs to the CGCL Aggregator modules can be divided into a number of groups. Inputs also come from four types of devices namely Hydro, PV (Solar) Wind and Sheddable Loads (SEL), although this could be extended. The CGCL Factory which creates the agents first, captures data that is generic to all devices. It does this only once at the beginning of the simulation for speed and passes this data in the form of lists and `Numpy` arrays to other parts of the code. The CGCL implementation code module takes inputs from the database that are agent specific like device data and constants and uses data passed to it in arrays from the factory module. Specific inputs from the SmartNet database are as follows:

Scenario:

- `Scenario_Scenario`: provides data on scenario including name , number of time steps to be simulated

Scheduler:

- `Scheduler_Scheduler`: provides data on latency and for what time horzon - eg Aggreagtor to be 12 periods forward form time t. Sent indirectly via scheduler module

Device Constants:

- `device_HydConstants`: Hydroelectric generators parameters e.g. marginal cost, power capabilitym, etc.
- `device_PvConstants`: Photovoltaic generators parameters
- `device_SelConstants`: Sheddable loads parameters
- `device_WindConstants`: Wind generators parameters

Device Profiles

- Hydro generators
    - `profiles_HydPowerProfile`: contains the connection between the related profile and the constants table.
    - `profiles_HydPower`: power production forecast profiles – represents actual profiles
    - `profiles_HydPowerBaselineProfile`: contains the connection between the related profile and the constants table.

    - `profiles_HydPowerBaseline`: scheduled power production profiles from previous (e.g. intraday) market results.
- PV generators
    - `profiles_PvPowerProfile`: contains the connection between the related profile and the constants table.
    - `profiles_PvPower`: power production forecast profiles
    - `profiles_PvPowerBaselineProfile`: contains the connection between the related profile and the constants table.
    - `profiles_PvPowerBaseline`: scheduled power production profiles from previous (e.g. intraday) market results
- Sheddable loads
    - `profiles_SelPowerProfile`: contains the connection between the related profile and the constants table.
    - `profiles_SelPower`: power absorption forecast profiles.
- Wind generators
    - `profiles_WindPowerProfile`: contains the connection between the related profile and the constants table.
    - `profiles_WindPower`: power production forecast profiles.
    - `profiles_WindPowerBaselineProfile`: contains the connection between the related profile and the constants table.
    - `profiles_WindPowerBaseline`: scheduled power production profiles from previous (e.g. intraday) market results.

Network parameters

- `network_Node`: contains general information of each node (belonging sub-network…)
- `network_Network`: is the main table, contain the name of the network.
- `network_SubNetwork`: contain the list of sub-networks (the transmission networks, and the distribution networks) in which the networks is divided.
- `network_SubNetworkType`: type of sub-networks (transmission or distribution)

Bids (Market bids)

- `bids_QtBid`: main record for bid at time
- `bids_Qbid`: represents a bid for a forward time horizon. Linked to `bids_QtBid` record via foreign key. Also an output.

- `bids_QBidSegment`: within each time step (i.e. `bids_Qbid`) , aggregators can bid price and volume segments (both for up and down flexibility)also an output (e.g. bid volume at a price P[4]

Clearing (Market bids cleared)

- `Clearing_QBidSegmentVariables`: cleared bids containing accepted fractions and link to `bids_Qbidsegment` records

3.8.3 Input from other modules

3.8.3.1 Inputs for CGCL Aggregator Factory

The only direct input is from the `scheduler` which provides calls on methods that initialise aggregators. The `scheduler`

- Sends an initialisation signal to set up the Factory Actor
- Starts the aggregation logic when required (via execution function in mode `"aggreg"`)
- starts the disaggregation logic when required (via execution function in mode `"disagg"`)

3.8.3.2 Inputs for CGCL aggregator implementation

- Generic data such as base and actual profile types – sent for all types of devices (Hydro, PV, Wind and SEL), passed to the agent in a call as a tuple of lists from agent factory
- Cleared Bids
  - `clearing_QbidSegmentsVariables`: provides cleared bids sent by the aggregator to the market. This data input provides details on how much the market cleared as a fraction. Note the CGCL aggregator will accept partial bids.
- Control signals from Scheduler via CGCL Factory
  - Mode `"aggreg"`: Starts aggregation process which takes device data and marginal costs and clusters data into price buckets or segments so that it can be bid to the market via a database write.
  - Mode `"disagg"`: Starts the disaggregation process.

[4] Note that the aggregator bids P0 and P1 but [2] defined that P0=P1 for this type of aggregator

During the execution process, data such as the scenario, the time step `atT`, the market latency `atToForLatency` and the number of periods to be bid `forHorizon` are sent to the agent implementation during this control signal process.

3.8.4 Flow chart of the module

As described above, the CGCL structure is divided in two main modules (Python packages), called **CGCL Factory** and **CGCL Implementation**. In the sections that follow we provide descriptions of the various functions in the two main modules or packages, and present flowcharts of the logic inside these various functions.

3.8.4.1 CGCL Aggregator Factory Module

The aggregator factory class or module consist of three main methods. These are

- `AggregatorCGCL.initialize` (Figure 29 and Figure 30)
  - It prepares all the data needed for the whole simulation period and stores profile data used by all device types in this aggregator. Also includes data for the scenario, e.g. market horizons, latency, etc. It is launched once by scheduler main system.
  - Initialise callas the function `fill_in_profile_data`
- `AggregatorCGCL.execute(modestring)` (Figure 31 and Figure 32)
  - Routine that is called from the scheduler. It passes a `modestring` which tells the routine whether to aggregate (`“aggreg”`) or disaggregate (`“disagg”`). It also passes scenario data, the time period horizon and latency .
  - The execution routine cycles through each agent stored in the agent directory and calls the appropriate routine inside each agent object (aggregate or disaggregate)
  - The initial design had agents writing bids variables directly to the database, resulting in millions of data writes. This proved to be slow resulting in hour long writes per time period. The latest design uses global lists to store SQL write commands. This is a much faster process as the millions of outputs can be stored in these lists and written once during the aggregation process as a bulk create.
  - As similar issue occurs with writes to the disaggregator set-points out tables.

- `AggregatorCGCL.fill_in_profile_data`
    - Is called many times by the Factory module. Data profiles are read in for many devices and converted into `Numpy` arrays for later manipulation.
    - This routine uses scenario data to create a blank array for all time periods
    - The routine checks to see if the database data in array form has the appropriate period. If it does not it copies the last known record into the empty data value slot. This array is used as the input to calculations

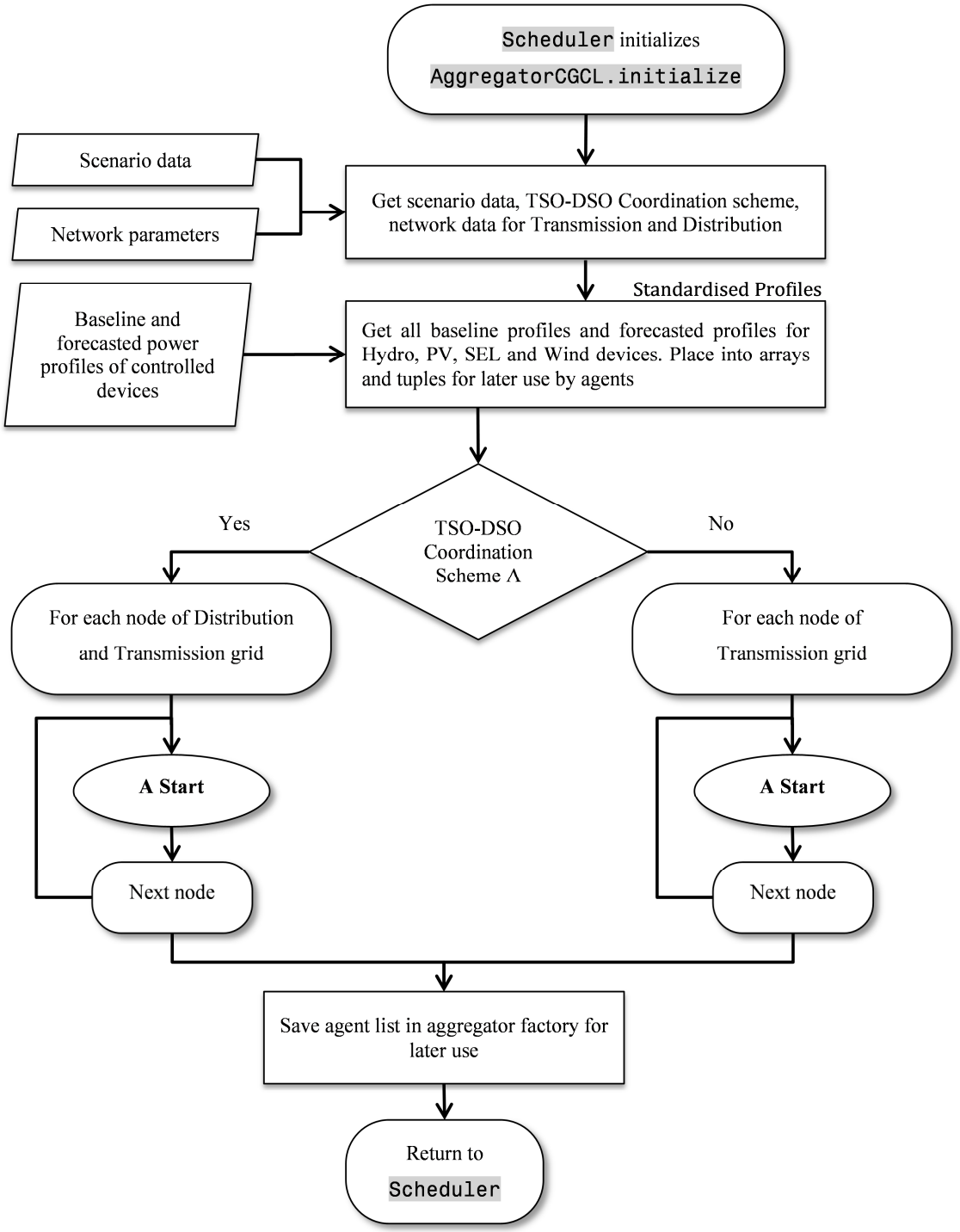


Figure 29 – Flow diagram of `AggregatorCGCL.initialize`

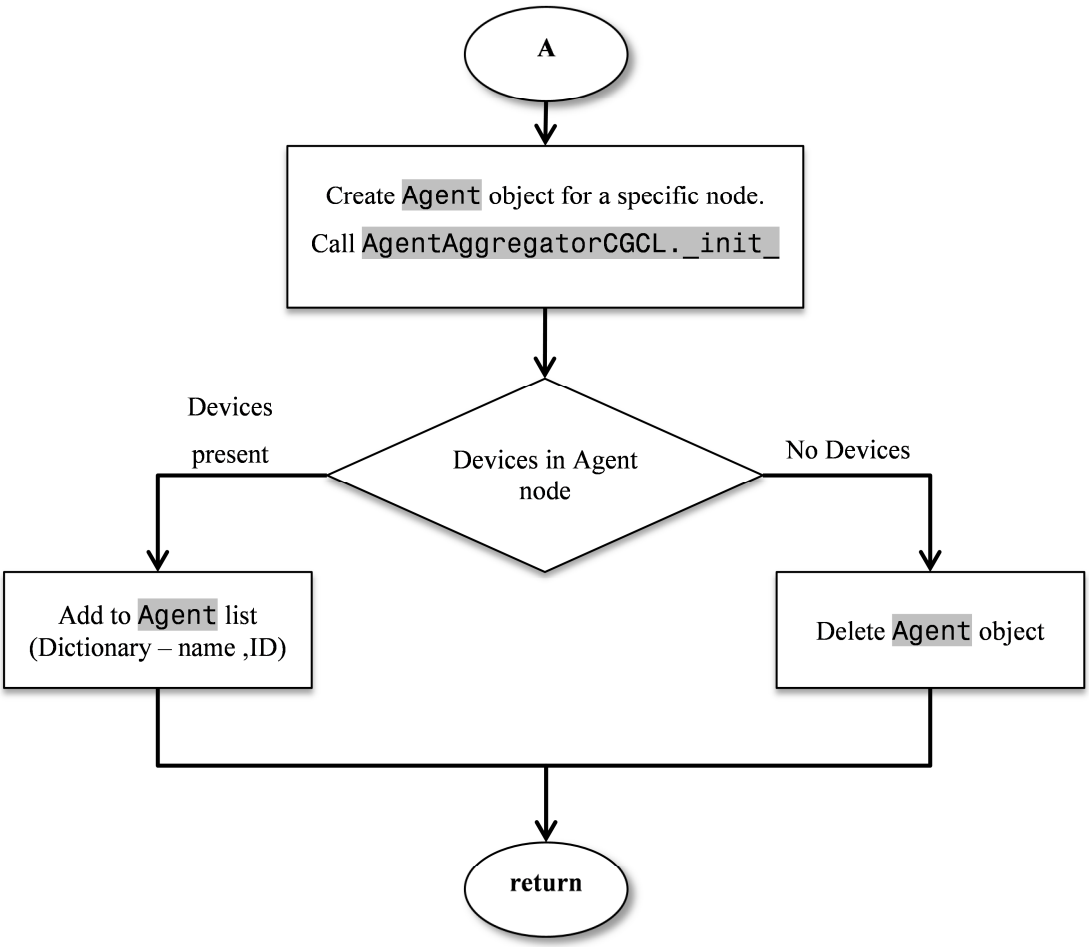


Figure 30 – Flow diagram of `AggregatorCGCL.initialize` *– creation of agents*

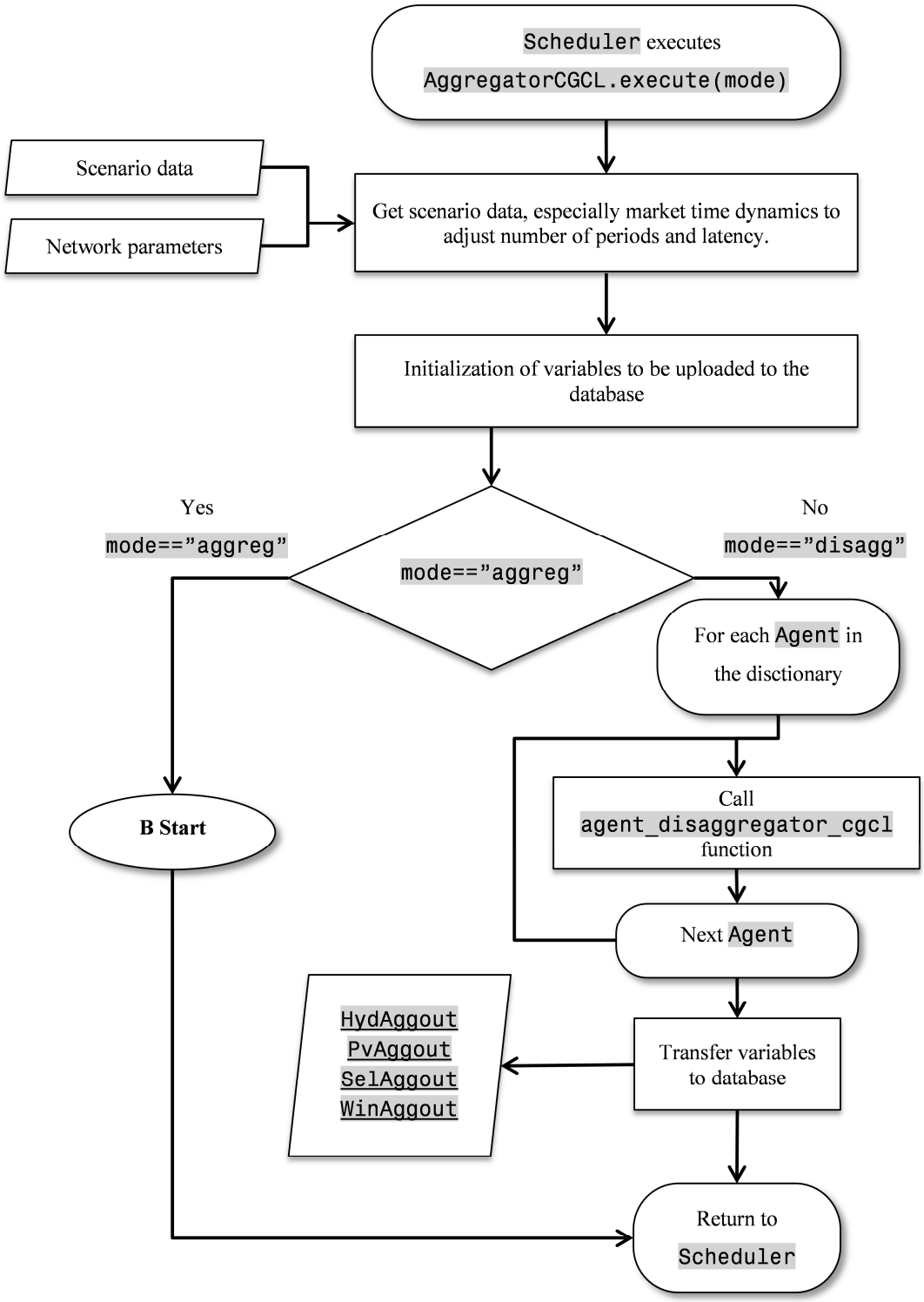


Figure 31 – Flow diagram of `AggregatorCGCL.execute`

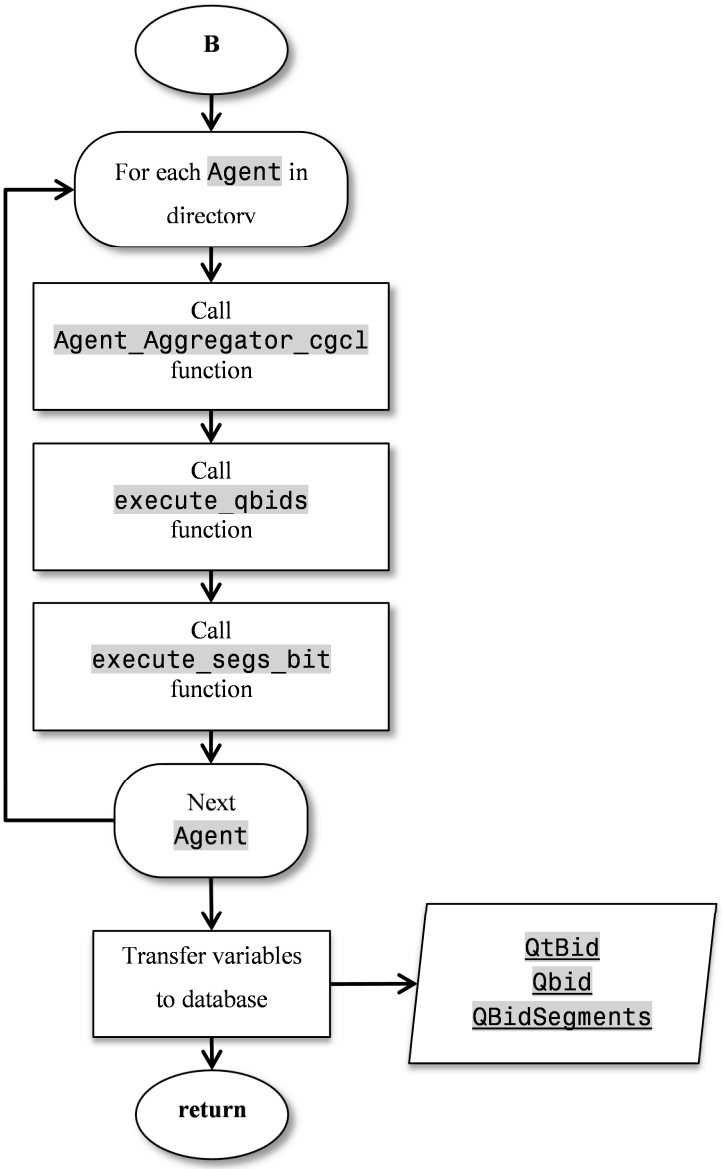


Figure 32 – Flow diagram of `AggregatorCGCL.execute` *aggregation and creation of bids*

3.8.4.2 CGCL Aggregator Implementation Module

The aggregator implementation class is more complex than the factory as it contains the underlying logic and control logic for each of the individual aggregators. We have used an agent based method to create one aggregator (if required) at the appropriate node in the network. In a large system we would have tens of thousands of aggregator agents. Each agent stores its own data such as device profiles (both day ahead and their expected or actual profiles) and performs its own calculations and stores the results of those calculations within its own memory. It takes device bids from four types of devices namely Hydro, PV (Solar) , Wind and sheddable loads (SEL), clusters that data into price buckets or segments and effectively bids this data to the market by storing those bids into the `QtBid`, `Qbid` and `QBidSegments` tables:

- `QtBid` is the main record that represents the collection of bids made at a node at a particular time step that may be for many future time periods and for many price ranges;
- Associated with the `QtBid` are the records associated with each forward period bid. E.g. at time 0 we will bid for the next hour in fifteen minute intervals and supply in this instance 4 `Qbids` for those forward time periods. This is represented by a `QBid` record, one for each forward time bid.
- Associated with each of the forward time steps is a set of `Qbidsegment` records, which can be thought of as a flexibility supply curve providing different volumes at different prices. The curve represents both upward flexibility bids and downward flexibility bids.

Each aggregator agent performs its own calculations and updates the tables as necessary. A minimum bid size of 1 kW (this is parameter driven) is provided so not all `Qbid` and `Qtbid` records may be sent. In some time periods three `Qbid` records might be sent and in others only one may be sent for a particular `Qtbid`. Logic has to deal with this irregularity. We utilise a map list that maps actual segment bids to columns in the `finalbidmatrix`, to deal with this irregularity. Bids for a particular time period are stored within the agent as a matrix `finalbidmatrix` (which has a Up and Down versions). The matrix represents the clusters/segments as well as the devices in that cluster including details on its type, volumes (active and reactive power), its bid (price) and so on. This matrix approach allows the logic in the module to unpick cleared bids and disaggregate them to individual devices associated with aggregator.

The Market is cleared by the market module which stores the cleared bids in the `QbidSegmentVaribales` table. This is linked via foreign keys to the `QbidSegment` tables stored by the aggregator routine, described above.

The scheduler triggers the disaggregation routine by sending a signal to the aggregator factory. The aggregator factory uses the agent list (python dictionary) to send a control signal (call to function) to each aggregator in turn, to tell it to disaggregate. This routine cycles through each agent and calls the disaggaregation routine within the agent aggregator.

The aggregator implementation class contains the logic of the aggregator agents and its initialisation code. It has a number of functions listed below:

- `_init_`
  - Is called when the agent object is created by the Agent Factory.
  - Takes data from the aggregator factory.
  - Creates internal storage in lists and arrays for later use.
  - Calls `initialise` function.
- `initilaise`
  - Initializes all required variables for the simulation of an aggregator agent, including the list of devices associated with the aggregator, device constants and variables and sets initial settings.
- `grab_initial_data`
  - This function/routine takes all the initial data required for the agent aggregator at each node.
  - Uses this data to create individual device profiles for each device for the whole simulation. This is done once when the agent is created to save time. Device data is created for all curtailable devices.
  - Cost data for each device is obtained and stored in `Numpy` based matrices.
  - Additional data is added to the matrices/arrays so that they can be filtered and sliced later.
- `agent_aggregate_cgcl`
  - Called by the aggregation factory module.
  - Calls aggregation logic and is used to create bids for the market.
  - Calls `generate_qt_bid` routine to write bids to a global array, so that the Factory can write them to the appropriate databases.
- `agent_disaggregate_cgcl`
  - Called by the aggregator factory in its execute routine. This factory routine cycles around all the agents and triggers the disaggregation routine inside each of them.
  - The disaggregation routine makes use of the `Qtbid`, `Qbid`, `QbidSegments` and `QbidSegmentVariables` records and internal arrays such as the `finalbid` matrix and `bidmatrix` to unpick the bids and disaggregate. The use of codes for

device types allows us to filter `Numpy` arrays to extract data for particular devices and therefore allows us to more easily write to the appropriate device output table.

- Data is sent to a routine called `disaggregate_cgcl_core` that starts to perform the `aggout` part of the code (consisting of writing data to the aggregator set-point tables). The routine cycles through all the devices on the node (or nodes) attributed to this aggregator and checks to see whether this device is present in memory. If it is not the device is left at its set-point (i.e. the baseline or day-ahead profile value for this time period), else it is changed.
- A record is written to the `disaggreagtor_setpoint_out` table (there is one table for each type of device technology). Records are therefore appended to global lists and written to the database in one go. This is only done after all aggregator agents have updated all of their records related to the selected time step.

- `disaggregate_cgcl_core`
  - It creates a table aimed at storing the reference to the device to which the set-points should be changed and by how much. Changes can be upward and downward.
  - It retrieves the cleared data from the `clearing_QbidSegmentVariables` table and uses the previously internally stored table to unpick the bids to apportion them to individual devices. Where the market clears or accepts the full volume of a particularly bucket or segment, the module logic accepts all bids from all the devices assigned to this particular bucket. In the case where the market accepts a fraction of the bid segment, we must apportion volumes to individual devices.
- `generate_qt_bids`
  - Called by `agent_aggregate_cgcl` function (described above). Takes data stored internally inside the selected agent that contains data on the bids and writes `Qtbid` record – the first part of three parts, to record a bid
- `execute_qbidsd`
  - It creates a list of `Qbid` records by appending to a global list `write_qbid_list_allagents`. Each agent adds or appends to this list. Once all agents have added to this list for a particular time period, the aggregator factory writes as a bulk create to the database
- `execute_segs_bit`
  - It creates a list of `QbidSegments` records by appending to a global list `write_q_list_allagents`. Each agent adds or appends to this list. Once all agents have added to this list for a particular time period, the aggregator factory writes as a bulk create to the database table.

3.8.5 Output to database

There are two main types of output that that aggregator provides to the database. The first concerns the bids to the market (`QBid`, `Qtbid`, `QtBidSegments`) and the second, set point outputs from the disaggregation process , to be used by the physical layer.

3.8.5.1 Aggregation Bids to Market

The aggregation process clusters all bids from four types of devices and collates them into a number of buckets or segments to be bid to the market. The parameters `max_number_stacks_up` and `max_number_stacks_down` in the aggregator factory, define the max number of buckets/segments that can be bid to the market. This is currently set to ten for both up and down bids and for all aggregator agents, but can be set differently for each individual aggregator agent.

- `QtBid`: Main record sent by aggregator at a node at the time when the bid is submitted. This record allows the routine to retrieve from the database all the bids related to all future time steps within the market clearing horizon.
- `Qbid`: Records future time horizon bids for a particular time step. Essentially it is a placeholder for segment bids (or bid curve). Record allows us to recover bid segments at this future time horizon.
- `QbidSegments`: Segments which creates the bid curve in the quantity-price dimensions.

3.8.5.2 Disaggregation Outputs

After the disaggregation process, new device set points are determined and all aggregator set points are updated even ones that did not change and are marked with a time stamp. As the described aggregator deals with four types of devices, the disaggregation process has to correctly identify the type of device that has both a change and no change to its set-point and then change the value in the appropriate `disaggregator_setpoints_{device}out tables`. There are four of them that updated by the CGCL Aggregator module:

- `disaggregator_setpoints_hydout`: set-points from disaggregator to run-of-the-river hydroelectric power plants.
- `disaggregator_setpoints_pvout`: set-points from disaggregator to solar power plants.

- `disaggregator_setpoints_winddout`: set-points from disaggregator to wind power plants.
- `disaggregator_setpoints_selout`: set-points from disaggregator to sheddable loads (e.g. Street Lamps)

3.9 Electrical Energy Storage unit aggregation module

3.9.1 Brief description of the module

In this section the integration of the Electrical Energy Storage (EES) unit aggregation module in the SmartNet simulation platform is described. Under the smart grid concept, an EES aggregator will participate in SmartNet market (ancillary service market) on behalf of the EES owner, aiming to maximize its profit. The EES unit's aggregation model is a profit maximizing optimization problem, where a linear programing is used as a mathematical tool. In detail presentation of the EES unit aggregation optimization algorithm is provided in [2]. The EES unit aggregation model is coded through a mathematical modelling language (AMPL) and a Python interfaces have been created to enable AMPL model integration with the database and the rest of the simulation platform. The integration procedure is depicted in Figure 33. According to it, *Interface 1* provides database-AMPL (`DB2AMPL`) communication by reading the corresponding tables from the database and creating a `{file}.dat` for AMPL model. Then AMPL aggregation model is simulated with `{file}.dat` input parameters. When the simulation is complete, the AMPL writes simulation results (the optimal bids) on `{file}.log`. In a similar way, the *Interface 2* establishes the communication between AMPL-database (`AMPL2DB`) by reading AMPL `{file}.log` file and writing the optimal bids to the respective tables in the database (market module).

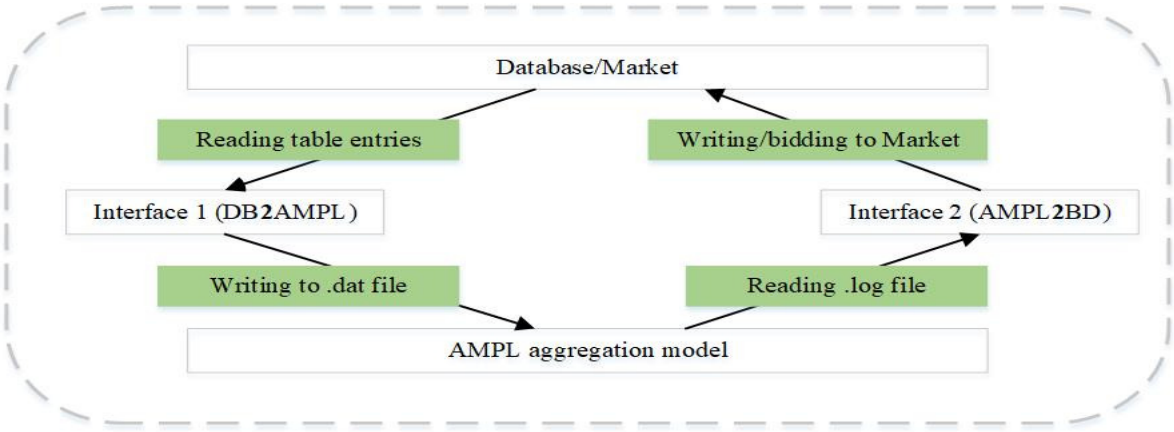


Figure 33 – The EES unit aggregation module integration procedure with the database

3.9.2 Input from database

The following tables provide the input for the EES interface;

Device Constants:

- `device_StoConstants`
- `device_StoCategory`

Scenario:

- `scenario_Scenario`

Profiles:

- `profiles_StoPowerProfiles`
- `profiles_NodeHasNodePriceProfile`
- `profiles_ NodePriceProfile`
- `profiles_NodePrice`
- `profiles_StoPower`
- `profiles_ NodeHasNodeDeltaCostProfile`
- `profiles_ NodeDeltaCost`

Network:

- `network_Node`
- `network_Network`
- `network_SubNetwork`
- `network_SubNetworktype`

phylay_setpoints:

- `phylay_setpoints_StoPhyOut`

3.9.3 Input from other modules

The `Scheduler` model provides the following parameters to the interface:

- Aggregation Mode (`"aggreg"`)
- Scenario identifier
- A starting time step
- Market latency
- Market horizon

3.9.4 List of functions of the module

By launching the python script `AggregatorSTO.execute("aggreg")` three modules are called: `queries.py` contains the required functions to read the data from the database, `readDbForAmpl.py` module aggregates the data read by `queries.py` and finally `aggregator.py` sends the bids to the market.

- `queries.py` contains the following functions:
  - `queries.retrieve_constants`
  - `queries.get_timeStepLengthInSeconds`
  - `queries.get_CB`
  - `queries.get_priceProfile_query`
  - `queries.get_stoConstants_query`
  - `queries.get_stoPhyOut`
  - `queries.get_TimeRleated`
  - `queries.get_accepted_bids`
  - `queries.get_cleared_price`
  - `queries.get_ActivePowerDeltaCost`
  - `queries.create_actor`
  - `queries.get_nodes_list`

- `readDbForAmpl.py` contains the following functions:
  - `readDbForAmpl.write_first_block`
  - `readDbForAmpl.write_second_block`
  - `readDbForAmpl.write_third_block`
  - `readDbForAmpl.write_fourth_block`
  - `readDbForAmpl.write_fifth_block`
  - `readDbForAmpl.write_sixth_block`
  - `readDbForAmpl.write_seventh`
  - `readDbForAmpl.write_eighth_block`
  - `readDbForAmpl.write_ninth_block`
  - `readDbForAmpl.make_Files`
  - `readDbForAmpl.bid`
  - `readDbForAmpl.find_Ampl_file_to_append_cleared_bids`
  - `readDbForAmpl.allSubNets`
  - `readDbForAmpl.subNets_items`

- `readDbForAmpl.trans_and_Distri_nodes`
- `readDbForAmpl.arrange_Trans_and_distr_nodes`
- `readDbForAmpl.get_set_of_EVS`
- `readDbForAmpl.create_list_of_Transmission_nodes`

- `aggregator.py` contains the following functions:
  - `aggregator.initialise`
  - `aggregator.execute`
  - `aggregator.aggreg_bid`
  - `aggregator.create_qSegment`
  - `aggregator.create_QPHalfPlaneConstraint`
  - `aggregator.get_cleared_bids`

3.9.5 Flow chart of the module

The simulation block prepares the files (optimization models) to be processed by AMPL and their structure is highly dependent on the chosen coordination scheme. Once all the files `{file}.dat` are generated by accessing to the simulation database then the AMPL model runs and outputs the results as bids that are sent back to the database by means of `{file}.log`. The flowchart of the module is reported in Figure 34.

3.9.6 Output to database

Bids

- `bids_QBid`
- `bids_QBidSegment`
- `bids_QtBbid`

Constraints

- `constraints_QPDiscConstraint`

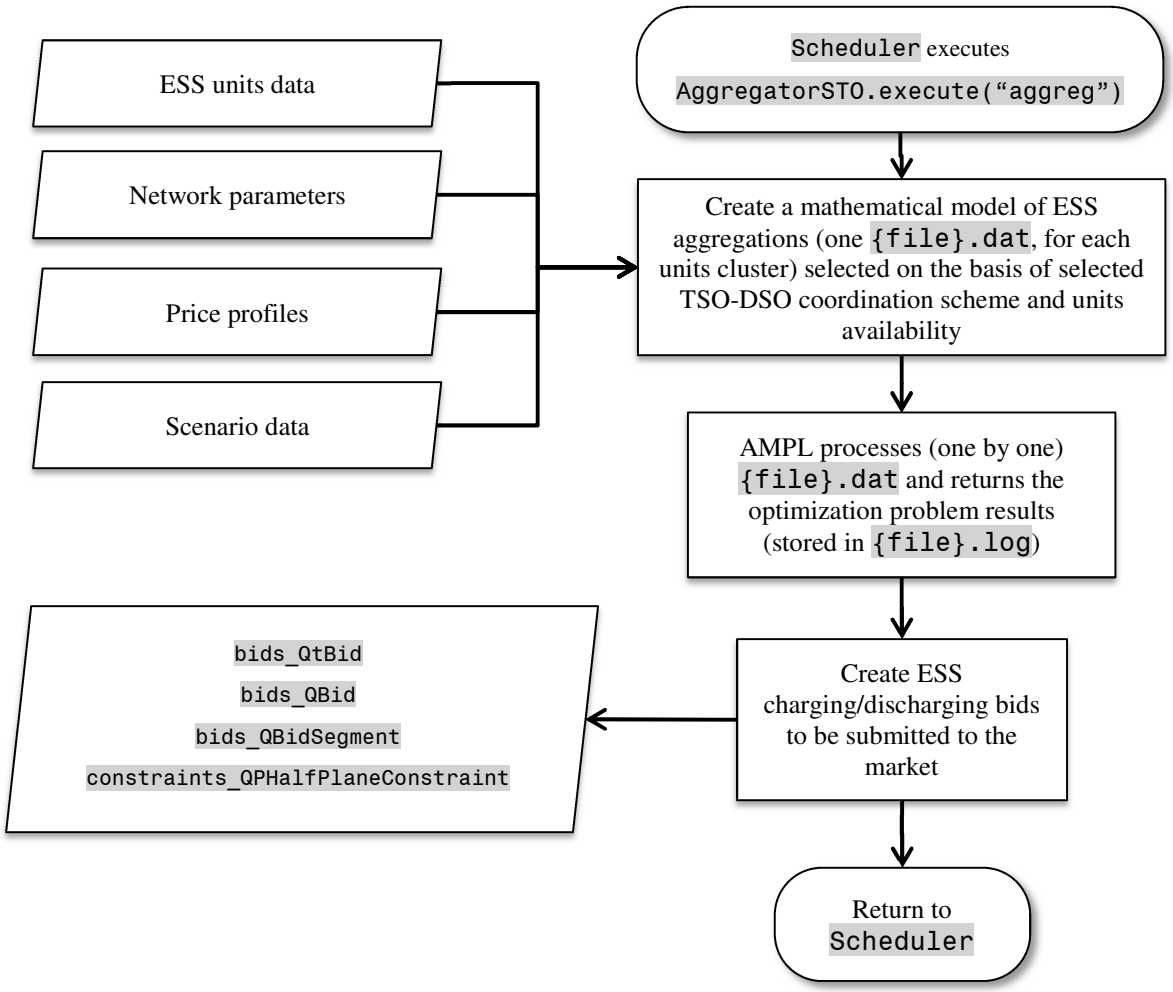


Figure 34 – Flow diagram of `AggregatorSTO.execute("aggreg")`

3.10 Electrical Energy Storage unit disaggregation module

3.10.1 Brief description of the module

Electrical Energy Storage (EES) unit disaggregation model is a cost minimizing optimization problem, where the main objective of the function consists of supplying the accepted charge/discharge quantity with the lowest possible cost. Detailed description of the EES unit disaggregation model is provided in [2]. As for the aggregation module, the EES unit disaggregation functions are coded in AMPL and Python which communicates through *Interface 1* and *Inferface 2* (`DB2AMPL` and `AMPL2DB` respectively). The activations returned by the market module are coded within the mathematical model (`{file}.dat`) and processed by AMPL, which returns `{file}.log` converted by a python module in the set-points of each ESS unit (Figure 35).

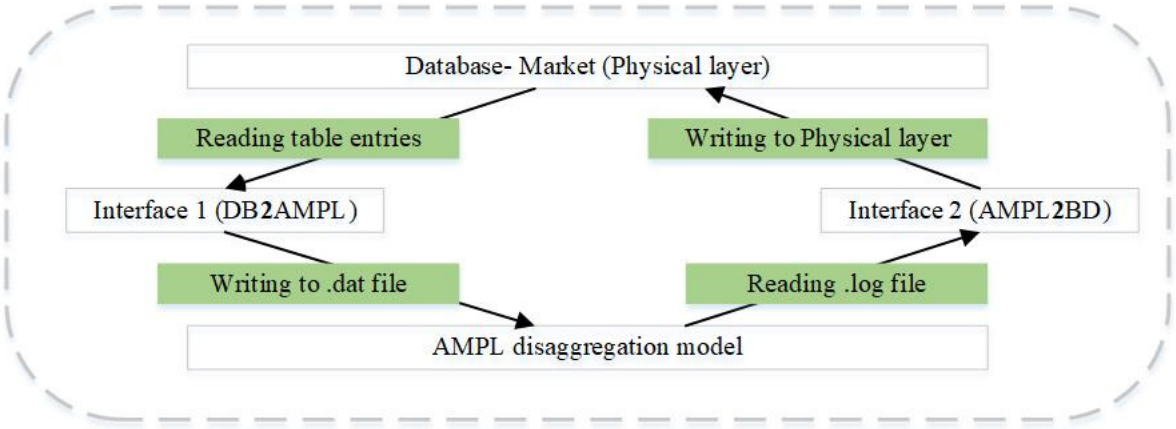


Figure 35 – The EES unit disaggregation module integration procedure with the database

3.10.2 Input from database

Unlike the aggregation parts where there is a need to access various tables in the database, in the disaggregation part all is needed to check how many bids were accepted if any and check the status of the clearing market. Once the new data is obtained, it is then appended to the existing data for optimization model to solve it.

Market clearing:

- `clearing_NodeVariables`
- `clearing_QBidVariables`

Bids:

- `bids_Qbid`
- `bids_QbBidSegment`
- `bids_QtBid`

3.10.3 Input from other modules

The `Scheduler` model provides the following parameters to the interface;

- Aggregation Mode (`"disaggreg"`)
- Scenario identifier
- A starting time step
- Market latency
- Market horizon

3.10.4 List of functions of the module

In order to disaggregate ESS units activations, the `Scheduler` launches the python script `AggregatorSTO.execute("disagg")` which is calling the following functions coded within `aggregator.py` module:

- `aggregator.write_dissagreg_ampl_file`
- `aggregator.sendDisAggregationLogDataToDatabase`

3.10.5 Flow chart of the module

The disaggregation process begins by checking the bids stored within the database, querying the ones that have been accepted. Once these information is obtained the requested activations are grouped for each node of the network (nodal resolution is depending in the TSO-DSO coordination scheme) and an AMPL model is constructed for that node. Similar to the aggregation process, the AMPL optimization model runs each individual `{file}.dat` to perform disaggregation and the results are stored within `{file}.log` (one for each node) which is then immediately written back into the database. The flowchart of the module is represented in Figure 36.

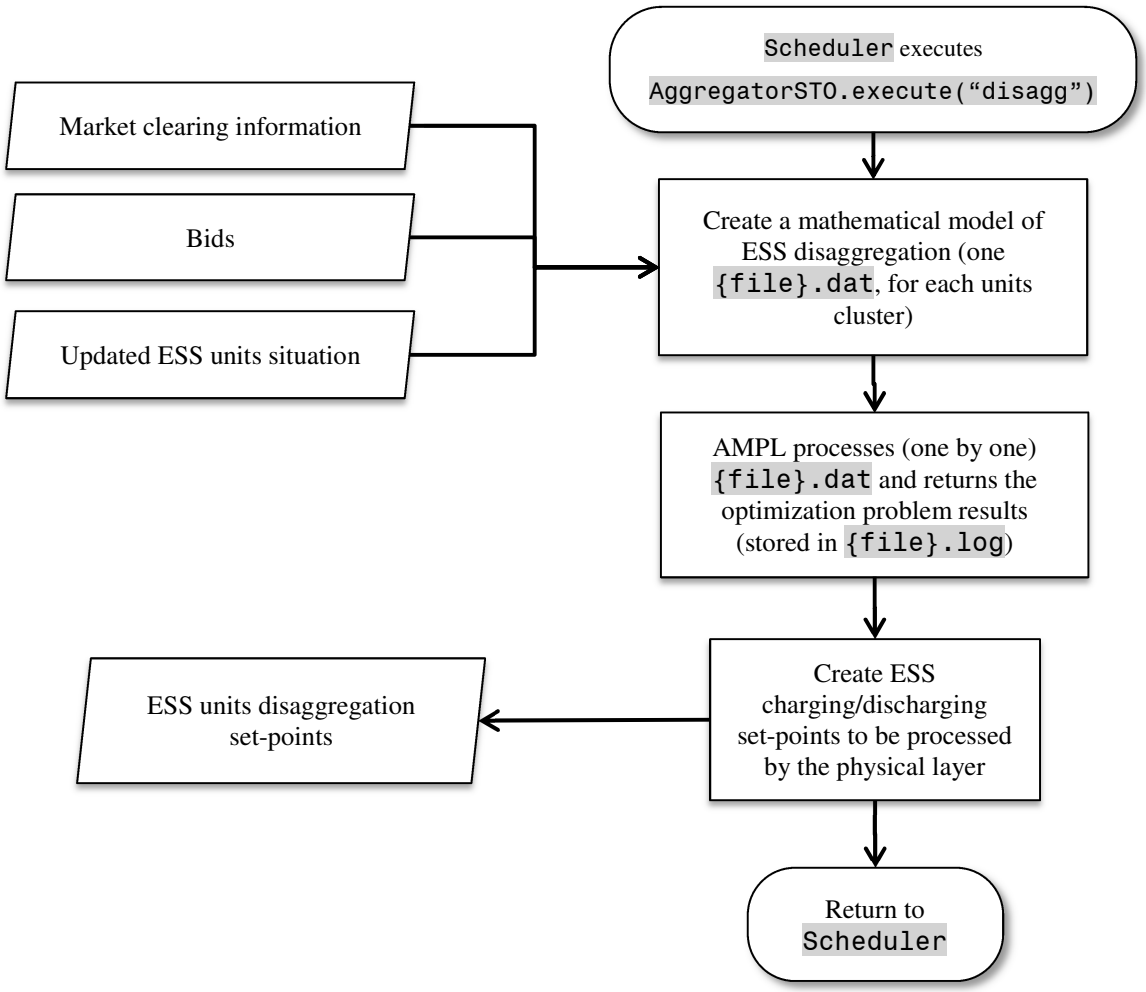


Figure 36 – Flow diagram of `AggregatorSTO.execute("disagg")`

3.10.6Output to database

Once the disaggregation data is calculated it is then written back into the database in the following table.

Disaggregator_setpoints:

- `disaggregator_setpoints_StoAggOut`

4 Market layer

4.1 Brief description of the module and flowchart

The market layer encompasses the clearing platform of the integrated reserve market. The integrated reserve market architecture aims at allowing flexible resources coming from both transmission and distribution networks to compete in the same ancillary services market. As shown in Figure 37, the market receives bids (from aggregators), network models (from physical layer) and forecasted network imbalance (from scenario) as inputs. Based on the provided inputs, the market is ready to run the constrained-optimization problem known as the "market clearing". This is the core algorithm which is responsible to calculate the optimal volume of power exchange (cleared quantity) and the associated value of power injection or off-take at each node (cleared price). In a final step, the results of the market clearing are sent to the aggregators, which in turn dispatch the resources accordingly (disaggregation module).

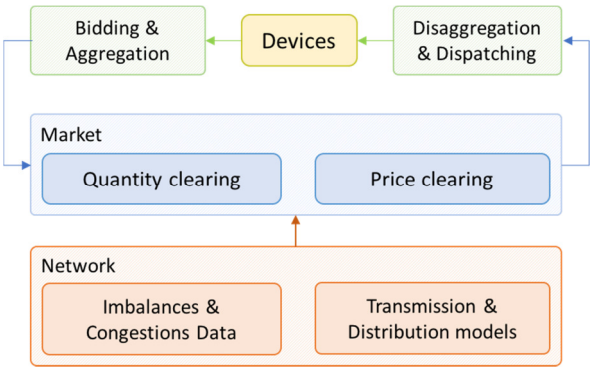


Figure 37 – Simplified block diagram of the main interactions of the market block with other simulation layers

In order to map the dynamics of different flexibility resources while expressing the constraints of assets, aggregators, and operators, the market module allows both simple bids (specifying quantities and prices) and complex bids, i.e. simple bids on which further constraints (e.g. ramping constraints, exclusive bids) are applied. Concerning the network constraints, there is a trade-off between the level of details of the network and the computational time: it cannot be very simplified (otherwise it creates a big demand of unwanted measurements because the physical constraints of network will not be considered in the market clearing algorithm) but it cannot be too complex (in order to maintain the algorithm computationally tractable). Therefore, a proper network model is chosen based on the type, topology, and

size of the power network. Voltage constraints and reactive power are included for the distribution grid, whereas a DC model for the transmission network is adopted.

In the clearing module, methods for finding the optimal values of traded quantity (of energy) and the resulting price (of the corresponding quantity) are developed. The clearings of quantity and price are not separable tasks, but rather procedures belonging to the same module.

Furthermore, four coordination schemes for the acquisition of ancillary services between the transmission system operator (TSO) and the distribution system operator (DSO) have been considered in the simulation. There are different requirements regarding the parameter settings, input/output data and arrangement of modules for each TSO-DSO coordination scheme, as well as additional local optimization algorithms for the decentralized schemes.

Focusing on inputs and outputs that the market reads respectively writes to the database, we can summarize this in Figure 38.

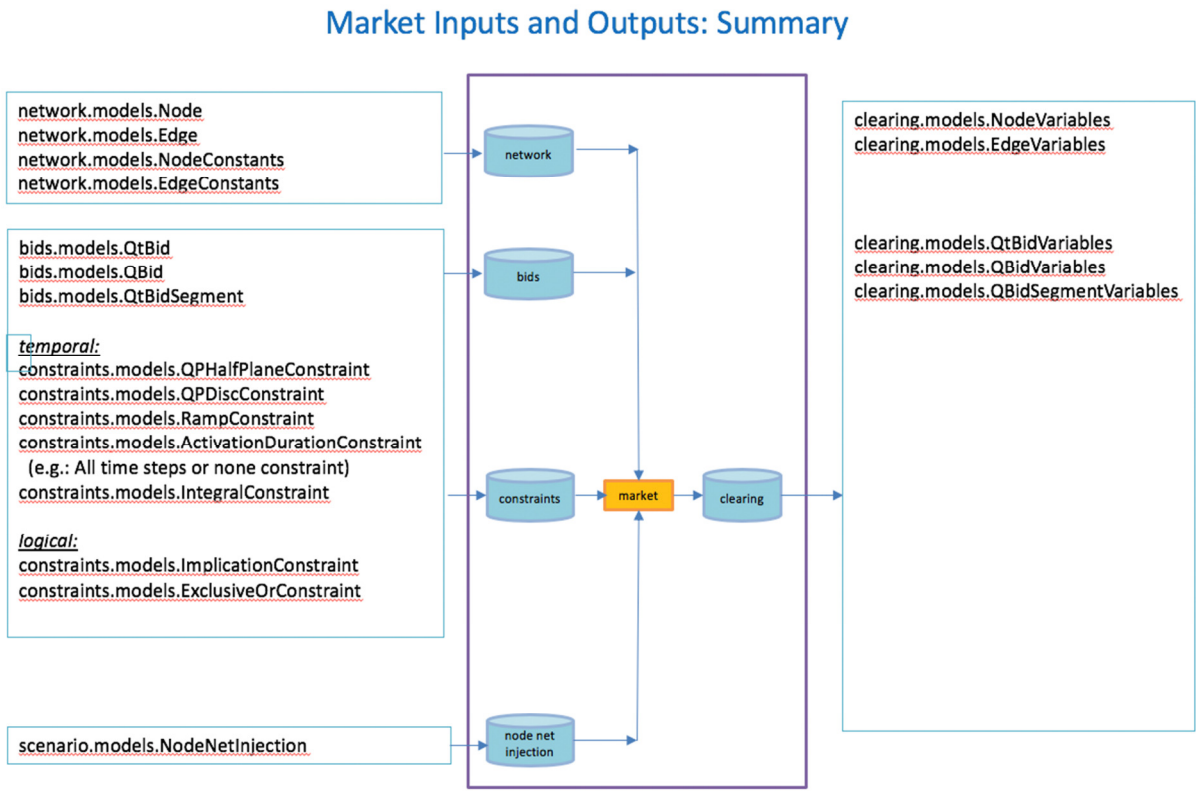


Figure 38 – Summary of market inputs and outputs

The following sections describe the inputs and output of the market block in detail. Note that, apart from these database tables and fields, there are no other inputs or outputs to and from the market, except market internal files, like AMPL model files and log files. These market internal files are of no use to the other blocks in the SmartNet simulator.

4.2 Inputs from database

The inputs of the market layer are firstly the one related to the network description, secondly the bids and bid constraints (coming from the aggregators) and finally the forecasted net injections per node coming from scenario. The database tables that are used as market input are listed below, together with and explanation of the main related parameters (columns of the table).

Network Model:

The market needs to know the network description to be able to model the power flows of the lines as a function of the power exchange of the nodes. This includes the network topology and electrical parameters of lines, the be able to calculate active, reactive power flows and losses. The network model is described here as an input to the market block, but is also an input to other blocks. Other blocks will refer to this section if it's an input to their blocks as well.

The network is composed by the following fields, which are all read by the market block:

- `Network.models.Network`
- `Network.models.SubNetwork`
- `network.models.Node`
- `network.models.Edge`
- `network.models.NodeConstants`
- `network.models.EdgeConstants`

Market Bids:

Bid related inputs are the following.

- `bids.models.QBidSegment`
- `bids.models.QBid`
- `bids.models.QtBid`

There are class functions to plot bids in vector graphic format for debugging. These also allow to do curve crossing and graphically discover the crossing point. An example is given in Figure 39.

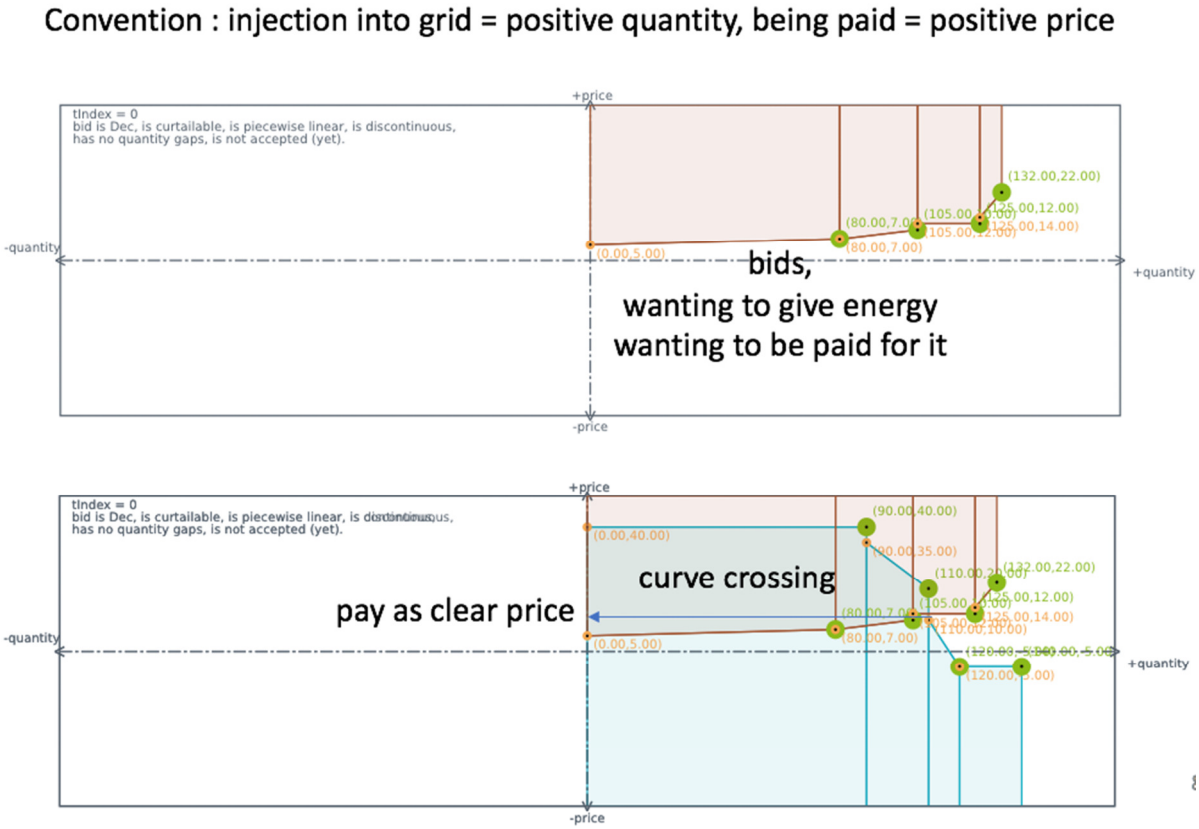


Figure 39 – Illustrative bid curve plots

Market Bid Constraints:

It is perfectly possible to only construct bids (`QtBid`, `QBid`, `QBidSegment`) combinations into the market without adding any constraint over them. These are called *naked bids*. However, sometimes temporal constraints (like ramp constraints) or logical constraints (like conditional constraints: if accept bid1, also accept bid2) are needed for some of the simulated flexible units. For this reason the following constraint types have been modeled on the basis of the different aggregator typologies:

- `constraints.models.QPHalfPlanceConstraint`
- `constraints.models.QPDiscConstraint`
- `constraints.models.rampConstraint`
- `constraints.models.ActivationDurationConstraint`
- `constraints.models.IntegralConstraint`
- `constraints.models.ImplicationConstraintsOnQBids`
- `constraints.models.ImplicationConstraintOnQtBids`
- `constraints.models.ExclusiveChoiceConstraintOnQBids(_List)`

- `constraints.models.ExclusiveChoiceConstraintOnQBids_QBid`
- `constraints.models.ExclusiveChoiceConstraintOnQtBids(_List)`
- `constraints.models.ExclusiveChoiceConstraintOnQtBids_QtBid`

NodeNetInjection:

- `scenario.models.NodeNetInjectionProfile`
- `scenario.models.NodeHasNodeNetInjectionProfile`
- `scenario.models.NodeNetInjection`

4.3 List of functions

The market layer is composed by several functions, and the main ones can be listed as follows:.

- `market.initialise(scenarioId, atT=-1)`: function to be performed only once, so outside the scheduler loop.
- `market.generateAmplAtTIndependentAmplDataFile`: function to generate the part of the AMPL data file that is not changing from time iteration to the next. This is mainly the network data, which, in the SmartNet simulations, is assumed to not change over time.
- `market.setTimeParameters(atT, atToForLatency, forHorizon)`: sets market parameters, feed form the scheduler.
- `market.preprocessBids(scenario)`: for a scenario with coordination scheme A specified, bids have to be relocated to the transmission node that is connected to the root node of the distribution network of the node that the original bid was destined for. This function is for debugging the market only.
- `market.preprocessBidNetInjection(scenario, atT, atToForTLatency, forHorizon)`: In case of coordination scheme A, the node net injection has to be adapted as well, to reflect that all of it is happening in the parent node of the distribution network root node. This function is for debugging the market only.
- `market.execute(scenario, atT, atToForTLatency, forHorizon, workWithCoordinationSchemeFiles=True, minQBidSegmentQuantityInMW=0)`: The main block (here: market) function called by the scheduler. When `workWithCoordinationSchemeFiles` is `False` the function supports an older code version that predates the use of coordination schemes. `True` is the default and to be used for all SmartNet simulations. `minQBidSegmentQuantityInMW` is a parameter that can be increased to let the market ignore really small bids.

- `market.testExec`: a function like execute but with more settable arguments. `Execute` calls `testEcex` with some arguments set to fixed values.
- `writeTemplateSetsFile()`, `generateAmplDataFile()`, `readSubstituteWrite()` are just some of the main functions to generate the optimisation problem as AMPL language files. `correctQtBidAndWarn()` is a function that checks for errors against the market rules, format or tolerances in Bids and produces warnings in the log, which can be sent to bidders.
- `parallelExecute()` is a function that, for coordination scheme C, writes many independent problems that can be executed in parallel, to files, then writes a `makefile` to execute these in parallel and calls the make on the `makefile` to do so, then combines the outputs into one big log file. This can then be parsed by the standard output parsing function without it needing to know that parallelism was ever used.
- `writeOutSvgAndPdfFiles()`: writes out scalable vector graphic files, so visualisations, of the network grid with the market outcome in terms of power flows, voltages, loading annotated on top of it. This is very useful for debugging or illustration purposes. An example of a part of an as such generated picture for a radially shaped distribution grid with market output on top is given in Figure 40.

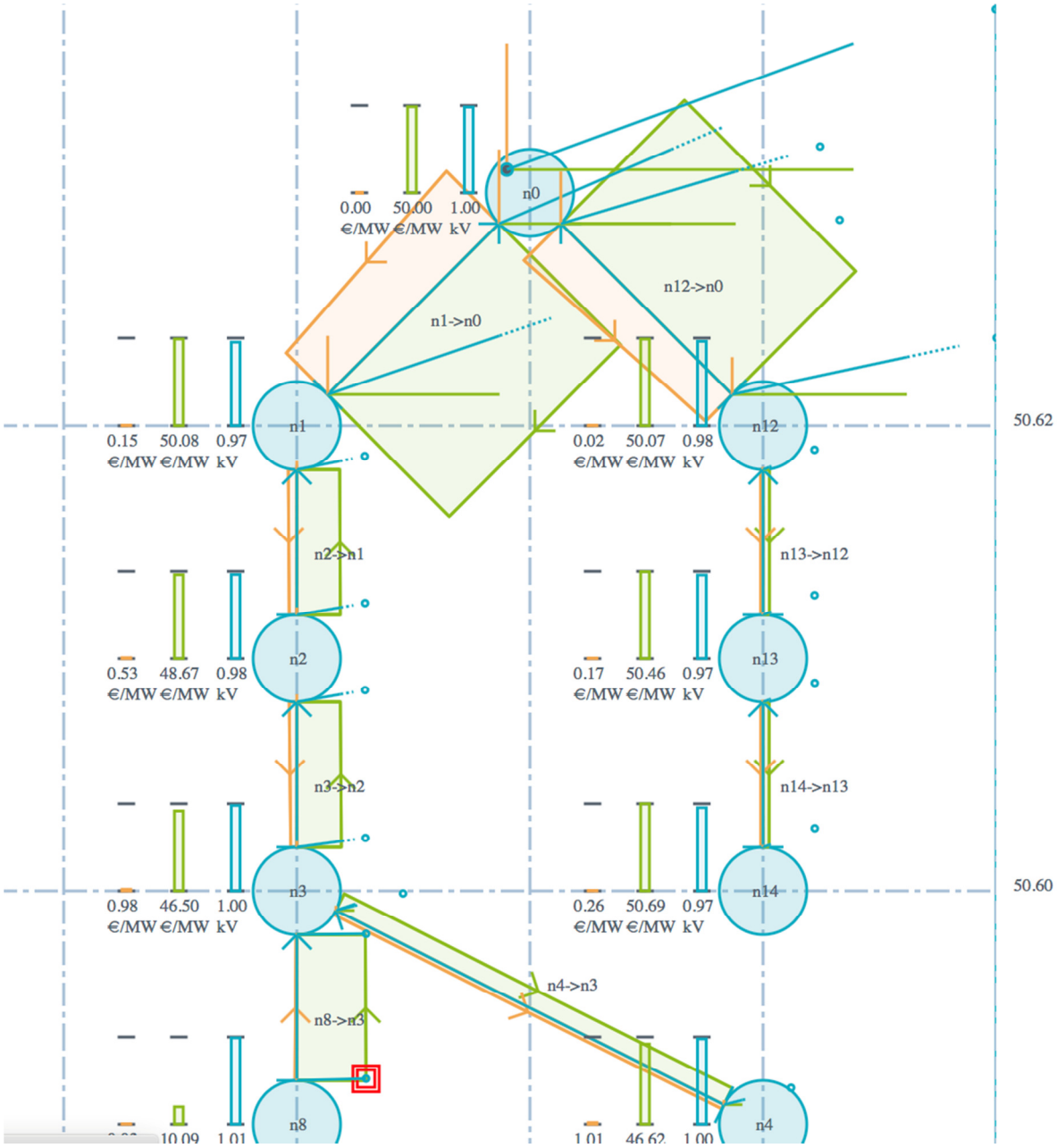


Figure 40 – Illustrative distribution network plotted with electrical variables (market results)

4.4 Flow chart

The main flow chart is summarized in Figure 38 already. However as for the technological implementation a different representation is given in Figure 41.

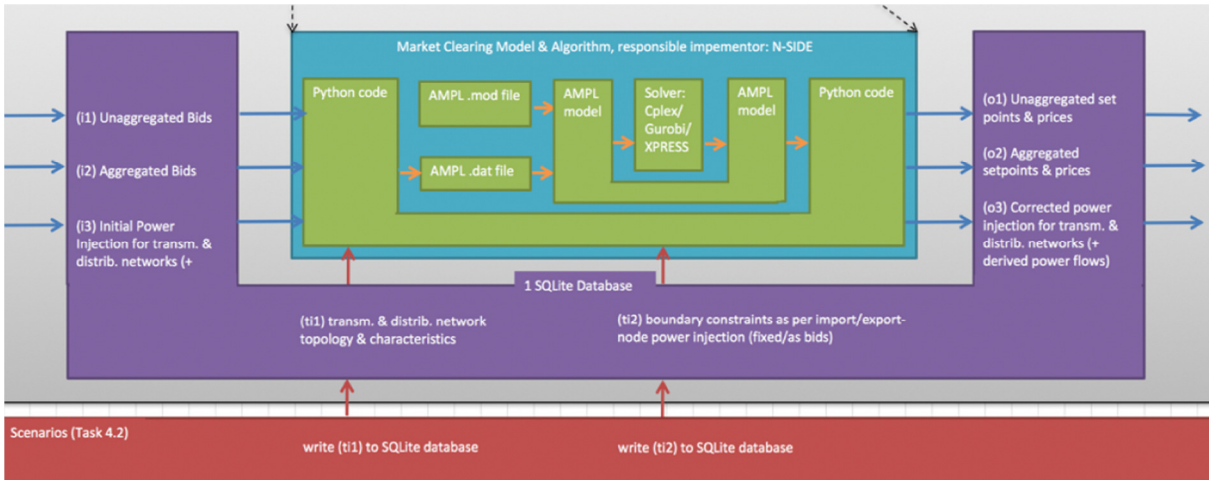


Figure 41 – Technical flow diagram of the market layer process

Figure 41 indicates in the purple area, the names of the main tables that are inputs or outputs to and from the market block. In the inner turquoise block, in the outer green U shaped area, we can see that the code handling database reads and writes is coded in python programming lanquage. The inner green modules reveal that the python code writes a set of files in the AMPL language (amongst which the main ones are a `.dat(a)` file and a `.mod(el)` file). These files formulate the optimisation problem in a concise way. The `.mod` file holds a model that is independent of the input data. The `.dat` file holds all the data that is dependend on an iteration, or a coordination scheme and such. AMPL executable is then called, asking it to solve this optimisation problem. The AMPL executable than generates log files in a particular format, which is then parsed by python code again, which puts the resulting optimisation variable values in the database.

4.5 Outputs to database

The outputs of the market are the market clearing data. More specifically, these entail the accepted quantity (fractions) for all `QBidSegements` for the specific time step of the auction (`forT`). Also a price per node is returned, since the implemented market is assumed to adopt a a pay-as-clear remuneration approach.

Market Clearing:

There are three main market outputs as variable values determined by the market: Bid related, Node related and Edge related outputs. In more detail, these are the following.

- `clearing.models.QBidSegmentVariables`

- `clearing.models.QBidVariables`
- `clearing.models.NodeVariables`
- `clearing.models.EdgeVariables`

A graphical summary of the 5 tables returned the market outputs and their relations is reported in Figure 42. In this picture it can be noticed that each of the tables `QtBid`, `QBid` and `QBidSegment` have their respective variable counterpart tables that reference them. As for `Nodes` and `Edges`, these also have market variables associated to them and these tables make a reference (forerign key) to the scenario.

Figure 42 – Tables returned by the market layer and their relations

There are three other results, mainly for debugging purposes, the market writes to the database:

- `clearing.MarketDSOAggregation`
- `clearing.AggDSOBids`
- `clearing.models.DistributedTimingTrace`

5 Physical layer

5.1 Brief description of the module

The physical layer simulates the physical behaviour of the network (transmission and distribution) and of all the connected controllable devices. The physical layer is simulated in three main steps:

1. The set-points (resulting from the disaggregation process) are applied to the devices taking into account their actual status and physical constraints.
2. The networks states, both for distribution and transmission, are computed. The physical layer optimizes the use of resources in order to avoid congestions not detected by the market. At this step the secondary frequency regulation (i.e. automatic Frequency Restoration Reserve - aFRR) is activated in order to compensate residual energy imbalance.
3. The states of devices are updated taking into account the regulation requested at the previous points.

5.1.1 Updated of devices status according to disaggregation set-points

The aggregation processes introduces approximations in the physics of the controlled devices, especially when several devices have to be managed and all the variables cannot be monitored. In addition, some of the variable involved in the estimation of available flexibility might be affected by forecasting error. For this reason the physical layer corrects the set-points of devices on the basis of a more detailed model and real time profile. Once these set points have been corrected, the state of the power system can be computed.

5.1.2 Simulation of network and automatic asset

The simulation of the entire power system (transmission and distribution network) can result in a complex problem to be managed by a single computational step. Taking advantage of the decoupling between transmission and distribution systems (tap-changing transformers are assumed to be operational in all primary substations) the effect of the transmission system voltage on the distribution in is considered negligible. With this assumption, each distribution network is simulated separately by using the most updated status of the connected devices and, thanks to the network model (physical parameters, constraints and controllable asset), and obtaining the voltage of its nodes, loading of its lines and the power exchange with the upstream network (transmission system). Subsequently, once the power exchange in correspondence of the primary substations is available, also the transmission network can simulated. Both distribution and transmission networks are simulated by using a standard Optimal Power Flow (`PyPower`) software which has been adapted according to the simulation platform needs. In the optimization of networks, the algorithms use both the DSO or DSO assets and the available

controllable resources, selected by the scenario, to avoid current and voltage congestions by modulating (in case of necessity) their active and reactive power exchanges (actions on the active power set-points represent the unwanted measures) within the flexible resources capabilities.

Static compensators (STATCOMs) and On Load Tap Changers (OLTC) are providing significant contribution to transmission system stability (especially in terms of voltage regulation). However, the simulation tool adopted (`PyPower`) does not allow a straightforward integration of these asset. Thus, for simplicity reasons, these assets are not simulated (except at distribution level) and lager voltage regulation margins have been implemented.

After having simulated the entire power system the residual total imbalance is calculated in order to compensate it with aFRR resources. The calculated aFRR needs are then shared between all the resources (both at distribution and transmission levels) that can provide this service. The power exchange of nodes is then accordingly updated by performing another Optimal Power Flow (OPF) aimed at solving the congestions possibly caused by aFRR activations (for sake of simplicity, aFRR effects on lines loading are recalculated for transmission network only).

5.1.3 Update of devices status according to network behaviour

After the simulation of the entire network, the devices status are updated based on possible requests computed with the OPF for solving network congestions. In these calculations the new set-points and status already respect the internal constraints of the controlled units since the active and reactive power capabilities of resources have been preliminarily computed taking into account the devices characteristics. The updated state of devices are then reported to aggregators for the next simulation steps.

5.2 Input from database

The input of the physical layer can be divided in two group. The first group refers to the data used to simulate the devices, while the second group refers to the data used for the network simulation. The input data used by the devices are divided according to the typologies of devices and they describe all the devices characteristics used both for aggregators and physical layer simulations. The table that contain all the constants characteristics of devices are:

Device Constants:

- `device_ChpConstants`
- `device_ConConstants`
- `device_HydConstants`
- `device_PvConstants`

- device_SelConstants
- device_StatConstants
- device_StoConstants
- device_TclConstants
- device_WetConstants
- device_WindConstants

The active power profiles of the devices are reconstructed from the following tables. Depending on the complexity of the device technologies, the power profile of each of them can be described by using multiple tables. The presence of two tables is the minimum requirement: one contains the time profile, while the second is used to link the time profile with the corresponding power unit stored in the constant table. In this way the same profile can be assigned to multiple devices.

Device Profiles

- Combined Heat and Power generators
  - profiles_ChpPowerProfile
  - profiles_ChpPower
  - profiles_XiDemandHeatProfile
  - profiles_XiDemandHeat

- Conventional generators
  - profiles_ConPowerProfile
  - profiles_ConPower

- Hydroelectric generators
  - profiles_HydPowerProfile
  - profiles_HydPower
  - profiles_HydPowerBaselineProfile
  - profiles_HydPowerBaselines

- Solar generators
  - `profiles_PvPowerProfile`
  - `profiles_PvPower`
  - `profiles_PvPowerBaselineProfile`
  - `profiles_PvPowerBaseline`

- Sheddable loads
  - `profiles_SelPowerProfile`
  - `profiles_SelPower`

- STACOM
  - `profiles_StatPowerProfile`
  - `profiles_StatPower`

- Storage devices
  - `profiles_StoPowerProfile`
  - `profiles_StoPower`

- Thermostatically controlled loads
  - `profiles_TclConfortTempProfile`
  - `profiles_TclConfortTemp`
  - `profiles_TclAvailabilityProfile`
  - `profiles_TclAvailability`
  - `profiles_TclMaxTempProfile`
  - `profiles_TclMaxTemp`
  - `profiles_TclMinTempProfile`
  - `profiles_TclMinTemp`
  - `profiles_TclInternalThermalGainProfile`
  - `profiles_TclInternalThermalGain`
  - `profiles_TclEnvThermalGainProfile`
  - `profiles_TclEnvThermalGain`
  - `profiles_ExternalTempProfile`
  - `profiles_ExternalTemp`

- Atomic Loads – Wet Appliances
  - `profiles_WetApplianceModel`
  - `profiles_WetApplianceProfile`
  - `profiles_WetApplianceBootingDistribution`

- Wind generators
  - `profiles_WindPowerProfile`
  - `profiles_WindPower`
  - `profiles_WindPowerBaselineProfile`
  - `profiles_WindPowerBaseline`

Finally, in order to simulate the interaction of the physical devices with the aggregators (and the market) the set-point sent by the aggregators are processed by accessing to the following tables (which are divided per device technology):

Disaggregator set-points

- `disaggregator_setpoints_ChpAggOut`
- `disaggregator_setpoints_ConAggOut`
- `disaggregator_setpoints_HydAggOut`
- `disaggregator_setpoints_PvAggOut`
- `disaggregator_setpoints_SelAggOut`
- `disaggregator_setpoints_StatAggOut`
- `disaggregator_setpoints_StoAggOut`
- `disaggregator_setpoints_TclAggOut`
- `disaggregator_setpoints_WetAggOut`
- `disaggregator_setpoints_WindAggOut`

The data of the network are stored in two categories of table: one for the constant data, the other for the power profiles. The constant data are stored in:

Network parameters

- `network_Node`
- `network_NodeConstants`
- `network_BusType`
- `network_Edge`
- `network_EdgeConstants`
- `network_Network`
- `network_SubNetwork`
- `network_SubNetworkType`
- `phylay_ControlSolutionDso`
- `phylay_ControlSolutionTso`

The power profile of each node, which contain all the information of devices not participating to the market, is contained in the following tables:

Node power profile

- `profiles_NodePower`
- `profiles_NodePowerProfile`
- `profiles_NodeHasNodePowerProfile`

In addition there are a set of support tables, which are used for storing temporary state both for devices and networks:

Device and Network Variables

- `phylay_NodeVariables`
- `phylay_EdgeVariables`
- `devices_ChpVariables`
- `devices_ConVariables`
- `devices_HydVariables`
- `devices_PvVariables`
- `devices_SelVariables`
- `devices_StatVariables`
- `devices_StoVariables`
- `devices_TclVariables`
- `devices_WetVariables`
- `devices_WindVariables`

5.3 Input from other modules

The only direct input is from the scheduler, which shares scenario ID and the time step resolution to be used for the simulation.

5.4 List of functions of the module

For each device category there is a set of function used for simulating its behaviour and the evolution of the internal states.

- Combined heat and power generators
  - `devices_ChpAggToVar`: this function takes the set-points of aggregators, checks the constraints and writes them in the variables support tables.
  - `devices_ChpVarToDevOut`: this function takes the state of the devices before the simulation of the network (which are stored in the variables tables) and writes them in the devices table.
  - `devices_ChpVarToPhyOut`: this function takes the state of the devices (stored in the variables tables) after the simulation of the network (so that it considers possible re-dispatching due to network congestion management) and writes them in the devices table.
  - `devices_checkChp`: this is a sub-function used to check if the set-points respect the charge constraints and in case it corrects them.
  - `devices_uptdateChp`: this is a sub-function used to check if the set-points respect the physical constraints. In case they are not, the function corrects them, opportunely updating the state of the device.

- Conventional generators
  - `devices_ConAggToVar`: this function takes the set-points of aggregators, it checks the constraints and it writes them in the variables support tables.
  - `devices_ConVarToDevOut`: this function takes the state of the devices before the simulation of the network (which are stored in the variables tables) and writes them in the devices table.

- `devices_ConVarToPhyOut`: this function takes the state of the devices (stored in the variables tables) after the simulation of the network (so that it consider possible re-dispatching due to network congestion management) and writes them in the devices table.
- `devices_updateCon`: this is a sub-function used to check if the set-points respect the constraints, in case correct them, and then update the state of the device.

- Hydro generators
  - `devices_HydAggToVar`: this function takes the set-points of aggregators, it checks the constraints and it writes them in the variables support tables.
  - `devices_HydVarToDevOut`: this function takes the state of the devices before the simulation of the network (which are stored in the variables tables) and writes them in the devices table.
  - `devices_HydVarToPhyOut`: this function takes the state of the devices (stored in the variables tables) after the simulation of the network (so that it consider possible re-dispatching due to network congestion management) and writes them in the devices table.
  - `devices_updateHyd`: this is a sub-function used to check if the set-points respect the constraints, in case correct them, and then update the state of the device.

- Photovoltaic generators
  - `devices_PvAggToVar`: this function takes the set-points of aggregators, it checks the constraints and it writes them in the variables support tables.
  - `devices_PvVarToDevOut`: this function takes the state of the devices before the simulation of the network (which are stored in the variables tables) and writes them in the devices table.
  - `devices_PvVarToPhyOut`: this function takes the state of the devices (stored in the variables tables) after the simulation of the network (so that it consider possible re-dispatching due to network congestion management) and writes them in the devices table.
  - `devices_updatePv`: this is a sub-function used to check if the set-points respect the constraints, in case correct them, and then update the state of the device.

- Sheddable loads
  - `devices_SelAggToVar`: this function takes the set-points of aggregators, it checks the constraints and it writes them in the variables support tables.
  - `devices_SelVarToDevOut`: this function takes the state of the devices before the simulation of the network (which are stored in the variables tables) and writes them in the devices table.
  - `devices_SelVarToPhyOut`: this function takes the state of the devices (stored in the variables tables) after the simulation of the network (so that it consider possible re-dispatching due to network congestion management) and writes them in the devices table.
  - `devices_updateSel`: this is a sub-function used to check if the set-points respect the constraints, in case correct them, and then update the state of the device.

- Static Compensators (STATCOM)
  - `devices_StatAggToVar`: this function takes the set-points of aggregators, it checks the constraints and it writes them in the variables support tables.
  - `devices_StatVarToDevOut`: this function takes the state of the devices before the simulation of the network (which are stored in the variables tables) and writes them in the devices table.
  - `devices_StatVarToPhyOut`: this function takes the state of the devices (stored in the variables tables) after the simulation of the network (so that it consider possible re-dispatching due to network congestion management) and writes them in the devices table.
  - `devices_updateStat`: this is a sub-function used to check if the set-points respect the constraints, in case correct them, and then update the state of the device.

- Storages
  - `devices_StoAggToVar`: this function takes the set-points of aggregators, checks the constraints and writes them in the variables support tables.
  - `devices_StoVarToDevOut`: this function takes the state of the devices before the simulation of the network (which are stored in the variables tables) and writes them in the devices table.

- `devices_StoVarToPhyOut`: this function takes the state of the devices (stored in the variables tables) after the simulation of the network (so that it considers possible re-dispatching due to network congestion management) and writes them in the devices table.
- `devices_checkSto`: this is a sub-function used to check if the set-points respect the charge constraints and in case it corrects them.
- `devices_uptdateSto`: this is a sub-function used to check if the set-points respect the physical constraints. In case they are not, the function corrects them, opportunely updating the state of the device.

- Thermostatically controlled loads
  - `devices_TclAggToVar`: this function takes the set-points of aggregators, it checks the constraints and it writes them in the variables support tables.
  - `devices_TclVarToDevOut`: this function takes the state of the devices before the simulation of the network (which are stored in the variables tables) and writes them in the devices table.
  - `devices_TclVarToPhyOut`: this function takes the state of the devices (stored in the variables tables) after the simulation of the network (so that it consider possible re-dispatching due to network congestion management) and writes them in the devices table.
  - `devices_TclControl`: this is a sub-function used to check if the set-points respect the temperature constraints and in case correct them.
  - `devices_TclControlPower`: this is a sub-function used to check if the set-points respect the physical constraints. In case they are not, the function corrects them, opportunely updating the state of the device.
  - `devices_updateStateFirstOrder`: this is a sub-function used to check if the set-points respect the constraints. In case they are not, the function corrects them, and then update the state of the device (with a first order state-space model).
  - `devices_uptdateTcl`: this is a sub-function used to check if the set-points respect the constraints. In case they are not, the function corrects them, and then update the state of the device (with a second order state-space model).

- Atomic loads (wet appliances)
  - `devices_WetAggToVar`: this function takes the set-points of aggregators, it checks the constraints and it writes them in the variables support tables.

  - `devices_WetVarToDevOut`: this function takes the state of the devices before the simulation of the network (which are stored in the variables tables) and writes them in the devices table.
  - `devices_WetVarToPhyOut`: this function takes the state of the devices (stored in the variables tables) after the simulation of the network (so that it consider possible re-dispatching due to network congestion management) and writes them in the devices table.

- Wind generators
  - `devices_WindAggToVar`: this function takes the set-points of aggregators, it checks the constraints and it writes them in the variables support tables.
  - `devices_WindVarToDevOut`: this function takes the state of the devices before the simulation of the network (which are stored in the variables tables) and writes them in the devices table.
  - `devices_WindVarToPhyOut`: this function takes the state of the devices (stored in the variables tables) after the simulation of the network (so that it consider possible re-dispatching due to network congestion management) and writes them in the devices table.
  - `devices_updateWind`: this is a sub-function used to check if the set-points respect the constraints, in case correct them, and then update the state of the device.

There is also a list of common sub-functions aimed at checking further constraints violations.

- `devices_checkRectangularCapability`: check if the set points respect the rectangular capability of simulated devices.
- `devices_checkCapacityCapability`: check if the set points respect the charge constraints (i.e. energy capacity) of simulated devices.
- `devices_checkRampCapability`: check if the set points respect the ramp capability of simulated devices.
- `devices_checkCircularCapability`: check if the set points respect the circular capability of simulated devices.

The network state is mainly computed by a modified version of `PyPower`, however it has been necessary to write a set of functions that create the input with the correct control configuration:

- `phylay_powerFlowDsoReactive`: this function compute the Optimal Power Flow (OPF) of a distribution network having assumed that only the reactive regulation of devices is possible.
- `phylay_powerFlowDsoOpf`: this function compute the Optimal Power Flow (OPF) of a distribution network in which both reactive and active power can be dispatched.
- `phylay_powerFlowDso`: this function compute the Power Flow (PF) of a distribution network without any possible regulation from the simulated devices.
- `phylay_powerFlowDsoFast`: this function is used for small networks to compute a simplified Power Flow (PF) of a distribution network in order to speed up the simulation process.
- `phylay_powerFlowTsoReactive`: this function compute the Optimal Power Flow (OPF) of a transmission network having assumed that only the reactive regulation of devices is possible.
- `phylay_powerFlowTsoOpf`: this function compute the Optimal Power Flow (OPF) of a transmission network in which both reactive and active power can be dispatched..
- `phylay_powerFlowTso`: this function compute the Power Flow (PF) of a transmission network without any possible regulation from the simulated devices.
- `phylay_secondaryRegulation`: this function, given the imbalance of a power system, compute the automatic secondary regulation for each device participating to the service.
- `phylay_secondaryRegulationSubNetwork`: this function is used in coordination scheme C where a local (distribution) secondary regulation is performed in order to correct local imbalance. The function, given the imbalance of a sub-network, computes the secondary regulation for each device participating to the regulation.
- `phylay_powerFlowDsoSecondary`: this function network sums all the secondary regulation contributions of the devices connected to distribution networks and assigns them to the corresponding transmission node.
- `phylay_typeBusConversion`: this function converts the `network_BusType` into the `PyPower` format
- `phylay_createCaseBus`: this sub-function is used to create the nodes input in the `PyPower` format.
- `phylay_createNodeT`: this sub-function is used to create slack node in `PyPower` format.
- `phylay_createCaseBranch`: this sub-function is used to create the branches input in `PyPower` format.
- `phylay_createtriplets`: this sub-function is used to create the generators input in `PyPower` format. For each generator, a triplet of devices is created in order to divide upward and downward contribution and allowing a better modulation of power.

- `phylay_createtripletsReactive`: this sub-function is used to create the generators input in `PyPower` format when only reactive regulation is possible. For each generator, a triplet of devices is created in order to divide upward and downward contribution and allowing a better modulation of power.
- `phylay_writeToPhylayOut`: this function is used to store the state of nodes and branches returned by the simulator to the output tables.

5.5 Flow chart of the module

The physical layer structure is divided in two main blocks, called **PHYLAY 1** and **PHYLAY 2**. The `phylay1` has the objective of computing the application of the disaggregator set points to the resources, taking into account a detailed model of devices. The possible differences between the disaggregators set point and their application are caused by two main reasons:

- Aggregators uses simplified models with respect to the simulated physics of the devices.
- Forecast errors on the profiles.
- A change in the state of devices occurred in previous time steps in order to manage a local unexpected congestion (re-dispatch).

The final power exchange are then used to compute the network state. The `phylay1` does not compute the evolution of the state of devices (e.g. the state of charge of storages), because this is done only when the congestion management of the network is performed in `phylay2`. In fact the set point of devices can change due to re-dispatching measures (i.e. unwanted measures) activated by network operators.

5.5.1 PHYLAY 1

The general structure of `phylay1` is described in the flowchart reported in Figure 43. The algorithm takes into account the previous state of devices, the device parameters, the real time profile and the aggregator set-points in order to compute the exchange of power of each single device. The results are stored in the support tables and in the device set-points tables.

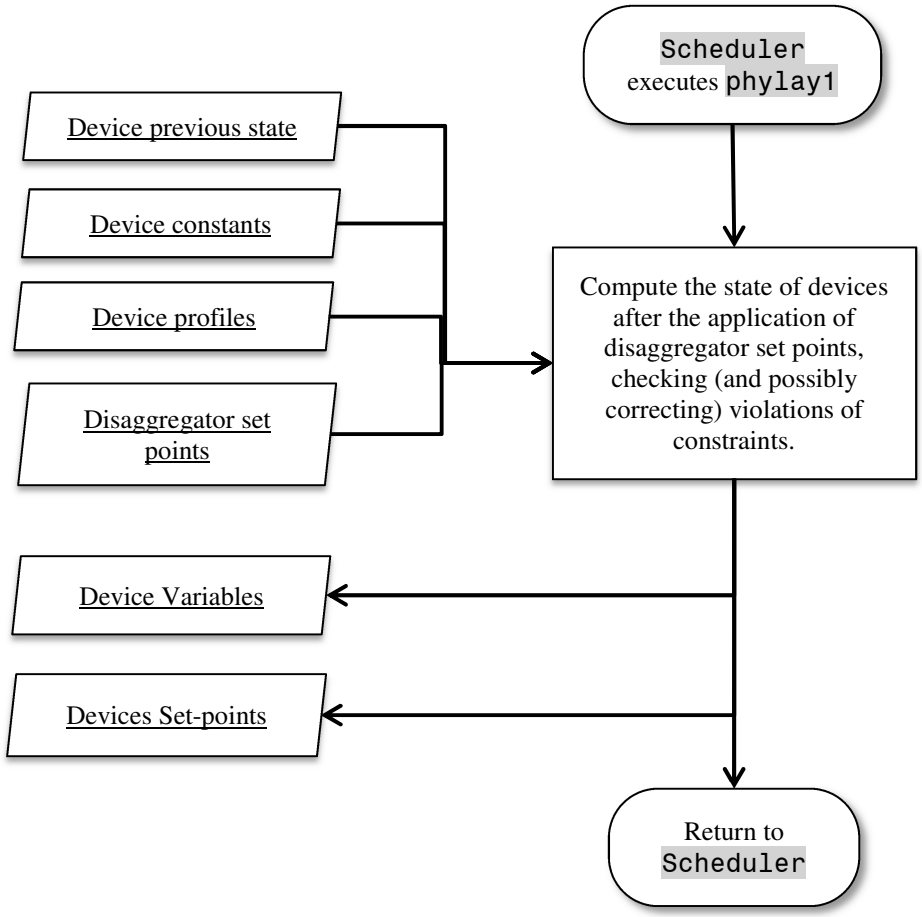


Figure 43 – Flow diagram of PHYLAY 1

Within `phylay1`, each device is simulated and the computation process can be described separately with dedicated flowcharts (Figure 44÷Figure 53).

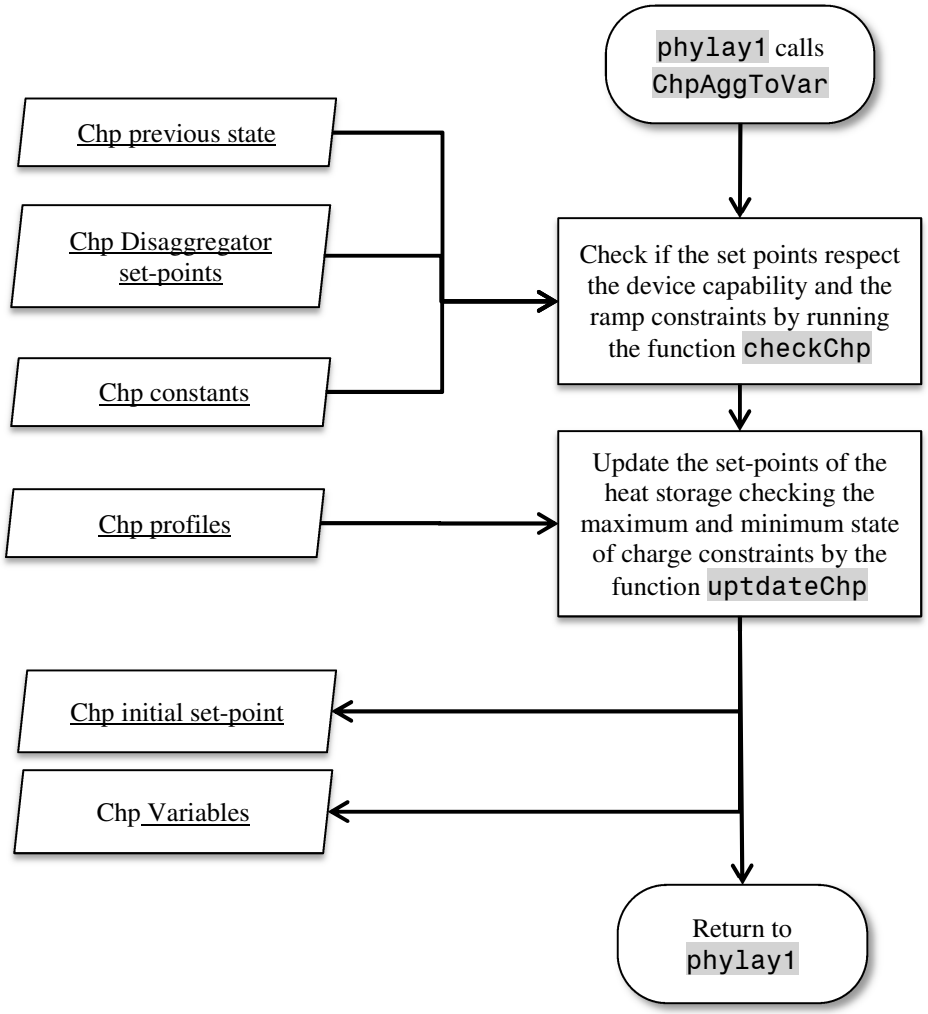


Figure 44 – Flow diagram of PHYLAY 1 – Combined Heat Power devices

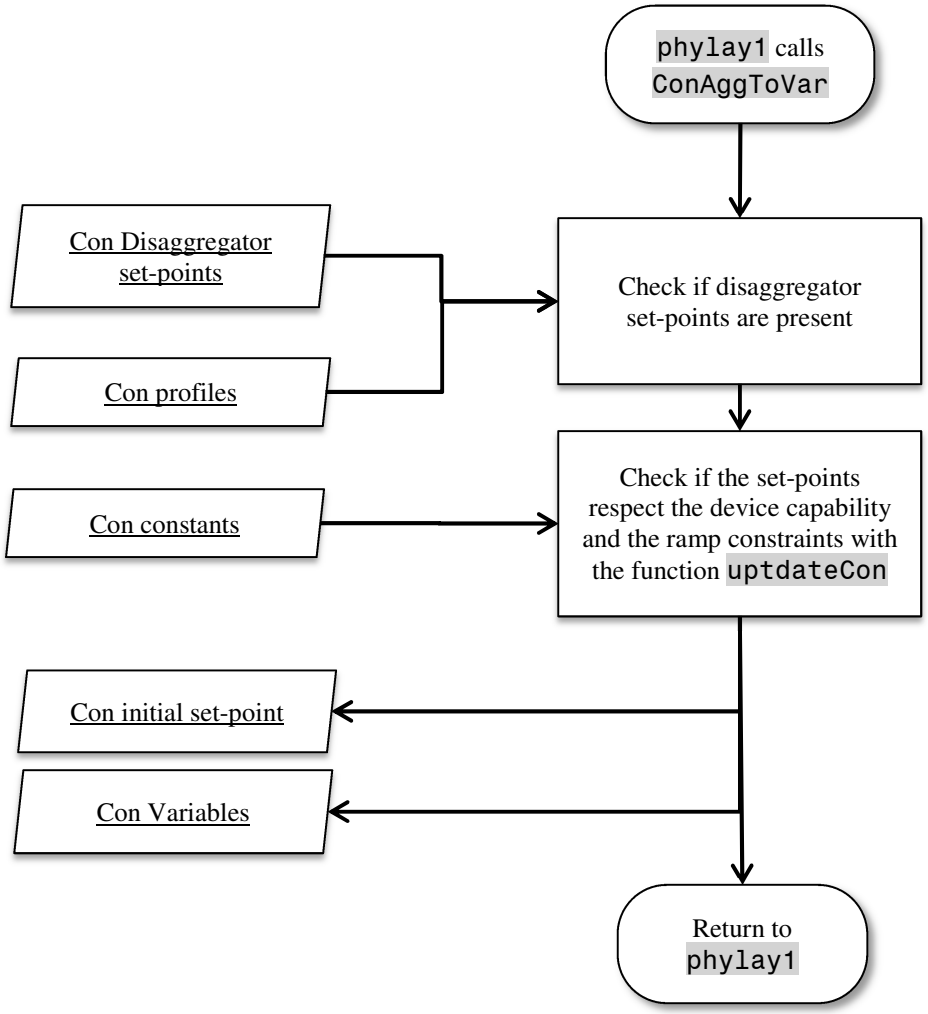


Figure 45 – Flow diagram of PHYLAY 1 – Conventional generators

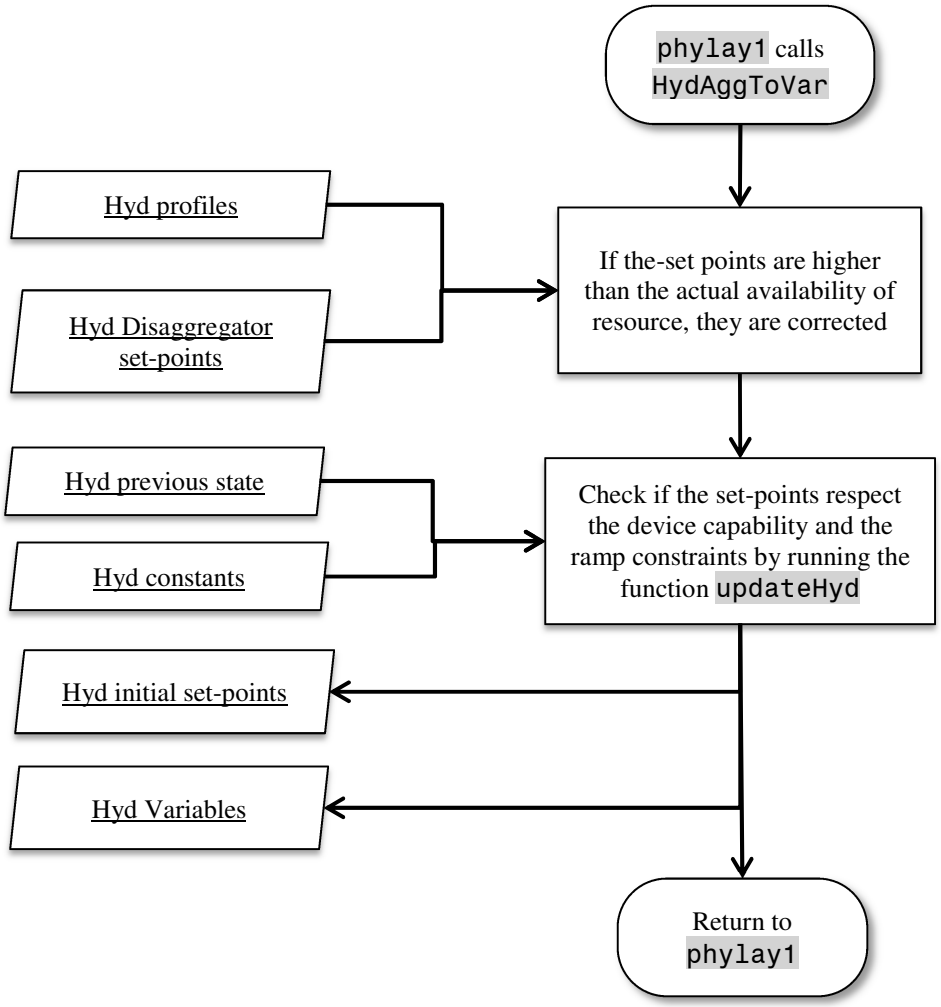


Figure 46 – Flow diagram of PHYLAY 1 – Hydro generators

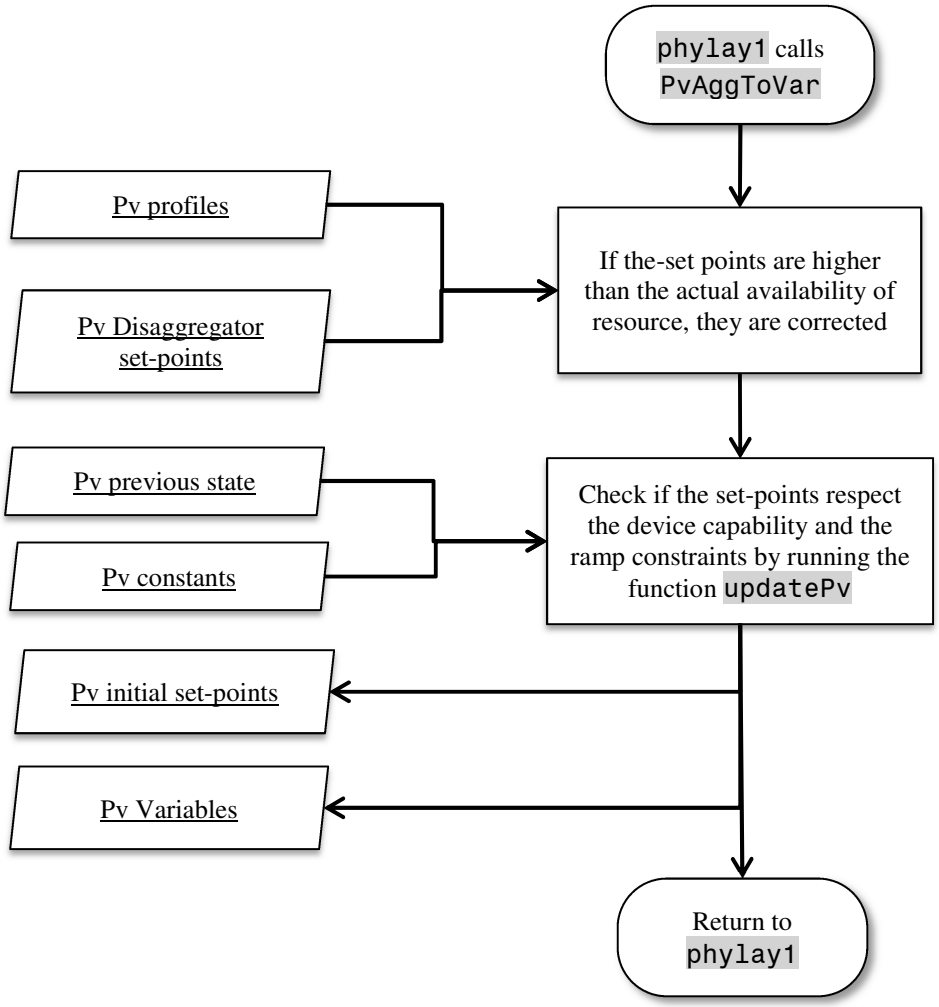


Figure 47 – Flow diagram of PHYLAY 1 – Photovoltaic generators

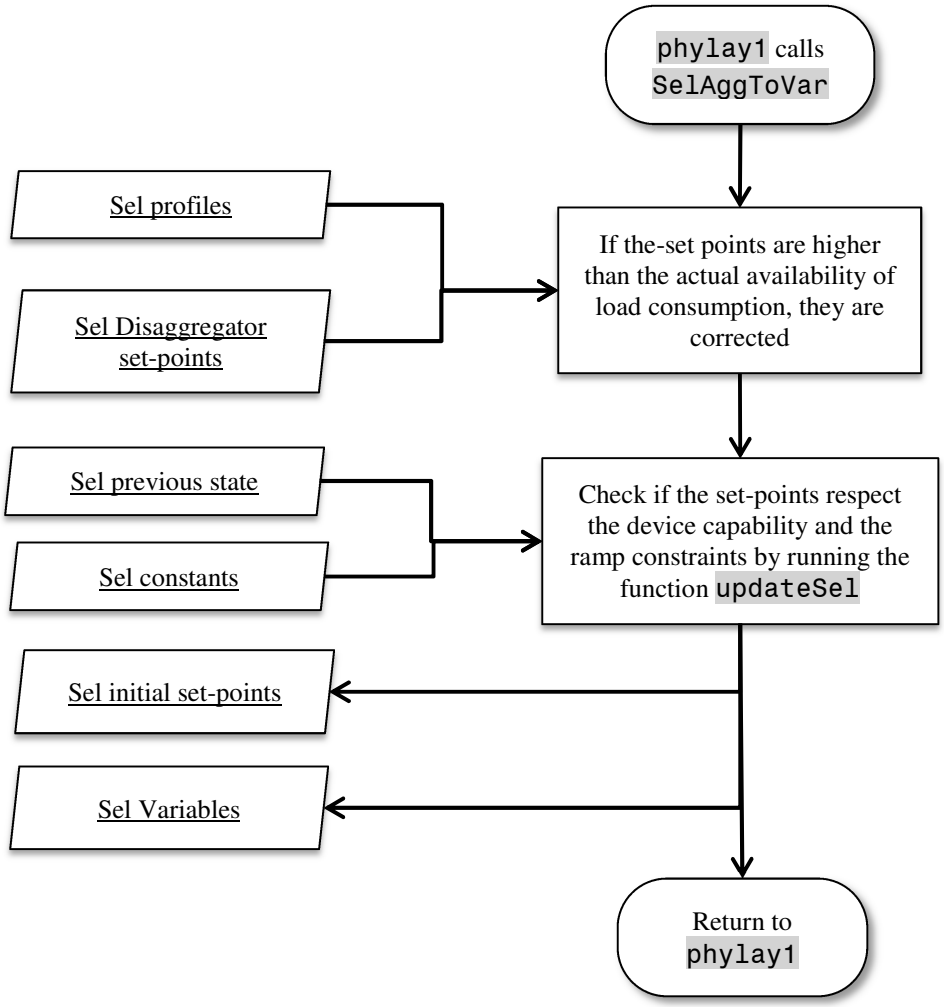


Figure 48 – Flow diagram of PHYLAY 1 – Sheddable loads

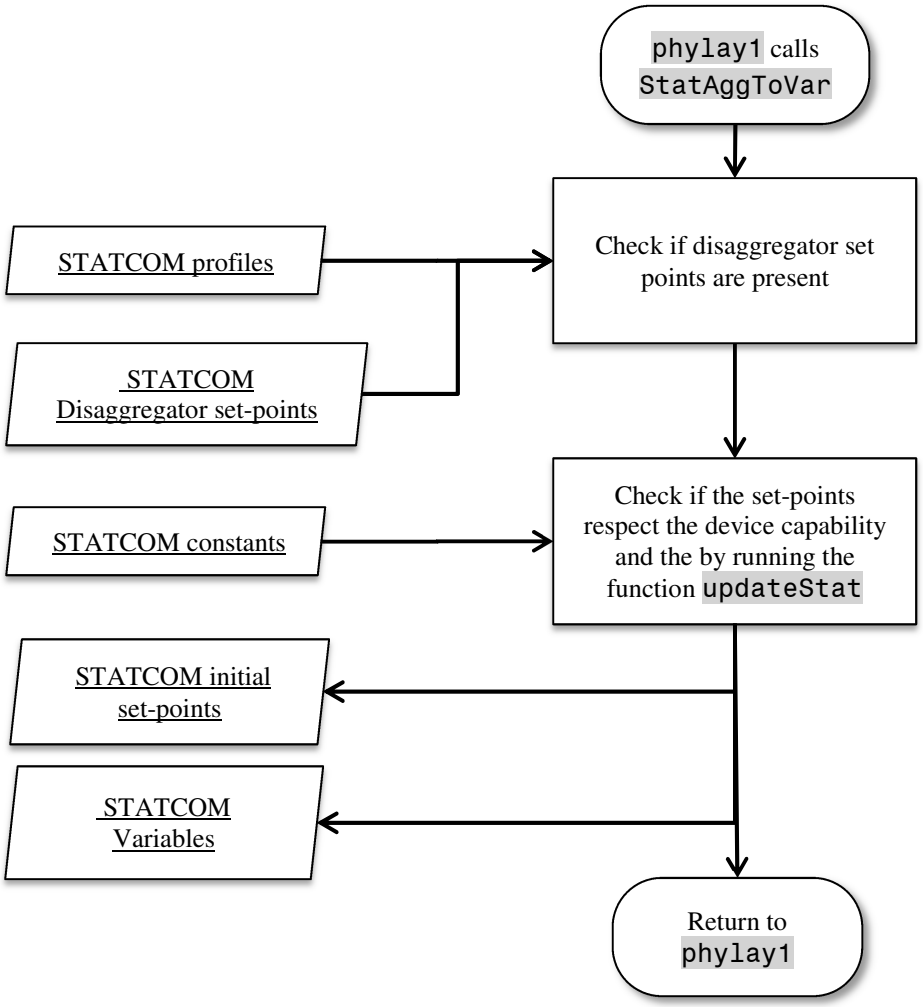


Figure 49 – Flow diagram of PHYLAY 1 – Static compensators

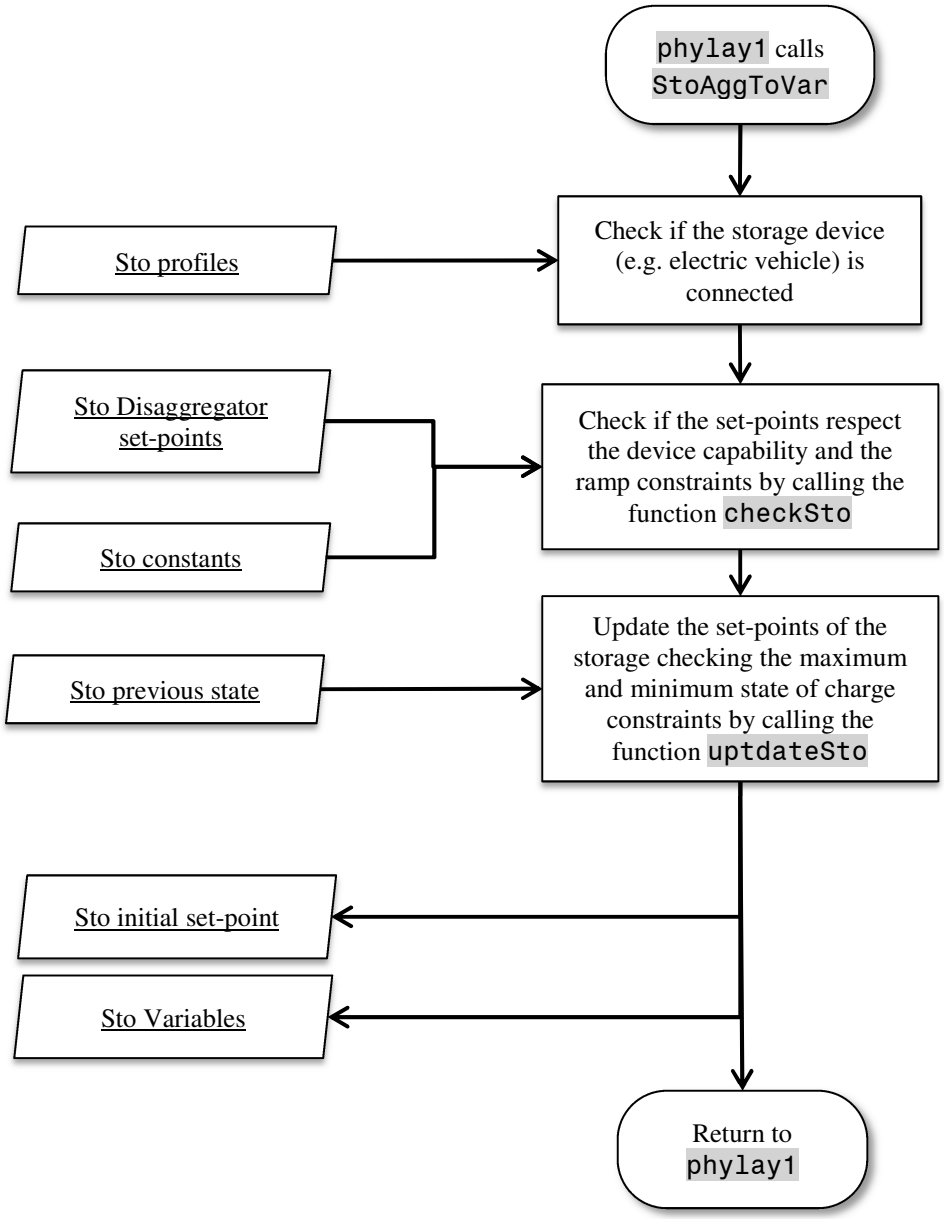


Figure 50 – Flow diagram of PHYLAY 1 – Storage-base devices

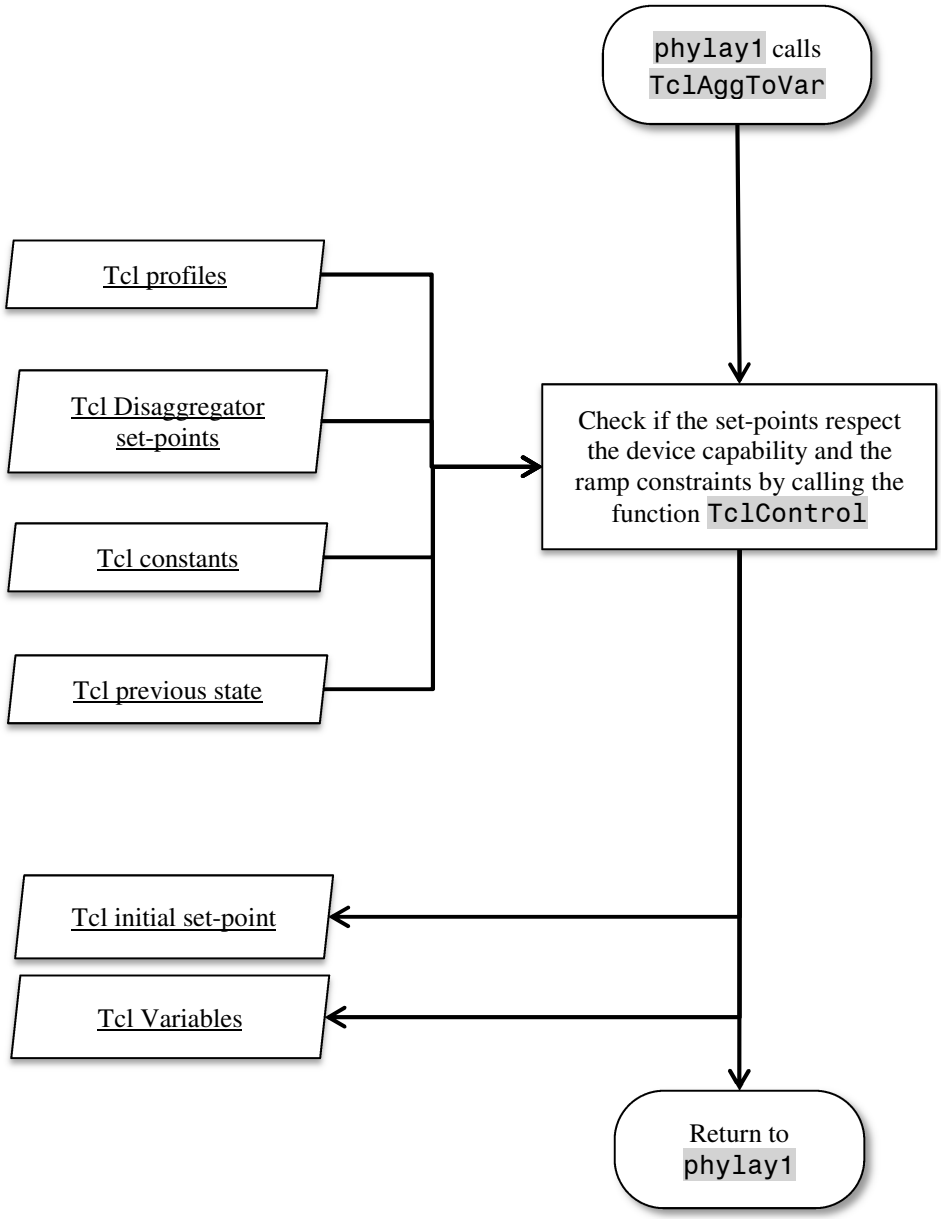


Figure 51 – Flow diagram of PHYLAY 1 – Thermostatically controlled loads

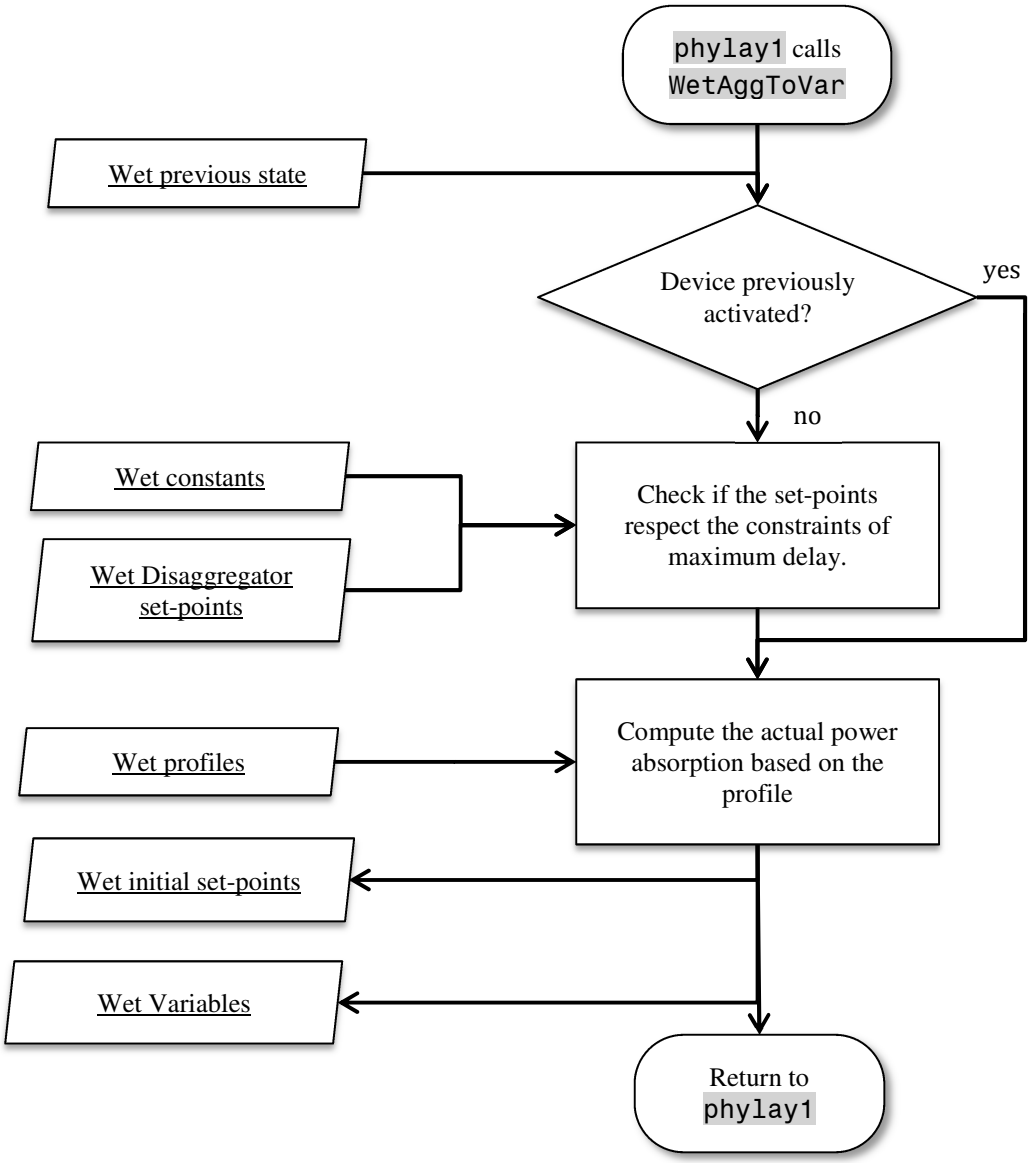


Figure 52 – Flow diagram of PHYLAY 1 – Atomic loads

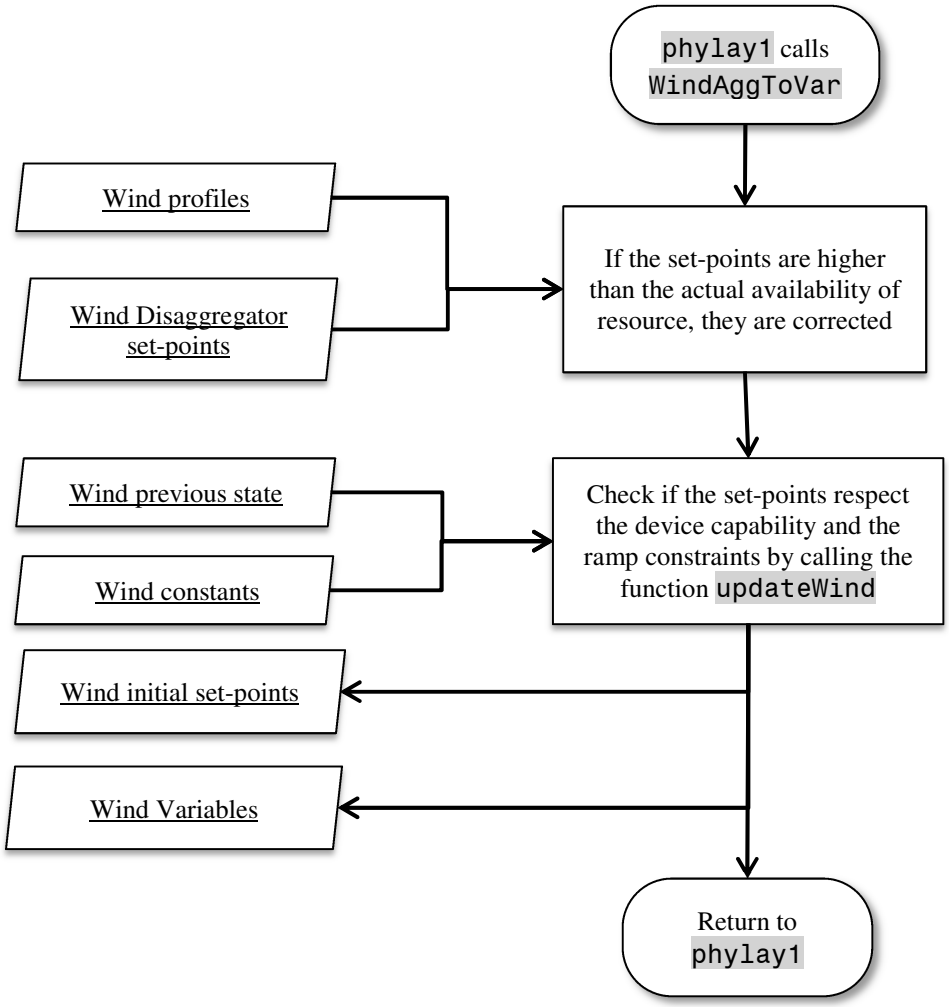


Figure 53 – Flow diagram of PHYLAY 1 – Wind generators

5.5.2 PHYLAY 2

The `phylay2` block is more complex: it computes the electrical variables of the networks and the evolution of the state of devices. At first, distribution networks are simulated by means of an Optimal Power Flow (OPF). The control scenario of the network (e.g. controllable distribution transformers, etc.) is specified by the fields present in the table `ControlSolutionDso`, which can be modified together with the tables reporting the characteristics of devices (e.g. reactive power limit capabilities). According to the flowchart reported in Figure 54, the OPF of distribution network computes the necessary action from the asset of DSO and from the local resources in order to solve possible congestions (if any).

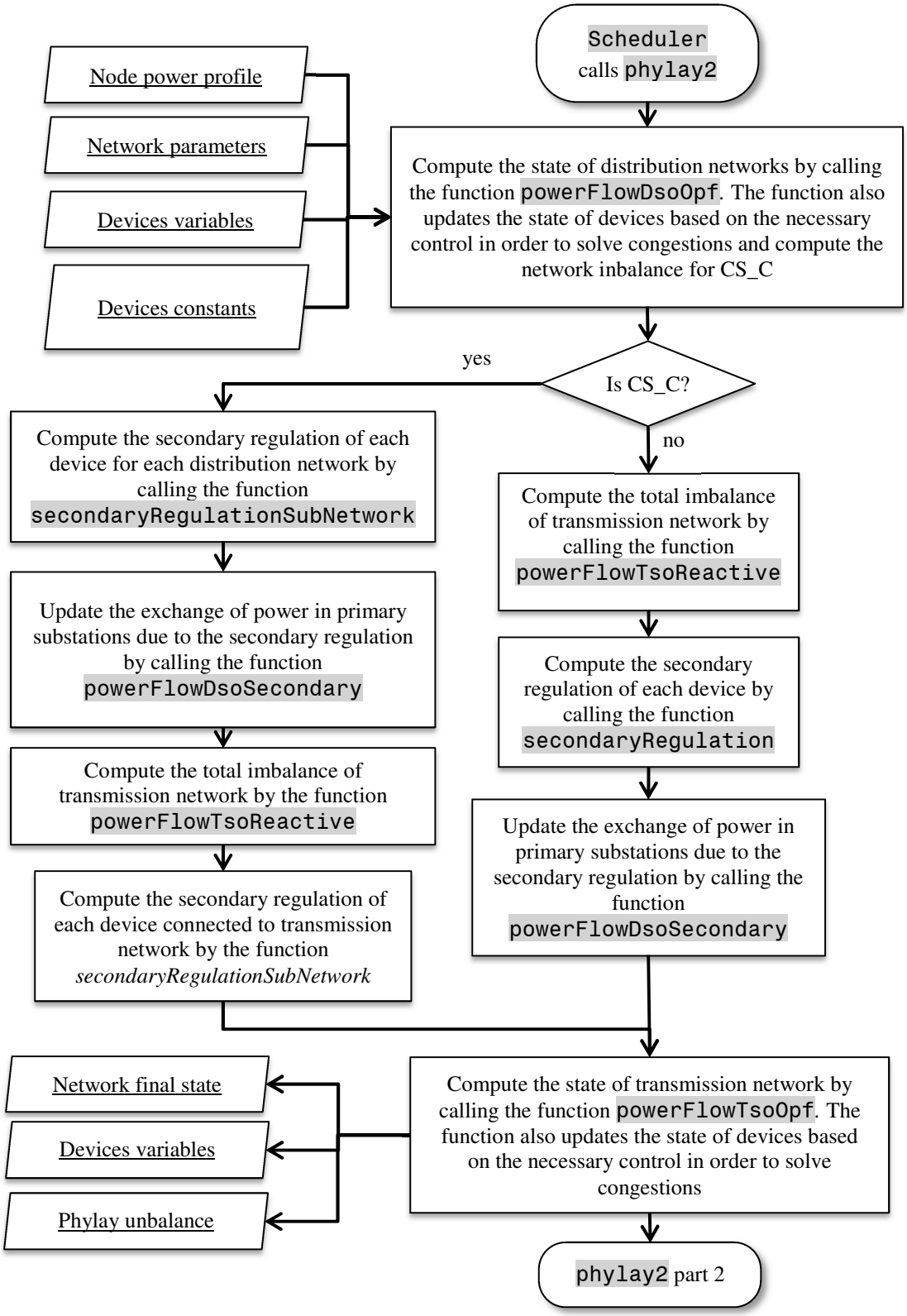


Figure 54 – Flow diagram of PHYLAY 2 – Part 1

After that, the states of devices are updated in order to take into account possible control actions used to solve congestions (Figure 55). These new set-points are decided by a dedicated OPF function in which the flexible elements are constructed in order to make the devices respecting their capability and charge constraints.

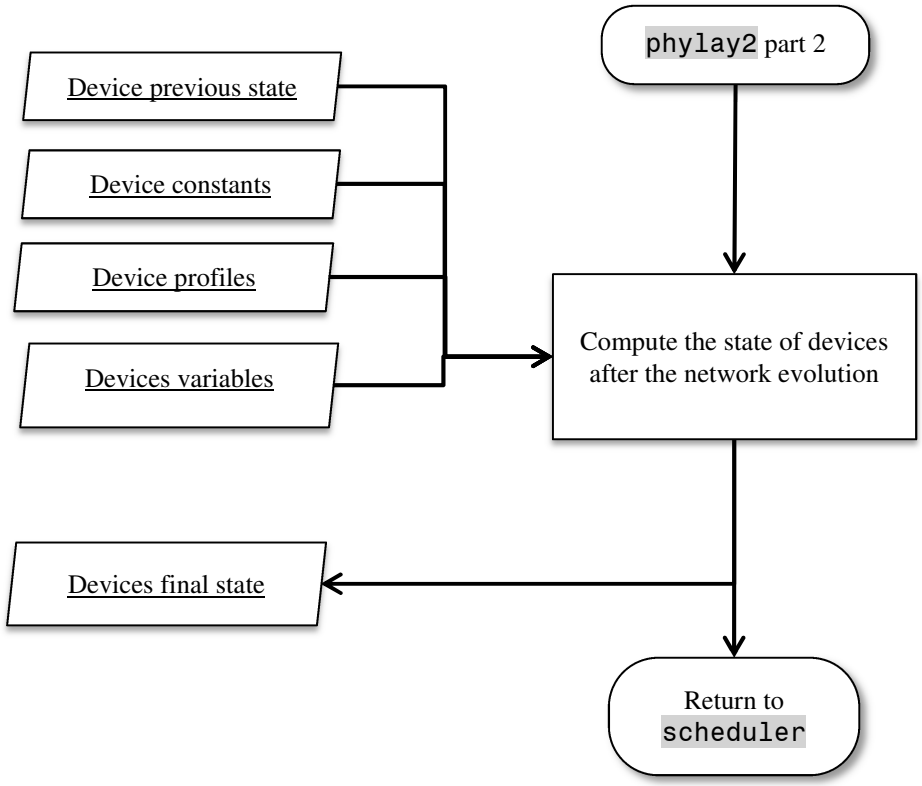


Figure 55 – Flow diagram of PHYLAY 2 – Part 2

The simulation of the operations carried DSO and TSO is performed with similar functions, which are adapted for the specific characteristics of the two types of network. The functions aimed at simulating DSO operations are described in section 0, while the ones carried out by TSO are reported in section 0. In the following section, the individual functions aimed at computing the evolution of the state of devices are then described. These functions take the correction of set-points processed during the network operation and apply them to the device models in order to obtain their new internal states (section 5.5.2.3).

5.5.2.1 Simulation of DSO operations

There are two main functions that allow to compute the state of distribution network in different control configurations:

- `powerFlowDsoOpf`: Optimal Power Flow (OPF) of distribution network with active and reactive power modulation from local resources and the asset of DSO.
- `powerFlowDsoOpfReactive`: Optimal Power Flow (OPF) of distribution network with reactive power modulation from local resources and the asset of DSO.

The structure of the two functions is quite similar. These routines create the electrical structure of networks in the `PyPower` format and they convert the devices in the `PyPower` format for programmable generators. The model of generators take into account the constraints of devices (e.g. maximum power) in modulating active and reactive power. Also the possible asset of DSO (i.e. tap-changing transformers, static compensators, etc.) is taken into account within the optimization model.

The functions can possibly take into account the effects of state estimation error. In this case, two network simulations are carried out. The first one is computing an OPF having as input the state of devices to which an error (representative of state estimation uncertainty) is added. This routine returns the set-points to be applied to actual devices. At this point, a new network simulation is carried out with them.

The flow diagram of the DSO operation simulator is reported in Figure 56 and Figure 57, representing the data preparation and computation steps respectively.

5.5.2.2 Simulation of TSO operations

The functions that compute the state of the transmission network and simulates TSO actions are quite similar to the routines aimed at representing the DSO networks and operations. The two main differences are:

- multiple slack generators are added in correspondence of the networks borders (which simulates neighbouring countries);
- the exchange of power of nodes is computed considering also the exchange with distribution networks.

Figure 58 and Figure 59 reports the flow diagrams illustrating the algorithms adopted for the simulation of transmission network simulation.

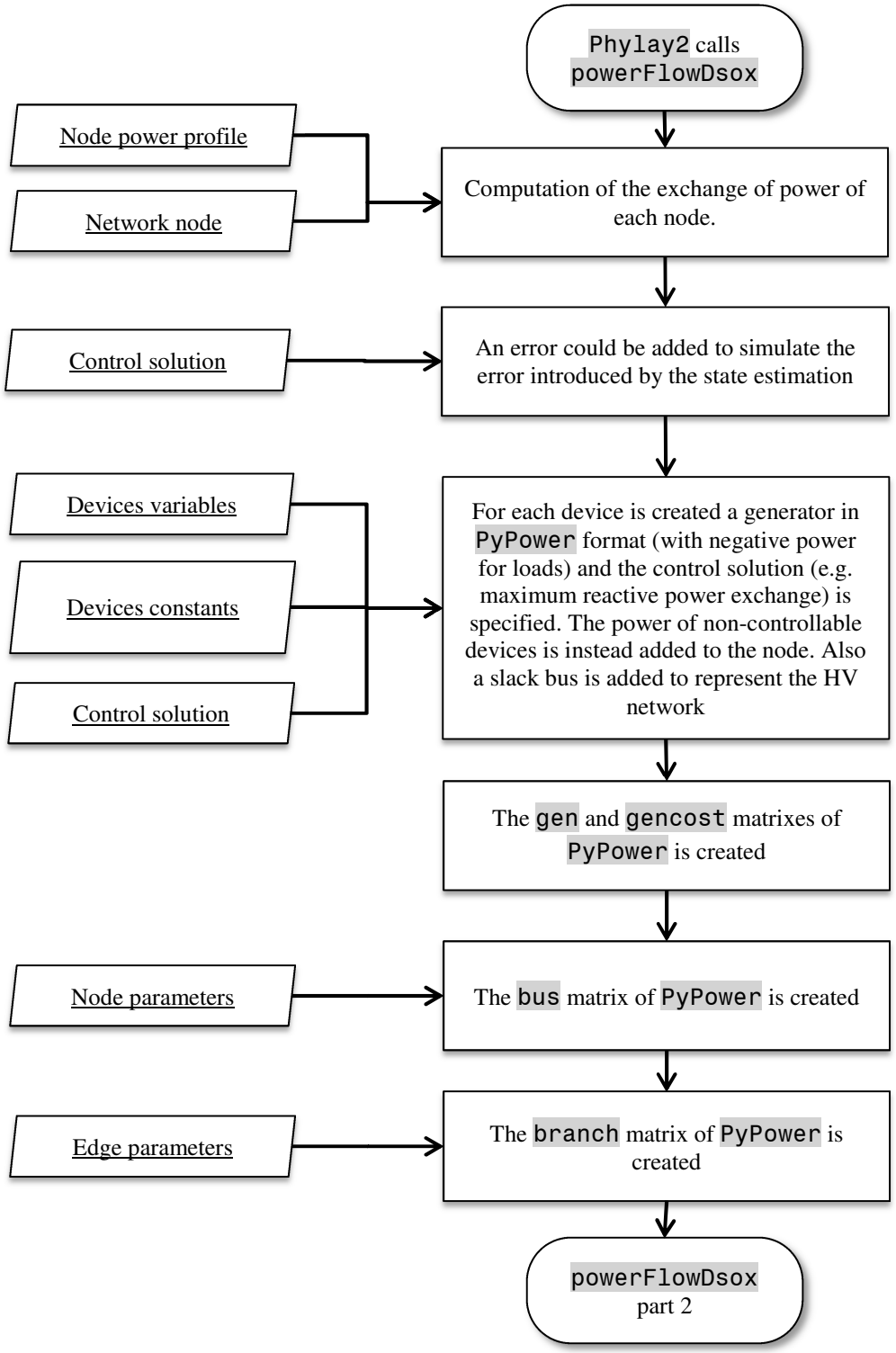


Figure 56 – Flow diagram of `powerFlowDsox` *– data conversion to* `PyPower` *format*

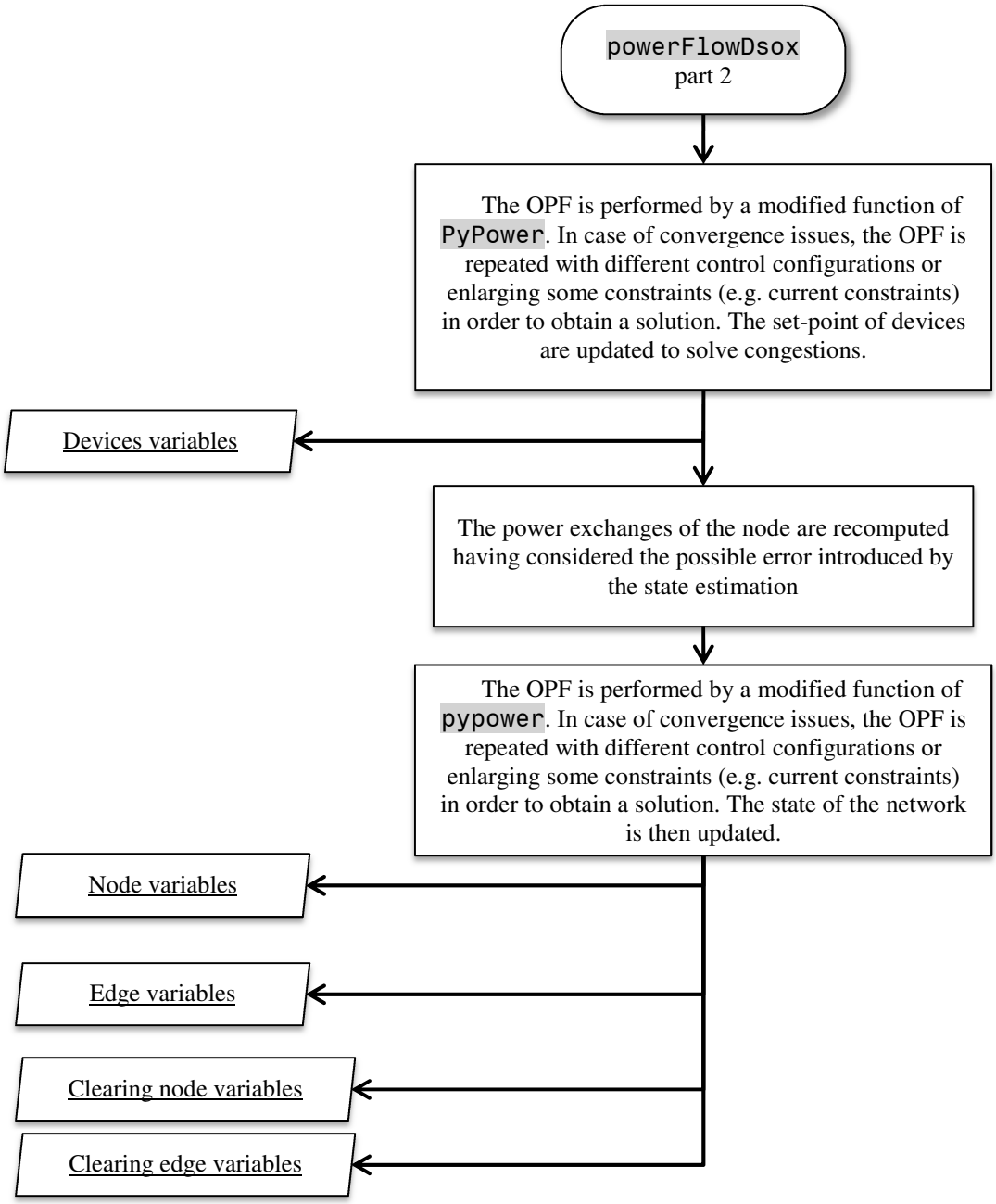


Figure 57 – Flow diagram of `powerFlowDsox` *– distribution network simulation*

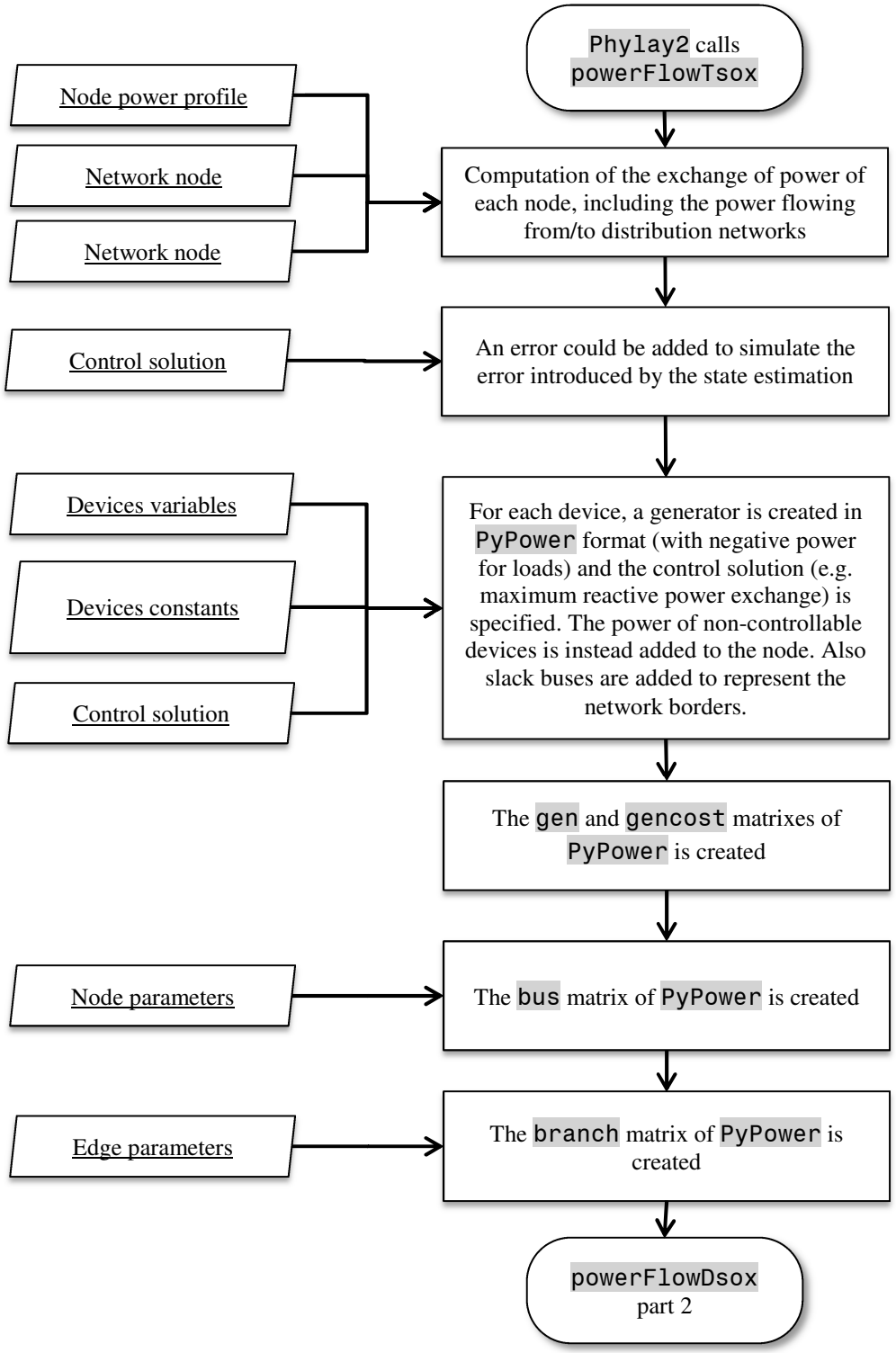


Figure 58 – Flow diagram of `powerFlowTsox` *– data conversion to* `PyPower` *format*

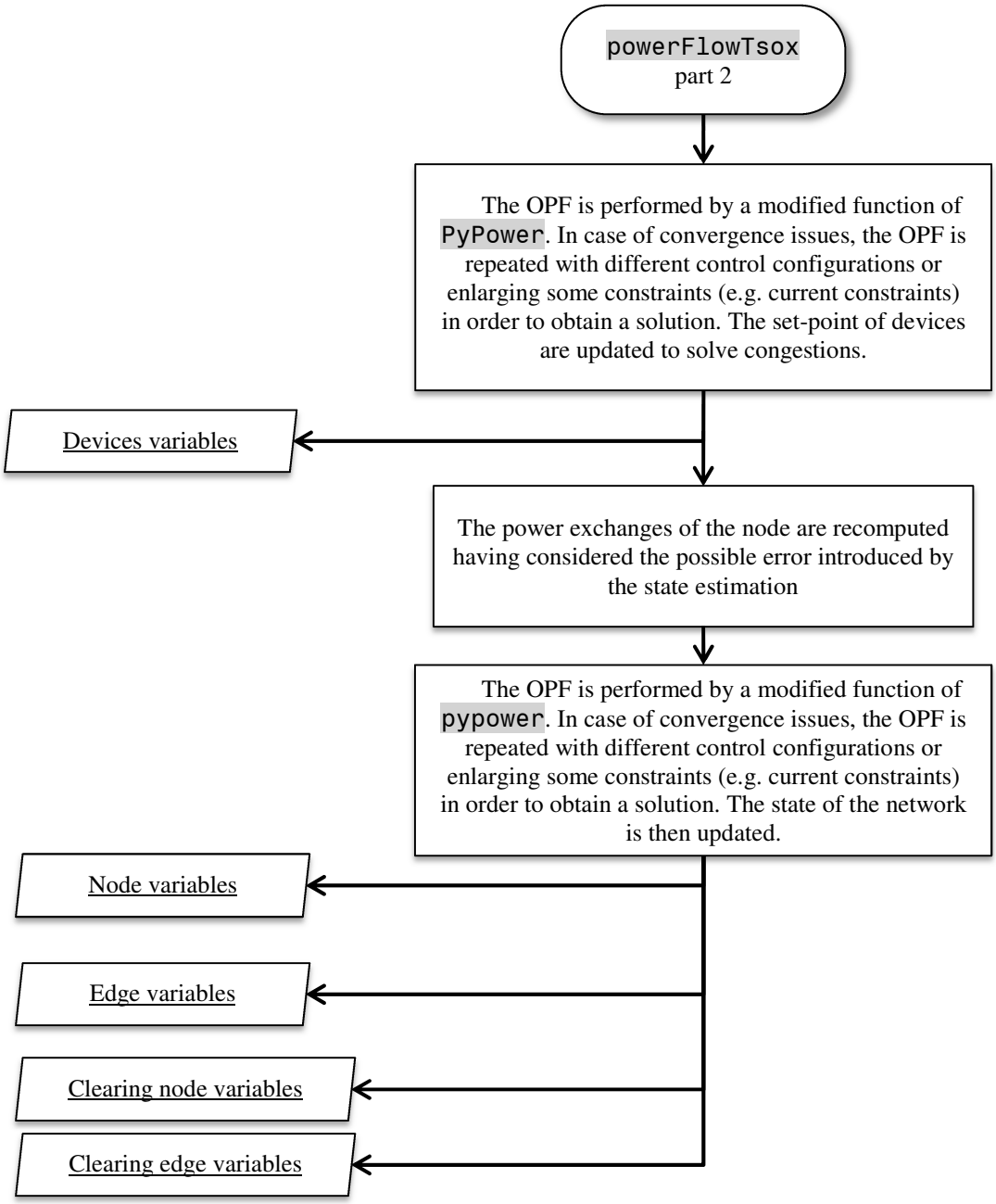


Figure 59 – Flow diagram of `powerFlowTsox` *– transmission network simulation*

5.5.2.3 Updated states of devices

As anticipated above, the new set-points are applied to each single devices according to the algorithm described in Figure 60÷Figure 69. Their applicability is checked by comparing them with the power and state-of-charge capabilities.

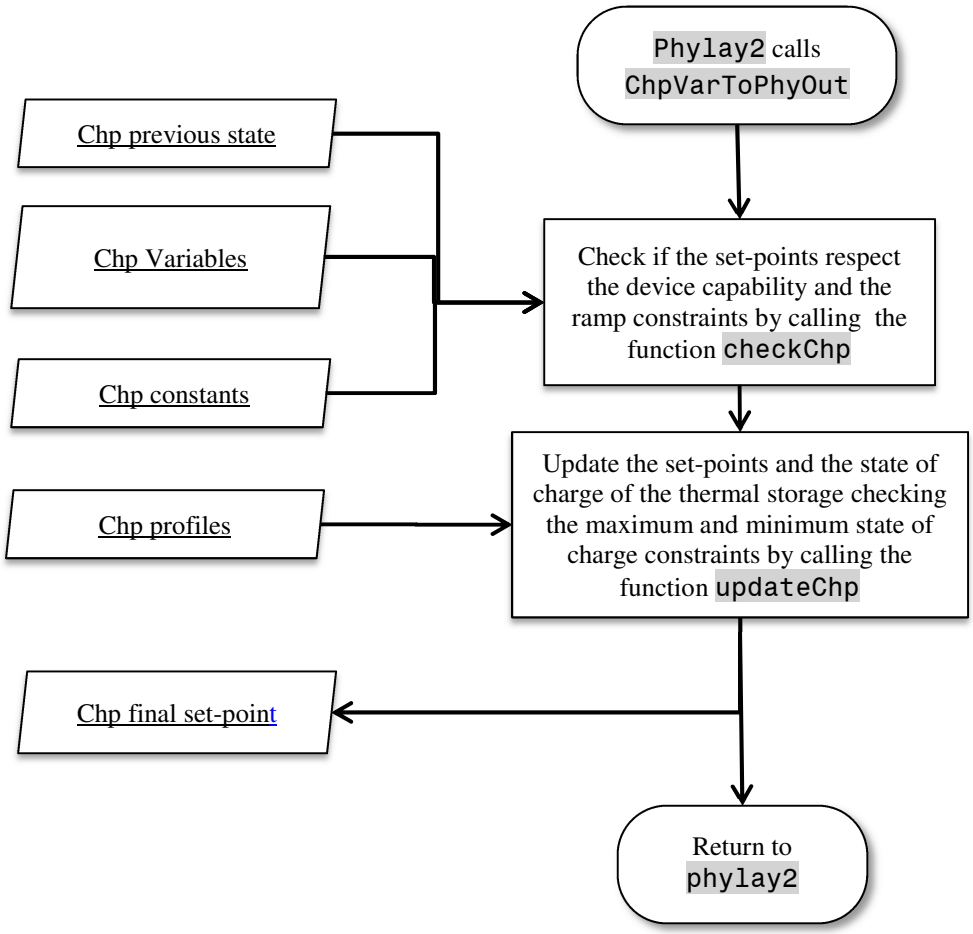


Figure 60 – Flow diagram of PHYLAY 2 – Combined Heat Power devices

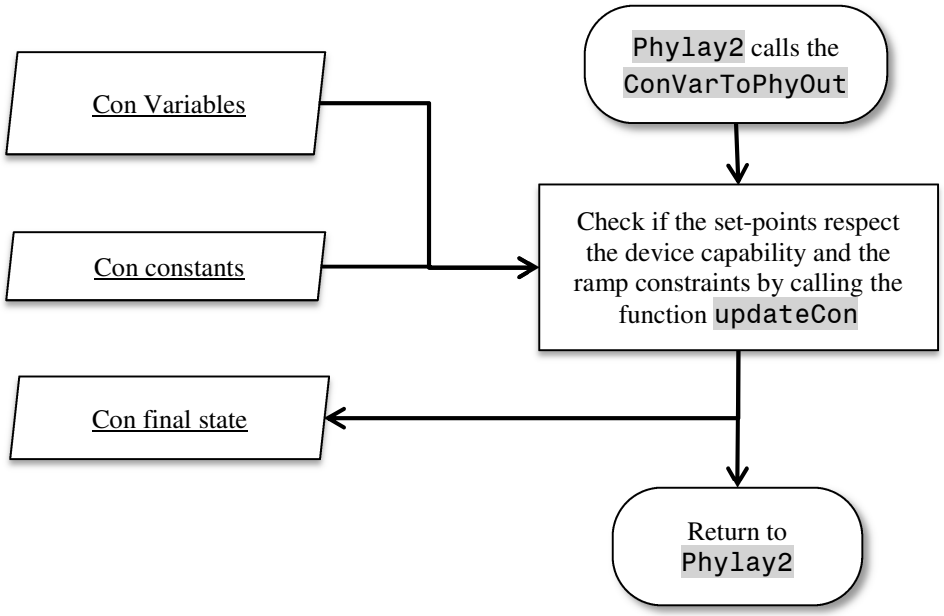


Figure 61 – Flow diagram of PHYLAY 2 – Conventional generators

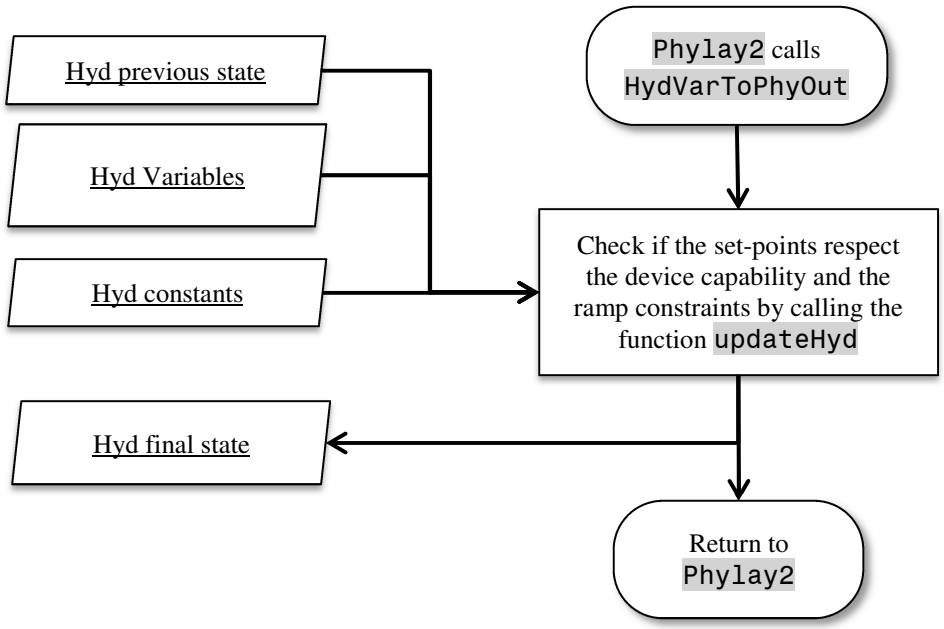


Figure 62 – Flow diagram of PHYLAY 2 – Hydro generators

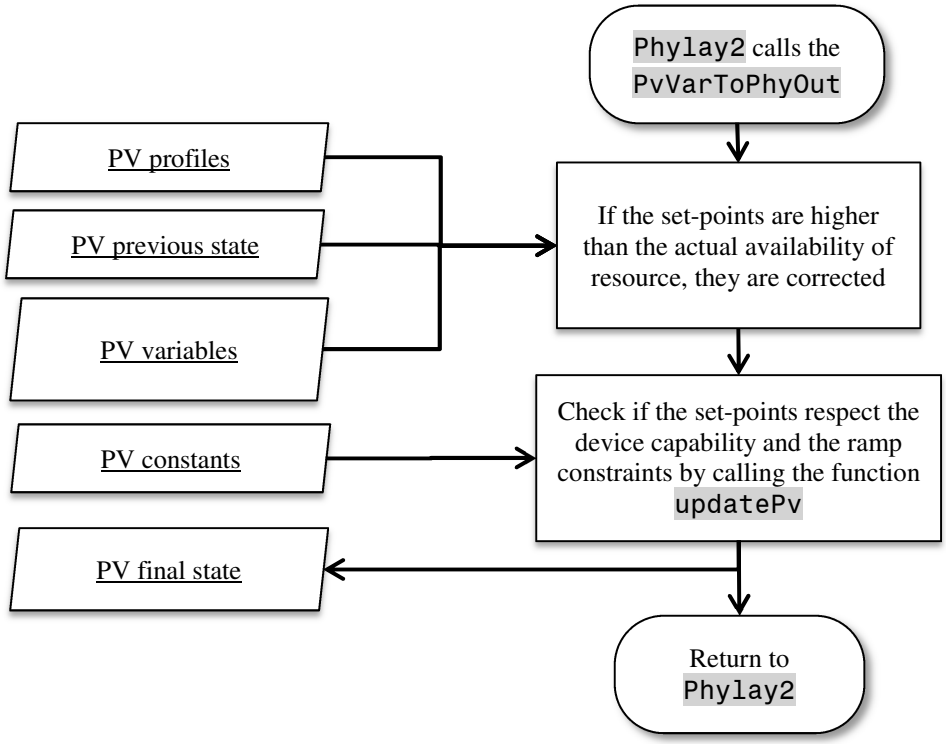


Figure 63 – Flow diagram of PHYLAY 2 – Photovoltaic generators

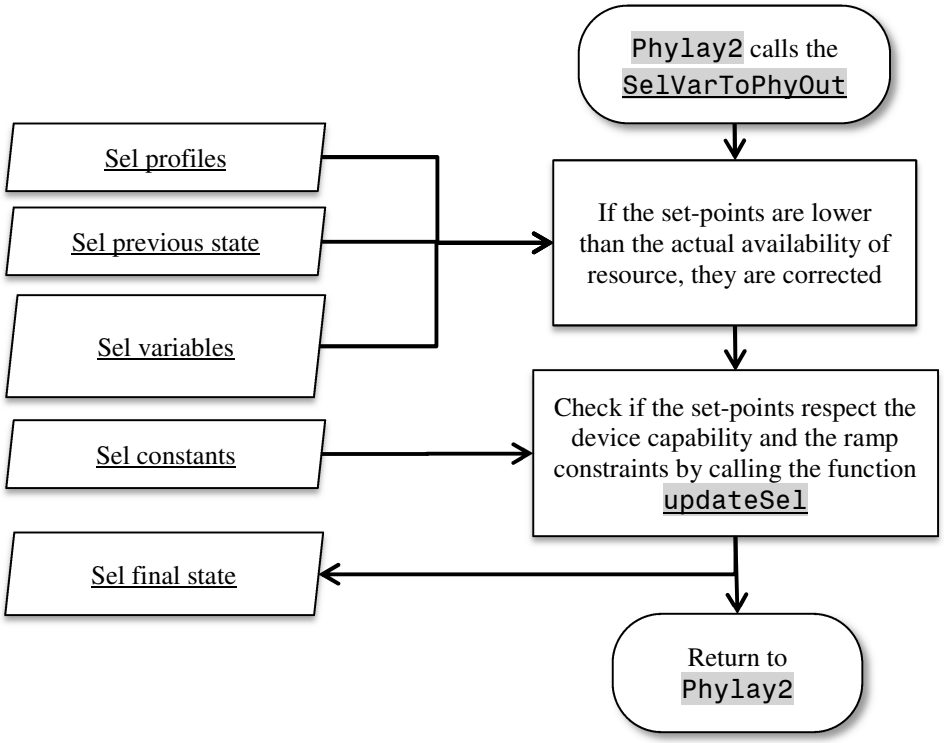


Figure 64 – Flow diagram of PHYLAY 2 – Sheddable loads

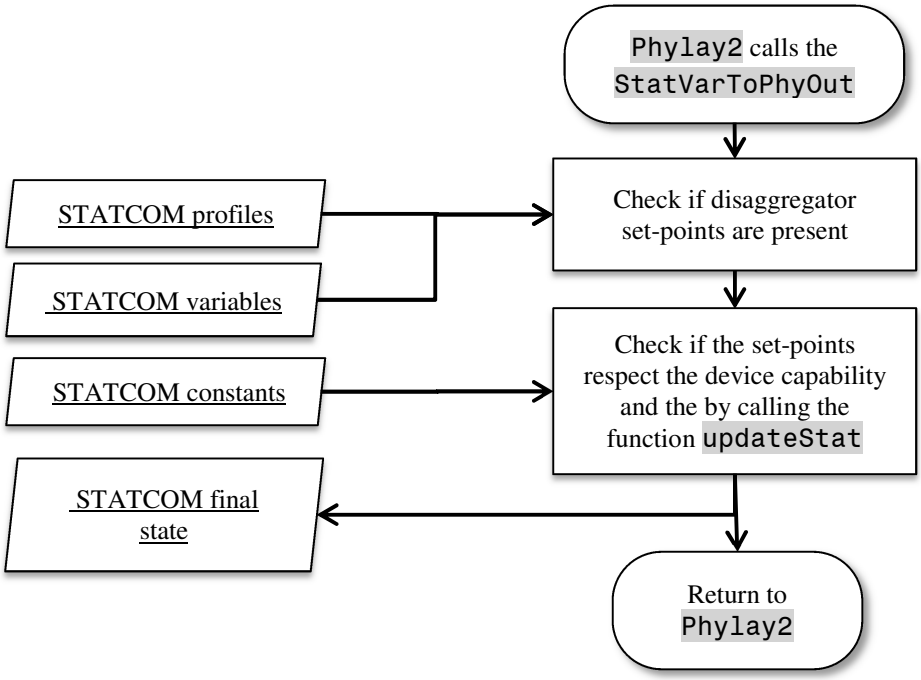


Figure 65 – Flow diagram of PHYLAY 2 – Static compensators

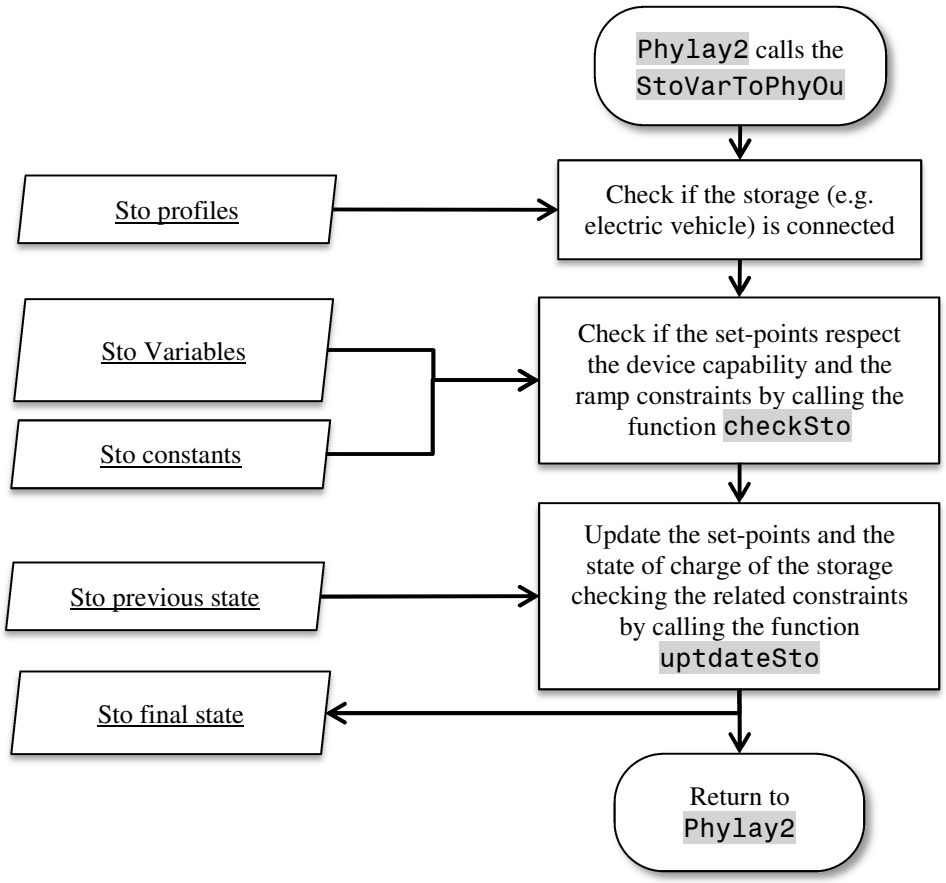


Figure 66 – Flow diagram of PHYLAY 2 – Storage-base devices

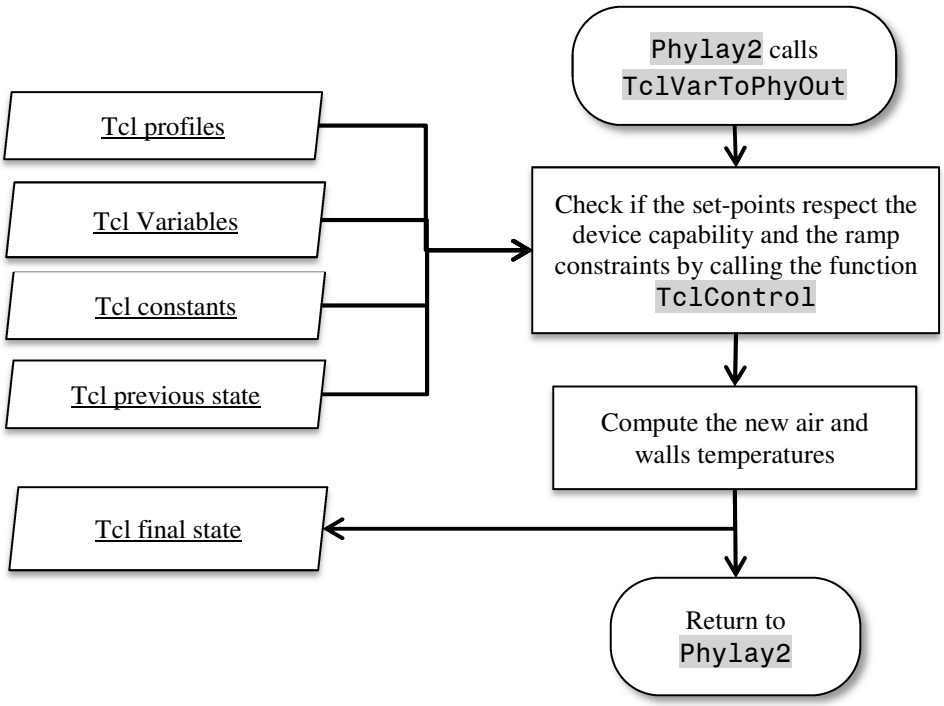


Figure 67 – Flow diagram of PHYLAY 2 – Thermostatically controlled loads

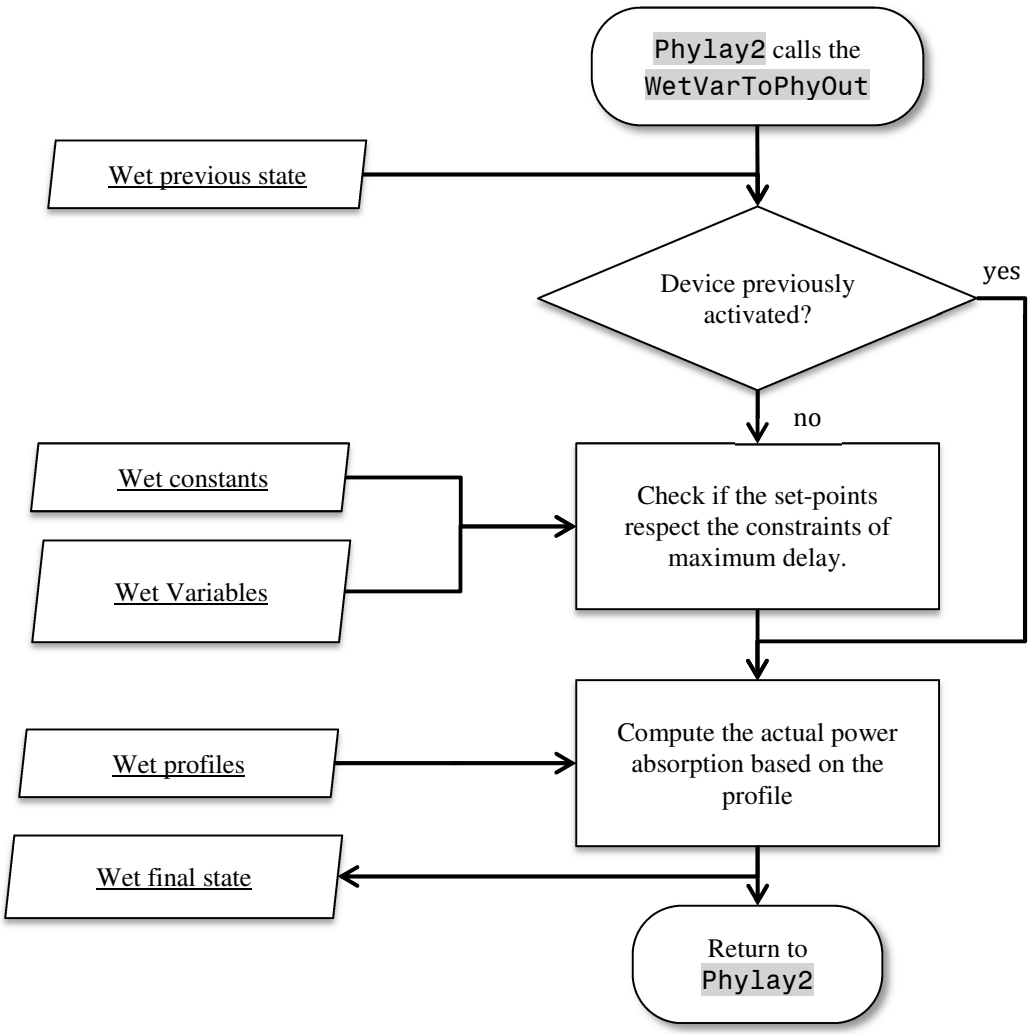


Figure 68 – Flow diagram of PHYLAY 2 – Atomic loads

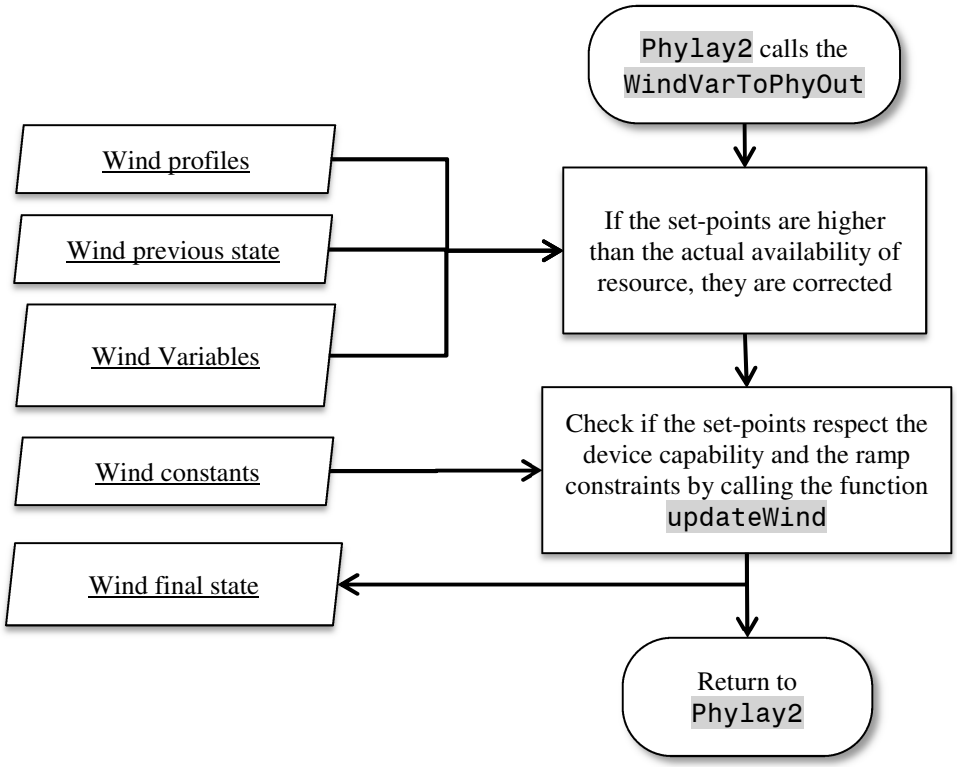


Figure 69 – Flow diagram of PHYLAY 2 – Wind generators

5.6 Output to database

The initial state of devices, which is the results of the application of aggregators set-points, are stored in the following tables:

Initial state of devices

- `devices_setpoints_ChpDevOut`
- `devices_setpoints_ConDevOut`
- `devices_setpoints_HydDevOut`
- `devices_setpoints_PvDevOut`
- `devices_setpoints_SelDevOut`
- `devices_setpoints_StatDevOut`
- `devices_setpoints_StoDevOut`
- `devices_setpoints_TclDevOut`
- `devices_setpoints_WetDevOut`
- `devices_setpoints_WindDevOut`

The final state of devices, resulting from the network evolution (action taken by network operators), are stored in the following tables. They are also used as an input in the next time step iteration because they store the internal state of devices (e.g. state of charge of storages).

Final state of devices

- `phylay_setpoints_ChpPhyOut`
- `phylay_setpoints_ConPhyOut`
- `phylay_setpoints_HydPhyOut`
- `phylay_setpoints_PvPhyOut`
- `phylay_setpoints_SelPhyOut`
- `phylay_setpoints_StatPhyOut`
- `phylay_setpoints_StoPhyOut`
- `phylay_setpoints_TclPhyOut`
- `phylay_setpoints_WetPhyOut`
- `phylay_setpoints_WindPhyOut`

The state of nodes and branches are stored in the following tables. These table are also the output table of the market. The field *writer* (=*phylay/market*) is used to distinguish the two type of output.

Final state of network

- `clearing_NodeVariables`
- `clearing_EdgeVariables`

Finally, some summarizing parameters, which describe the state of each sub-network, are stored in the table:

- `phylay_Unbalances`

6 Database tables

6.1 Devices

These tables refers mainly to devices. The input data used by the devices are divided based on the device typology and they describe all the devices characteristics used both for aggregators and physical layer simulations.

6.1.1 Device Constants:

The table that contain all the constants characteristics of devices are:

- `device_ChpConstants`: Combined Heat Pump parameters
- `device_ConConstants`: Conventional Generators parameters
- `device_HydConstants`: Hydroelectric generators parameters
- `device_PvConstants`: Photovoltaic generators parameters
- `device_SelConstants`: Sheddable loads parameters
- `device_StatConstants`: STATCOM parameters
- `device_StoConstants`: Storage parameters
- `device_TclConstants`: Thermostatically controlled loads parameters
- `device_WetConstants`: Wet appliances (Atomic Loads) parameters
- `device_WindConstants`: Wind generators parameters

6.1.2 Device Profiles

The profiles of the simulated resources are saved in the following tables. For each typology of devices, two tables are provided: one contains the time profile, while the second is used to link the time profile with the corresponding device in the constant table. In this way the same profile can be assigned to multiple devices.

- CHP generators
  - `profiles_ChpPowerProfile`: contains the connection between the related profile and the constants table.
  - `profiles_ChpPower`: power production profiles
  - `profiles_XiDemandHeatProfile`: contains the connection between the related profile and the constants table.
  - `profiles_XiDemandHeat`: thermal load profiles

- Con generators
  - `profiles_ConPowerProfile`: contains the connection between the related profile and the constants table.
  - `profiles_ConPower`: power production profiles
- Hydro generators
  - `profiles_HydPowerProfile`: contains the connection between the related profile and the constants table.
  - `profiles_HydPower`: power production forecast profiles
  - `profiles_HydPowerBaselineProfile`: contains the connection between the related profile and the constants table.
  - `profiles_HydPowerBaseline`: scheduled power production profiles from previous (e.g. intraday) market results
- PV generators
  - `profiles_PvPowerProfile`: contains the connection between the related profile and the constants table.
  - `profiles_PvPower`: power production forecast profiles
  - `profiles_PvPowerBaselineProfile`: contains the connection between the related profile and the constants table.
  - `profiles_PvPowerBaseline`: scheduled power production profiles from previous (e.g. intraday) market results
- Sheddable loads
  - `profiles_SelPowerProfile`: contains the connection between the related profile and the constants table.
  - `profiles_SelPower`: power absorption forecast profiles
- STACOM
  - `profiles_StatPowerProfile`: contains the connection between the related profile and the constants table.
  - `profiles_StatPower`: reactive power absorption forecast profiles
- Storage devices
  - `profiles_StoPowerProfile`: contains the connection between the related profile and the constants table.
  - `profiles_StoPower`: power exchange forecast profiles
- Thermostatically controlled loads
  - `profiles_TclConfortTempProfile`: contains the connection between the related profile and the constants table.
  - `profiles_TclConfortTemp`: comfort temperature profile

- `profiles_TclAvailabilityProfile`: contains the connection between the related profile and the constants table.
- `profiles_TclAvailability`: availability profile of devices (0/1)
- `profiles_TclMaxTempProfile`: contains the connection between the related profile and the constants table.
- `profiles_TclMaxTemp`: maximum temperature profile
- `profiles_TclMinTempProfile`: contains the connection between the related profile and the constants table.
- `profiles_TclMinTemp`: minimum temperature profile
- `profiles_TclInternalThermalGainProfile`: contains the connection between the related profile and the constants table.
- `profiles_TclInternalThermalGain`: thermal power profile produced by other devices injected In the air
- `profiles_TclEnvThermalGainProfile`: contains the connection between the related profile and the constants table.
- `profiles_TclEnvThermalGain`: thermal power profile produced by other devices injected In the wall
- `profiles_ExternalTempProfile`: contains the connection between the related profile and the constants table.
- `profiles_ExternalTemp`: external temperature profile

- Atomic Loads – Wet Appliances
  - `profiles_WetApplianceModel`: contains the connection between the related profile table, the Booting distribution table and the constants table.
  - `profiles_WetApplianceProfile`: power absorption profile
  - `profiles_WetApplianceBootingDistribution`: time step when the devices start its cycle
- Wind generators
  - `profiles_WindPowerProfile`: contains the connection between the related profile and the constants table.
  - `profiles_WindPower`: power production forecast profiles
  - `profiles_WindPowerBaselineProfile`: contains the connection between the related profile and the constants table.
  - `profiles_WindPowerBaseline`: scheduled power production profiles from previous (e.g. intraday) market results

6.1.3 TCL Aggregator internal tables

The TCLs have additional tables used by the related aggregator.

- `aggreg_AggregatorTcl`: contains the id and name of the TCL aggregator
- `aggreg_AvailabilityProfile`: contains the connection between the related profile and the parameter values
- `aggreg_AvailabilityStep`: TCL availability profiles values (0/1)
- `aggreg_BidConfig`: contains configuration parameters to build TCL bids including the number of temperature set points and control durations to be used (control variables)
- `aggreg_ComfTempProfile`: contains the connection between the related profile and the parameter values
- `aggreg_ComfTempStep`: TCL comfort temperature profiles values
- `aggreg_Device`: TCL devices main technical characteristics (nominal power, efficiency, etc.)
- `aggreg_Envelope`: TCLs envelope thermal parameters
- `aggreg_ExtTempProfile`: contains the connection between the related profile and the parameter values
- `aggreg_ExtTempStep`: external temperature profiles values
- `aggreg_ExtTGProfile`: contains the connection between the related profile and the parameter values
- `aggreg_ExtTGStep`: external thermal gains profiles values
- `aggreg_IntTGProfile`: contains the connection between the related profile and the parameter values
- `aggreg_IntTGStep`: internal thermal gains profiles values
- `aggreg_MaxTempProfile`: contains the connection between the related profile and the parameter values
- `aggreg_MaxTempStep`: TCL maximum temperature profiles values
- `aggreg_MinTempProfile`: contains the connection between the related profile and the parameter values
- `aggreg_MinTempStep`: TCL minimum temperature profiles values
- `aggreg_Tcl`: includes a list of all TCLs involved in the simulation and defines the values of all their parameters
- `aggreg_TclStatus`: includes the status of each TCL at the beginning of the simulation (temperature set-point, indoor and envelope temperatures, …). This information is updated from the physical layer module at the beginning of the simulation.
- `aggreg_TimeStep`: time-steps data

- `tcls_FlexProfSet`: set of control durations that can be applied to the TCLs in the portfolio
- `tcls_TempSetPointSet`: set of temperature set-points that can be applied to each TCL in the portfolio
- `aggreg_FlexCalculation`: contains internal information of the TCL Aggregation module related to the simulations of the individual flexibility profiles and their mapping with the `qbidsegments`.
- `aggreg_BidProfile`: contains internal information of the TCL Aggregation module related to the calculation of the `qbidsegments`.
- `aggreg_BidCalculation`: contains internal information of the TCL Aggregation module related to the calculation of the `qbids`.

6.1.4 Disaggregator set points

The third necessary element for the processing of device evolution consists of the set-points from aggregators, which are stored in the following tables.

- `disaggregator_setpoints_ChpAggOut`: set points for Combined Heat Pumps
- `disaggregator_setpoints_ConAggOut`: set points for conventional generators
- `disaggregator_setpoints_HydAggOut`: set points for hydro generators
- `disaggregator_setpoints_PvAggOut`: set points for photovoltaic generators
- `disaggregator_setpoints_SelAggOut`: set points for sheddable loads
- `disaggregator_setpoints_StatAggOut`: set points for STATCOM
- `disaggregator_setpoints_StoAggOut`: set points for storages
- `disaggregator_setpoints_TclAggOut`: set points for thermostatically controlled loads
- `disaggregator_setpoints_WetAggOut`: set points for atomic loads
- `disaggregator_setpoints_WindAggOut`: set points for wind generators

6.1.5 Device and Network Variables

In addition, there are a set of support tables which are used for storing temporary state of devices:

- `phylay_NodeVariables`: contains the state of nodes
- `phylay_EdgeVariables`: contains the state of branches
- `devices_ChpVariables`: contains the state of Combined Heat Pumps
- `devices_ConVariables`: contains the state of conventional generators
- `devices_HydVariables`: contains the state of hydro generators

- `devices_PvVariables`: contains the state of photovoltaic generators
- `devices_SelVariables`: contains the state of sheddable loads
- `devices_StatVariables`: contains the state of STATCOM
- `devices_StoVariables`: contains the state of storages
- `devices_TclVariables`: contains the state of thermostatically controlled loads
- `devices_WetVariables`: contains the state of atomic loads
- `devices_WindVariables`: contains the state of wind generators

6.1.6 Initial state of devices

The initial state of devices, after the application of set-points provided by the disaggregation routines, are stored in the following tables:

- `devices_setpoints_ChpDevOut`: sate of combined heat pumps
- `devices_setpoints_ConDevOut`: sate of conventional generators
- `devices_setpoints_HydDevOut`: sate of hydro generators
- `devices_setpoints_PvDevOut`: sate of photovoltaic generators
- `devices_setpoints_SelDevOut`: sate of sheddable loads
- `devices_setpoints_StatDevOut`: sate of STATCOM
- `devices_setpoints_StoDevOut`: sate of storages
- `devices_setpoints_TclDevOut`: sate of thermostatically controlled loads
- `devices_setpoints_WetDevOut`: sate of atomic loads (wet appliances)
- `devices_setpoints_WindDevOut`: sate of wind generators

6.1.7 Final state of devices

The final state of devices, after the network evolution, are stored in the following tables. They are also used as an input in the next time step iteration because they also store the internal state of devices (e.g. state of charge of storages).

- `phylay_setpoints_ChpPhyOut`: sate of combined heat pumps
- `phylay_setpoints_ConPhyOut`: sate of conventional generators
- `phylay_setpoints_HydPhyOut`: sate of hydro generators
- `phylay_setpoints_PvPhyOut`: sate of photovoltaic generators
- `phylay_setpoints_SelPhyOut`: sate of sheddable loads
- `phylay_setpoints_StatPhyOut`: sate of STATCOM
- `phylay_setpoints_StoPhyOut`: sate of storages
- `phylay_setpoints_TclPhyOut`: sate of thermostatically controlled loads

- `phylay_setpoints_WetPhyOut`: sate of atomic loads (wet appliances)
- `phylay_setpoints_WindPhyOut`: sate of wind generators

6.2 Network Model

The market and physical layers need to know the network description to be able to model the power flows of the lines as a function of the power exchange of the nodes. This includes the network topology and electrical parameters of lines, the be able to calculate active, reactive power flows and losses. The network model is described here as an input to the market block, but is also an input for the other simulation layers.

6.2.1 Network parameters

The network model is composed by the following fields, which are all read by the market block.

- `Network.models.Network`: We consider a single network per simulation in SmartNet. Since we study interactions between DSO and TSO, their respective networks are both modelled as '`SubNetworks`' of this one Network. In SmartNet, we consider only scenario's where the Network has at most one Transmission `SubNetwork` and where the Network can have zero or any positive number of Distribution Networks.
- `Network.models.SubNetwork`: Subnetworks are of type 'D' (distribution) or type 'T' (transmission). A `SubNetwork` mentions in its network field that it is part of the Network. In SmartNet, a distribution `SubNetwork` is radial. A transmission network can be radial or meshed. For Distribution networks we have functions to automatically derive the root node. (For Transmission networks of course the root node does not generally exist.)
- `network.models.Node`: The Node in a network is where lines connect. It's also where power is injected or off-taken. The Node refers in its subnetwork field to the `SubNetwork` it belongs to.
- `network.models.Edge`. Edges model the lines in the physical grid. They describe the connections along which power flows from node to node. An `Edge` has a `fromNode` and an `uptoNode`. An Edge can belong to one `SubNetwork` if both `fromNode` and `uptoNode` belong to the same Network or crosses `SubNetwork` boundaries otherwise.
- `network.models.NodeConstants`: Node-constants describe the physical parameters of the network that are node related. `NodeConstants` refer in node to the `Node` the constants are for. We separated the `NodeConstants` from its Node to be able to change the list of `NodeConstants`, while not having to change anything in the `Node` class. The list of used `NodeConstants` parameters is:

  - `node`: node the `NodeConstants` refer to
  - `nodeBusType`: (1=PQ, 2= PV, 3 = reference, 4 =isolated)
  - `nodeBoundary`: (0 = not a boundary node, 1 = is a boundary node)
  - `s_sh`: nodal shunt conductance (real part of shunt admittance `y_sh`, [UNIT:S])
  - `b_sh`: nodal shunt susceptance (imaginary part of shunt admittance `y_sh`, [UNIT :S])
  - `U_nom`: nominal voltage [UNIT:kV]
  - `U_min`: minimal voltage [UNTI:kV]
  - `U_max`: maximal voltage [UNIT:kV]
  - `P_min`: minimal active power [UNTI:MW]
  - `P_max`: maximal active power [UNIT:MW]
  - `Q_min`: minimal reactive power [UNIT:MW]
  - `Q_max`: maximal reactive power [UNIT:MW]
  - `S_injectionMin`: maximum apparent power [UNIT:MW] Only used for debugging.
  - `S_offtakeMax`: maximum apparent power [UNIT:MW] Only used for debugging.
  - `g_sh_pu`: series resistance (per unit) real part of the series impedance `z_l_s` [UNIT:Ohm]
  - `b_sh_pu`: series susceptance (per unit) real part of the nodal shunt admittance `y_sh` [UNIT:S]
  - function `translateToPu()` translates all constants to per unit values.
- `network.models.EdgeConstants`: Edge-constants describe the physical parameters of the network that are edge related. `EdgeConstants` refer in edge to the `Edge` the constants are for. We separated the `EdgeConstants` from its Edge to be able to change the list of `EdgeConstants`, while not having to change anything in the `Edge` class. The list of used `EdgeConstants` parameters is:
  - `r_l_s`: series resistance (real part of series impedance `z_l_s`) [UNIT:Ohm]
  - `x_l_s`: series reactance (imaginary part of series impedance `z_l_s`) [UNIT:Ohm]
  - `g_ij_sh`: conductance (real part of the admittance `y_ij_sh`) of the i side shunt [UNIT:S]
  - `b_ij_sh`: susceptance (imaginary part of the admittance `y_ij_sh`) of the i side shunt [UNIT:S]
  - `g_ji_sh`: conductance (real part of the admittance `y_ji_sh`) of the j side shunt [UNIT:S]
  - `b_ji_sh`: susceptance (imaginary part of the admittance `y_ji_sh`) of the j side shunt [UNIT:S]

- r_l__pu: per unit series resistance (real part of series impedance z_l_s) [UNIT:Ohm]
- x_l_s_pu: per unit series reactance (imaginary part of series impedance z_l_s) [UNIT:Ohm]
- g_ij_sh_pu: per unit conductance (real part of the admittance y_ij_sh) of the i side shunt [UNIT:S]
- b_ij_sh_pu: per unit susceptance (imaginary part of the admittance y_ij_sh) of the i side shunt [UNIT:S]
- g_ji_sh_pu: per unit conductance (real part of the admittance y_ji_sh) of the j side shunt [UNIT:S]
- b_ji_sh_pu: per unit susceptance (imaginary part of the admittance y_ji_sh) of the j side shunt [UNIT:S]

- U_ij_sh_rated: rated max voltage amplitude difference between on line from node i to node j [UNIT:kV]
- U_ji_sh_rated: rated max voltage amplitude difference between on line from node j to node i [UNIT:kV]
- I_ij_sh_rated: rated max current on line from node i to node j [UNIT:A]
- I_ji_sh_rated: rated max current on line from node i to node j [UNIT:A]
- S_ij_sh_rated: rated max om power amplitude difference on line from node i to node j [UNIT:MW]
- S_ji_sh_rated: rated max om power amplitude difference on line from node j to node i [UNIT:MW]
- S_ij_sh_rated: rated max om power amplitude difference on line from node i to node j, used by the physical layer [UNIT:MW]

- a_ij_min: rated min on tap changing ratio of transformer on line from i to j [UNIT:1]
- a_ij_max: rated max on tap changing ratio of transformer on line from i to j [UNIT:1]
- nTaps_ij: number of taps on tap changing transformer on line from i to j [UNIT:1]
- a_ji_min: rated min on tap changing ratio of transformer on line from j to i [UNIT:1]
- a_ji_max: rated max on tap changing ratio of transformer on line from j to i [UNIT:1]

  - `nTaps_ij`: number of taps on tap changing transformer on line from j to i [UNIT:1]

  - `phi_ij_min`: min on angle range of phase shifter on line from i to j [UNIT:rad]
  - `phi_ij_max`: max on angle range of phase shifter on line from i to j [UNIT:rad]
  - `phi_ji_min`: min on angle range of phase shifter on line from j to i [UNIT:rad]
  - `phi_ji_max`: max on angle range of phase shifter on line from j to i [UNIT:rad]

- `network_BusType`: different type of node (PV, PQ…)
- `network_SubNetworkType`: type of sub-networks (transmission or distribution)
- `phylay_ControlSolutionDso`: contains information about how a network is operated (e.g. use of OLTC…)
- `phylay_ControlSolutionTso`: contains information about how a network is operated (e.g. use of OLTC…)

6.2.2 Network Variables

In addition there are a set of support tables are used for storing temporary state of networks:

- `phylay_NodeVariables`: contains the state of nodes
- `phylay_EdgeVariables`: contains the state of branches

6.2.3 Final state of network

Some summary parameters that describe the state of each sub-network are stored in the table:

- `clearing.models.NodeVariables`:
  - `scenario`: simulation scenario the other fields in this table refer to
  - `writer`: `'market'` or `'phylay'` because only those two process blocks write to this storage block
  - `node`: node the other fields in this table refer to
  - `atT`: `atT` the other fields in this table refer to [UNIT:1]
  - `forT`: `forT` the other fields in this table refer to [UNIT:1]
  - `vsq`: value of voltage squared obtained for (`scenario`, `wroter`, `node`, `atT`, `forT`) [UNIT:kV^2]
  - `v`: value of voltage obtained for (`scenario`, `writer`, `node`, `atT`, `forT`) [UNIT:kV]
  - `acceptedActivePower`: sum of all accepted active power over all `QBids` obtained for (`scenario`, `writer`, `node`, `atT`, `forT`) [UNIT:MW]

- `acceptedReactivePower`: sum of all accepted reactive power over all `QBids` for (`scenario`, `writer`, `node`, `atT`, `forT`) [UNIT:MW]
- `clearedPriceActivePower`: nodal price for active power for (`scenario`, `writer`, `node`, `atT`, `forT`) [UNIT:EUR/MW]
- `clearedPriceReactivePower`: nodal price for reactive power (usually around some % of active power price) [UNIT:EUR/MW]
- `vAngle`: voltage phasor angle [UNIT:rad]
- `vReal`: voltage phasor real part [UNIT:1]
- `vImaginary`: voltage phasor imaginary part [UNIT:1]

- `clearing.models.EdgeVariables`:
  - `scenario`: simulation scenario the other fields in this table refer to
  - `writer`: `'market'` or `'phylay'` because only those two process blocks write to this storage block
  - `edge`: edge the other fields in this table refer to
  - `atT`: `atT` the other fields in this table refer to [UNIT:1]
  - `forT`: `forT` the other fields in this table refer to [UNIT:1]
  - `theta_ij`: voltage angle difference between node i and node j [UNIT:rad]
  - `P_ij_i`: active power flowing from i to j, measured at i [UNIT:W]
  - `Q_ij_i`: reactive power flowing from i to j, measured at i [UNIT:W]
  - `P_ij_j`: active power flowing from i to j, measured at j (not the same as `P_ij_i` if lossy) [UNIT:W]
  - `Q_ij_j`: reactive power flowing from i to j, measured at j (not the same as `Q_ij_i` if lossy) [UNIT:W]
  - `a_ij`: transformer ratio: `a_ij = U_i' / U_i` [UNIT:1]
  - `a_ji`: transformer ratio: `a_ji = U_j' / U_j` [UNIT:1]
  - `csq_ij`: square of edge current in A^2 [UNIT:A^2]
  - `c_ij`: edge current in A [UNIT:A]
  - `c_ij_Angle`: (edge) voltage phasor angle (difference) in radians [UNIT:rad]
  - `c_ij_Angle_Real`: real part of `c_ij_Angle` [UNIT:1]
  - `c_ij_Angle`: Imaginary part of `c_ij_Angle` [UNIT:1]
- `phylay_Unbalances`: contains summary parameters that describe the state of each sub-network

6.3 Market tables

The following tables are related to the market module interactions with the database. These tables are written by the aggregators in order to submit the bids to the market and, after the execution of the clearing functions, they are read by the other blocks (disaggregators and physical layer) in order to apply the market directives to the physical devices and network..

6.3.1 Market Bids:

Bid related inputs are the following.

- bids.models.QBidSegment: A QBidSegment is just a pair of quantities (quantity0, quantity1) and a related pair of prices (price0, price1). The sign/direction-correspondence-convention here is that quantities in the bid correspond to injection into the grid (node) and prices in the bid correspond to what is asked to be paid by the bidder from the market to the bidder. But both quantities and prices can be positive and negative in general. This allows all four (quantity-sign, price-sign combinations).
    - `quantity0`: described above
    - `quantity1`: described above
    - `price0`: described above
    - `price1`: described above
    - `qBid`: the `QBid` that this segment is part of (see below for `QBid`)
    - `segmentIndex`: an integer that is positive for positive quantities and negative for negative quantities. It indicates the order of segments for the case that bidders directly bid complete merit order curves. This allowed direct curve visualization per bidder. However, bidders can bid al their bids as separate segments, all belonging to a different `QBid` as well.

- `bids.models.QBid`: A `QBid` is a collection of one or more `QBidSegments`. This is realized by `QBidSegments` referring to their `QBid` in the `QBidSegment` field.
    - `qtBid`: the `QBid` the `QBid` is part of
    - `relForT`: the relative time that the `QBid` is for. The absolute time it is for it (absolute) `forT = qtBid.forT + QBid.relForT`. Note that the (integer) time (step) the bid is put in the system is called `atT` in SmartNet and the (integer) time (step) of the auction that the bid is meant to participate in, is called `forT`. So you bid at `'atT'` for an auction consideration for `'forT'`.

- `bids.models.QtBid`: A `QtBid` is a collection of `QBids`, each with their respective time indication. This is realized by a `QBid` referring to its `QtBid` in the field `qtBid` and specifying the `relForT` as well (see discussion in `QBid` just above). This makes that a time profile of quantities and prices can be bid.
    - `actor`: the id of the person/entity bidding
    - `forT`: the (absolute, integer) time (index) of the auction that the bid is to be considered.
    - `node`: reference to the node the bid is for. Notice that we don't bid on zones, but on specific nodes. We also have nodal prices and not zonal prices as market output.
    - `qtBidNr`: a number only used by the bidder (and not the market) for his own reference.
    - `Name`: a name only used by the bidder (and not the market, apart for logging) for his own reference.

There are class functions to plot bids in SVG format for debugging. These also allow to do curve crossing and graphically discover the crossing point. An example is given in Figure 70.

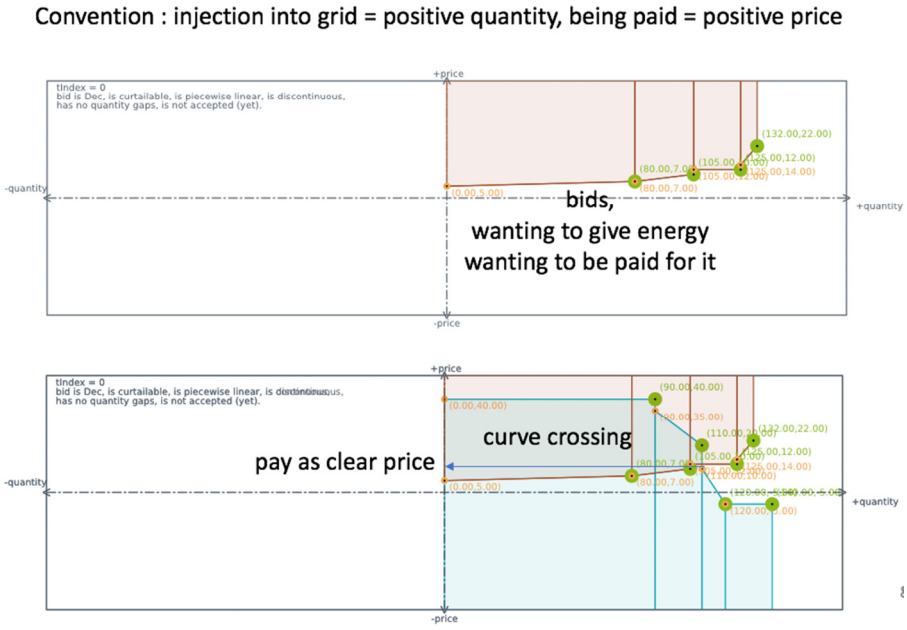


Figure 70 – Graphical representation of bids submitted to the market module

6.3.2 Market Bid Constraints:

It is perfectly possible to only bid bids (`QtBid`, `QBid`, `QBidSegment`) combinations into the market without adding any constraint over them. We call these bid 'naked bids'. However, sometimes temporal constraints (like ramp constraints) or logical constraints (like conditional constraints: if accept bid1, also accept bid2) constraints are needed in reality. So we also modelled the following constraints types. The constraints were chose on an as-needed-by-aggregator-basis. So we think they reflect what is needed in reality.

- `constraints.models.QPHalfPlanceConstraint`: Essentially, constraining the (active power, reactive power) 2-dimensional feasible space for a bid of aggregated devices can be defined per time step separately, so per `QBid` separately. However, we defined the constraint on a `QtBid`, in case the bidder wants to easily define the same feasible half-planes for all `QBids` in the `QtBid` in one constraint. Note that multiple half planes can be defined, so that any convex area can be nicely approximated to any desired accuracy (e.g.: triangular, rectangular capability, circular capability). This convex capability is common in devices and such convex areas have the nice property that also the capability of the aggregated devices results in a convex area of the same shape.
  - `qtBid`: the `qtBid` these half-planes constraints are applicable to.
  - `fromRelForT`: the lowest `forT` in the `QtBid` the half-planes constraint will be applied to
  - `uptoRelForT`: the highest `forT` in the `QtBid` the half-planes constraint will be applied to. Note that the same constraints will be applied for all `QBids` with a `relForT` within [`fromRelForT`, `uptoRelForT`]
  - `lineQtoPSlope`: the slope of the line defining the half-plane in (active (`P`), reactive(`Q`))-power space. Note that we consider the slope of `Q` as a (linear) function of `P`. Zero is possible to specify here. Infinity is not allowed. Infinity corresponds to the case of a vertical line in (`P`,`Q`) space and to the case of no restriction of `Q`, so a constraint with slow Infinity is redundant and not needed since already contained implicitly by the restriction on active power only in a segment to the range [`QBidSegment.quantity0`, `QBidSegment.quantity1`].
  - `lineQOffset`: The reactive energy offset (also called *absis*) off the line that describes the half-plane in (`P`,`Q`) space. So this is the value of reactive energy on this line when `P=0`.
  - `constrainToUpFromLine`: indicates which side of the line (above/below) is the feasible area the (`P`,`Q`) feasible points are restricted to by this constraint. When `True`, the upper part of the line indicates the feasible space, when `False`, the lower part fo the line indicates the feasible space.

- `constraints.models.QPDiscConstraint`: Essentially, constraining the (active power, reactive power) to a circular area, centered around d (`0`,`0`). This is defined because some devices have this shape of capability and also their aggregated bids then have a circular shaped capability.
  - `qtBid`: the `qtBid` these disc constraints are applicable to.
  - `fromRelForT`: the lowest `forT` in the `QtBid` the disc constraint will be applied to
  - `uptoRelForT`: the highest `forT` in the `QtBid` the disc constraint will be applied to. Note that the same constraints will be applied for all `QBids` with a `relForT` within [`fromRelForT`, `uptoRelForT`]
  - `maxApparentPower`: The radius of the disc describing the capability of the bid in (`P`,`Q`) space. The inside of this disc indicates the feasible space.

- `constraints.models.rampConstraint`: This is a temporal constraint, so should for sure be defined on `QtBids` since these objects have the time concept defined in them.
  - `qtBid`: the `qtBid` the other fields refer to
  - `fromRelForT`: the first time index in `forT` terms that the constraint is defined on
  - `uptoRelForT`: the last time index in `forT` terms that the constraint is defined on
  - `activePowerIncrement`: a step (can be positive or negative) in MW from time step `fromRelForT` to time step `uptoRelForT`. How this value is used, as an upper or lower bound, depends on the next field. [UNIT:MW]
  - `imposeIncrementAsMinimum`: Boolean, when `True`, it means that a minimum increment is imposed between the accepted active power values in the entire `QBid` (summed over all its `QBidSegments`) for `fromRelT` and for `uptoRelT`.
- `constraints.models.ActivationDurationConstraint`: defined on a `QtBid`, this constraint allows to specify that if any acceptance happens in a `QtBid` for it can never be for fewer than x subsequent time steps. This avoids to many activations and deactivations in a short time, with the objective to avoid wear-and-tear.
  - `qtBid`: the `qtBid` the other fields refer to
  - `deltaAlphaLo`: the number of time steps that should always be equal or less than any duration of an activation occurring in the `QtBid` horizon.
  - `fullyInsideQtBidHorizon`: If `True`, then a `QtBid` will never see activation in its horizon if the corresponding deactivation (happening at least `deltaAlphaLo` time steps later due to this constraint) does not fit in the horizon anymore. Default value is `True`.

- `constraints.models.IntegralConstraint`: This constraint allows to set upper and/or lower limits on the total accepted energy value between two time steps of a `QtBid`. This can (and is in SmartNet) used for battery modelling.
  - `qtBid`: the `qtBid` the other fields refer to
  - `fromRelForT`: the first time index in `forT` terms that the constraint is defined on
  - `uptoRelForT`: the last time index in `forT` terms that the constraint is defined on
  - `energyLo`: the lower bound on the energy that can be accepted in this `QtBid` during the interval [`fromRelT`, `uptoRelT`] [UNIT:MWh]
  - `energyHi`: the upper bound on the energy that can be accepted in this `QtBid` during the interval [`fromRelT`, `uptoRelT`] [UNIT:MWh]
- `constraints.models.ImplicationConstraintsOnQBids`: between any combination of a (`QtBid`, `QBid`) and another (`QtBid`, `QBid`) an implication relation can be set up, meaning that if the first `QBid` is accepted (meaning all its `bidsegments` are accepted above their respective minimal acceptance levels) the second `QBid` should be accepted (same meaning for that `QBid`). For this specification of both `QtBids` and `QBids` is needed. We also ask the bidder to specify the `forT`. While the `forT` is in principle deductible information, this allows the market to perform a check that he/she indicated the `QBids` that correspond to these `forT` values. So the fields are.
  - `ifQtBid`
  - `ifQBid`
  - `ifForT`
  - `thenQtBid`
  - `thenQBid`
  - `thenForT`
- `constraints.models.ImplicationConstraintOnQtBids`: A constraint, similar to the previous one, but directly between the acceptance values of two `QtBids`, is also defined. The `QtBid` acceptance fields are defined as `True` when some (not nothing) of a `QtBid` has been accepted. The fields for this constraint are only two.
  - `ifQtBid`
  - `thenQtBid`
- `constraints.models.ExclusiveChoiceConstraintOnQBids(_List)`: This is the parent class defining a new list constraint. It just contains an `ID` in the database and no other fields. As such, we can make any number of lists.
- `constraints.models.ExclusiveChoiceConstraintOnQBids_QBid`: In this table an entry is made for each `QBid` that needs to be added to the exclusive choice list (defined by a reference to

constraints.models.ExclusiveChoiceConstraintOnQBids(_List).ID). So we have a reference to that list and a reference to the QBid that ends up in that list. As such we can make any list have as many QBids as we want. The market will generate exclusive constraints for all QBids belonging to the same list. It will do so for all lists.

- constraints.models.ExclusiveChoiceConstraintOnQtBids(_List): This is the parent class defining a new list constraint. It just contains an ID in the database and no other fields. As such, we can make any number of lists.
- constraints.models.ExclusiveChoiceConstraintOnQtBids_QtBid: In this table an entry is made for each QtBid that needs to be added to the exclusive choice list (defined by a reference to constraints.models.ExclusiveChoiceConstraintOnQtBids(_List).ID). So we have a reference to that list and a reference to the QtBid that ends up in that list. As such we can make any list have as many QtBids as we want. The market will generate exclusive constraints for all QtBids belonging to the same list. It will do so for all lists.

6.3.3 Price profiles

The price associated to each node is contained in the following tables.

- profiles_NodeDeltaCost: node delta cost data
- profiles_NodeDeltaCostProfile: contains the connection between the related profile and the parameter values
- profiles_NodeHasNodeDeltaCostProfile: contains the connection between the nodes and the node delta cost profiles
- profiles_NodePrice: node price data
- profiles_NodePriceProfile: contains the connection between the related profile and the parameter values
- profiles_NodeHasNodePriceProfile: contains the connection between the nodes and the node price profiles

6.3.4 NodeNetInjection:

This table contain the profiles used in the market to compute the imbalance.

- scenario.models.NodeNetInjectionProfile: Defines the profile name (irrespective of how many data will be contained by it in the NodeNetInjectiopn class). This is the parent class defining a profile name only (apart form an implicit ID). The idea of this is that in this way, multiple nodes can be reusing the same profile without

having to define the profile more than once. The profile itself can contain many values (for (`node`, `atT`, `forT`) tuples), so that reduces work and database space used.
    - `name`: only a name is needed here
- `scenario.models.NodeHasNodeNetInjectionProfile`: Defines the link between a node and a profile.
    - `scenario`: simulation scenario
    - `node`: node it refers to
    - `nodeNetInjectionProfile`: refers to the first class in `NodeNetInjection`
- `scenario.models.NodeNetInjection`: Defines the profile records. The Node net injection is defined by the scenario. It is the net (as injection – off-take) amount of MW that is supposed to be injected per node. This information is needed by the market in the power balance equations it needs to respect at all times. Indeed, in all nodes, all incoming flows (injection, power flowing into the node via lines) should be equal to all outgoing flows (off-take, power flowing out of the node via lines). The scenario generates these values from the knowledge of the devices that are connected to these nodes and their respective power production or consumption, based on forecasts performed at `atT` for time steps into the future: `forT`.
    - `nodeNetInjectionProfile`: refers to the first class in `NodeNetInjection`
    - `atT`: time it is predicted at [UNIT:1]
    - `forT`: time it is predicted for [UNIT:1]
    - `P_fix`: active power value predicted [UNIT:MW]
    - `Q_fix`: reactive power value predicted [UNIT:MW]

Helper functions are defined to search for the most recent prediction for a `forT` value at a certain node. These functions hide the complexity of the three classes needed to store this efficiently.

6.3.5 Market Clearing:

The outputs of the market are the market clearings. More specifically, these entail the accepted quantity (fractions) for all `QBidSegements` for the specific `forT` of the auction. Also a price per node is outputted, since the implemented market is a pay-as-clear market.

There are three main market outputs as variable values determined by the market: Bid related, Node related and Edge related outputs (see section 6.2.3). In more detail, these are the following.

- `clearing.models.QBidSegmentVariables`:
    - `qbidSegment`: reference to the `qBidSegement` the other fields are applicable to
    - `atT`: the integer time index the bid is made at (in fact directly derivable from `QBid` and `QtBid` fields that this segment belongs to)

- `forT`: the integer time index of the auction the bid is to be considered for (also derivable from `QBid` and `QtBid` fields that this segment belongs to)
- `acceptedFraction`: a fractional number between 0 and 1 that indicates what fraction was accepted from this bid segment. Together with cleared nodal prices these are the main result of the market clearing [UNIT:1]
- `acceptedQuantity`: the quantity in MW that was accepted from this bid segment [UNIT:MW]
- `accepted`: a Boolean value that is `True` when any (non-zero) fraction of this bid segment was accepted, `False` otherwise. So this field's value is directly derivable from `acceptedFraction` [UNIT:True/False]

- `clearing.models.QBidVariables`:
  - `qBid`: the `QBid` these variables refer to
  - `atT`, `forT`: (see: `clearing.models.QBidSegmentVariables` above)
  - `acceptedActivePower`: The sum of all `accceptedPower` over all the `QBidSegments` belonging to this `QBid`.
  - `acceptedReactivePower`: The total reactive power for this `QBid`. Note that, contrary to active power, it is not possible to spread this reactive power over the individual `QBidSegments` in this `QBid`. Active reactive power constraints are only defined on the level of `QBid` and not at the level of `QBidSegments`, so segments do not define nor constrain reactive power. Segments do define and constrain active power by their `quantity0` and `quantity1` fields.
  - `accepted`: a Boolean field that reports whether or not all the `QBidSegments` contained in this `QBid` are accepted at or above their minimal acceptance level.
  - Note that the `QBid` cleared price is not stored here since it is a nodal property rather than a per `QBid` property.
- `clearing.models.QtBidVariables`:
  - `qtBid`: reference to the `qtBid` its fields apply to.
  - `atT`: (see: `clearing.models.QBidSegmentVariables` above). Note that a `QtBid` can refer to multiple, be it consecutive, `forT` values. This can be derive from the `QBids` that belong to this `QtBid`.

Figure 71 summarizes the five tables that the market outputs and their relations. In this picture we can see that each of the tables `QtBid`, `QBid` and `QBidSegment` have their respective variable counterpart tables that reference them. As for `Nodes` and `Edges`, these also have market variables associated to them and these tables make a reference (forerign key) to the scenario.

Figure 71 – Tables returned by the market layer and their relations

There are three other results, mainly for debugging purposes, the market writes to the database, like:

- `clearing.MarketDSOAggregation` which summarizes for the same (`scenario`, `atT`, `forT`, `node`, `subnetwork`) tuple, information on aggregated DSO bid injection, aggregated DSO bid cost and up and down injection.
- `clearing.AggDSOBids` which summarizes per (`scenario`, `atT`, `forT`, `node`, `subnetwork`, `qtBid`, `qBid`, `segment`, `quantities`, `prices` and `SOCP` and `power slacks`) for easier debugging.
- `clearing.models.DistributedTimingTrace`: This table is written to by market, but by also most other blocks in order to assemble the time it takes for functions in these blocks. It is a logging table useful to see which block functions take the highest fraction of the time spent during simulation and to know where to focus on to reduce total simulation time.
  - `scenario`: The simulations scenario (`Country`, `Coordination Scheme`, `simulation version`)
  - `atT`: For every `atT` we typically produce time log points in this table
  - `block`: name of the block the function belongs to (free text)
  - `function`: name of function (free text)
  - `subnet`: subnetwork name (if applicable, freetext)

- `startedAt_Seconds`: the time the function was started. The duration of the function can be derived from the start `startedAt_Seconds` of the next function in this table minus the `startedAt_Seconds` of the current function on this table.

7 References

[1] Gerard H., Rivero E., Six D., SmartNet project Deliverable 1.3, "Basic schemes for TSO-DSO coordination and ancillary services provision". December 2016. Available on-line at: http://smartnet-project.eu/wp-content/uploads/2016/12/D1.3_20161202_V1.0.pdf [last accessed in 18 June 2019].

[2] Dzamarija M., Plecas M., Jimeno J. et al., 2018, SmartNet project Deliverable 2.1, "Aggregation models". May 2018. Available on-line at: http://smartnet-project.eu/wpcontent/uploads/2018/05/D2.1_20180524_V1.0.pdf [last accessed in 18 June 2019].

[3] Leclercq G., Pavesi M., Gueuning T. et al., SmartNet project Deliverable 2.2, "Network and market models". February 2019. Available on-line at: http://smartnet-project.eu/wpcontent/uploads/2019/02/2019215113154_D2.2_20190215_V1.0.pdf [last accessed in 18 June 2019].