PNNL-28420 Ver. 4

The Transactive Energy Network Template Metamodel

Version 4

October 2024

DJ Hammerstrom
D Raker

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor Battelle Memorial Institute, nor any of their employees, makes **any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights**. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or Battelle Memorial Institute. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RL01830

Printed in the United States of America

**Available to DOE and DOE contractors from
the Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062
www.osti.gov
ph: (865) 576-8401
fox: (865) 576-5728
email: reports@osti.gov**

**Available to the public from the National Technical Information Service
5301 Shawnee Rd., Alexandria, VA 22312
ph: (800) 553-NTIS (6847)
or (703) 605-6000
email: info@ntis.gov
Online ordering: http://www.ntis.gov**

PNNL-28420 Ver. 4

The Transactive Energy Network Template Metamodel

Version 4

October 2024

DJ Hammerstrom
D Raker

Prepared for
the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Pacific Northwest National Laboratory
Richland, Washington 99352

Summary

While transactive energy, which is defined as an allocation of electricity based on dynamically discovered values or prices, has been extensively studied, its uptake and use has been slow. This report describes a tool, the *transactive network template*, which should hasten the creation and uptake of transactive energy networks.

Some basic principles of transactive energy are familiar from existing wholesale electricity markets. Locational prices are calculated today for zones within bulk electric transmission systems. Locational prices differ while accounting for the locational costs of electricity generation and the losses and constraints incurred when electricity is transmitted from generators and distributed to consumers. A transactive energy network might include these transmission zones. However, current research strives to apply transactive energy also in electricity distribution circuits, buildings, and even for individual generating and consuming devices. At the same time, researchers explore how to apply transactive energy in real time during increasingly shorter time intervals.

Automated computational agents become necessary as transactive energy becomes applied to smaller circuit zones and at faster dynamic timescales. A transactive energy network is an example of a multi-agent system. Each zone in the network is represented by its *transactive agent*, which makes decisions for and acts on behalf of a business entity that is responsible for and manages one of the circuit regions. A transactive energy network is also an example of a decentralized, distributed control system. Control decisions and responsibilities are distributed among the network's transactive agents. The transactive agents are independent; that is, there typically is no centralized authority or oversight function. Instead, transactive agents exchange transactive signals and thereby negotiate the prices and quantities of electricity that they will exchange.

Initially, the circuit regions and responsibilities of transactive agents appear to be very dissimilar. Each circuit region may comprise transmission, distribution, or building-level circuits. Each has a unique position and electrical connectivity within the transactive energy network. Each possesses unique assets that either generate or consume electricity, and these (e.g., renewable energy generator, diesel generator, aggregate utility load, building load, space conditioning, refrigerator, etc.) may further differ in their price flexibility and in their strategies for responding to dynamic electricity prices. Given such diversity, an implementer's first inclination might be to start from scratch to define all these devices and to engineer their seemingly unique interactions.

Given that each implementer's perspective may be narrow within a transactive energy network, it is unlikely that uniquely engineered systems would interact well. This is where the transactive network template is applicable. The transactive network template is a metamodel that has been developed to guide implementers as they configure their own transactive agent within a network of such agents. The object-oriented design of the transactive network template provides basic code object types that may be used and extended by implementers to represent each of the assets in their circuit region. These objects further facilitate the transactive agent's necessary computations, which are divided among responsibilities to schedule power usage, balance electric supply and demand, and coordinate the exchange of electricity with the other transactive agents.

This report addresses the conceptual transactive network template design. Implementers are directed to more formal design documents and reference implementations. A Python™-based[1] reference implementation of the transactive network template has been coded, and three implementations have been configured to represent a national laboratory and two university campuses. Version 2 of the transactive node template generalizes the market class and its methods to facilitate multiple, and more diverse market

[1] Webpage available at https://www.python.org.

coordination mechanisms than were facilitated by and demonstrated using Version 1. Version 3 includes new Appendix B, which addresses the designs of methods that would make dynamic prices track approved electricity rates. In the future, the author wishes to make the transactive network template more generally applicable to networks that require more accurate power flow.

Development of the transactive network template is jointly funded by the U.S. Department of Energy (DOE) Energy Efficiency and Renewable Energy and the DOE Office of Electricity. In late 2015, one of the first projects to be funded by the DOE Grid Laboratory Modernization Laboratory Consortium was the Clean Energy and Transactive Campus project, led by Pacific Northwest National Laboratory. DOE funds were matched by an investment by the Washington Department of Commerce through its Clean Energy Fund. The transactive network template was developed to guide the implementation of transactive energy networks within this project’s scope.

Glossary and Acronyms

	local asset	A device or system that generates or consumes electricity and that is managed by a *transactive agent*. An object class within the *transactive network template*. A *transactive agent* knows all states and strategies of its *local assets*. A *transactive agent* schedules its *local assets*, but it does not transact with them.
	locational marginal price	Formal definitions exist in wholesale electricity markets. More generally, the change in cost that accompanies a change in electricity supply at a given location and time.
	transactive agent	An entity that manages a circuit region using an implementation of the *transactive network template*. A *transactive agent* must negotiate with other *transactive agents* to coordinate energy flow and value within its *transactive energy network*.
	transactive energy network	A set of *transactive agents*, plus the allowed transaction pathways between the *transactive agents*.
	transactive neighbor	Each member of a pair of *transactive agents* that transact with one another by exchanging *transactive signals.*
	transactive network template	An abstracted, object-oriented metamodel of one *transactive agent's* perspective within a multi-agent *transactive energy network*. An extensible set of base object classes that was developed using the *Unified Modeling Language*®. A reference implementation of the transactive network template base classes has been instantiated in Python™[1] for use with the VOLTTRON™[2] platform.
	transactive signal	The information that *transactive neighbors* exchange. A set of records, each of which represents a pairing of price and quantity within a forecast time interval.
UML®	*Unified Modeling Language*	A standard visual modeling language intended to be used for modeling business and similar processes, analysis, design, and implementation of software-based systems.[3] The standard is maintained by the Object Modeling Group®.

[1] Webpage available at https://www.python.org.

[2] Webpage available at https://volttron.org/.

[3] Webpage available at https://www.uml-diagrams.org.

Contents

Figures

Tables

Introduction

The *transactive network template* is an extensible set of base object classes that was developed using the Unified Modeling Language™ (UML).[1] The transactive network template facilitates the configuration and operation of one agent within a cyber-physical network of such agents. It is designed to facilitate decentralized transactive energy systems, which automate and coordinate decentralized control decisions of distributed devices that generate or consume electricity.

Development of the transactive network template is jointly funded by the U.S. Department of Energy (DOE) Energy Efficiency and Renewable Energy and the DOE Office of Electricity. In late 2015, one of the first projects to be funded by the DOE Grid Laboratory Modernization Laboratory Consortium was the Clean Energy and Transactive Campus project, led by Pacific Northwest National Laboratory. DOE funds were matched by an investment by the Washington Department of Commerce through its Clean Energy Fund. The transactive network template was developed to guide the implementation of transactive energy networks within this project's scope.

MATLAB®[2] and Python®[3] reference implementations of the transactive network template were coded. The MATLAB implementation is useful for research exploration, but it is not suitable for field use. The Python reference implementation has been implemented on various VOLTTRON™[4] communication platforms for use in actual networks. Furthermore, the Python reference implementation was used to configure three transactive energy network implementations—one representing a transactive energy network comprised of buildings on the Pacific Northwest National Laboratory campus and their upstream campus, municipality, and wholesale electricity suppliers; and two representing two U.S. university campuses.[5]

The word *template* highlights the ability of the transactive network template to guide and facilitate specific agent implementations. Its object-oriented design enforces a degree of standardization, and its classes provide the standard properties that will be needed by an agent. Furthermore, needed behavioral methods are enforced to standardize inter-agent transactions and integration of the agent's local assets. At the same time, extensibility is supported for the assets themselves, which often possess additional unique properties and energy behaviors. Extensibility is quite naturally supported when using object-oriented code design. Libraries of asset models (e.g., for specific building loads and demand-responsive assets) should evolve by inheritance and extension of the base transactive network template object classes. The base classes themselves should not be casually modified.

This report is intended for a nontechnical reader, not the coder or implementer. It introduces concepts and features of the transactive network template, but it does not provide enough detail to create another valid reference implementation. It is important for the reader to understand why the transactive network template is a *metamodel*, but the reader is not expected within this report to interpret the UML diagramming which allowed its design as an object-oriented metamodel. If still more detail is needed, contact Donald J. Hammerstrom[6] regarding the design document or Hung Ngo[7] regarding the Python reference implementation code.

[1] Webpage available at https://www.uml-diagrams.org.

[2] Webpage available at https://www.mathworks.com/products/matlab.html.

[3] Webpage available at https://www.python.org.

[4] Webpage available at https://volttron.org/.

[5] Katipamula S, RG Lutes, S Huang, J Lian, H Ngo, and DJ Hammerstrom. 2019. *Coordination of Behind-the-Meter Distributed Energy Resources for Transactive Grid Services: Multi-Building*. PNNL-XXXXX, Pacific Northwest National Laboratory, Richland, Washington.

[6] Donald.Hammerstrom@pnnl.gov

[7] Hung.Ngo@pnnl.gov

Implementations of the transactive network template are intended to facilitate a single agent's perspective within a multi-agent transactive network. That agent will be referred to in this report simply as "the agent" or "this agent." When necessary, references to "other network agents" or "another network agent" or "neighboring agent" should be understood to refer to other members of the multi-agent network besides the one central to the given implementation. Terse, precise class names were used in the transactive network template, but more readable names and descriptions have been used in this report (e.g., a *LocalAsset* object is referred to as a "local asset object").

The structural and behavioral aspects of the transactive network template are addressed in Section 2, and further improvements to the transactive network template are suggested and discussed in Section 3.

The Transactive Network Template Metamodel

The transactive network template is a *metamodel* in that it is a model of models. Its object-oriented UML design may guide reference implementations that may use different software code languages. Regardless, every reference implementation should reproduce the same transactive network template base classes and base class behaviors. In principle, transactive agents residing in the same transactive network may choose entirely different reference implementations of the transactive network template, and these agents should still properly exchange transactive signals and interact.[1]

Object-oriented designs are separable into their structural and behavioral elements. Section 2.1 introduces base classes of the transactive network template, thereby providing a structural overview of the structural elements that are available to model an agent's assets and position within a transactive network. Section 2.2 introduces the most important behavioral responsibilities of the transactive agent and how those responsibilities are allocated among the available transactive network template classes. The three fundamental computational responsibilities of a transactive agent are to 1) to balance electrical power in the circuit region that is managed by the agent, 2) schedule the power to be generated or consumed by the agent's local assets, which decisions may be price-responsive, and 3) conduct transactions and coordinate electricity exchanges with other agents.

Sections 2.3 and 2.4 address what it means to configure and extend a transactive network template code implementation respectively.

Base Transactive Network Template Code Classes

This section introduces the structure of the transactive network template by introducing its base classes. These classes are the structural elements available to design, configure, and operate a transactive network template implementation. Objects must be instantiated from these base classes (or from classes whose parentage can be tracked back to these base classes) from the perspective of a single transactive agent and its electric circuit region that it represents.

A transactive network template implementation provides useful object classes that one transactive agent may configure, specialize, and use to plan and manage electricity supply usage in the circuit region for which it is responsible. Three of the transactive network template's most important code classes are shown in Figure 1. The *market class* manages the transactive agent's balancing of electricity supply and demand. The *local asset class* interfaces with and schedules the devices and systems in the circuit region that the transactive agent manages. The *neighbor class* coordinates the transactions and manages the exchange of transactive signals with neighboring transactive agents. The neighbor and local asset classes may be specialized and instantiated as many times as is necessary to represent all the transactive agent's assets and transactive neighbors.

The transactive network template features more object classes than the important ones featured in Figure 1. High-level information about all the transactive network template base classes is summarized in Table 1. The first column gives both the base class name (bold, italicized) and its brief description. The second column lists properties that the base class manages on behalf of the transactive agent, and the third column lists the classes' most important behavioral responsibilities.

[1] More precisely, the transactive records should interoperate at the business and syntactic levels. The transactive network template is agnostic about interoperability in physical communication layers. Neighboring agents may have to negotiate their choice of communication carrier and choose from available low-level communication protocols.

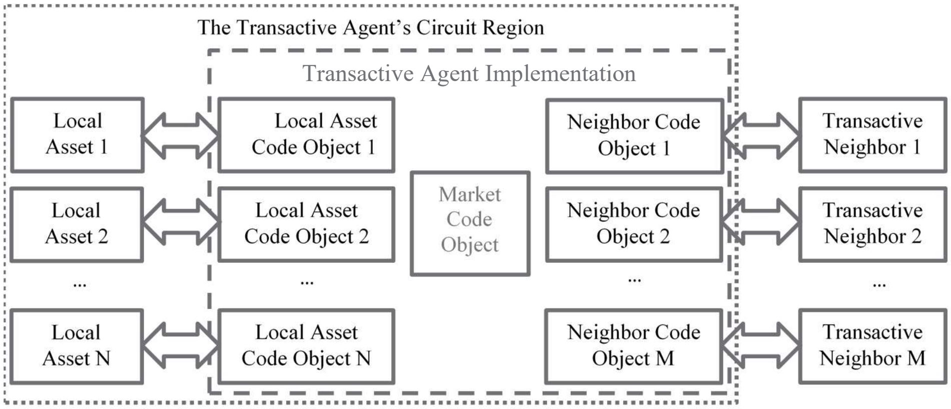


Figure 1. Simplified Representation of a Transactive Agent's Transactive Network Template Code Implementation and Its Interfaces to the Transactive Agent's Local Assets and Transactive Neighbors

1.1 Scheduling, Balancing, and Coordination Objectives

There are three very important computational responsibilities managed by transactive agents that implement the transactive network template: 1) balancing, 2) scheduling, and 3) coordination. The transactive network template has been designed to make these three computations as separable and independently achievable as possible.

1.1.1 The Balancing Responsibility

Many readers will possess conceptual understanding of the market principles that are central to the balancing objective. Given updated power schedules and price flexibilities of all the agent's local assets and neighbors, an agent's market object calculates an electricity price that balances supply and demand among all electricity entering and exiting the agent's circuit region.

The agent's circuit region is treated as a "copper plate," which means it possesses undifferentiated electrical circuit properties and incurs no transport losses within the circuit region. There may be only one voltage in the circuit region. While exchanges with neighbor objects may be modified to reflect electricity that is lost upon importing electricity into the circuit region, local assets reside in the circuit region and typically will not incur transport losses.[1] The current transactive network template version addresses only real electric power transactions and the balancing of real power.[2]

[1] These "copper-plate" principles are intentional and should not be violated. Electricity must have one unit price across the agent's entire circuit region at any given time. If an implementer feels compelled to assign multiple electricity prices within one agent's circuit region, the regions should be separated and granted their own agents and transactive network template implementations.

[2] Future versions should address reactive power and voltage management, which would require successively detailed calculations of both real and reactive power generation and consumption within the agent's circuit region and complex power transport between agents and their circuit neighbors. Effects like transport losses can be estimated until such new transactive network template versions can be completed. These future versions are addressed in Section 3.0.

Table 1. Important Properties and Responsibilities of Transactive Network Template Base Class Objects

Class Object	Important Properties	Responsibilities
myTransactiveNode—The agent's transactive node. It represents an agent's perspective from within its circuit region.	• Lists of information source, local asset, market, meter, and neighbor objects that the asset must interact with.	• None.
LocalAsset and ***LocalAssetModel***[(a)]—A local device or system that must be scheduled by the transactive agent—a generation or load device that lies within the agent's circuit region and is "owned by" the asset. The transactive agent knows the asset's entire status and strategy.	• Cost parameters for calculating production costs. • Default and active vertices for representing the asset's flexibility to the locational price. • Lists of meter and information sources. • The object's name and description. • The asset's default, minimum, maximum, and scheduled powers. • The asset's production, total production, dual, and total dual costs.	• Given forward electricity prices, an asset schedules its own power and its flexibility to change its electric power generation or consumption in response to changes in those forward prices. • Given the asset's scheduled powers, the asset updates its production and dual costs, which are used by the market object to determine convergence of the transactive agent's balancing objective.
Neighbor and ***NeighborModel*—**Locations outside the agent's circuit region with which the agent may exchange electricity. Transactive neighbor objects are further managed by other transactive agents of the transactive network and expect to exchange transactive records with this agent.	• The neighbor's default, minimum, maximum, and scheduled powers. • Cost parameters that may be used to calculate production costs.[(b)] • Default and active vertices that are used to represent neighbor flexibility on a marginal supply or demand curve. • Demand-charge parameters. • Lists of meter and information sources. • The object's name and description. • Production, total production, dual, and total dual costs. • Ready, sent, and received transactive records. • Boolean indicator stating whether neighbor is transactive or not. • Transport loss factor that may be used to estimate transport losses for electricity imported from the neighbor.	• Given a series of forward electricity prices, the *Neighbor* schedules its own power and its flexibility to changes in those forward prices. • Given the neighbor's scheduled powers and forward prices, the neighbor updates its production and dual costs. • For every neighbor object, the agent must prepare, send, receive, and check for convergence among transactive record signals. • If demand charges are in play for the neighbor object, it must update the thresholds and impacts of the demand charges.

Class Object	Important Properties	Responsibilities
Market	• Aggregate flexibility is stored as a list of aggregate active vertices. • Aggregated generation total generation, demand, total demand, and net powers. • Aggregated production, total production, dual, and total dual costs. • Convergence criterion threshold concerning the agent's balancing objective. • Convergence status. • Default and actual marginal electricity prices. • Market forecast horizon and clearing schedule. • Time interval duration. • Current market state. • Identity of preceding and any corrected markets.	• The market object must balance electric supply and demand for the agent, which includes these following responsibilities. • Initiate updating of and sum local assets' and neighbors' production and dual costs. • Initiate updating of and sum of local assets' and neighbors' powers and flexibility scheduling. • Maintain a current list of active market time intervals. • Instantiate new successive market objects as they become needed and relevant. • Manage market lifetime and event timing using a state machine.
TimeInterval—A time interval object.[c]	• Market object in which the time interval is relevant. • The object's name. • The interval's starting, duration, activation, and clearing times in the given market object, as well as its calculation timestamp.	• Update market state. • A construction method exists to enforce class structure.
IntervalValue—Measured or calculated data that belong within their specified time interval. For example, forecasted price and power data must have their respective time interval specified.	• Market object in which the value is relevant. • Object name identifier. • Source class and object that created value. • The value and its measurement type and units of measure. • Time interval in which the value resides.	• A construction method exists to enforce class structure.
TransactiveRecord—transactive record. A set of these records constitute a signal between this agent and one of its transactive neighbors.	• Tuple of cost, marginal price, and power, and the time interval in which the tuple is relevant.[d] • Indicator whether the record represents scheduled power or an inflection point on a piecewise linear supply or demand curve.	• A construction method exists to enforce class structure.
Vertex—An inflection point between lines of a piecewise linear marginal supply or demand curve.	• Tuple of cost, marginal price, and power.	• A construction method exists to enforce class structure.
MarketState—Enumeration of possible market states.	• The set {Inactive, Explore, Tender, Transaction, Delivery, Publish, and Expired}	• None.

Class Object	Important Properties	Responsibilities
MeterPoint—A meter source of data.[e]	• Meter description and name. • Measurement value and type and unit of measure. • Sample time and interval and next scheduled sample time. • Store time, storage interval, and next scheduled repository store time.	• Read a meter. • Store meter data.
InformationServiceModel—Information service or, more generally, a source of information other than a simple meter.[f]	• Information source (e.g., service and its location). • Information type and units of measure. • Object name. • Sample time and duration, and next scheduled sample time.	• Update information (e.g., forecast outdoor temperature from a weather forecasting service).

[a] Two classes of types object and model were used for various base classes of the transactive network template. The intention was to group object properties that reside with objects in static time and behavioral properties and methods that reside modeled with objects in predicted time series. There is some value to this approach, but the distinction will be mostly ignored within this report.

[b] Fueled generator cost functions are the basis for most theory underlying wholesale electricity markets. The transactive network template therefore facilitates calculation of production costs using a quadratic function of power. Interestingly, these conventional cost functions are not particularly useful for determining the production costs of other distributed generation resources.

[c] Time interval objects, once instantiated, remain affixed to their delivery time period. The time interval object can therefore keep track of a market's status, which facilitates potentially rich market timing practices. The transactive network template should be resilient to missed calculations and system down times because of the persistence of time interval objects.

[d] Many additional tuple elements were defined for a transactive record to support anticipated future functionality. Unimplemented, untested features will not be discussed in this report.

[e] As for *InformationServiceModel*, reference implementations are tending to ignore this class. It was hoped that this class would facilitate platform independence.

[f] The reference implementations largely ignore this class. Access to some information sources will be found to have been provided by a communication platform, as was the case with VOLTTRON. It was hoped that this class would facilitate platform independence.

All suppliers, consumers, importers, and exporters of electricity are treated similarly and symmetrically by the transactive network template. A consistent sign convention is enforced from the perspective of the agent and its circuit region that the agent manages. Power and electricity entering the agent's circuit region via generation or importation are assigned positive values; power and electricity exiting the agent's circuit region via consumption or exportation have negative values. Bidirectional power flows and energy storage assets are permitted and supported. The transactive network template allows an electricity consumer to become a generator or an exporter to become an importer from one time interval to the next. An important consequence of this sign convention is that, from the agent's perspective, electric power balance has been achieved if the sum of all electricity generation, consumption, importation, and exportation is zero.

The balancing process is initiated upon the agent's market object inviting all its neighbor and local asset objects to update their schedules, price flexibilities, and production and dual costs. These scheduling computations by the transactive network template neighbor and asset objects are precisely the responsibilities that are to be discussed in Section 0. The market object sums the objects' responses to determine net power balance, net available price flexibility, and various costs that indicate whether the balancing process has converged.[1] Template Version 1 facilitated only an iterative consensus negotiation mechanism that could be solved using either of two alternative solution methods: Method 1 conducts a sub-gradient search, in which each time interval's price is nudged up or down based on the magnitude and sign of the calculated duality gap. This first method requires many iterations, but it becomes necessary when objects cannot, or choose not to, reveal the flexibility of their scheduled powers to price changes. The balancing process ends when the duality gap has been driven to a suitably small magnitude. The simpler, preferred Method 2 requires many fewer iterations. If all transactive network template neighbor and asset objects reply to their market object with accurate, piecewise linear representations of their price flexibility for each active time interval, then the clearing price may be accurately determined through interpolation, thus requiring few if any iterations. The duality gap still is used by Method 2 to indicate convergence, but very few iterations were needed for the simple reference implementations to date.

The authors strongly advocate for the simpler Method 2. One of its assumptions is that all objects' production (and consumption) costs are represented by quadratic, monotonically increasing cost functions. An implication of this common assumption is that the derivative of the quadratic cost function—its marginal supply or demand curve—is affine. Any power function comparable to marginal prices is therefore piecewise linear, as demonstrated by **Error! Reference source not found.**, which shows a cleared balancing between a price-responsive supplier and an inflexible consumer.

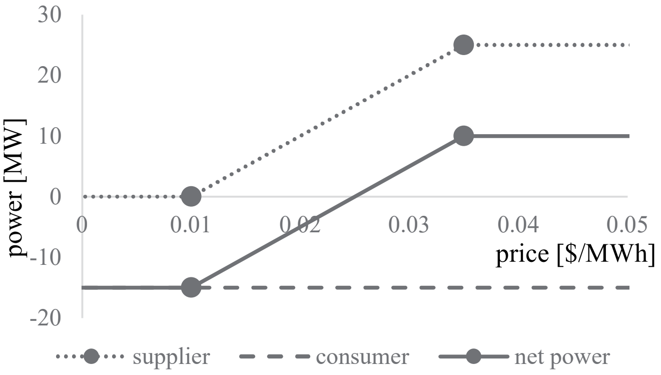


Figure 2. A Simple Example Clearing between a Flexible Supplier and Inflexible Consumer

The price-sensitive supplier's offer is represented by three line segments: It will never produce electricity at any price below $0.01/MWh. It will produce a linearly increasing power between that price and $0.035/MWh. Finally, it can produce no more than 25 MW at any price higher than that. In this case, only two inflection points, or *vertices*, were needed to represent a relatively sophisticated price-responsive production offer from the supplier.

Curves' tails are always presumed to extend horizontally from the smallest priced vertex to negative infinity and from the greatest priced vertex to positive infinity. The price-inflexible consumer in **Error! Reference source not found.** is represented by a single horizontal line at -15 MW. This entire bid from the consumer may
be represented by the single vertex (∞, -15).

Net power is calculated by adding powers at each marginal price. In fact, if the curves are all piecewise linear, only power at vertex marginal prices must be added, and the power at all other marginal prices

[1] More precisely, the magnitude of the *duality gap*, which is a difference between *primal* and *dual* costs, indicates whether the agent has converged upon a satisfactory balance of supply and demand throughout a set of forecast time intervals.

may be found by interpolation. The corresponding clearing price occurs where the net power curve (again piecewise linear) is zero, meaning that supply and demand are of equal magnitude (i.e., balanced).

Incidentally, the transactive network template allows and supports negative marginal prices. A negative marginal price indicates that balancing clears to the left of $0/MWh, which requires that a supplier was willing to reduce its production, or a consumer was willing to increase its consumption, as marginal price became negative. While marginal price remains negative, suppliers must pay to produce, and consumers must be paid to consume electricity.

Startup and Future Balancing Issues

System startup of transactive network template processes creates interesting challenges for the balancing process. The market object must supply default marginal prices to the agent's local asset and neighbor objects, which any price-responsive object will require if it is to schedule its power. While this might seem to be an issue only during system startup, new future time interval objects are constantly being spawned as the system marches through time, and a default marginal price must be assigned to these new time intervals as well to jumpstart the balancing of power in the new time intervals.

Base transactive network template classes may possess additional class properties and methods that are not discussed in this report. In most cases, these features were included in the transactive network template to support anticipated future functionality. We list some anticipated functionality here but warn the reader that these capabilities are not tested and may be incomplete:

- *Reserve margins*. Given an increasing interest in grid resilience, properties have been proposed to help keep track of aggregate system reserve margin. Unused, super marginal production capacity (and potentially unengaged demand reduction, as well) might be claimed as reserve margin. The available aggregate reserve margin might indicate the transactive network's resilience. Alternatively, a requirement might be stated for a minimum allowable reserve margin, and the market class clearing process might be extended to account for the achievement and cost of this requirement.
- *Asset engagement*. Wholesale electricity markets perform unit commitment to preplan whether large generators need be engaged (ready) or not. This practice is particularly important for assets that take a long time to start up and shut down or that incur costs upon doing so.
- *Fixed and fixed*, *avoidable costs*. Fixed production costs are not typically used for unit commitment and dispatch, but fixed costs comprise a substantial fraction of final electricity bills. And some wholesale electricity markets are finding ways to re-compensate producers for fixed, avoidable expenses (e.g., startup cost) they might incur. For these and other reasons, transactive systems may need to harmonize discrepancies and unfairness due to fixed and fixed, avoidable costs.

The Scheduling Responsibility

Given a series of forward prices, each neighbor and local asset object must be able to 1) predict its power generation or consumption, 2) calculate its predicted flexibility to change its schedule in response to changes of said forward prices, and 3) calculate its various production and dual costs, which may include the costs of both electricity and utility. These responsibilities are referred to here as *scheduling*. Each local asset and neighbor object should be able to independently schedule itself. Each object acts in its self-interest on behalf of its transactive agent.

The scheduling process is initiated when the local asset or neighbor objects are invited by the transactive network template market object to schedule themselves. The scheduling processes differ slightly for neighbor and local asset objects, as will be addressed in the next two subsections.

Scheduling of a Neighbor Object

Because the neighbor object represents a neighboring agent and its remote circuit region, this agent (meaning the one central to the transactive network template) knows little about the remote circuit region and its motivations and therefore uses a more standardized, hard-coded approach in its scheduling of the neighbor object. The neighbor object maintains a copy of the last supply or demand curve that was received from the neighboring agent in the form of a transactive signal. These curves are represented by the inflection vertices, as has already been discussed in conjunction with **Error! Reference source not found.**. Therefore, the scheduled power is simply the power at which the neighbor object's saved supply or demand curve intersects the current marginal price in its respective time interval. The neighbor's price flexibility is precisely represented by the neighbor object's saved supply or demand curve that it received also from the neighbor transactive agent via a received transactive signal. The production cost (or its equivalent gross consumer surplus), excluding a constant term, may be calculated by integrating the neighbor object's saved supply or demand curve over the object's viable power from its minimum production (maximum consumption) to maximum production (minimum consumption). The dual cost is equal to the calculated production cost (gross consumer surplus), less the energy income (outflow).

Scheduling a Local Asset Object

During the scheduling process for local asset objects, the market object expects the same calculated results as from neighbor objects, but the scheduling of local asset objects may be much more diverse and complex and highly specialized. This complexity can be accommodated for the local asset objects, however, because the status of a local asset object is always fully transparent to its agent. Any strategy of a local asset is fully known by its agent and should exist to serve the interests of the agent, as well.

Consider a battery energy storage site as an example local asset represented by its corresponding local asset object. An asset's owner might pose a strategy to optimize the arbitrage value of the energy to be stored into batteries (negative power and energy, by the transactive network template sign convention) and released back to the electric power grid (positive power and energy). This is certainly feasible to do given a series of forward electricity prices. However, the optimization strategy must consider the site's limited energy storage capacity, the inverters' charge and discharge power capacities, and the batteries' current state of charge. This optimization is very challenging, but the optimization strategy might still be configured to suit not only this example, but also the needs of other battery energy storage owners who have similar basic objectives. More sophisticated battery owners might further value battery lifetime, backup reserve capacity, maintenance periods when battery availability is to be limited, and so on.[1] The potential objectives are unbounded. Therefore, many local asset battery energy storage objects will become further specialized to address permutations of more and more operational objectives, resulting in increasingly richer optimization strategies. For these reasons, the transactive network template establishes an interface by which local asset objects must respond their power generation or consumption schedules, but the transactive network template must not formalize or standardize the means by which the optimal power schedule is calculated.[2]

We continue the battery energy storage example and consider its responsibility to calculate its price flexibility. Assume that an optimal power schedule has been found. Further define *residual flexibility* as a

[1] Upon including valued objectives in addition to electricity cost and income, one is now optimizing a *utility cost* function.

[2] In fact, many empirical and simplified heuristic methods and decision functions have evolved for predicting transactive power schedules from prices. Analytical optimization should be preferred for power scheduling when explicit forward prices exist, as is the case today for the transactive network template. Simplified approaches should strive to approximate the results of a true optimization strategy.

trajectory on the marginal price plane if one time interval's price were to be perturbed while leaving all other forward prices unchanged. The point at the scheduled power value and current locational electricity price is necessarily a member of the residual flexibility curve. All the other points on the trajectory may be found while using the same optimization strategy as was used to find the scheduled power.

The opportunities represented by the residual flexibility curve are therefore respectful of the opportunities the asset has in the other time intervals because the multi-period optimization principals are retained unchanged. It is important for implementers to understand this ideal approach (using perturbation to determine residual flexibility) even if shortcuts are taken.[1]

Regardless how residual flexibility is calculated, it should become represented by its piecewise linear supply or demand curve. Doing so takes advantage of concise storage and simple calculations, as was discussed in conjunction with **Error! Reference source not found.**.

The final scheduling objective for the local asset object, to calculate production cost (gross consumer surplus) and dual cost, proceed as for neighbor objects as discussed earlier.

Much as the transactive network template market object provided default prices to facilitate bootstrap startup of the transactive network template balancing process, both local asset and neighbor objects must supply a default scheduled power and default active vertices to ensure startup and resiliency of the scheduling processes. If an agent's transactive network template implementation is to successfully start up, then there must exist a feasible initial solution to the balancing computation—there must exist a price at which summed supply and load balance one another.

These default values are particularly needed by a neighbor object, which lacks knowledge of the neighboring agent's power needs (or offers) and price flexibility until that neighbor agent choses to reveal such information by finally sending its transactive signal. Local asset objects are less dependent on these defaults during bootstrap startup, but the defaults may still be used should scheduling calculations fail.

The Coordination Responsibility

Pairs of transactive neighbor agents transact; that is, they exchange transactive signals. In doing so, each agent plays its role in the coordination of energy allocation throughout the entire transactive network of decentralized, independent transactive agents. Neighboring transactive agents should be independent. Each has its own unique local assets and its own unique position within the transactive network. A neighboring transactive agent cannot even be expected to have been derived from the same transactive network template reference implementation or to have been coded using the same computer language. A transactive agent can decide when to send its own computed transactive signal to one of its neighboring agents, but it cannot insist and control precisely when that neighboring transactive agent shall reply with its (i.e., the neighboring agent's) transactive signal. The transactive network template coordination and signaling processes must therefore be flexible to accommodate other agents' independence.

A *transactive record* pairs a forecast time interval with a record number, price, and electricity quantity.[2] Neighboring transactive agents must agree on the format of a transactive record so that exchanged records may be accurately interpreted by both neighbors. Better yet, an entire transactive network should agree upon and enforce a standard transactive record format (along with compliant transactive network template

[1] Again, if optimization principles were not applied while calculating a power schedule, then there is no strong support for calculating price flexibility. The local asset's intertemporal constraints may be lost and violated.

[2] The transactive network template transactive record class currently includes other properties such as reactive power and voltage that will be needed for future transactive network template capabilities and versions.

reference implementations) as a condition of joining a transactive energy network. The transactive network template specifies content information and structure of transactive records, but it is agnostic concerning low-level communication protocols that must be negotiated between transactive neighbors.

At least one transactive record must be sent for each active time interval, thus revealing the agent's aggregate electric power that, according to its computations, is scheduled to be exchanged. This record is assigned as "Record 0". When only this one single transactive record is sent, the sender is saying it can offer no flexibility to change its scheduled power as a function of the time interval's electricity price.

If additional transactive records are sent concerning an active time interval, these records are numbered successively using integers {1, 2, …}. These additional transactive records represent the agent's residual flexibility—inflection points ("active vertices") of the agent's supply or demand curve in the active time interval. At least two inflection points are necessary to represent residual flexibility.[1]

Neighboring transactive agents must agree which time intervals are eligible for transactions and should therefore be represented among the transactive records that are sent and received.[2] The set of transactive records for all the active time intervals is the transactive record *signal*.

If the transactive signals between neighbors can be represented by relatively few transactive records, each representing an inflection point of a piecewise linear supply or demand curve, then the transactive signals remain small and should not require much communication bandwidth. The transactive network template is agnostic concerning the choice of communication carrier. An agent may be, but is not necessarily, internet connected. Future transactive network template reference implementations should not specify or require any one single communication protocol.

The coordination process is driven by the agent's decision to send its transactive signal to a neighboring transactive agent. It should do so upon recognizing either of two events: 1) local conditions have significantly changed since a transactive signal was last sent or 2) the agent and its neighboring agent disagree concerning the price or average electric power that is to be exchanged.

To determine these events, the agent saves and compares three versions of the transactive signal: 1) the last transactive record signal that was sent, in fact, by this transactive agent to the neighboring transactive agent, 2) the current transactive signal that is up-to-date and ready to be sent to the neighboring transactive agent, but has not yet been sent, and 3) a copy of the most recent transactive record signal received by this transactive agent from the neighboring transactive agent. The transactive agent should send its transactive signal if the prepared and sent transactive record signals differ by more than a configurable threshold magnitude. The agent should also resend its transactive signal if the prepared and received transactive signals differ by more than another configurable threshold magnitude, providing the neighbor's transactive signal has been received since one was last sent. The transactive agent believes coordination between itself and its neighboring agents has converged while neither event compels it to resend its transactive record signal.[3]

[1] Specifically, this means there may be sets of transactive records numbered {{0}; {0, 1, 2}; {0, 1, 2, 3}; …}, but never {0, 1}, for a given active time interval.

[2] This determination concerning *active* time intervals and other attributes and timing of the transactions are specified by the transactive network template market object. Neighboring agents must therefore agree on transaction rules as specified by their respective transactive network template market objects.

[3] This fully event-driven design is very flexible and forgiving. The system can ride through temporary communication outages, but it cannot distinguish satisfied neighbors from those experiencing failed communications. Therefore, a recommended practice is for an agent to send a transactive signal at least once during each market time interval. Future transactive network template versions should also discount the reliability of stale transactive signals that were not updated when expected.

The process and timing of inter-agent signaling, as directed by the market object, defines whether a transactive negotiation is a bilateral auction, consensus negotiation, or other negotiation mechanism. The transactive network templates base classes must provide methods and properties that are flexible and sophisticated enough to facilitate many different mechanisms. This flexibility was enhanced in Version 2, to be discussed later in this report. Had Version 1 focused on only bilateral auctions, which have intrinsic assumptions concerning a direction of power flow and one-time price discovery, it may have been impossible to later extend the facilitation to deep multi-agent networks and iterative negotiation mechanisms.

Also central to the facilitation of alternative negotiation mechanisms is the exchange of net supply and demand curves between neighboring agents. The *net* supply or demand curve is the sum net flexibility of all an agent's neighbor and local asset objects, *excluding the transactive neighbor agent to which the transactive record signal pertains*. This practice comes quite naturally to those conducting bilateral auctions in that electric load is aggregated and bid into a supply market. But this aggregation must also take place in the creation of supply bids if alternative diverse, iterative, decentralized negotiations are to be facilitated. The principle may be exemplified using **Error! Reference source not found.**, which shows a supply, demand, and summed net curve. If the supply curve is that revealed by a neighboring agent that supplies this agent, then its transactive signal would exclude the supply curve, leaving only the constant demand curve to be included in the transactive record signal. The points are that 1) coordination may occur between two transactive neighbors independently from any aggregation and 2) the timing of signal exchanges is not necessarily limited to prescribed market periods, power flow direction, or heartbeat dependencies.

If an agent and a neighboring agent are to initiate successful transactions, each must configure generous default representations of the other's power capacity range. Each neighbor has an assigned minimum and maximum power capacity that represents such capacity. The electric power received from or sent to a neighbor may be constrained either by the abilities of the two neighbors to generate and consume electricity or by the capacity of the electricity transport elements (e.g., conductor, transmission line, distribution line, transformer, fuse, breaker, etc.) between the two neighbors. The capacity range may be narrowed to powers either above or below zero if a neighbor agent is anticipated to always be an electricity supplier or consumer.

Each agent may compute a still narrower power range to represent its residual flexibility range, wherein it could be enticed to operate given correspondingly enticing prices. The agent's residual flexibility may extend to either or both the capacity constraints. On the other hand, if the transactive agent truly has no price flexibility, its residual flexibility becomes a single scheduled power within the allowed power capacity range.

The relationships of these various hard and soft power constraints are shown in Eq. (1). Bars below and above average powers p represent minima and maxima, respectively, of the indicated ranges.[1] Transactive records should represent the scheduled power and soft price flexibility range, if any. An agent should offer as much price flexibility as it can (and will) via its transactive records. A transactive record between two transactive neighbor agents should never be computed to lie outside the hard capacity constraint range that they share. The logic of this paragraph defines and enforces the impacts of price-responsiveness and capacity constraints during the transactive network template's coordination of power exchange between neighboring transactive agents.

[1] Observe that the transactive network template sign convention applies here. The minimum $\underline{p}_{\text{capacity}}$ for a neighbor object that consumes electricity from this agent is a negative number that represents its *maximum* average electric power consumption during the time interval, a negative number.

$$\overbrace{\underline{p}_{\text{capacity}} \le \underbrace{\underline{p}_{\text{flexibility}} \le p_{\text{scheduled}} \le \overline{p}_{\text{flexibility}}}_{\text{price flexibility range}} \le \overline{p}_{\text{capacity}}}^{\text{power capacity range}} \tag{1}$$

Discussion has thus far avoided the interpretation of the agent's electricity price. For implementations that support only a shallow network (e.g., utilities bidding their aggregate local asset flexibilities into a wholesale electricity market), the interpretation might not matter much. Local asset objects will schedule themselves given any forward price schedule, regardless of its interpretation. However, for deep, greatly decentralized networks, the interpretation of the price must be consistent with the market process and determines whether assets are meaningfully coordinated among the many levels of the transactive network. Marginal price is the preferred interpretation of a transactive agent's price. It is the locational marginal price that meaningfully compares resource opportunities and determines which distributed resources should be dispatched.[1]

Whereas local assets were said to reside on a lossless copper plate representation of the agent's circuit region, neighbor objects are remote, do not lie on the lossless copper plate, and should account for any losses that are incurred as electricity is imported into the agent's circuit region. A parameter of the neighbor class allows an implementer to configure the full-load percentage loss when importing $\underline{p}_{\text{capacity}}$ from the corresponding neighbor object. The lost electricity may then be estimated for any imported average electric power magnitude.[2]

Losses should be applied to only imported (i.e., purchased) electricity. The importer must account for electricity losses and the cost impacts of energy losses.[3] The average power to be imported from the neighboring transactive agent must be decremented by the anticipated electricity power loss. The effective local price of the imported electric power must be increased because more power must be purchased than will, in fact, be received.

Transactions between neighboring transactive agents may also address demand charges if such practices exist between them. Neighbor class properties have been provided to configure and apply such demand charges. Specific practices may differ, but the typical practice is to add substantial monthly charges based on the highest demand that one agent is supplied by another during the prior calendar month. The demand charges attributable to relatively few high-demand periods in the month often constitute a substantial fraction of the electricity customer's monthly bill. The base transactive network template neighbor object monitors the actual demand power and compares it with highest demand power thus far in the calendar

[1] Marginal price has a formal definition in wholesale electricity markets, but the definition should be relaxed somewhat for use with decentralized transactive networks. Here, we simply refer to the marginal price function that results upon differentiation of utility cost functions. Marginal prices differ by location and time. The important point is that transactive network template prices should be derived everywhere using principles of cost minimization and should not include arbitrary factors or offsets that are not founded in market principles.

[2] The first version of the transactive network template does not account for reactive power flow, so losses may only be *estimated* while presuming constant power factor.

[3] Having the agent that imports electricity incur losses and the costs of lost energy may be politically offensive, but it is computationally much more straightforward and advantageous. An agent's electricity price exists for electricity at that agent's location. Losses are a cost of moving the electricity. If the exporter were to account for losses, it would need to keep track of not only its locational price, but also all the shadow prices that would accompany the values of electricity that might be exported to many neighboring agents. If the electricity importer accounts for energy losses, it must simply compare and choose from among the values of locally generated power resources and the corrected (increased) price of importing a neighbor's electricity. A buyer naturally should make such resource decisions.

month. The demand-charge price is added to the marginal price at and above the power demand that would exceed the month's prior peak demand.[1]

What It Means to C*onfigure* a Transactive Network Template Code Implementation

When an implementer configures a transactive network template implementation, he establishes an agent's perspective and initializes the behaviors and properties of the market, neighbor and local asset objects with which the agent must interact. This process might eventually become somewhat automated, but the first implementations have had few, relatively time-invariant neighbor and asset objects, so automating their registry has not been needed. Instead, these objects can be instantiated and configured once using a script.

If an existing class adequately represents a needed object and its capabilities, the implementer may simply instantiate that class with the new object's name and configure its properties. If new properties or methods are needed to represent a needed object and its capabilities, an existing class must be *specialized*.

What It Means to S*pecialize* a Class

A class may be specialized by creating a new class that inherits properties and methods from another. All the inherited properties and methods may be used by an object of the new, specialized class. However, the parent class's inherited properties and methods may be redefined to suit the needs and capabilities of the new class. Specialization is to be used by the transactive network template to extend its usage to new and unique objects. For example, the transactive network template base local asset class may be specialized to represent battery energy storage. The new battery energy storage class may then be further specialized to include the utility value of battery lifetime during the scheduling process, and so on.

Specialization will be most used and useful for local asset objects. Ideally, libraries of specialized asset classes will become developed over time and made available to implementers.

Specialization should never alter the interfaces between transactive network template base classes because doing so may affect the stability of other existing implementations that use the prior transactive network template interfaces.

[1] This mix of energy and demand prices is not mathematically vigorous. The intention is to forecast and approximate the real cost impacts of the demand charges in real time so that the impacts might be avoided. Some approximation is unavoidable.

Introduction to Version 2

In late 2019, the transactive network template was revised to facilitate multiple markets and different types of market negotiation practices. Whereas the initial implementation had facilitated only a type of iterative consensus negotiation between neighboring transactive agents, template Version 2 facilitates nearly any combination of consensus and auction negotiations, the temporal interactions of which may be designed during configuration of the objects in the various market classes. The coordination of multiple markets may be exemplified, for example, by a shaping market that is refined or corrected by day-ahead markets, the hours of which are then corrected by hourly real-time markets, and so on.

The following important differences are observed between iterative, consensus market mechanisms and auction market mechanisms and must be addressed if a transactive network is to broadly facilitate both such market interactions:

- <u>Price availability during the scheduling of assets</u>. A consensus mechanism simultaneously discovers both price and quantity. When an elastic asset is called to schedule its power under a consensus market, it is explicitly provided forward prices that are also being actively discovered. On the other hand, market prices are unknown, but may be predicted, at the time an elastic asset is called upon to schedule its power under an auction market. This difference has implications for the sources of price information that is available druing the scheduling of assets under alternative market types.
- <u>Market timing models</u>. Alternative market negotiation practices differ in how each defines market clearing and the types and timing of negotiation activitites that must take place in respect to the market clearing event. Especially for distributed, nested auctions, timing states must define the times at which assets are scheduled, bids are aggregated, and prices desceminated.
- <u>Agent timing coordination</u>. The distinctions betweeen market negotiation practices are largely determined by the coordination of transactive signals between neighboring transactive agents, not changes to the markets' balancing calculations and solution methods. This distinction would be missed if the template had facilitated negotiations without assigning each transactive agent distinct aggregation and price-discovery responsibilities.

Version-1 implementations instantiated market classes just once during initial system configuration and instantiated new market time intervals as needed. That approach was found to limit the facilitation of multiple, simultateous markets. In Version 2, new market objects must be instantiated for each member of a market series and its unique market clearing time. A series of market objects is derived from the same parent market class, and all its market time intervals have the same duration. More than one series of market objects may be derived from the same parent market class, but market series must be derived from different parent market classes if they require different negotiation activities (e.g., consensus versus auction) or have different purposes (e.g., day-ahead versus real-time) in the system. Therefore, each market object is assigned its own set of market time intervals as the market object itself is instantiated. Some new logic was required in Version 2 to make sure that all the new market objects are initiated as they become relevant and needed.

Given these fundamental differences between market treatments in Versions 1 and 2, the following improvements were made to the transactive network template for Version 2 and will be described in further detail in the remainder of this report:

- <u>Market state machine</u>. The base market class has been given a state machine that defines market states, the basic transitions between the states, and extensible methods (which may be redefined by children market classes) for the activities during each transition and while in each state.

Whereas timing was weakly designed in Version 1, Version 2 works in perpetually by having the configuration script loop through calls to each active markets' state machines.

- Market instantiation. The market state machine is not really relevant to a market object before a market object has become instantiated. Prior to then, another market object must be relied upon to instantiate the new market object when it is relevant and needed, and the new market object then begins to transition through its own market state machine.

- Market price prediction model. Should a long set of forward prices not be discovered in parallel with the scheduling of an agent's assets, the market must offer a model by which prices of active market time intervals may be forecasted.

- Asset price prediction model. Similarly, if there exist few or no meaningful foreward prices within the forward horizon over which an asset must assess its opportunity costs, it must somehow forecast prices over this forward time horizon and thereby assess when electricity is a bargain or expensive.

- Balancing iterations moved to market state machine. In Version 1, the markets' balancing method was inherently iterative. The fact that the private method (balancing occurs at each agent, not between agents) is iterative is not itself a problem, but intrinsic iterations are unwise because they potentially tie up the agent's calculations from performing other timely responsibilities.

Series of Market Objects and Relationships between Series of Market Objects

In Version 2, a market object defines exactly one market clearing and all the timing and activities relevant to that market clearing. This differs from Version 1, in which a market object could have many, uncountable market clearing events.

A *series of market objects* must (1) derive from the same parent market class, (2) share the same market interval duration, and (3) collectively define market delivery across the time continuum.

A simple series might, for example, consist of hour-long delivery periods, each having a market clearing event defined some duration prior to the beginning of its hour-long delivery period. Each market object is instantiated from the same market class, which defines its properties, including the specification of its single, hour-long delivery period. If a market object is instantiated for each successive hour, all time may be covered by the resulting set of delivery hours (see Figure 3).

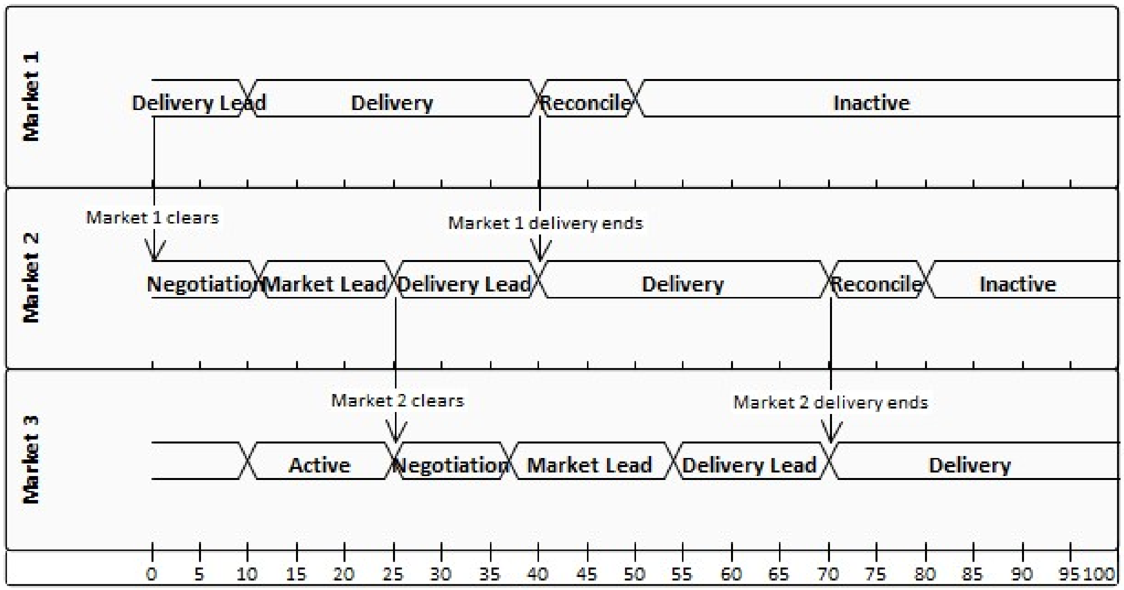


Figure 3. Coordination between Successive Market Series Objects. The market delivery periods in this example do not overlap, but the markets' delivery periods entirely cover the time continuum.

Consider another example series of market objects defining a rolling window of 24 hour-long market time intervals. Well before midnight, a set of 24 hour-long time intervals is defined from 00:00 until just before 24:00 (i.e., midnight tomorrow), and these time intervals are cleared altogether by the market (whatever that might mean for this particular series) just before midnight. A new market object is instantiated well before and cleared just before 1:00 for the 24 hours beginning at 1:00, and so on. Each market object defines a single clearing event even though the market time intervals are being allowed to overlap from one market object to the next.

The relationships between *different* series of market objects may entail refinement and correction. Consider a series of market objects defining a series 1-hour delivery periods and their clearing events. This first series may be refined by a second series of market objects, each addressing a single 15-minute market time interval. Specifically, each 15-minute time interval in the second market series is being used to refine the hour from the first market series that it subdivides. These two series might be, but are not necessarily, derived from the same parent market class. Each series individually provides full coverage of time continuum, but they are different series of market objects because they have different time interval durations. The second series having 15-minute intervals may be used to refine the coarser 1-hour intervals of the first series. Figure 4 demonstrates a market being refined by a market that subdivides its predecessor's market time intervals.

In the prior example, the succeeding market might have renegotiated all the electricity as it refined the coarser time intervals. The term *correction* refers a special type of refinement in which the quantities and prices from the prior market (the one that is to be corrected) are treated as commitments, leaving the new market to negotiate not all the electricity again, but only the changes from the scheduled powers that were cleared in the prior market that is being corrected. A commitment flag has been provided for each market object to indicate whether its cleared outcomes should be refined (commitment = false) or corrected (commitment = true).

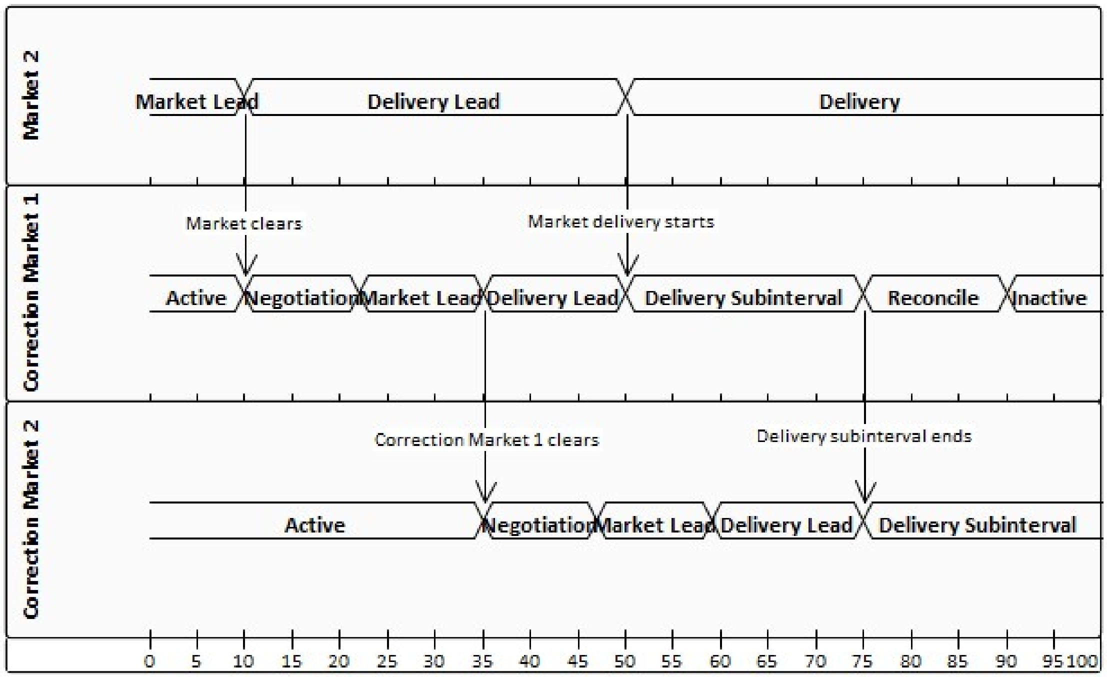


Figure 4. Example of a Market Series Timing Diagram that Subdivides and Corrects another Market Object

*Requirement*s:

- *Each market object defines exactly one market clearing event.*
- *All market objects in a series of market objects are derived from the same parent market class.*
- *All market objects in a series of market objects use the same market time interval duration.*
- *Collectively, the delivery periods of a series of market objects entirely cover continuous time.*
- *Negotiations should begin for a market object after the prior market object in its own series has cleared.*
- *Negotiations should proceed to refine or correct any market time interval after the market object owning the time interval to be refined or corrected has cleared.*

Instantiation of New Market Objects

Template Version 2 has provided versatility of market behaviors by better managing market object lifetimes. Many market objects must now become instantiated, live, and expire according to a market state machine. Many market objects must be reliably instantiated as they are needed. This section discusses how this challenge is accomplished. Once instantiated, each market object can transition through its defined state machine and important activities that have been assigned to its transitions and states can be performed.

At least one of an agent's market classes must have a method to determine when a new market object should be instantiated. Each market object keeps track of the time of its market clearing and the next market clearing time, as well. Each market object should assess whether the time has come for the next market object to become instantiated. A market object is relevant and needed only during the its lifetime defined by the state machine. The time for a new market object may be determined from the current time and the durations of all the states that must occur in a new market object before the next market clearing. The new market object is created and added to the agent's list of active market objects, and this new market object possesses all the information it needs to determine when still another one will be needed and relevant. Each market maintains a pointer to the market that precedes it (and likely created it) within its series of market objects.

Multiple series of refinement markets could, in principle, propagate each series of market objects in the fashion just described in the prior paragraph. However, complex market relationships including refinements and corrections should all be driven from one such process rather than from multiple independent processes. One dominant series of market objects—likely the one having longest time intervals and earliest relative clearing times—should instantiate both its own market series members and all the succeeding markets that refine or correct the market objects being created in its own series. This consolidation of the responsibility to instantiate needed market objects simplifies the creation of pointers from new markets to the ones that are being refined or corrected.

*Requirement*s:

- *At least one parent market class and one of its corresponding series of market objects must provide a method by which new series market objects, and potentially those of refining or correcting objects, are instantiated.*
- *Series of market objects keep track of whether their cleared outcomes should be refined (commitment = false) or corrected (commitment = true).*
- *A market object maintains a pointer ("priorMarket") to the market object it succeeds in a series of market objects.*
- *A market object maintains a pointer ("refinedMarket") to the market object it refines or corrects.*

Base Market State Machine

Once instantiated, a market object transitions through its market state machine, thus inducing all the market's needed actions and events. The following market states may represent either consensus or auction market processes. Regardless of the states' names, the actions undertaken during the various states may be unique for each different type of market negotiation.

- Inactive. A market object should be instantiated into this state and remains in this state until activated.

- Active. A market object transitions to this state at a defined time prior to its market clearing and remains in this state for the duration of a defined activation lead time.

- Negotiation. A market object transitions to this state at a defined time prior to its market clearing and remains in this state for the duration of a defined negotiation lead time. As the name implies, many markets are actively negotiated among agents during this state.

- Market Lead. A market object transitions to this state a defined time prior to its market clearing and remains in this state until the market object has cleared. The purpose of this state is to provide a duration for calculations and actions that must be completed before a market object clears.

- Delivery Lead. A market object transitions to this state when the market object clears and remains in this state until the first market interval must be delivered. The purpose of this state is to provide time for calculations and actions that must be completed between market clearing and delivery.

- Delivery. A market object transitions to this state when its time intervals begin to be delivered (a defined time after market clearing) and remains in this state until the last time interval has been delivered.

- Reconcile. A market object transitions to this state after the last market time interval has been delivered and remains in this state until the outcomes have been reconciled.

- Expired. A market object transitions to this state after it has been reconciled. Upon expiring, the market object is removed from the agent's list of active market objects.

A state machine defines conditions under which a market object may transition from one state to the next. The basic state machine is defined by the base market class and should rarely need to be overridden. Each transition and state, however, calls methods that may be overridden to define unique, critical market activities. The state machine's triggers, basic actions, and replaceable methods are summarized in Table 2.

Table 2. Methods Introduced by the Base Market Class for the Market State Machine

State	Methods Called	Triggers or Actions
Inactive		Initial state upon instantiation. Market is added to agent's list of active markets
	transition_to_active	Active period starts.
Active	while_in_active	
	transition_from_active_to_negotiation	Negotiation period starts.
Negotiation	while_in_negotiation	Negotiate.
	transition_from_negotiation_to_market_lead	Negotiation period ends.
Market Lead	while_in_market_lead	Collect market bids. Market calculations.
	transition_from_market_lead_to_delivery_lead	The market clears.
Delivery Lead	while_in_delivery_lead	Disseminate final market results. Prepare for asset controls.
	transition_from_delivery_lead_to_delivery	Delivery of market periods begins.
Delivery	while_in_delivery	Meter delivered electricity. Control scheduled electric power.
	transition_from_delivery_to_reconcile	Last delivery period ends.
Reconcile	while_in_reconcile	Reconcile transactions.
	transition_from_reconcile_to_expire	Market is reconciled. Market is removed from agent's list of active markets.
Expire		

Figure 5 provides another useful view of the state machine and its mappings to its replaceable methods. Additionally, several of the important objects typically created in the various methods are indicated, according to whether each is created by the iterative consensus (i.e., “iterative”) or bilateral auction (i.e., “auction”) negotiation mechanisms.

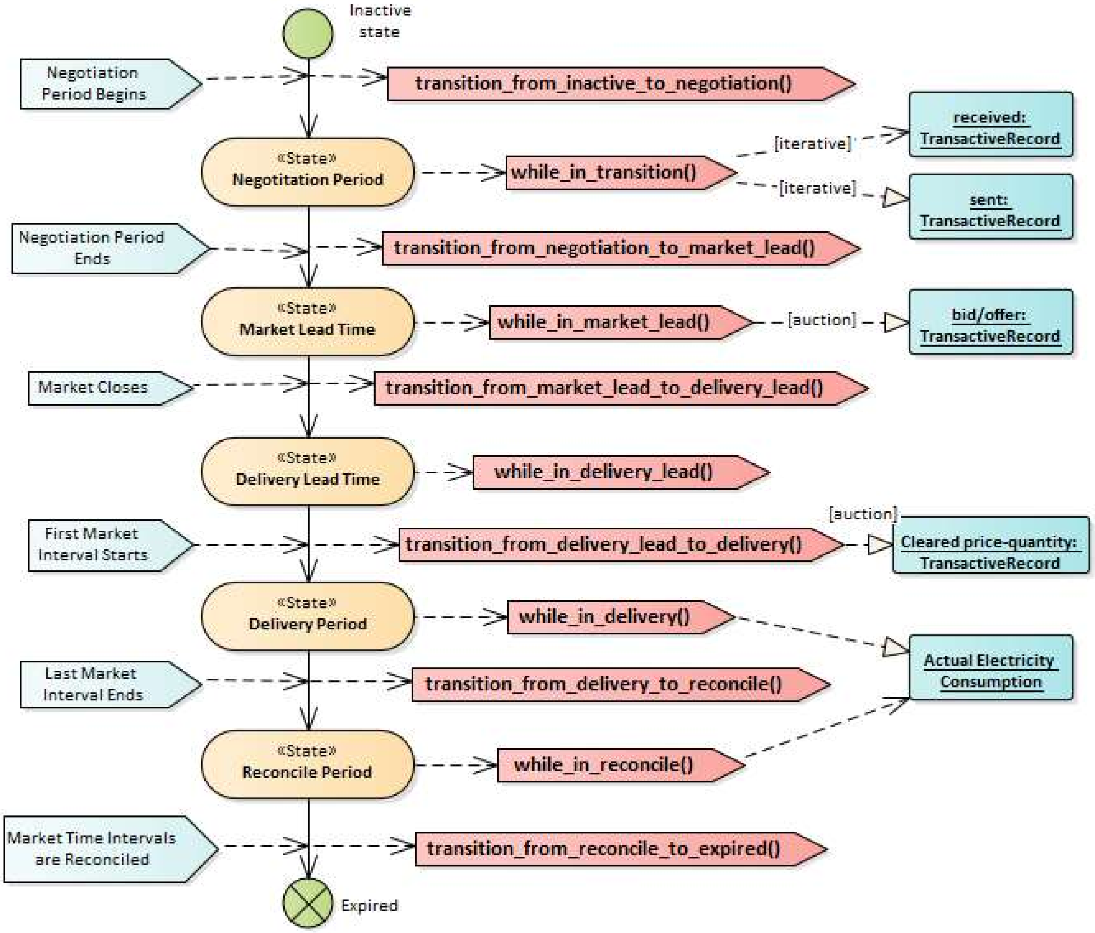

Inactive state
Negotiation Period Begins
transition_from_inactive_to_negotiation()
received: TransactiveRecord
[iterative]
«State» Negotitation Period
while_in_transition()
[iterative]
sent: TransactiveRecord
Negotiation Period Ends
transition_from_negotiation_to_market_lead()
«State» Market Lead Time
while_in_market_lead()
[auction]
bid/offer: TransactiveRecord
Market Closes
transition_from_market_lead_to_delivery_lead()
«State» Delivery Lead Time
while_in_delivery_lead()
[auction]
First Market Interval Starts
transition_from_delivery_lead_to_delivery()
Cleared price-quantity: TransactiveRecord
«State» Delivery Period
while_in_delivery()
Actual Electricity Consumption
Last Market Interval Ends
transition_from_delivery_to_reconcile()
«State» Reconcile Period
while_in_reconcile()
Market Time Intervals are Reconciled
transition_from_reconcile_to_expired()
Expired

Figure 5. The Lifetime of a Market Object based on its State Machine. Market activities may be assigned to these transitions and states to drive the needed market activities.

Consensus Versus Auction Timing using the Market State Machine

The market state machine and its methods may be used to define consensus, auction, and potentially other negotiation practices. Auction and consensus classes have been coded as children of the base market class. The base class’s state machine works for both of the new market classes, but the new market classes replace certain of the base classes transitional and state methods (see Table 2) to conduct actions at the right times and in the right order.

The timing of the most important consensus and auction activities are contrasted in Figure 6. The top panel of this figure is a representative timing diagram of market states and transitions. The actual durations of each state are unimportant and are configurable to the needs of each series of market objects.

The second panel of Figure 6 shows the major activities of an iterative consensus negotiation within the provided states. The negotiation period is most important as the window within which all prices and quantities must be iteratively negotiated. Locally, each agent must converge on the local electricity price and on the quantities to be generated or consumed by each local asset during the negotiation state. Furthermore, the agent must converge with all its neighboring agents concerning the prices and quantities of electricity to be exchanged during the negotiation state. The market lead and delivery lead states may be short, as they simply provide a short time for agents to prepare for the delivery of the negotiated electricity. Market clearing is not a distinct action in an iterative consensus negotiation, but, even if fictional, it is an important anchor for timing within the market state machine.

The third and fourth panels of Figure 6 show the activities of an *auction* market within the market states. Unlike the consensus approach, auctions inherently possess concepts of upstream and downstream within a distribution system, and neighbor agents are assigned one of these directions based on the expected power flow direction. An *upstream* neighbor is an electricity supplier and provides price discovery for the local agent; a *downstream* neighbor is an electricity consumer and aggregates demand bids to be cleared by the local agent during its balancing process. The third panel shows an auction agent's actions in respect to its upstream neighbors, and the fourth panel in respect to its downstream neighbors. Every auction agent is welcome to prepare and aggregate its own asset bids (and offers) during the negotiation state (not explicitly shown). The market lead state is used to send aggregated bids upstream and to receive aggregated bids from neighboring downstream agents (third panel). The delivery lead state is used to send cleared market offers to downstream neighbor agents and to receive such offers from upstream neighbor agents (fourth panel). Because the auction process is not iterative, additional logic is needed to make sure that bids and offers are coordinated throughout a string of upstream and downstream neighbor agents (i.e., throughout the transactive network). An agent should await bids from all its downstream agent neighbors before it finalizes its aggregated bid and sends it to its upstream neighbor agent. Then, during the delivery lead state, the agent should await results from its upstream neighbor's the market clearing before clearing its own market and sending the resulting offers to its downstream neighbor agents. The respective market states must be long enough for this information to be conveyed throughout the network. Future work is needed for template Version 2 to accommodate error cases when an agent fails to conduct its actions within the provided states.

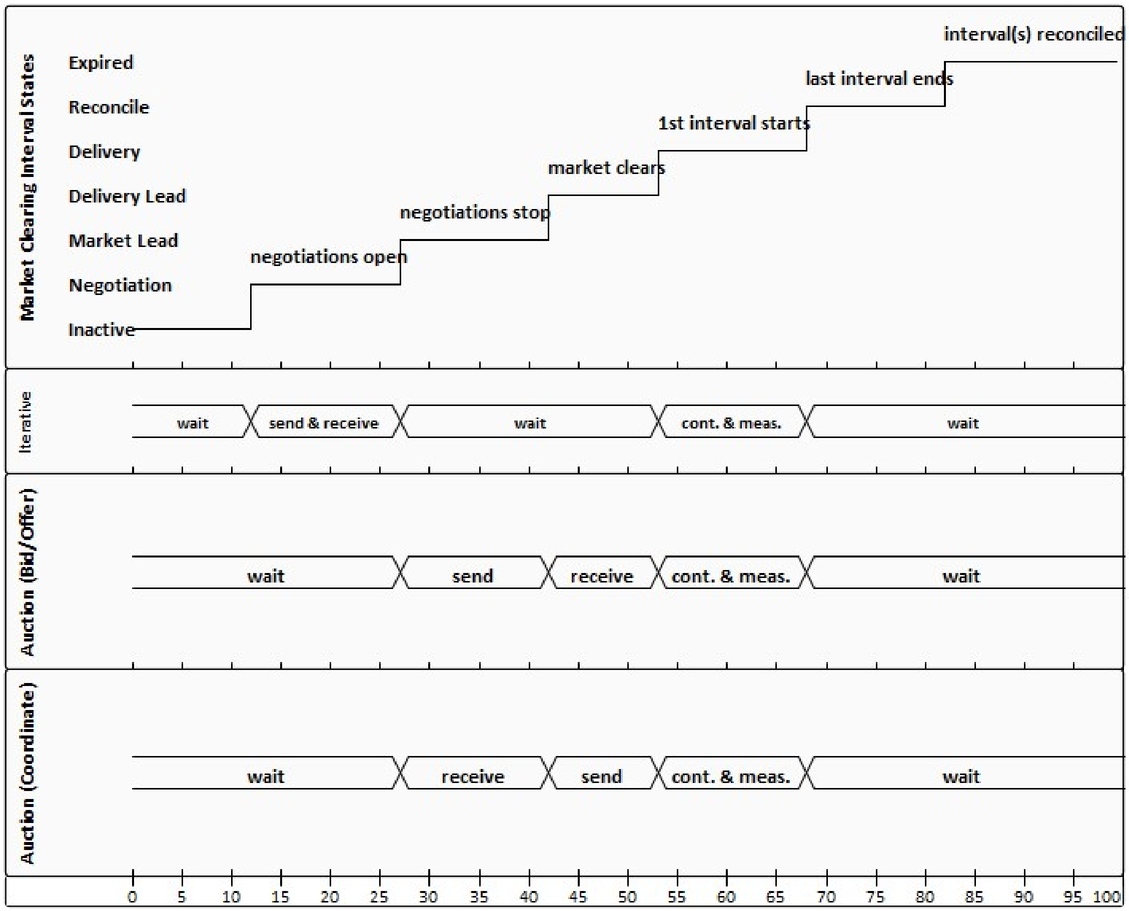
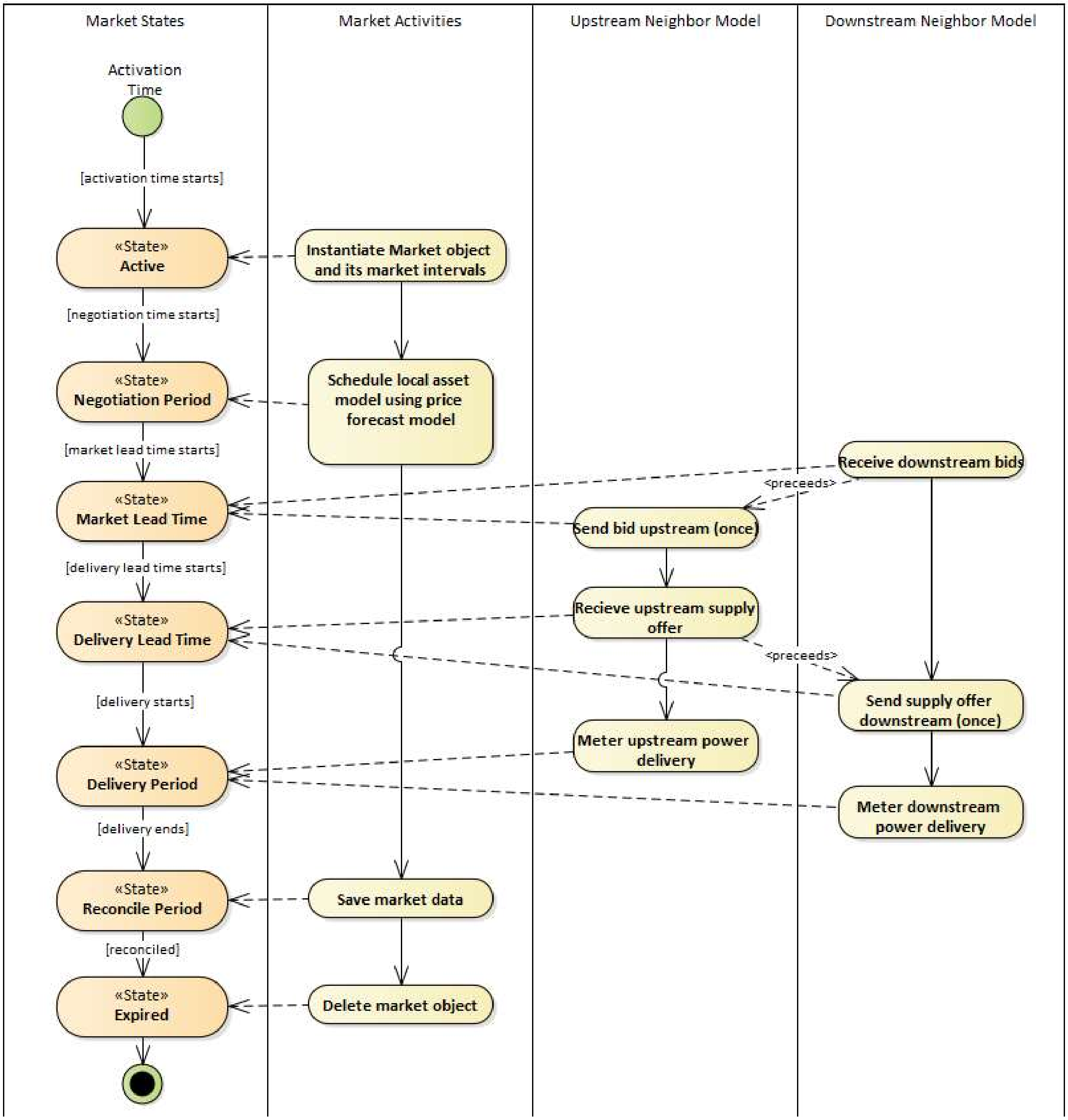


Figure 6. Market States and Transitions applied to Iterative Consensus and Auction Negotiations

Figure 7 and Figure 8 provide different views of the differences between auction and iterative consensus negotiation mechanisms, but these figures emphasize the activities that market objects and neighbor models (which may be further designated by “upstream” or “downstream”) are responsible in the various market states. Observe especially that an auction (i.e., Figure 7) possesses intrinsic ordering of neighbor activities that is not needed in iterative consensus negotiations.



Market States
Market Activities
Upstream Neighbor Model
Downstream Neighbor Model
Activation Time
[activation time starts]
«State» Active
Instantiate Market object and its market intervals
[negotiation time starts]
«State» Negotiation Period
Schedule local asset model using price forecast model
[market lead time starts]
Receive downstream bids
<preceeds>
«State» Market Lead Time
Send bid upstream (once)
[delivery lead time starts]
«State» Delivery Lead Time
Recieve upstream supply offer
<preceeds>
Send supply offer downstream (once)
[delivery starts]
Meter upstream power delivery
«State» Delivery Period
Meter downstream power delivery
[delivery ends]
«State» Reconcile Period
Save market data
[reconciled]
«State» Expired
Delete market object

Figure 7. Auction Market Activities Mapped into the Market State Machine

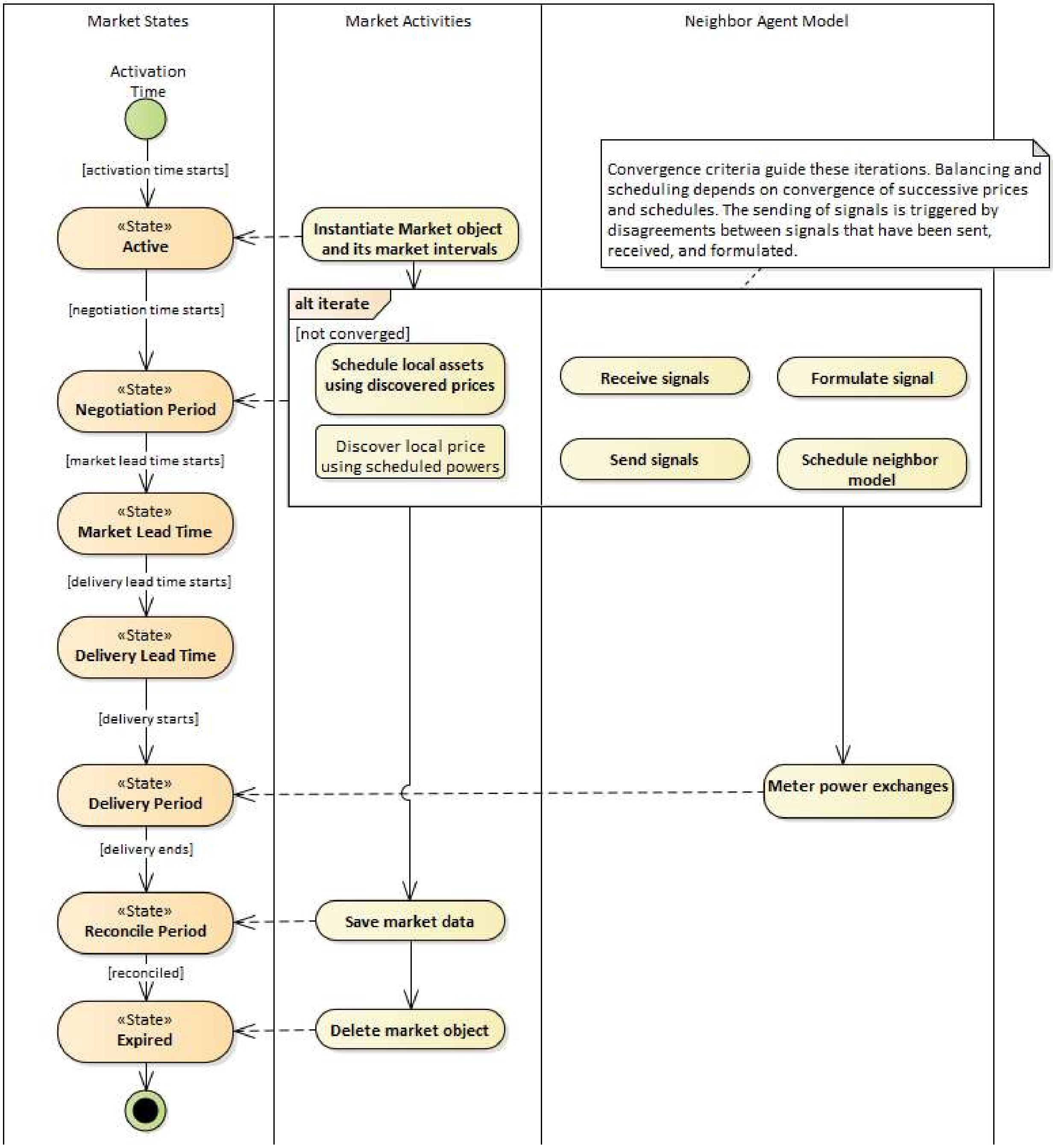

Market States
Market Activities
Neighbor Agent Model
Activation Time
[activation time starts]
«State» Active
Instantiate Market object and its market intervals
Convergence criteria guide these iterations. Balancing and scheduling depends on convergence of successive prices and schedules. The sending of signals is triggered by disagreements between signals that have been sent, received, and formulated.
[negotiation time starts]
alt iterate
[not converged]
Schedule local assets using discovered prices
Receive signals
Formulate signal
«State» Negotiation Period
Discover local price using scheduled powers
Send signals
Schedule neighbor model
[market lead time starts]
«State» Market Lead Time
[delivery lead time starts]
«State» Delivery Lead Time
[delivery starts]
«State» Delivery Period
Meter power exchanges
[delivery ends]
«State» Reconcile Period
Save market data
[reconciled]
«State» Expired
Delete market object

Figure 8. Market and Neighbor Model Activities during Iterative Market Consensus Negotiations

Market Price Prediction

The transactive network template, as originally implemented, used consensus negotiation over a rolling window of 24 hour-long market time intervals. This approach simultaneously discovers both prices and quantities, so the forward prices are meaningful. Version 2 extends applicability to auction negotiations, which perform price discovery only after bids have been submitted and which often feature single lone forward time intervals. Therefore, Version 2 offers methods to locally estimate prices for a market's forward time intervals should explicit, discovered prices be unavailable at the time bids must be calculated.

Given the possibilities of various levels of system sophistication, the markets preferences for forecasting forward prices should try these methods in this order until forecasts exist for each time interval that is to be cleared. The market object predicts prices for only its active forward time intervals, no more than that.

1. Actual market price. If a forward market time interval has already been assigned a price, this price should be accepted and used for the market's forward time intervals.

2. Price forecasted or discovered for this same time by a prior market object in the same market series. If a prior member of the same series of market objects possesses a price for the forward market time interval, this price should be adopted. This will occur only if sequential markets in a series overlap one another.

3. Price discovered for the same time in another market object that is being corrected. If a market object being refined or corrected possesses a price, this price should be adopted. Revised or corrected markets must always include the market time interval that is revising or correcting the first.

4. Price forecasted by the price model of the market series. A price model uses historical prices to predict future prices. A simple model was provided to the base market class to update and predicts average and standard deviation prices for the 24 hours in a day. If provided newly cleared prices upon transitions to delivery states (when no further changes to electricity prices should be possible), a market object can learn and predict its hourly trends. See Section 0 for further details.

5. A static default price of the market series. As a last resort, forward prices may be assigned the market series' static default, which is assigned during initial system configuration. This is, of course, less than ideal because it entirely fails to represent price dynamics.

Hourly Price Prediction Model

A list of average hourly prices and their standard deviations is provided from the base market class in Version 2, and a rudimentary method is provided to update and retrieve price data from this model. This section describes this price forecasting model which is now available for use by any market object that inherits from the base market class.

Market property "priceModel" is a list of 48 values. Given an hour *h* in the range 0 to 23, the average price for the hour is indexed in this list by *2*h*, and its standard deviation is indexed by *2*h+1*.

The base market class provides the low-pass filtering method

model_prices(*self, datetime, price, k*=*14*);

where parameter *datetime* is the time containing the prediction hour; parameter *price*, if provided, is a newly discovered price datum that updates the market object's hourly price data; and *k*, if provided, specifies the time constant of the updating function based on numbers of new price data. The default value *k=14*, for example, specifies a response time of two weeks (i.e., 14 new price data) if a new price datum is supplied once each hour.[1]

When the method is used for prediction, only parameter *datetime* is needed. The method simply looks up and replies the current average electricity price ($/kWh) and standard deviation electricity price ($/kWh) that corresponds to the hour or parameter datetime.

If the method is called with both parameters *price* and *datetime*, the method uses parameter *price* to update the data in the market object's list and replies with the updated price and updated standard deviation for the hour. The update of average price $\hat{\lambda}$ from provided *price* parameter λ_{new} is calculated as in (2), and the update of standard deviation price σ is updated as in (3).

$$\hat{\lambda}_{new} = \frac{(k - 1) * \hat{\lambda}_{old} + \lambda_{new}}{k} \tag{2}$$

$$\sigma_{new} = \left(\frac{(k - 1) * \sigma_{old}^2 + (\hat{\lambda}_{new} - \lambda_{new})^2}{k}\right)^{0.5} \tag{3}$$

This price forecast model is admittedly rudimentary. Implementers are invited to replace the method if greater sophistication is necessary.

Asset Price Prediction

Market objects call on assets to schedule their power and elasticity. Much as for the market, elastic assets may need to forecast prices farther into the future than prices are being supplied by the market object. Without knowledge of future prices, the asset cannot compare its current and future opportunity costs. The new asset property "scheduleHorizon" should state the future time horizon over which an asset can properly schedule its power. Typically, an elastic asset should have its scheduling horizon set to a duration that is about twice the time interval over which its scheduled energy can be shifted. A 24-hour horizon is sensible for many assets—water heaters, building air conditioning, and residential batteries, for example—that might shift their demand by approximately 12 hours. An asset's price horizon may be unique to the asset and the way it is scheduled.

Think of the scheduling horizon as the forward time over which the current price opportunities must be calibrated. The horizon may be affected both by an asset's capabilities and by expectations about market price patterns and volatility. Where prices have a strong diurnal trend, horizons for should be no shorter than 24 hours.

An asset could use many means to glean or predict expected future prices for the extent of its scheduling horizon. The following approaches are tried in this order until suitable price forecasts have been obtained for all included market time intervals:

1. Actual market price. If prices have been provided by a market object for its active time intervals, those prices should be used.

[1] Incidentally, early transactive auction implementations used only the prices from the preceding 24 hours to calibrate opportunity costs. That practice can be implemented when *k = 1*, in which case the stored prices are exactly the prior 24 updated prices. The calculated standard deviation is always zero when *k = 1*.

2. Prior market price. If a price is available from a prior market object in the same series of market objects as the one calling on the asset to schedule itself, that price should. This will occur only if successive market objects have delivery periods that overlap.

3. Corrected market price. Similarly, if a price is available from a market object that is being corrected, that price should be used.

4. Modeled price. As of Version 2, markets will offer prediction methods that may be mined for suitable price predictions. (See Section 0.)

5. Static default price. As a last resort, the market object's static default price may be adopted, but this approach is not especially useful for predicting and anticipating price dynamics.

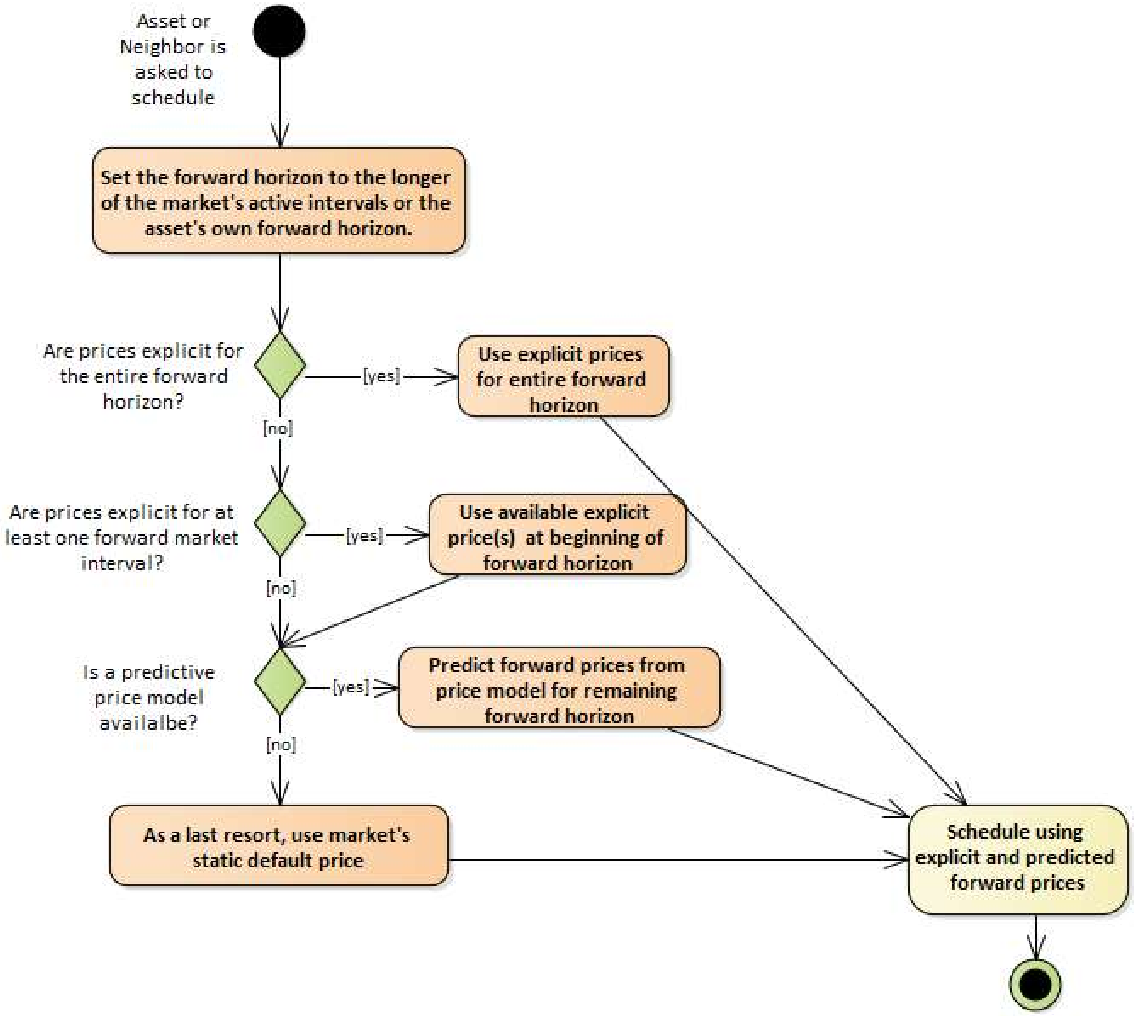

Asset or Neighbor is asked to schedule
Set the forward horizon to the longer of the market's active intervals or the asset's own forward horizon.
Are prices explicit for the entire forward horizon?
[yes]
Use explicit prices for entire forward horizon
[no]
Are prices explicit for at least one forward market interval?
[yes]
Use available explicit price(s) at beginning of forward horizon
[no]
Is a predictive price model availalbe?
[yes]
Predict forward prices from price model for remaining forward horizon
[no]
As a last resort, use market's static default price
Schedule using explicit and predicted forward prices

Figure 9. An Asset Model's Price Prediction Strategy prior to Scheduling

Correction Market Translations of Prior Bids and Offers

This section discusses a general strategy to use when a market object clearing is treated as a commitment, thus leading later markets that refine rather than replace the result of the prior corrected market clearing. If committed, generation and consumption in the corrected market must still be accounted at that market's prices, but incremental (or decremental) energies that become scheduled thereafter are subject to newly discovered correction market prices. The general ramification is that correction markets' bids and offers become translated by the power that was cleared in the market that is being corrected. Because only the differences from the prior market are being negotiated, there is likely greater price volatility in correction markets than in the market that is being corrected.

The following steps are performed when a new market is intended to correct a commitment in a prior market:

1. Preliminary assumptions of data availability. The market object being corrected is assumed to have cleared price and quantity for the time interval or intervals of interest. Asset and neighbor objects possess scheduled powers, which reveal their cleared quantities in the market object and time interval that is being corrected.

2. Neighbors are scheduled. Because neighbor models intentionally ignore intertemporal effects during their scheduling, their bids or offers in the new market may be had by simply translating their current demand or supply curves by the average electric power that was cleared and committed in the market that is to be corrected (see Figure 10).

3. Assets are scheduled. The supply or demand curves are updated over the asset's scheduling horizon using the strategies of Section 0 for predicting prices. The resulting demand or supply curves are then translated by the power that was cleared for the asset in the prior market that is being corrected.

4. The transactive agent balances the correction market. It uses both the immutable committed power quantities cleared in the market being corrected for each asset and neighbor and the new bids and offers from these entities that use only the *change* in power from the prior clearing. A correction market price is determined by the balancing process. The committed power quantities cleared in the market being corrected are again subtracted from the residual supply and demand curves that are then to be sent back to neighbors and assets.

5. The assets and neighbors note the average power quantities cleared in the correction market. The assets use both the scheduled prior commitments and corrections to control the assets during the delivery period.

Figure 10 offers examples how supply and demand curves from local asset and neighbors are translated using the power that had been cleared in the prior market that is being corrected. The top curves represent the supply and demand curves made into the prior market, and the bottom ones represent revised, translated supply and demand curves to be bid into the corresponding correction market. Cases (a) to (c) exhibit no change in supply or demand curve since the prior market. Their total scheduled powers would remain unchanged if the correction market were to clear at the same price as for the one being corrected. Case (d) exhibits a change in the general shape and position of its supply curve since the prior market. Such changes may occur if the asset's required utility changes, external stimuli change, or available flexibility changes or is unavailable now that the delivery period is nearer. Even if the correction market clears at the same price as the prior market, a correction must be made (i.e., a price paid) for the reduced power that this generation asset is now able to offer.

Figure 10a shows the correction of a generator offer. The prior market cleared at the dashed vertical line, which corresponds to the power commitment of the horizonal dashed line. In the correction market, the generator effectively has two offers: First, its immutable power commitment from the prior market is represented by an infinite horizontal line at the committed power level. Its updated supply curve (which happens to be identical to its prior supply curve in this case) is translated downward by the committed power level. Its scheduled power may be either increased or decreased, depending on the price that is discovered by the correction market.

Figure 10b follows similar logic as for Figure 10a, but this example is for a demand asset.

Figure 10c follows the same logic still again, but this time for a flexible battery asset. The price cleared in the prior market committed it to discharge (i.e., act like a generator) at its maximum offered discharge rate. After this new curve is translated downward by the committed power quantity, its residual flexibility cannot induce discharging energy regardless how high the price discovered by the correction market, but the battery system could be incentivized to reduce its discharge and eventually charge at low correction market prices.

Figure 10d shares the same generator supply curve into the prior market as for the generator of Figure 10a, but conditions are found to have changed for the generator since it made that prior offer. Its costs have decreased, as shown by a leftward shift in its offer; it has much less supply to offer, as shown by how low its powers become after its translation; and its flexibility has diminished, as shown by how little change in power is offered. Even if the correction market were to clear at the same price as the prior market, this generator must reduce its scheduled supply power and probably pay to do so in the correction market.

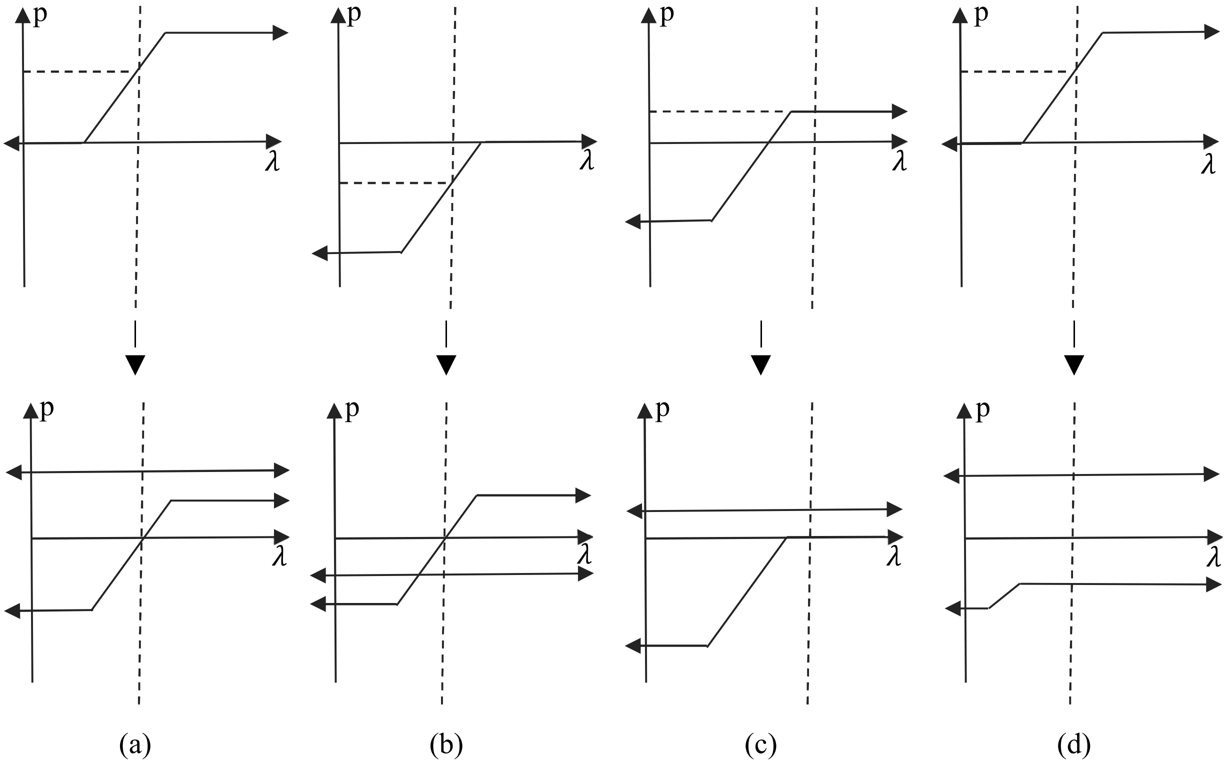


Figure 10. Correction Market Translations of Example Asset's or Neighbor's Supply or Demand Curves. Cases include corrections of (a) a generator offer, (b) a demand bid, (c) a battery system bid or offer, and (d) a generator, where its available flexibility and costs have changed since its offer into the market object that is being corrected. (*p* is power and *λ* is price)

Configuration Example

The current PNNL Transactive Campus Project is to be revised to demonstrate and test the transactive network template's new Version-2 functionality. Specifically, the current 24-hour rolling window of hourly consensus iterations is to be replaced by a single day-ahead bilateral auction that clears once per day at 9:45 prior to its first delivery hour at 10:00. Additionally, the results of the day ahead market are to be refined by a real-time bilateral auction that corrects commitments from each day-ahead hour using four subdivided 15-minute intervals. This section discusses critical configuration steps that will cause the desired market behaviors. Python configuration code will be used as an example. The intention of this section is to introduce important principles, not to offer complete code.

Assume the auction is being configured for agent using object "dAA". The following line would occur very early in its configuration script. This is important because its is this object that will keep track of all the active market objects:

```
myTN = myTransactiveNode()
```

State Machine Driver

Each agent's configuration script should conclude with a small code loop that has each current market object update its market state and all its corresponding activities and events. Assuming node "myTN" has been instantiated to represent the local transactive agent, this following snippet of Python code will recursively call on active market objects to move through their state machines.

```
For x in range(len(myTN.markets)
    myTN.spawn_markets()
    myTN.markets[x].events()
```

At least one market object must be configured to start the process. In our current example, a script must be used to instantiate the very first day-ahead market object and assign its properties. Thereafter, successive day-ahead markets will be instantiated as they are needed. Because the real-time market objects correct the day-ahead market objects, it is preferable to have the real-time correction markets instantiated at the same times as the day-ahead intervals that are to be corrected. This means that the day-ahead market's method spawn_markets() must be replaced to also generate the needed real-time market objects.

Day-Ahead Market Configuration

Many the auction day-ahead market's behaviors may be used directly from the auction base class, but the auction base class must be specialized here to have the day-ahead markets instantiate the corresponding real-time correction markets. Therefore, a preliminary step is to create new child class DayAheadAuction[1] and use this new class to redefine method spawn_markets():

```
class DayAheadAuction(Auction):
    …
    def spawn_markets():
        …
```

[1] The implementer is free to choose the class name, but it should be descriptive of the extension that is being made.

The very first day-ahead market is instantiated within a configuration script as follows:

1. Instantiate the first day-ahead auction market object from the newly extended auction class, which is a child of the transactive network template auction and base market classes:

   ```
   dAA = DayAheadAuction()
   ```

2. Assign the new auction object 24 market intervals. This, in conjunction with the defined interval duration, establishes the market delivery period duration:

   ```
   dAA.intervalsToClear = 24
   ```

3. Assign the interval duration. This, in conjunction with the number of intervals to clear, establishes the market delivery period duration:

   ```
   dAA.intervalDuration = deltatime(hours=1)
   ```

4. Assign the current market clearing time for the market that is currently in delivery. This may point to either 9:45 today or 9:45 yesterday (i.e., 15 minutes before delivery), depending on the current time. This assignment anchors the timing of all future market clearing times:

   ```
   dAA.marketClearingTime = datetime(year=2019, month=11, day=10, hour=9, minute=45,
           second=0, microsecond=0)
   ```

5. Assign the market clearing interval. Doing so establishes the series of future market clearing times:

   ```
   dAA.marketClearingInterval = deltatime(hours=24)
   ```

6. Assign the next market clearing time using the current market clearing time and market clearing time interval. This time will determine by when the next day-ahead market object will be needed.

   ```
   dAA.nextMarketClearingTime = dAA.marketClearingTime + dAA.marketClearingInterval
   ```

7. Assign a name to this market series, which should be the root of all future market object names in this series:

   ```
   dAA.marketSeriesName = “Day Ahead Market”
   ```

8. Assign state durations. These will be used to trigger certain state transitions in respect to market clearing times. It may be helpful to refer to the state machine of Figure 6. There is some freedom while assigning these durations. Each duration must be long enough to conduct all agent calculations, and the sum of all the lead times should not induce market negations before the prior market object has cleared.

   ```
   dAA.deliveryLeadTime = deltatime(minutes=15)
   dAA.marketLeadTime = deltatime(minutes=15)
   dAA.negotiationLeadTime = deltatime(minutes=30)
   dAA.activationLeadTime = deltatime(minutes=0)
   ```

9. *Confirm that the results of this day-ahead clearing are commitments to be corrected in the later real-time markets:*

```
dAA.commitment = True
```

10. The market should be instantiated in its “Inactive” market state. The market state will be quickly corrected by calls to method events() that exercise the market object’s state machine.

```
dAA.marketState = MarketState.Inactive
```

11. Place the configured day-ahead market into the agent’s list of markets. By doing this, the market object will be placed in a queue and matriculate through its state machine.

```
myTN.markets = [dAA]
```

Real-Time Market Configuration

As discussed above, real-time market object should be instantiated within the replaced spawn_markets() method of class DayAheadAuction, not in the configuration script. Therefore, the day-ahead market’s method spawn_markets() should instantiate not only the next new market object in its own series, but also all the real-time correction markets that will be needed to correct all the day-ahead time intervals. Let “new_dAA” reference the newly instantiated day-ahead market. The following code elements would be placed in the replaced spawn_markets() method:

1. The code inherited from the base market classes is probably adequate for the instantiation of the next day-ahead market object:

```
super().spawn_markets()
```

Let “new_dAA” reference the newly instantiated day-ahead market.

2. Real-time auction markets must be instantiated to fully cover the delivery period of the day-ahead market object. A loop is created, and real-time market objects are instantiated to cover the entire day-ahead delivery period. The base auction class is suitable for the real-time market objects because no new behaviors are needed. Because the instantiation steps are like those for the day-ahead markets above, the steps will not be separately enumerated:

```
deliveryStart = new_dAA.marketClearingTime – new_dAA.deliveryLeadTime
deliveryEnd = deliveryStart + new_dAA.marketClearingInterval

while deliveryStart < deliveryEnd

    rTA = Auction()

    rTA.intervalDuration = deltatime(minutes=15)
    rTA.deliveryLeadTime = deltatime(minutes=5)
    rTA.marketClearingTime = deliveryStart – rTA.deliveryLeadTime
    rTA.marketClearingInterval = deltatime(minutes=15)
    rTA.nextMarketClearingTime = rTA.marketClearingTime + rTA.marketClearingInterval
    rTA.marketLeadTime = deltatime(minutes=5)
    rTA.negotiationLeadTime = deltatime(minutes=5)
    rTA.activationLeadTime = deltatime(minutes=0)
    rTA.marketSeriesName = “Real-Time Auction”
    rTA.name = rTA.marketSeriesName + “_” + str(deliverStart)
    rTA.marketState = MarketState.Inactive

    myTN.markets.append(rTA)

    deliveryStart = deliveryStart + rTA.intervalDuration
```

Now, every time a new day-ahead market become instantiated, 96 real-time market objects will also be instantiated. Each markets remains in an inactive state in the agent's queue until it becomes time for it to transition into its negotiation state, as prescribed by the market state machine.

Suggested Future Work

The transactive network template was designed as a template for implementing transactive energy implementations in the electricity domain. Its object-oriented design establishes a structure of object types and object behaviors that will be needed to represent any circuit region within a transactive network. A useful separation of computational responsibilities has been designed into the transactive network template's base classes. Local asset objects schedule electricity consumption and generation and can thereby represent the price flexibility of an extremely diverse set of devices and systems. Market objects ensure that a local price is discovered that will balance the supply and demand of all electricity to be generated, consumed, or exchanged in the agent's circuit region. Neighbor objects manage the coordination that must occur between neighboring agents. The transactive network template further supports extensibility to address new grid objectives and, especially, to engage new local assets devices and systems and their unique scheduling needs and strategies.

The remainder of this section shall address future improvements of the transactive network template.

Power Flow Improvements

The current transactive network template version facilitates a simple pooled market approach to represent electricity exchange between transactive neighbors. Neighbors are presumed to be able to exchange electrical power within stated capacity limits. Reactive electric power and voltage are neither tracked nor managed in the transactive network. This simplification may be necessary and justified given the current reluctance and limited abilities of real-world entities to meter and, worse yet, accurately forecast their voltage or reactive power. Transport constraints and losses can be estimated as functions of electric power even though transport constraints and losses should be more accurately stated as functions of electric current. Admittedly, however, applications in microgrids and in circuits having high penetrations of intermittent renewable resources may necessitate voltage management that cannot be adequately addressed by the current transactive network template version.

Two improved versions of the transactive network template market base class are foreseen to better facilitate reactive power and voltage management. First, DC power flow principles should be applied to introduce reactive power flow and voltage and to estimate their interdependency. Voltages differences are estimated using DC power flow, so voltage constraints may be introduced and addressed upon its implementation. Reactive and real power flows are frequently decoupled in transmission system studies, where per-unit voltages lie close to unity and where transmission impedances are predominantly reactive, but this decoupling is not justified in general. The transactive network template can be applied to distribution and even smaller circuit regions. DC power flow models are generally stable, giving one hope that the decentralized application of these principles to the transactive network template might also be reliable and stable.

Eventually, a transactive network template market version using full, accurate AC power flow principles should be developed to accurately address circuit voltages. Unfortunately, the resulting equations are difficult to solve and may be found to introduce instabilities and reduced reliability to the transactive network template markets' balancing process.

The success of these improvements in the field may be limited by implementers' willingness to monitor and forecast their reactive power and voltage. Forecasts throughout the transactive network may, in fact, become less accurate if transactive agents misstate their voltages and reactive power needs and pay no penalty for doing so.

Whereas the current transactive network template version has its transactive neighbor and local assets similarly participate in the market object's balancing objective, future versions implementing DC and AC power flow principles will require new balancing calculations be used for the transactive neighbor objects. In the new versions, power exchange should not be independently asserted. Instead, power exchange is necessarily dependent upon the local circuit region's voltage, the neighbor circuit region's voltage, and the impedances of the interceding transport elements. A transactive agent may take actions to change its own complex voltage, but it cannot change other remote circuit regions' voltages in what might be a highly meshed electrical network. And its ability to do so might be heavily constrained by a constraint on local voltage, other neighbors' voltages, and constrained power flow capacities.

Support Financial Transactions

Pilot implementations of transactive systems have had their dynamic locational prices enforced to differing degrees. The transactive networks' dynamic prices have frequently been permitted to diverge from electricity billing practices and are therefore ignored or must be corrected when calculating actual customer bills. The current transactive network template design anticipated flags to mark agents' commitments to prices and quantities. A reconciliation market state was created to allow time for market outcomes to become resolved and settled. The revenue implications and practices for those who implement and participate in transactive networks may be significantly more dynamic than those that predominate today. Transition to a transactive world will be considered risky. The connections between transactive network processes and electricity billing practices must be facilitated and tested.

Training Tutorials

PNNL has attempted to teach the transactive node template to other potential implementers for the establishment of new transactive networks. Success has been mixed. The reference implementation is relatively new and fragile and is admittedly limited in the types of interactions and asset models that have been coded. The template introduces concepts and definitions that are foreign, at first, to many implementers.

The entire PNNL campus network implementation has been offered as an example, but this example has shortcomings. The implementation is closely integrated with the Volttron communication platform and is not easily exercised without also teaching and implementing the Volttron platform. Furthermore, its commercial building asset models are not fully compliant with the template's base classes.

For all these reasons, a simple reference implementation and tutorial are needed.

1. The student should develop one and only one agent's perspective without having to plan and implement an entire network. Working code for the tutorial's implementations should be downloadable and able to be run apart from any presumed computational or communication platforms.

2. The tutorial's starting point should possess just one example asset. The asset should at first have a constant, inelastic demand, but the tutorial would teach the student to create increasingly time-variant and price-responsive behaviors for the asset. The student would be led through the process for adding another new asset.

3. The tutorial must supply a non-transactive neighbor with which the student's agent interacts. The tutorial should then introduce the student to an emulated transactive neighbor with which transactive signals may be exchanged. The student would be taught to connect with a new transactive neighbor in the network.

4. Finally, the student would be taught to modify the provided market configuration, to implement anotehr market to refine the first, and to create other new market behaviors.

Having completed these tutorial steps, the student would be prepared to apply the transactive network template and its principles and construct a new transactive network.

Appendices

Appendix A: Recommended Relational Database Structure

This appendix depicts tables and views of a relational database that could be used to capture data from the transactive network template. For clearer presentation, the database has been parsed into multiple diagrams. Each diagram features one relational table (highlighted in yellow) and the important primary and secondary key relationships between the table and other database tables or enumerations. While current reference implementations of the transactive network template address data collection according to platform preferences (e.g., the Volttron environment), future versions of the transactive network template should facilitate this recommended database to make future system implementations more platform independent.

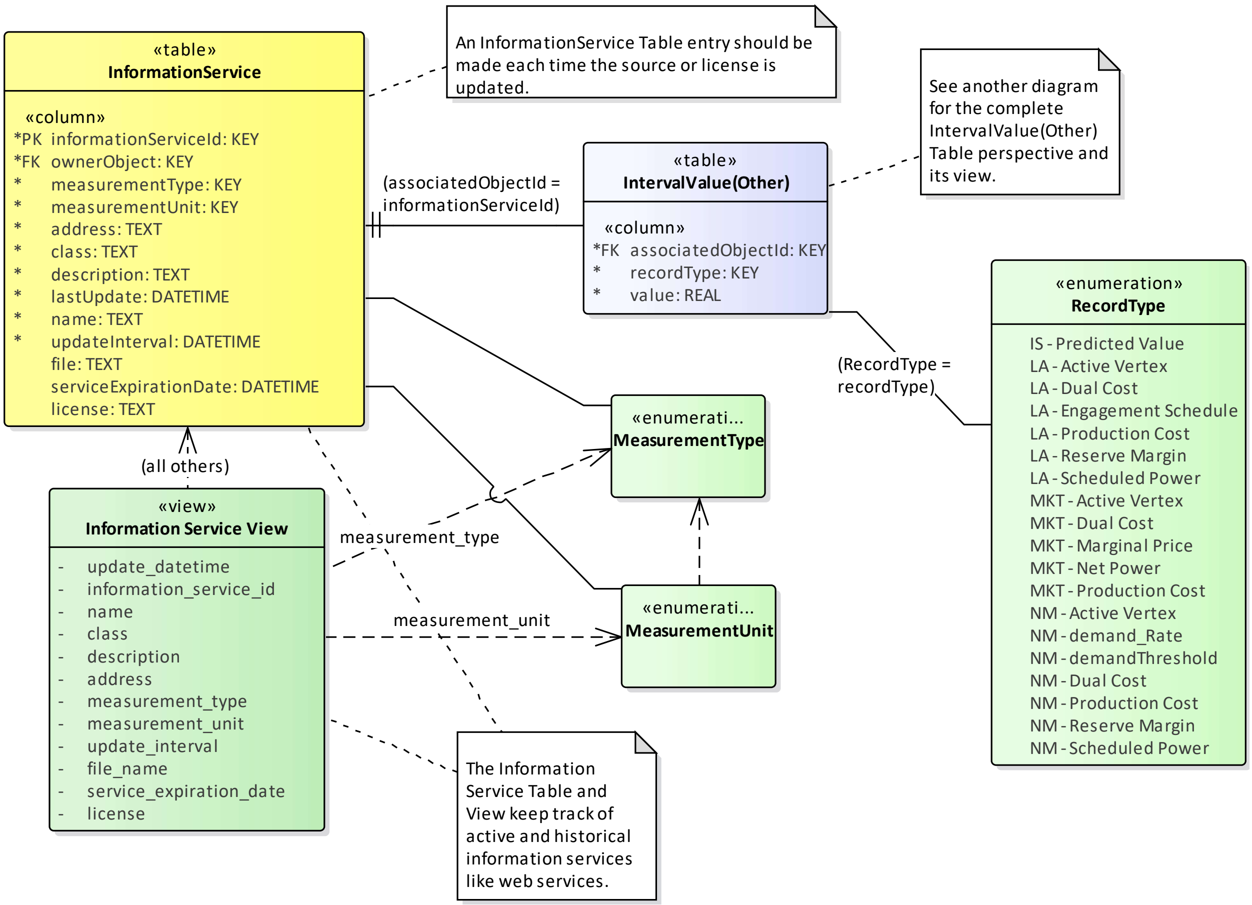
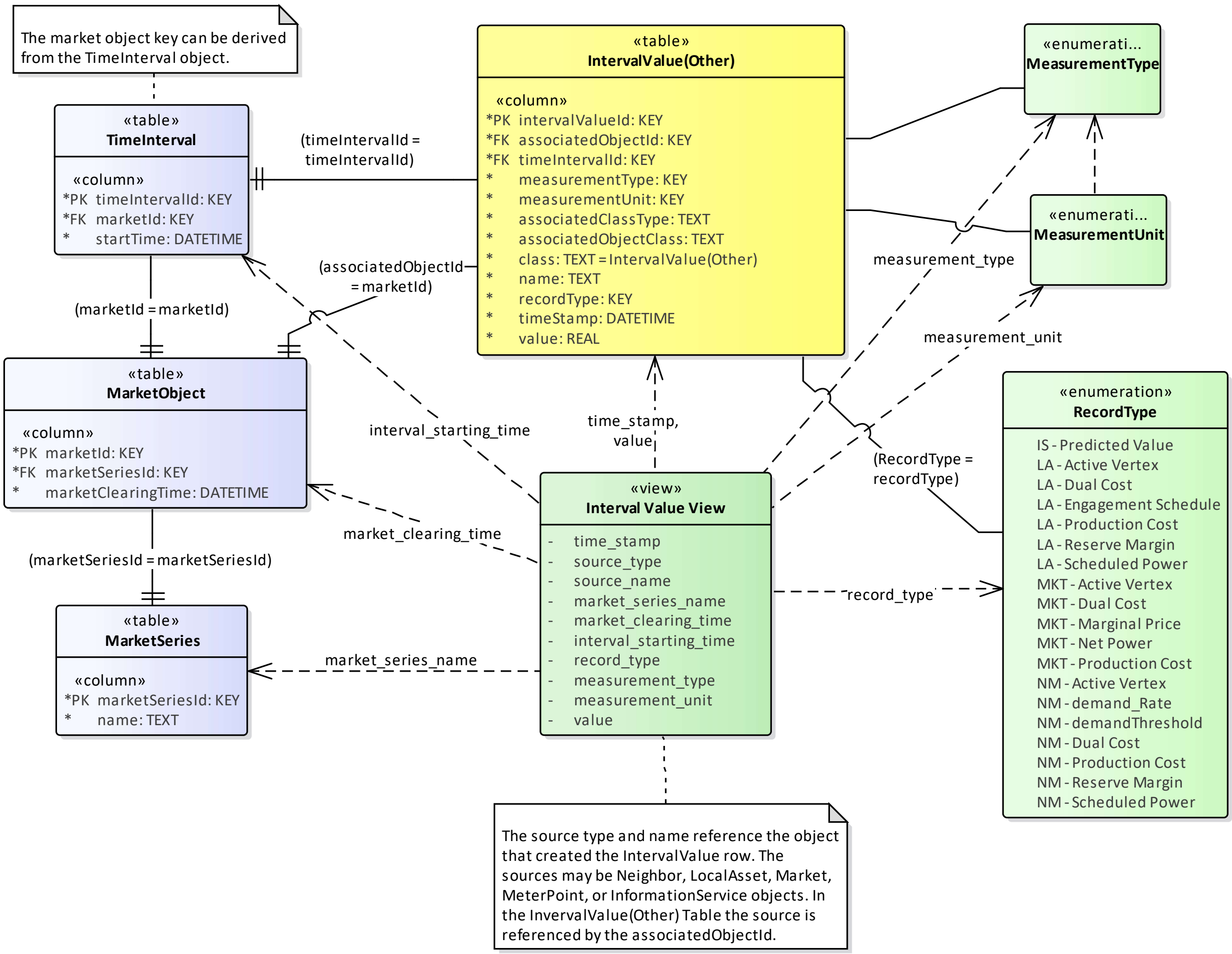


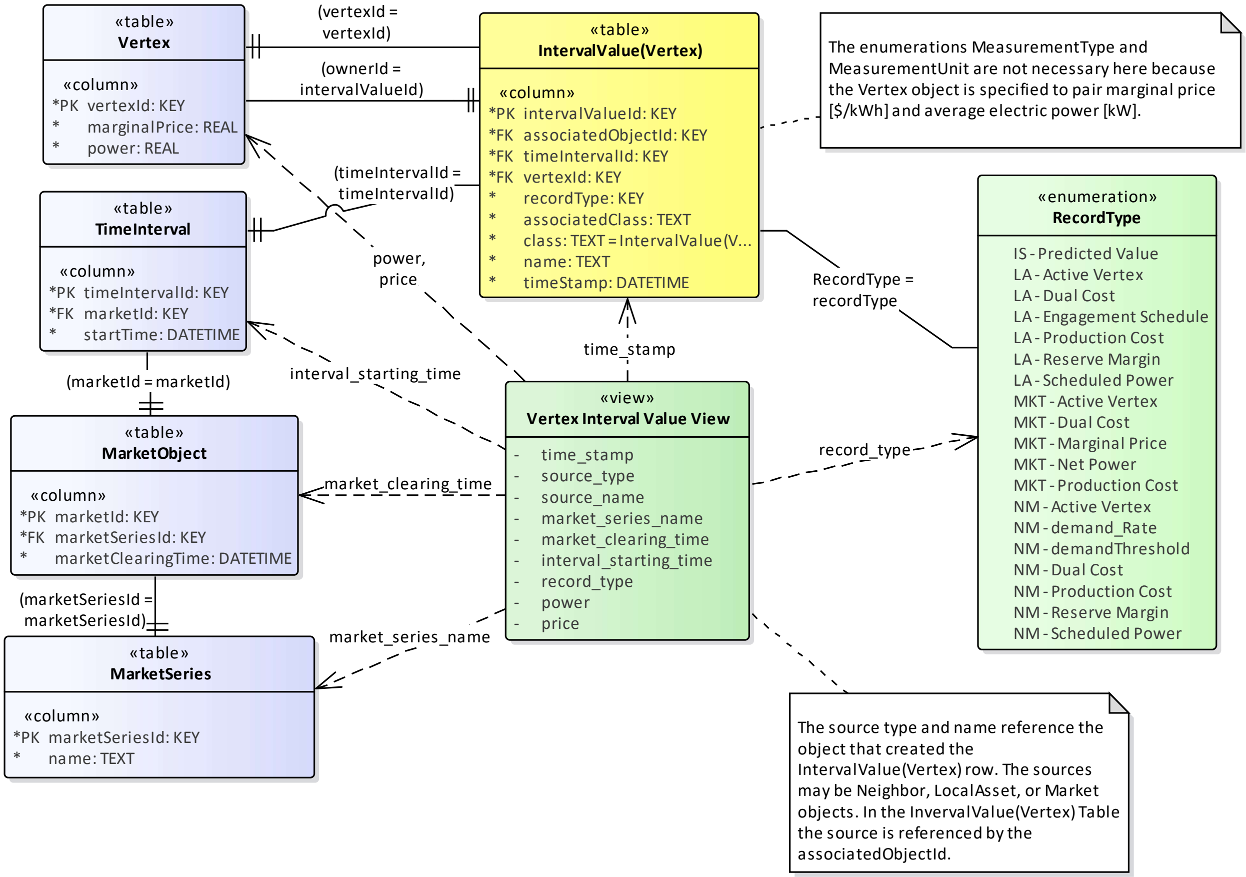
Figure A.1. Information Service Table Relationships



The market object key can be derived from the TimeInterval object.

The source type and name reference the object that created the IntervalValue row. The sources may be Neighbor, LocalAsset, Market, MeterPoint, or InformationService objects. In the InvervalValue(Other) Table the source is referenced by the associatedObjectId.

Figure A.2. Interval Value (for other than for Vertices) Table Relationships



The enumerations MeasurementType and MeasurementUnit are not necessary here because the Vertex object is specified to pair marginal price [$/kWh] and average electric power [kW].

The source type and name reference the object that created the IntervalValue(Vertex) row. The sources may be Neighbor, LocalAsset, or Market objects. In the InvervalValue(Vertex) Table the source is referenced by the associatedObjectId.

Figure A.3. Interval Value (for Vertices) Table Relationships

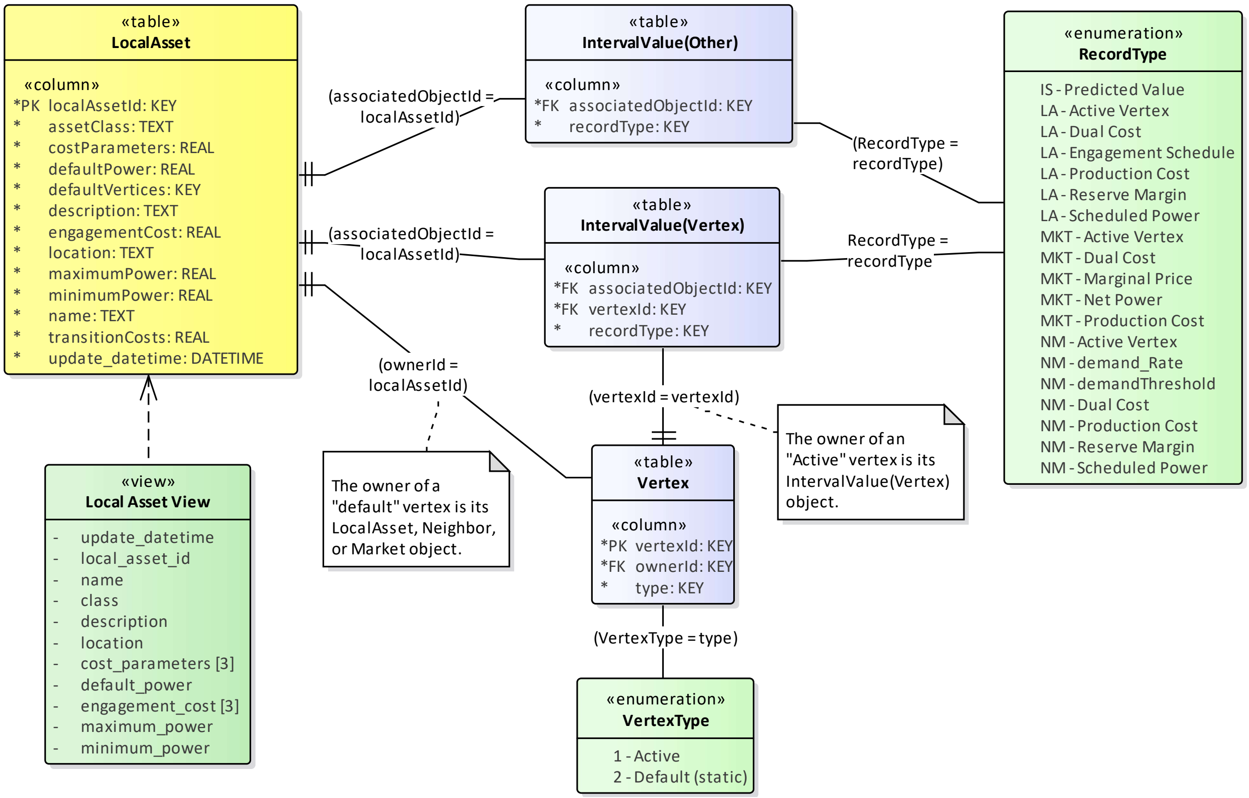


Figure A.4. Local Asset Table Relationships

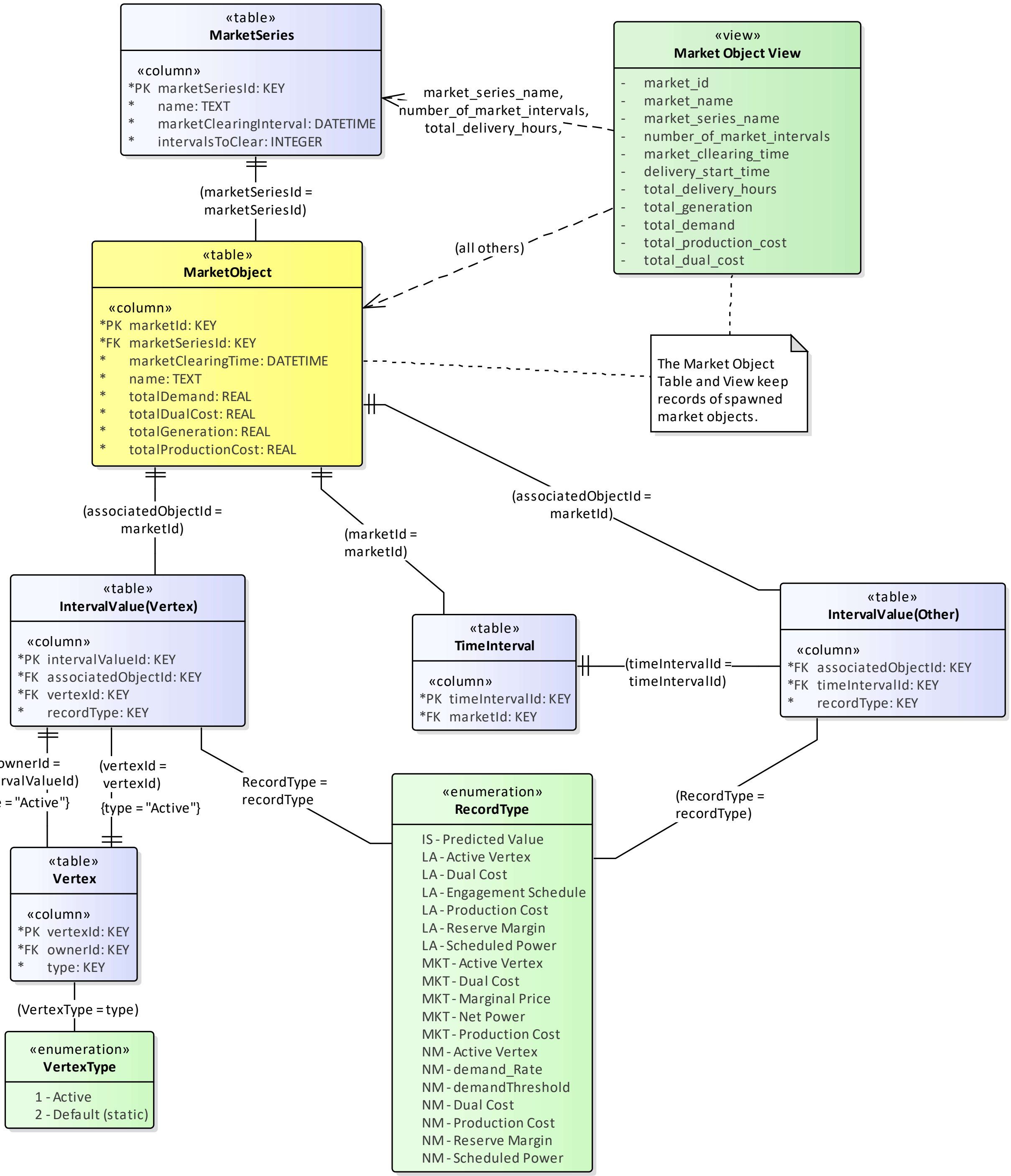
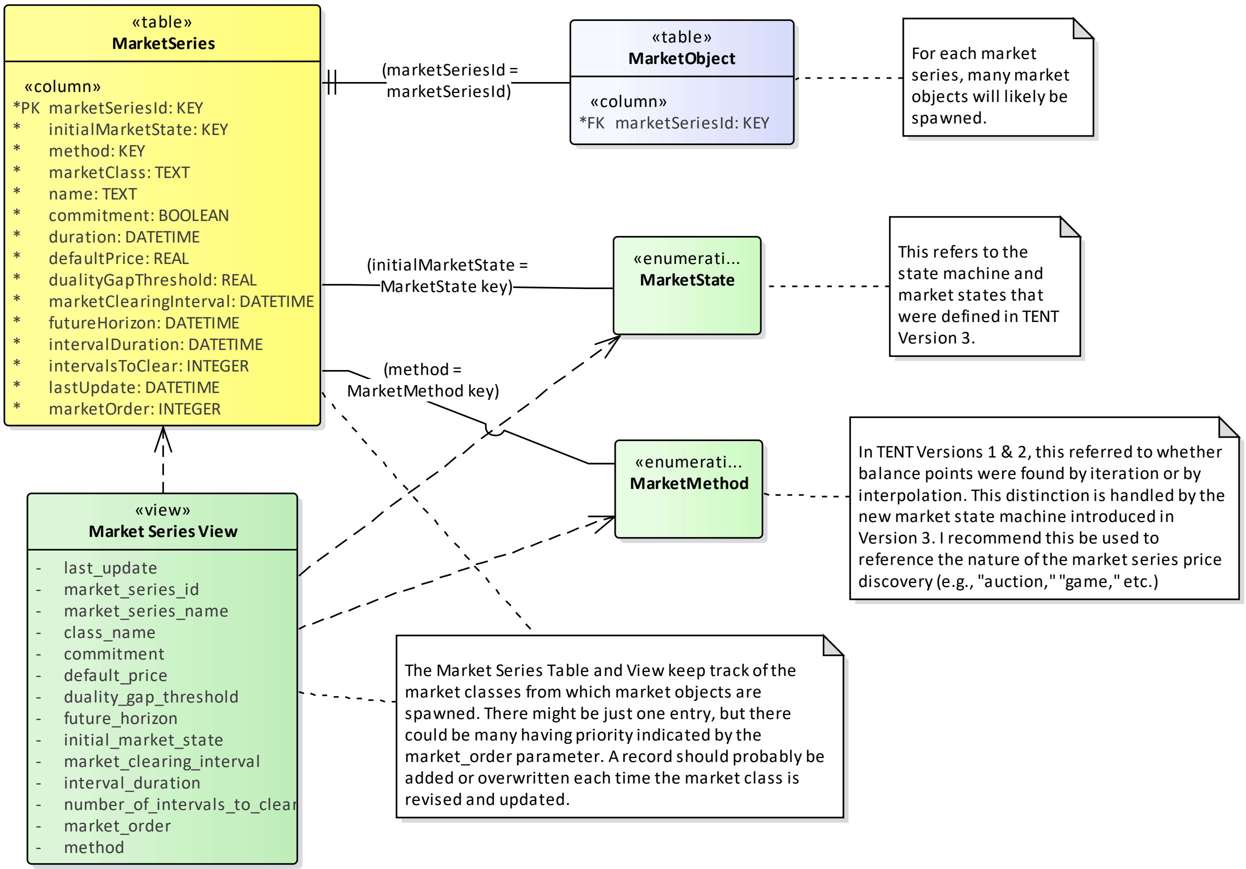


Figure A.5. Market Object Table Relationships



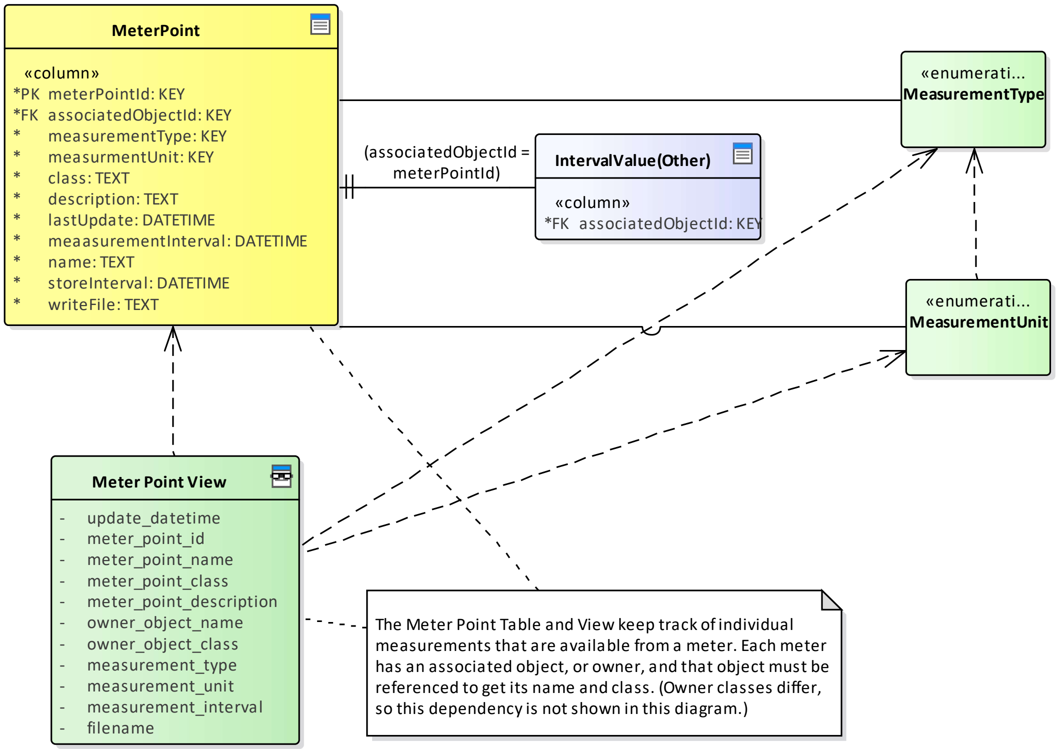
For each market series, many market objects will likely be spawned.

This refers to the state machine and market states that were defined in TENT Version 3.

In TENT Versions 1 & 2, this referred to whether balance points were found by iteration or by interpolation. This distinction is handled by the new market state machine introduced in Version 3. I recommend this be used to reference the nature of the market series price discovery (e.g., "auction," "game," etc.)

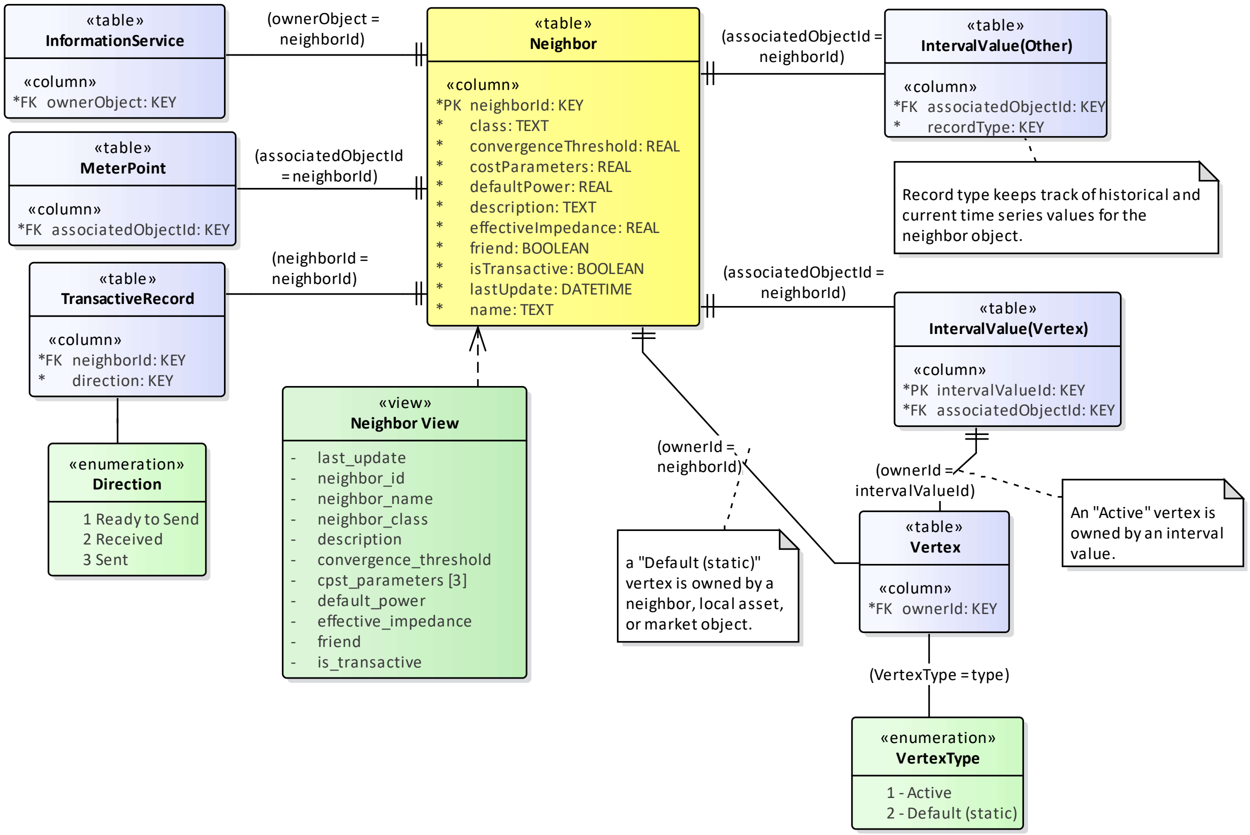
The Market Series Table and View keep track of the market classes from which market objects are spawned. There might be just one entry, but there could be many having priority indicated by the market_order parameter. A record should probably be added or overwritten each time the market class is revised and updated.

Figure A.6. Market Series Table Relationships



The Meter Point Table and View keep track of individual measurements that are available from a meter. Each meter has an associated object, or owner, and that object must be referenced to get its name and class. (Owner classes differ, so this dependency is not shown in this diagram.)

Figure A.7. Meter Point Table Relationships



Record type keeps track of historical and current time series values for the neighbor object.

a "Default (static)" vertex is owned by a neighbor, local asset, or market object.

An "Active" vertex is owned by an interval value.

Figure A.8. Neighbor Table Relationships

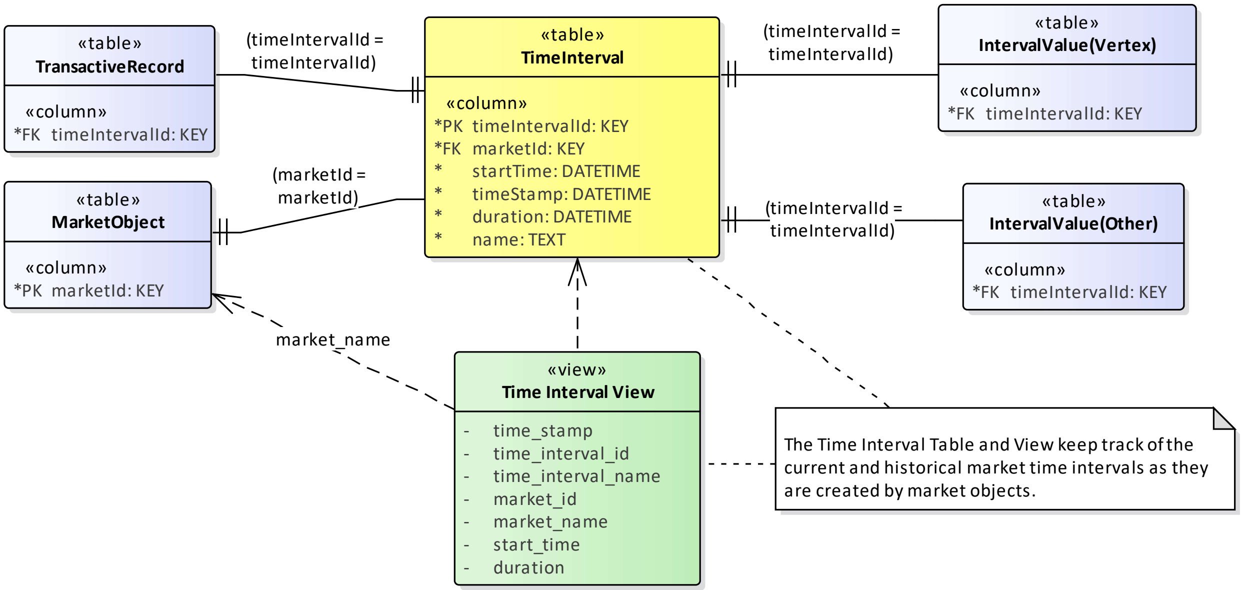


Figure A.9. Time Interval Table Relationships

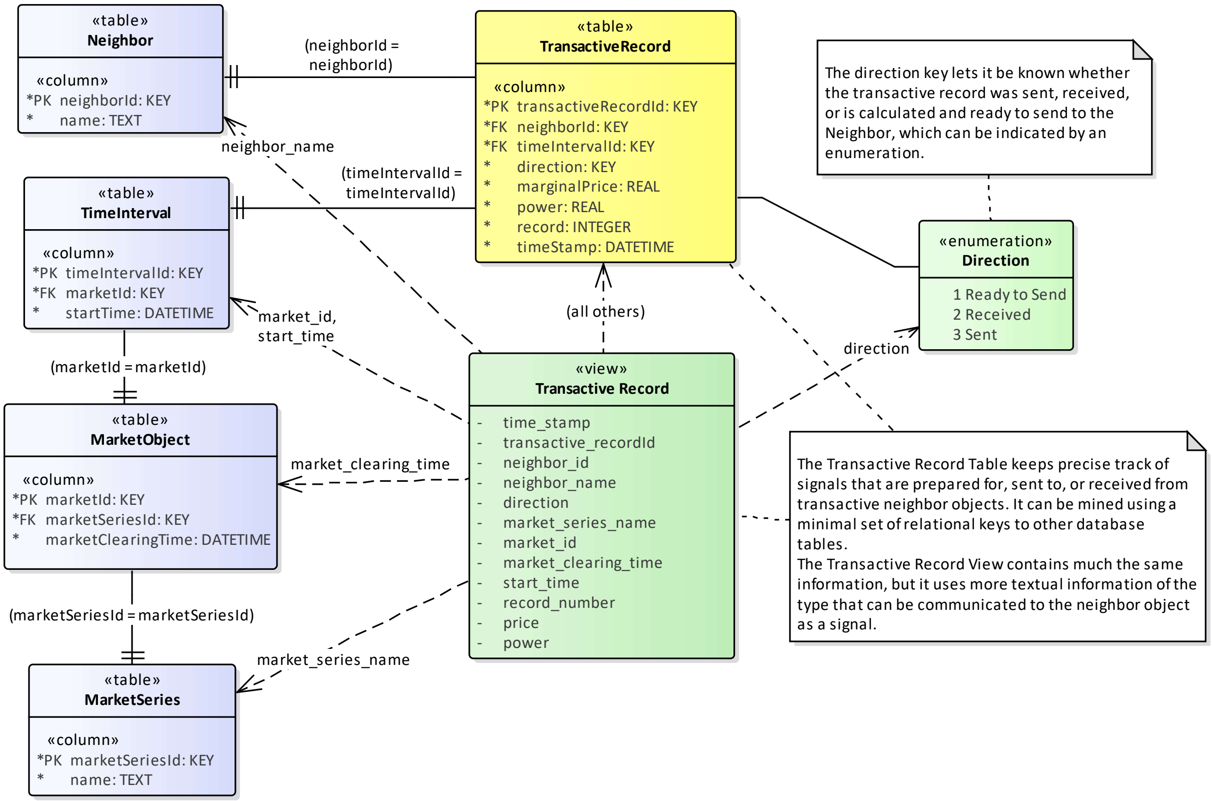


Figure A.10. Transactive Record Table Relationships

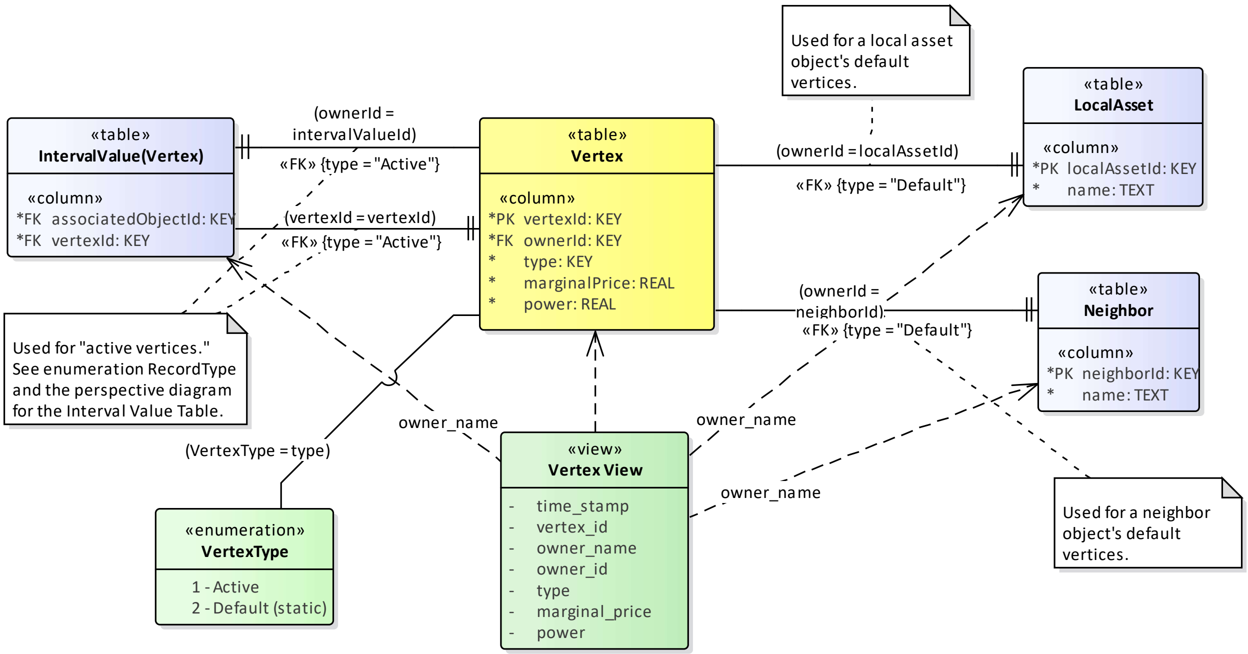


Figure A.11. Vertex Table Relationships

Appendix B: Methods to Harmonize Dynamic Prices and Approved Rates

This appendix introduces two novel TENT processes that may be used in the future to harmonize the dynamic and locationally unique electricity prices that TENT discovers with conventional, regulated cost-recovery practices, as first described in (Hammerstrom 2022). This appendix teaches how the two processes could be implemented in TENT, but the implementations of these processes will be deferred until the processes are needed by a TENT implementer. This appendix documents a potential future implementation of these processes in TENT, but the processes are not yet coded. The interactions of the distribution utility and retail customer with these new use cases are summarized in Fig. B.1. The implementation of these processes in TENT will introduce new price and quantity objects but should not displace existing ones.

The first process (B.1) forces actual cost recovery under dynamic electricity pricing to track approved cost recovery for an electricity customer class. The second method (Section B.2) makes sure that all members in a customer class receive similar average electricity prices, even if the dynamic price at a customer's location is affected by transport constraints or other constraints that may cause locational price discrepancies.

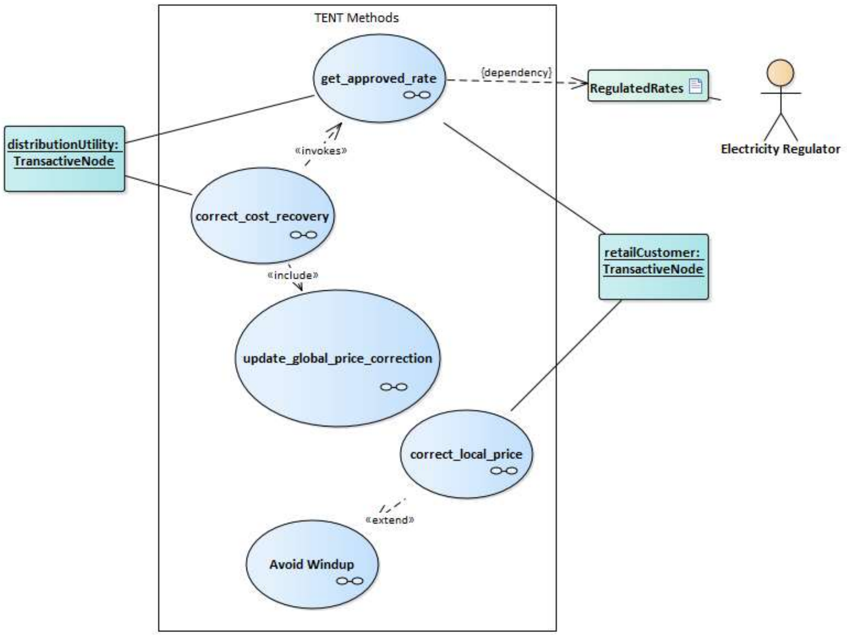


Figure B.1. New Use Cases

The methods are applicable when a transactive node represents a regulated electricity supplier like a distribution utility that offers regulated electricity rates. The retail customer is a member of an electric customer class, all members of which should expect similar access to electricity from their electricity supplier.

B.1 Correction of Distribution Utility Cost Recovery

We begin with the method that makes actual cost recovery track approved cost recovery. This sequence is summarized in Fig. B.2.

In parallel, the retail customers of a customer class send their transactive signals to the distribution utility, as already facilitated by TENT.

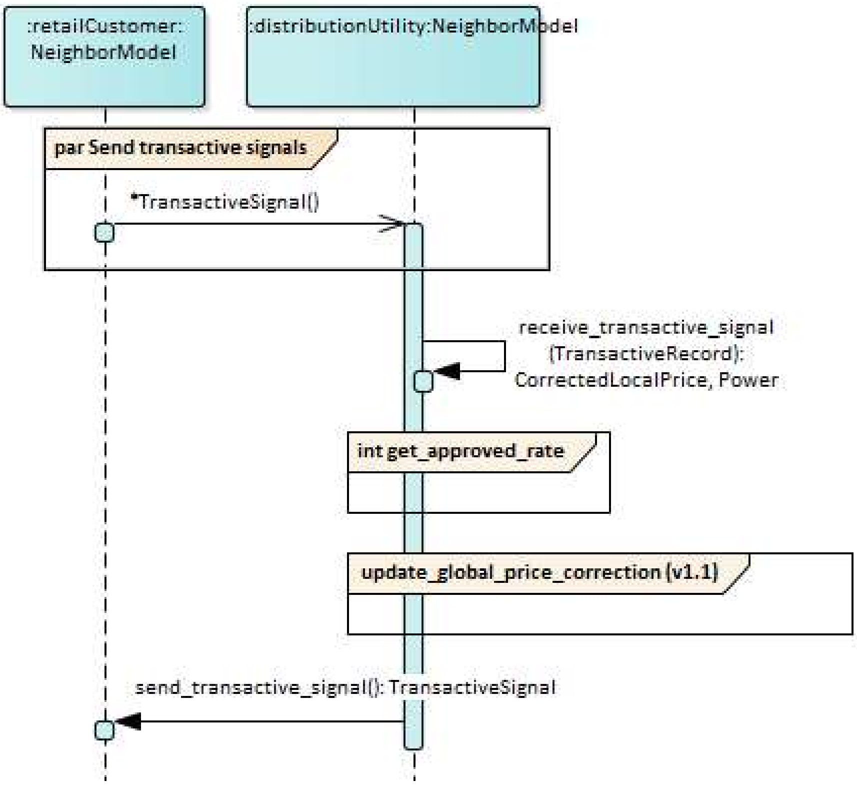

:retailCustomer:
NeighborModel
:distributionUtility:NeighborModel
par Send transactive signals
*TransactiveSignal()
receive_transactive_signal
(TransactiveRecord):
CorrectedLocalPrice, Power
int get_approved_rate
update_global_price_correction (v1.1)
send_transactive_signal(): TransactiveSignal

Figure B.2. UML Sequence Diagram of the method correct_cost_recovery()

The distribution utility accesses an approved electricity rate for the customer class. The method to do so is summarized in Fig. B.3. In the simplest case, the rates are static or change with a predicted pattern, in which case the rates may be retrieved from lookup tables. If the approved rates include dynamically changing prices, then such prices must be retrieved in real time. For example, critical peak pricing events and new market prices must be electronically queried or published. The method of querying may be unique to the approved rate and its source.

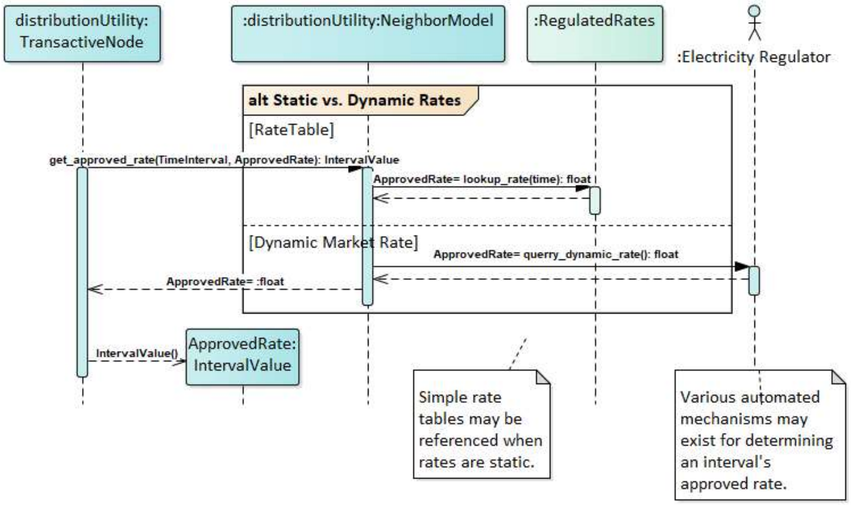


Figure B.3. Sequence Diagram for Method get_approved_rate()

Then the global price correction is updated. The price correction process is summarized by the activities of Fig. B.4.

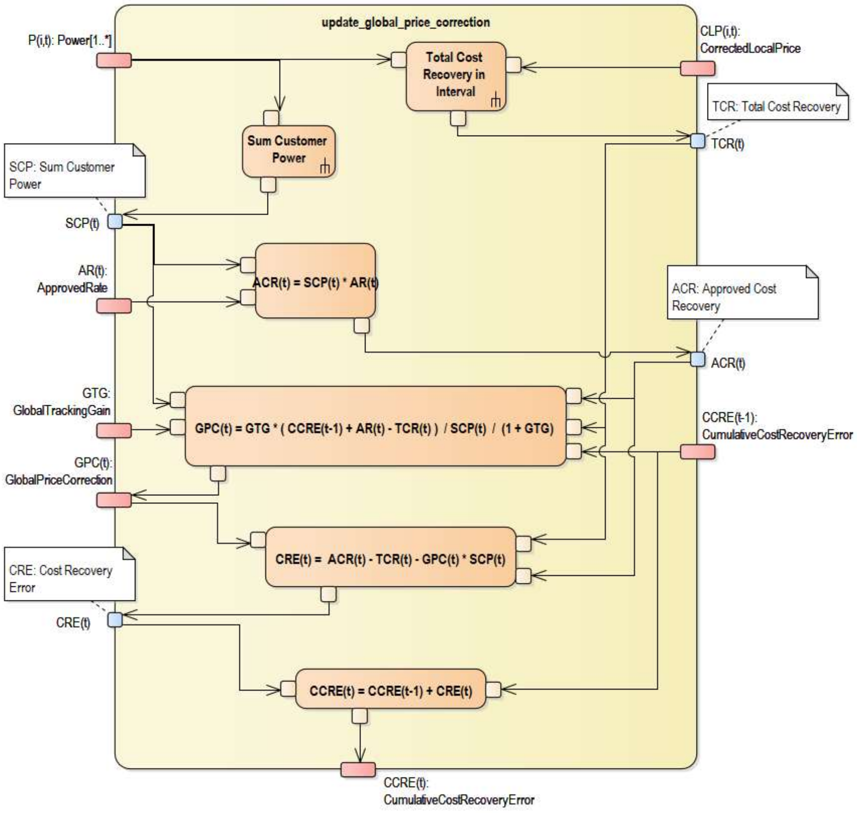


Figure B.4. Activity Diagram for Method update_global_price_correction()

Two activities initiate this process, as shown at the top of Fig. B.4. The first calculates the total electricity to be consumed in a market time interval *t* by the members *i* of a customer class, as shown in Fig. B.5. Scheduled average interval powers are already conveyed via TENT transactive signals. However, actual electricity consumption can be different from that which is scheduled in a time interval by TENT. It would be preferrable for the distribution utility to glean actual interval energy by using smart customer meters, not TENT signals.

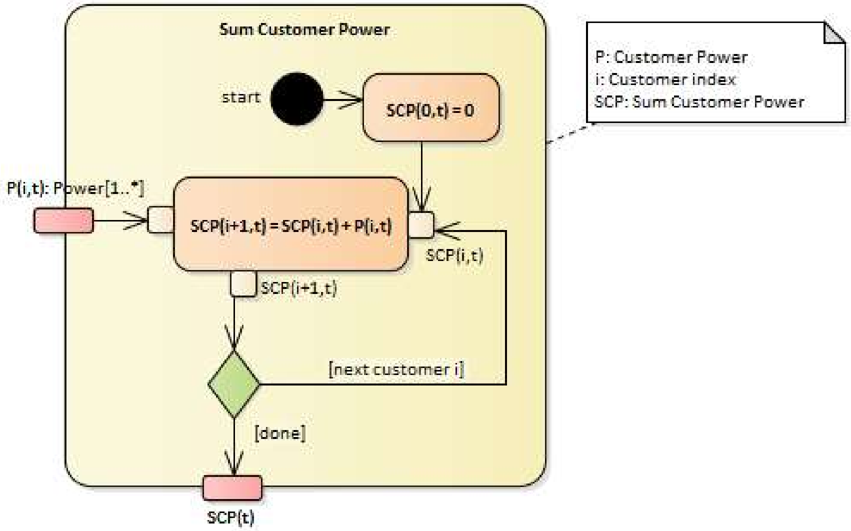


Figure B.5. Activity sum_customer_power()

The next activity of Fig. B.4 sums the total cost recovery, which is the sum product of customers' electricity and corrected dynamic prices in each market time interval. The calculation is shown in Fig. B.6. In each time interval, each customer's interval electricity (i.e., the same as the input to the activity diagram of Fig. B.5) is multiplied by the customer's corrected local price. The corrected local interval price will be a new object to TENT. It is the sum of the dynamic locational price, which exists in TENT already, the global price correction that the customer receives from the distribution utility, and the approved rate, which is also received from the distribution utility. The sum of these products is the total revenue from the customer class in the time interval.

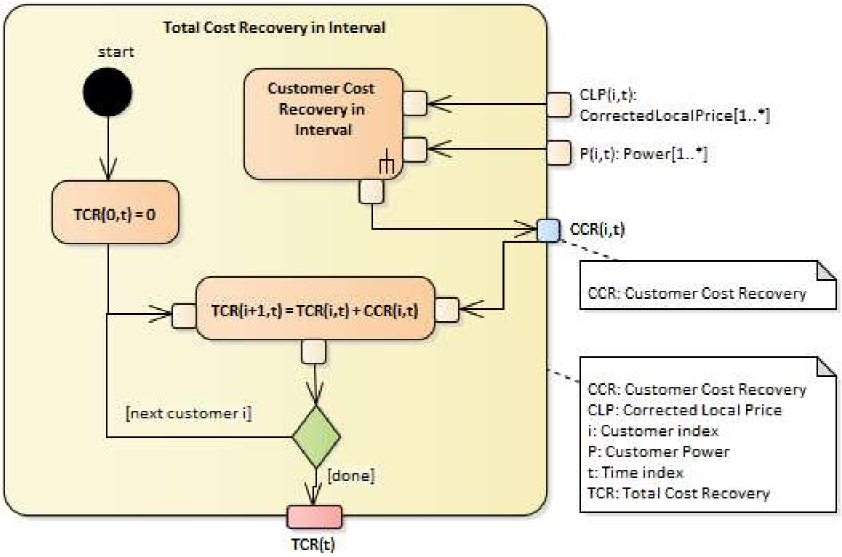

Total Cost Recovery in Interval
start
Customer Cost Recovery in Interval
CLP(i,t): CorrectedLocalPrice[1..*]
P(i,t): Power[1..*]
TCR(0,t) = 0
CCR(i,t)
CCR: Customer Cost Recovery
TCR(i+1,t) = TCR(i,t) + CCR(i,t)
[next customer i]
[done]
TCR(t)
CCR: Customer Cost Recovery
CLP: Corrected Local Price
i: Customer index
P: Customer Power
t: Time index
TCR: Total Cost Recovery

Figure B.6. Activity total_cost_recovery()

All remaining steps in update_global_price_correction() (i.e., Fig. B.4) are simple actions or functions that calculate:

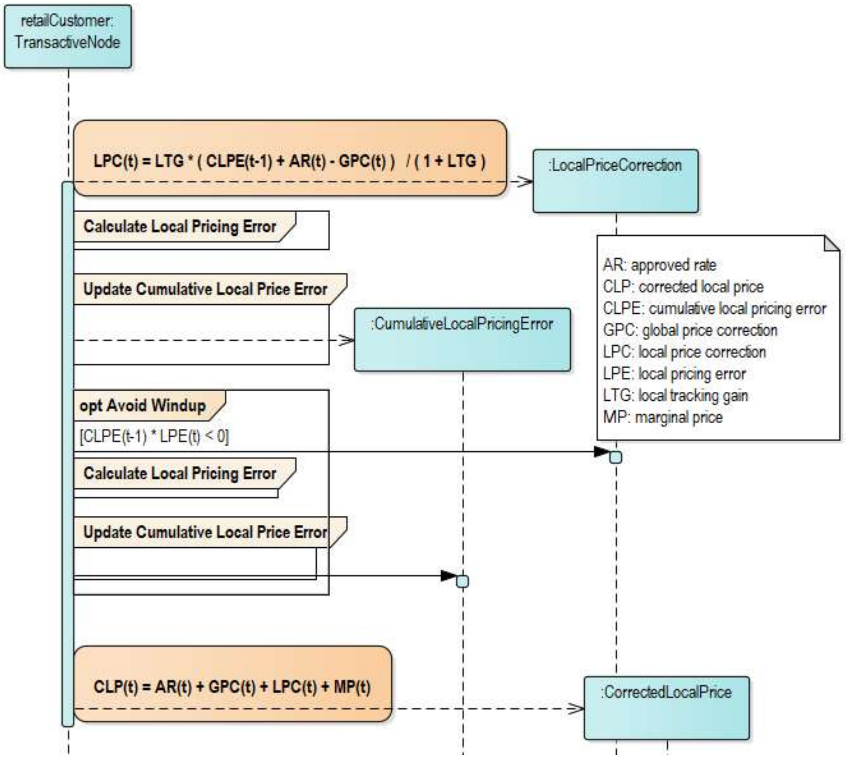
- the approved cost recovery in the time interval
- a global price correction for the market time interval
- a cost recovery error for the market time interval, and
- a cumulative cost recovery error.

Note that a constant global tracking gain is defined, which specifies the high-pass cutoff frequency with which the integrating filter tracks approved cost recovery.

B.2 Correction of Customer Prices

This section introduces the process by which average customer prices are made similar across an entire customer class. This is accomplished by having customers' corrected dynamic prices track an approved electricity rate. The calculation sequence is summarized in Fig. B.7.

The first action of Fig. B.7 updates the local price correction. The calculation uses a constant local tracking gain that defines the high-pass dynamics of the tracking process—typically defining a high-pass response period between 1 week and 1 month, or so. That is, the chosen gain should, upon excitation by a step change in the dynamic price, relax to the new condition within a period between 1 week and 1 month.



retailCustomer:
TransactiveNode
LPC(t) = LTG * (CLPE(t-1) + AR(t) - GPC(t)) / (1 + LTG)
:LocalPriceCorrection
Calculate Local Pricing Error
Update Cumulative Local Price Error
:CumulativeLocalPricingError
AR: approved rate
CLP: corrected local price
CLPE: cumulative local pricing error
GPC: global price correction
LPC: local price correction
LPE: local pricing error
LTG: local tracking gain
MP: marginal price
opt Avoid Windup
[CLPE(t-1) * LPE(t) < 0]
Calculate Local Pricing Error
Update Cumulative Local Price Error
CLP(t) = AR(t) + GPC(t) + LPC(t) + MP(t)
:CorrectedLocalPrice

Figure B.7. UML Sequence Diagram for Method correct_local_price()

Once the price correction has been calculated, it is relatively straightforward to then calculate the corresponding local pricing error for the market time interval (Fig. B.8) and to update a cumulative local pricing error (Fig. B.9).

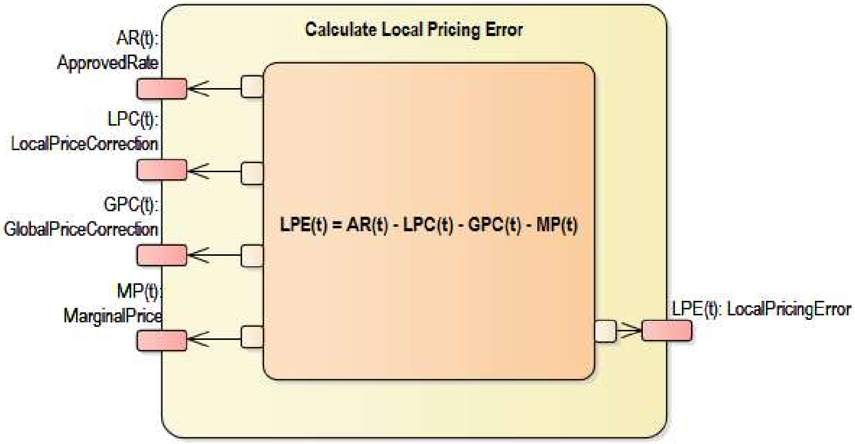

Calculate Local Pricing Error
AR(t):
ApprovedRate
LPC(t):
LocalPriceCorrection
GPC(t):
GlobalPriceCorrection
MP(t):
MarginalPrice
LPE(t) = AR(t) - LPC(t) - GPC(t) - MP(t)
LPE(t): LocalPricingError

Figure B.8. Activity calculate_local_pricing_error()

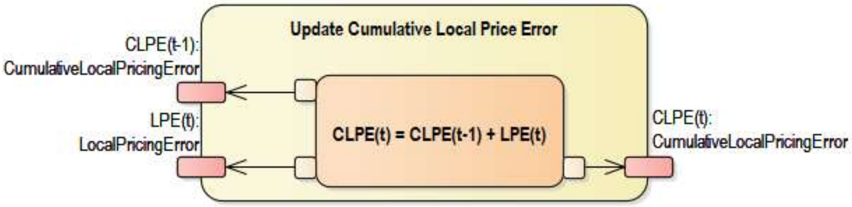

Update Cumulative Local Price Error
CLPE(t-1):
CumulativeLocalPricingError
LPE(t):
LocalPricingError
CLPE(t) = CLPE(t-1) + LPE(t)
CLPE(t):
CumulativeLocalPricingError

Figure B.9. Activity update_cumulative_local_price_error()

The process could end at this point (i.e., after the first use of "Update Cumulative Local Price Error" in Fig. B.7), but the results would be vulnerable to integrator windup and oscillatory behaviors when excited by sudden, extreme price changes. Optional sequence avoid_windup() may be invoked to mitigate integrator windup, as shown in Fig. B.7.

The details of method avoid_windup() are shown in Fig. B.10. The local price correction, local and cumulative pricing errors become recalculated if the local pricing error and cumulative local pricing error have different signs. If this condition is true, the local pricing error is reverted to its original value, and the local and cumulative pricing errors are again updated (Figs. B.8 and B.9).

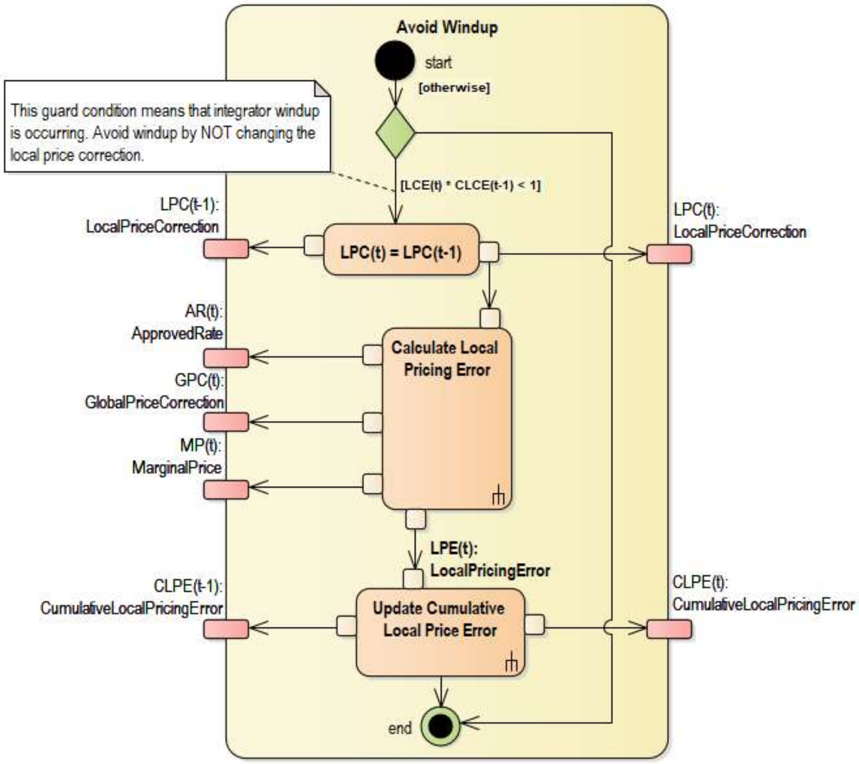

Avoid Windup
start
[otherwise]
This guard condition means that integrator windup is occurring. Avoid windup by NOT changing the local price correction.
[LCE(t) * CLCE(t-1) < 1]
LPC(t-1): LocalPriceCorrection
LPC(t) = LPC(t-1)
LPC(t): LocalPriceCorrection
AR(t): ApprovedRate
GPC(t): GlobalPriceCorrection
MP(t): MarginalPrice
Calculate Local Pricing Error
LPE(t): LocalPricingError
CLPE(t-1): CumulativeLocalPricingError
Update Cumulative Local Price Error
CLPE(t): CumulativeLocalPricingError
end

Figure B.10. Activity avoid_windup()

Finally, the corrected local price may be calculated using current global and local correction terms, as shown in the last action of Fig. B.7.

Reference

Hammerstrom, Donald James. 2022. "Harmonizaton of Dynamic Prices with Approved Rates." *IEEE Power Engineering Society General Meeting.* Denver, Colorado: IEEE.

Appendix C: Accommodating Matching Engines in TENT

TENT was designed to accommodate alternative types of electricity markets. Initial TENT implementations facilitated bilateral electricity markets and consensus methods that can discover an electricity price for each location and time interval. TENT has been extended to facilitate matching engines, which, like stock markets, asynchronously receive simple offers or bids ("aggressive orders") and attempt to match them from standing counteroffers or bids ("standing orders") that await to be matched. Principles of energy balance still underly the resulting electricity prices, but each bid or offer can result in a different price.

Whereas TENT facilitates rich supply and bid curves, as can be represented by linear interpolation between any nondecreasing ordering of Vertices, bids and offers submitted to market engines are typically simple "orders" that pair a single quantity with a strike price. The current work allows that practice to continue, but TENT also accommodates richer bid and offer curves, as well. If a match is successfully made, the result pairs a single price and quantity, regardless of the complexity of the bids and offers being matched.

Bids and offers into matching engines can include various constraints to be placed on the matching process. TENT was extended to accommodate some, but not all, these constraints.

- A distinction might be made between "limit orders" that include both a quantity and strike price and "market orders" that state a quantity that should be matched regardless of price. TENT already facilitated this distinction because the price of a VERTEX object can be assigned any price, including positive or negative infinity.

- An aggressive order may be flagged to indicate whether it should be included among standing orders if it is not initially matched. TENT can be extended to include this persistence flag, but this feature is not fully implemented and tested.

An aggressive order may be flagged to indicate whether its quantity must be entirely matched, or whether it may be partially matched, leaving an unmatched quantity remainder. For example, if an electricity generator has only binary on and off operational states, no option should exist to match only part of its offered generation.[1] TENT was extended to flag VERTEX objects to indicate the continuity of bids and offers, and matching engine processes were designed to respect the continuity or discontinuity of bids and offers.

Secondary matching strategies

Matching always uses price as its initial matching strategy. If price alone cannot alone be used to uniquely match orders, then the matched quantities of same-priced counteroffers must be matched using order age or quantity volume as a preference. These alternative strategies will not be needed and invoked as frequently for distributed transactive energy systems as for stock trading because the pools of standing orders will be smaller. TENT was extended to facilitate these two preferences.

[1] This does not preclude matching a single aggressive order using multiple standing orders, which makes the logic much more challenging.

Effects of partial fulfilment on matching

Each time an order is partially fulfilled, newly spawned orders must be created for the matched and unmatched quantities. These child orders must be traceable to the original parent order. Unmatched residuals are modified by reducing the magnitudes of all the order's vertices by the matched quantity.

New TENT Classes that Support Matching Engines

This appendix introduces two new TENT object classes and describes the extension of an existing base class. The class MatchingEngine is a child of class Market that facilitates the matching of an aggressive supply or demand order with existing counteroffers from a resting order book. Class Order and extended class Vertex support MatchingEngine markets and could also be used otherwise to simplify TENT code. These classes and related enumerations may be imported using these commands:

```
> from matching_engine import MatchingEngine
> from order import Order, Vertex, OrderStatus, SupplyOrDemand
```

Extended Vertex Class

Class order.Vertex extends original TENT class vertex.Vertex. Base class vertex.Vertex was intended to represent an inflection point in a quantity-versus-price supply or demand curve. While it has served its purpose well, Vertex objects have been challenging to instantiate. Some original vertex.Vertex properties have not proven to be especially useful. An extended order.Vertex object can be instantiated by supplying, at minimum, the object's price. A power quantity should normally be provided, as well. New order.Vertex properties "price" and "quantity" alias the existing vertex.Vertex properties "marginalPrice" and "power," respectively. Electricity price is still assumed to use measurement units "$/kWh," and quantity or power is still assumed to use measurement units "average kW." Here is an example instantiation of an extended order.Vertex object:

```
> new_vertex = Vertex(
    price=0.1,  # $/kWh
    quantity=100.0)  # kW_e
```

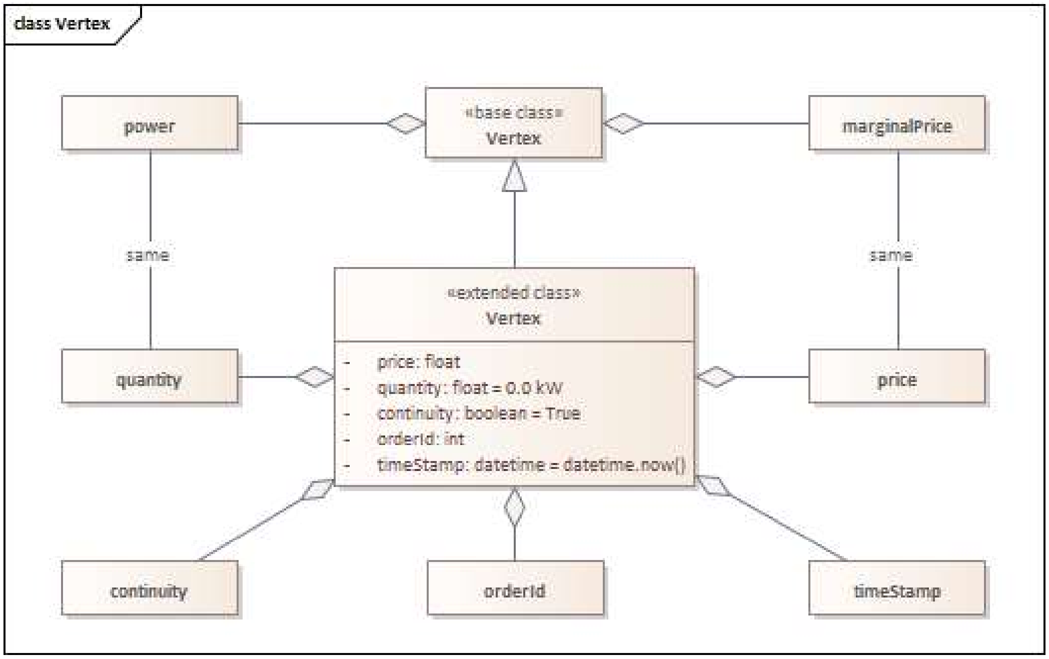
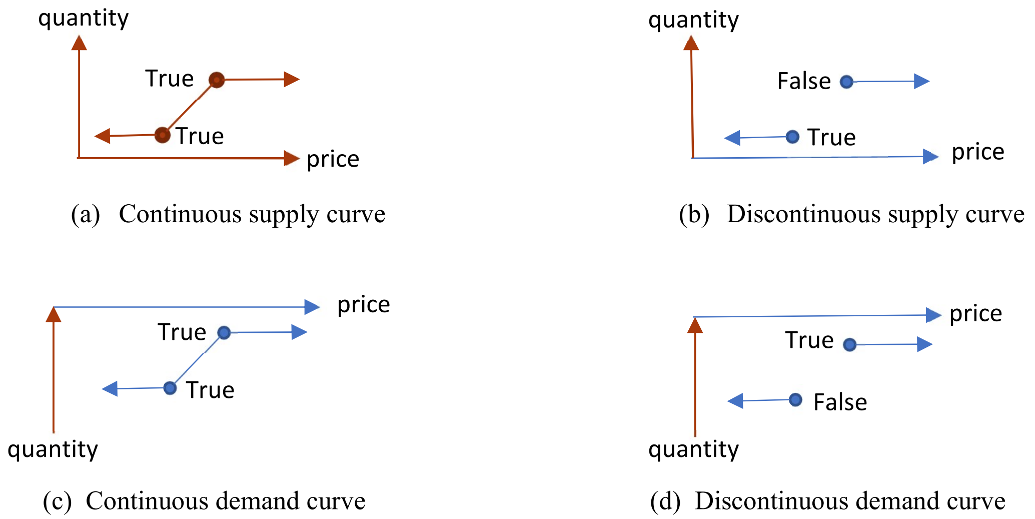


Figure 11. Vertex extends base class Vertex.

Extended ORDER.VERTEX properties:

- *timeStamp* (datetime): By default, *timeStamp* is set to the time and date at which the ORDER.VERTEX object is created. Ideally, *timeStamp* should match that of the ORDER.ORDER object to which the ORDER.VERTEX belongs.
- *orderId* (integer): This identifying integer should match the identifying integer of the ORDER.ORDER object to which the ORDER.VERTEX belongs.
- *Continuity* (Boolean): This property should be assigned True if the supply or demand curve is continuously defined between this ORDER.VERTEX object and the ORDER.VERTEX object representing the next smaller quantity magnitude of the supply or demand curve. The property should be assigned False if there is no quantity defined within that price range. See Figure 12. Observe especially how discontinuity is assigned in supply curves (Figure 12b) and demand curves (Figure 12d).



(a) Continuous supply curve

(b) Discontinuous supply curve

(c) Continuous demand curve

(d) Discontinuous demand curve

Figure 12. ORDER.VERTEX continuity assignments for ORDER.VERTEX objects in various continuous and discontinuous supply and demand curves

ORDER.ORDER Class

The new ORDER.ORDER class is used to collect one or more ORDER.VERTEX objects that belong together as a supply or demand curve. Prior implementations used class INTERVALVALUE to bind an individual VERTEX.VERTEX object with its market, time interval, the purpose that the VERTEX serves, and the system actor that is making the bid or offer. An INTERVALVALUE object having a VERTEX object as its value introduced challenging nuances. Complex price-quantity curves heretofore had to be recollected, manipulated, and resaved as individual VERTEX objects, each within its own INTERVALVALUE object. The ORDER.ORDER class simplifies this process and should lessen the chance that supply or demand curves get corrupted.

The following commands could be used to instantiate a new ORDER.ORDER object and its requisite parameter objects. “Datetime1”, “datetime2,” “datetime3,” and “timedelta” refer to times and dates and time durations that have been instantiated using python module DATETIME.

```
> a_time_interval = TimeInterval(
    activation_time: datetime = datetime1,
    duration: timedelta = timedelta(hours=1),
    market: Market = new_matching_engine,
    market_clearing_time: datetime = datetime2,
    start_time: datetime = datetime3)
> a_neighbor = Neighbor()
> new_order = Order(
    market: Market = new_matching_engine,
    time_interval: TimeInterval = a_time_interval,
    actor: Neighbor or LocalAsset = a  neighbor)
```

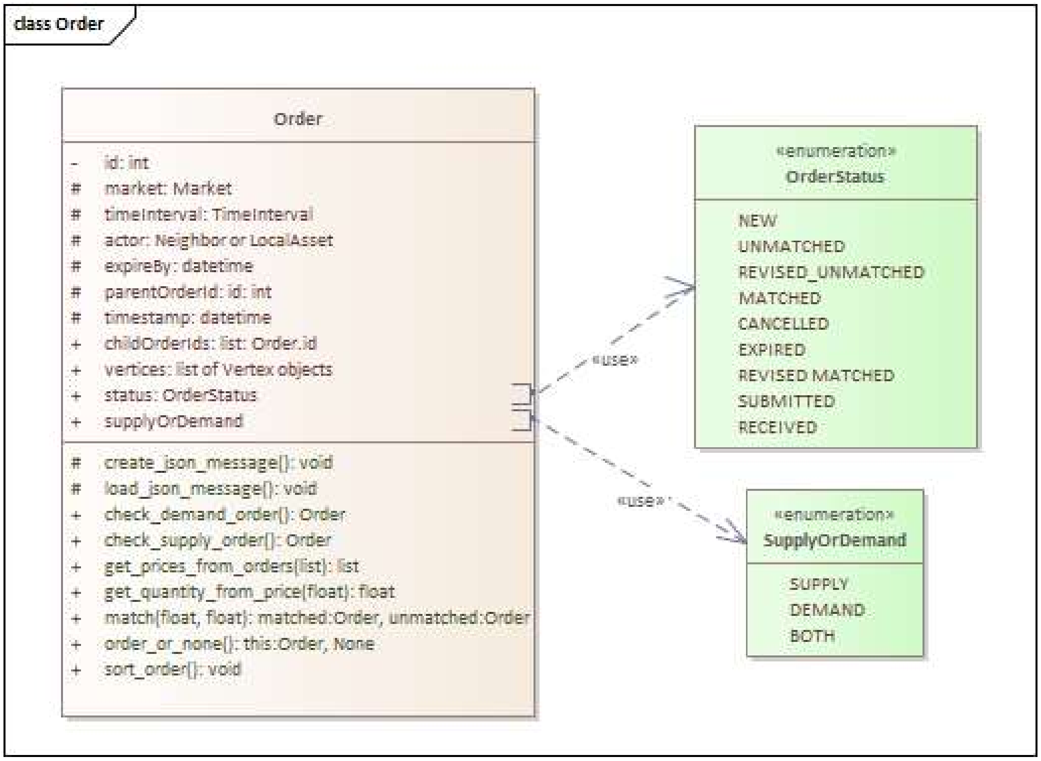

class Order
Order
- id: int
# market: Market
# timeInterval: TimeInterval
# actor: Neighbor or LocalAsset
# expireBy: datetime
# parentOrderId: id: int
# timestamp: datetime
+ childOrderIds: list: Order.id
+ vertices: list of Vertex objects
+ status: OrderStatus
+ supplyOrDemand
# create_json_message(): void
# load_json_message(): void
+ check_demand_order(): Order
+ check_supply_order(): Order
+ get_prices_from_orders(list): list
+ get_quantity_from_price(float): float
+ match(float, float): matched:Order, unmatched:Order
+ order_or_none(): this:Order, None
+ sort_order(): void
«use»
«use»
«enumeration»
OrderStatus
NEW
UNMATCHED
REVISED_UNMATCHED
MATCHED
CANCELLED
EXPIRED
REVISED MATCHED
SUBMITTED
RECEIVED
«enumeration»
SupplyOrDemand
SUPPLY
DEMAND
BOTH

Figure 13. Class ORDER.ORDER and two of its enumerations

ORDER.ORDER properties:

- *market* (MARKET): This property refers to a TENT MARKET object—typically the MATCHINGENGINE object in which this ORDER.ORDER object is active.
- *timeInterval* (TIMEINTERVAL): This property refers to a TIMEINTERVAL object in which this ORDER.ORDER object is active. The TIMEINTERVAL identifies the delivery time and duration. In a MATHCINGENGINE object, there exists a one-to-one pairing between this property and the market.
- *actor* (NEIGHBOR or LOCALASSET): This property points to a NEIGHBOR or LOCALASSET object that owns the supply or demand represented by this ORDER.ORDER object. An agent may work with the supply and demand of local devices or assets, or it may work with bids and offers from other neighboring agents in its transactive energy system.
- *expireBy* (DATETIME): This is the date and time at which the ORDER.ORDER object, if not acted upon, should expire.
- *parentOrderId* (integer): An ORDER.ORDER object may need to be revised as it is fixed or matched. This property should point to the ID of the parent ORDER.ORDER object that has been revised.
- *vertices* (ordered list of ORDER.VERTEX objects): This list of ORDER.VERTEX objects represents the ORDER.ORDER object's supply or demand curve. The list should represent supply or demand, but not both. The list should be ordered by increasing price and quantity.
- *Id* (integer): This is the unique identifier of the ORDER.ORDER object. This identifier is automatically assigned as the ORDER.ORDER is created and should not be changed thereafter.
- *timeStamp* (DATETIME): A matching engine may prioritize ORDER.ORDER objects by their age. Ideally, this timestamp should be set to the date and time at which the ORDER.ORDER object is placed in a

standing order book. By default, the timestamp is set to the time at which the ORDER.ORDER object is created.

- *status* (ORDER.ORDERSTATUS): This property keeps track of the status of the ORDER.ORDER object during its lifetime. See enumeration ORDERSTATUS. An ORDER.ORDER object may start out “new” and transition as it becomes revised, matched, superseded, expired, etc.
- *childOrderIds* (list of integers): As the ORDER.ORDER object becomes revised and superseded, this property should be used to list the new ORDER.ORDER objects that it parents.
- *supplyOrDemand* (SUPPLYORDEMAND): An ORDER.ORDER object should represent either supply or demand, not both. See enumeration SUPPLYORDEMAND.

ORDER.ORDER methods:

- *create_json_message*: This method returns a flat JSON file from an ORDER.ORDER object. The file is suitable to convey a bid or offer to a neighboring transactive energy system agent. The message may be interpreted by method *load_json_message*.
- *load_json_message*: This method may be used to interpret a message and reconstruct an order from a file that had been created by method *create_json_message*.
- *check_demand_order*: This method returns a proper demand ORDER.ORDER object from the original one. If the original ORDER.ORDER is already a proper demand order, then it is returned unchanged. If the original ORDER.ORDER object has no demand component, None is returned. If the original ORDER.ORDER object has both supply and demand components, its demand component is returned as a new revised Order object having the original ORDER.ORDER object as its parent.
- *check_supply_order*: This method returns a proper supply ORDER.ORDER object from the original one. If the original ORDER.ORDER is already a proper supply order, then it is returned unchanged. If the original ORDER.ORDER object has no supply component, None is returned. If the original ORDER.ORDER object has both supply and demand components, its supply component is returned as a new revised ORDER.ORDER object having the original ORDER.ORDER object as its parent.
- *get_prices_from_orders*: Given a list of ORDER.ORDER objects, this method retrieves their ORDER.VERTEX prices. Filtering is done to eliminate meaningless duplicates.
- *get_quanity_from_price:* A supply or demand quantity curve is necessarily a function of price. Given an electricity price, this method returns the ORDER.ORDER object quantity that corresponds to that price. None is returned if the supply or demand curve happens to be discontinuous (quantity is not defined) at that price.
- *match*: Given a matched counteroffer quantity and match price, this method returns two new ORDER.ORDER objects—one for the match and another for the ORDER.ORDER object’s remainder unmatched supply or demand curve. None is returned for the unmatched ORDER.ORDER object if there is no remainder supply or demand curve. This important method is discussed further below.
- *order_or_none*: This method simply returns the ORDER.ORDER object if it represents any supply or demand quantity. Otherwise, it returns None. This is a good method to use in conjunction with return statements so that meaningless ORDER.ORDER objects will not be propagated.
- *sort_order*: This method sorts an ORDER.ORDER object’s list of ORDER.VERTEX objects by price and quantity. The sorting process is different for supply and demand ORDER.ORDER objects.

After the MATCHINGENGINE market discovers a power quantity and price at which an aggressive order and standing order match, it calls method *match*(). See Figure 14. If this method is called about a standing order, the quantity magnitudes in the standing order's listed ORDER.VETEX objects are reduced by the matched quantity and a new revised ORDER.ORDER object is created for the remaining unmatched quantities of the give supply or demand curve. Typically, the original standing order is cancelled, but the old and revised ORDER.ORDER objects point to the other's ID using their *parentOrder* and *childOrderIds* properties. A new limit order object is also created for the matched quantity and price.

Any match causes both the aggressive order and its standing counteroffer to be matched and revised. So, the *match* method of class ORDER.ORDER is also called in respect to the aggressive order. Note that the sign of the quantity magnitude is that of the order's counteroffer, so the quantity signs are different for the standing and aggressive orders that are becoming revised. The revised quantity is the sum of the ORDER.ORDER object and matched quantities.

If no quantity of an order remains after the match, the unmatched order is returned as None.
These examples demonstrate calls made to method match() in respect to a standing order and an aggressive order.

```
> matched_order, unmatched_order = standing_order.match(
    match_price: float = 0.1,
    quantity: float = 100.0)
matched_order, unmatched_order = aggressive_order.match(
    match_price: float = 0.1,
    quantity: float = -100.0)
```

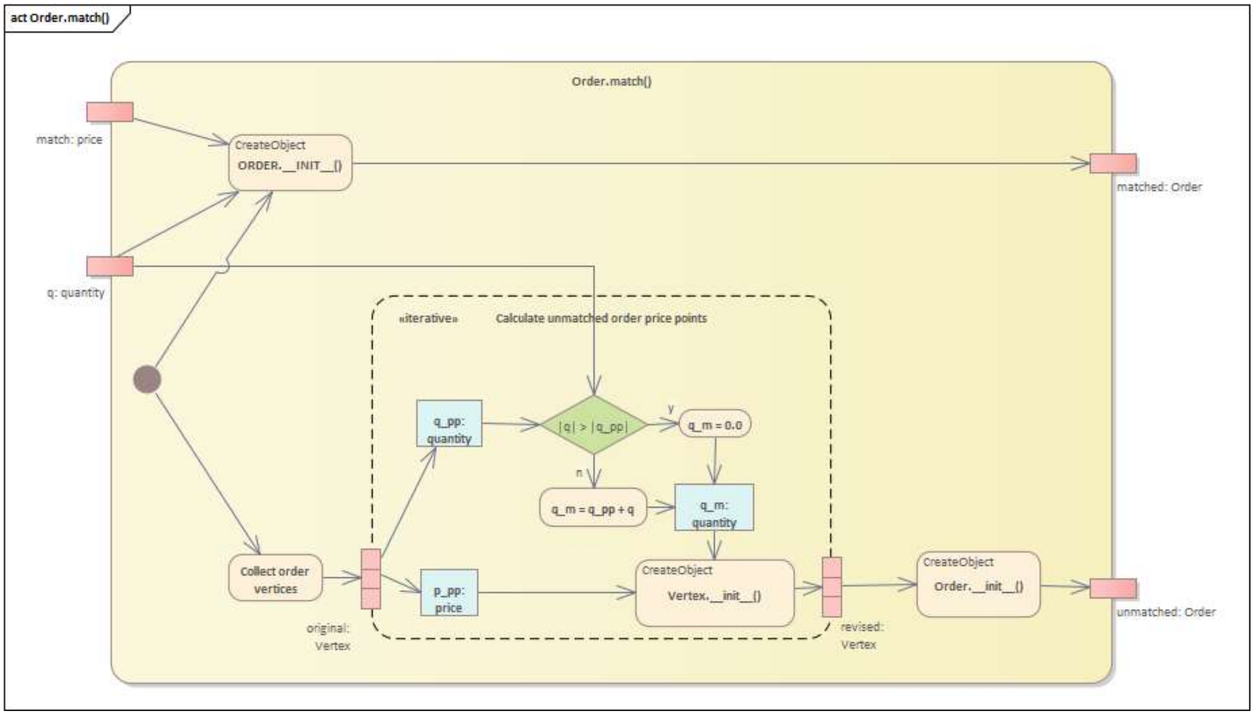


Figure 14. Given a matched quantity and price, the ORDER.ORDER class method *match* returns ORDER.ORDER objects representing the matched limit order and unmatched remainder after the transaction.

MATCHINGENGINE Class

The MATCHINGENGINE class inherits many useful attributes and methods from parent class MARKET. However, discussion will focus on new attributes.

MATCHINGENGINE properties:

- *matchingStrategy* (MATCHINGSTRATEGY): Price is always the dominant matching strategy. When price alone cannot arbitrate between multiple competing standing counteroffers, an alternative strategy must be invoked. There are many possible such strategies, but only two are defined here: order age and order volume. The first gives priority to ORDER.ORDER objects that are oldest. The latter gives priority to ORDER.ORDER objects that represent the greatest quantity magnitude.
- *hasMarket* (Boolean): By default, this property is set False and it is assumed that a transactive agent does not itself operate a matching-engine process. Any agent may, however, use the matching-engine process.
- *orderList* (list of ORDER.ORDER objects): This property holds a list of ORDER.ORDER objects that may be used as counteroffers when a new aggressive ORDER.ORDER object is received. This list is often called an “order book.” It is possible for listed ORDER.ORDER objects to be matched and revised, creating new ORDER.ORDER objects for the *orderList* or *matchList*.
- *matchList* (list of ORDER.ORDER objects): This property holds a list of ORDER.ORDER objects that represent successful match transactions. ORDER.ORDER objects should not be removed from this list.

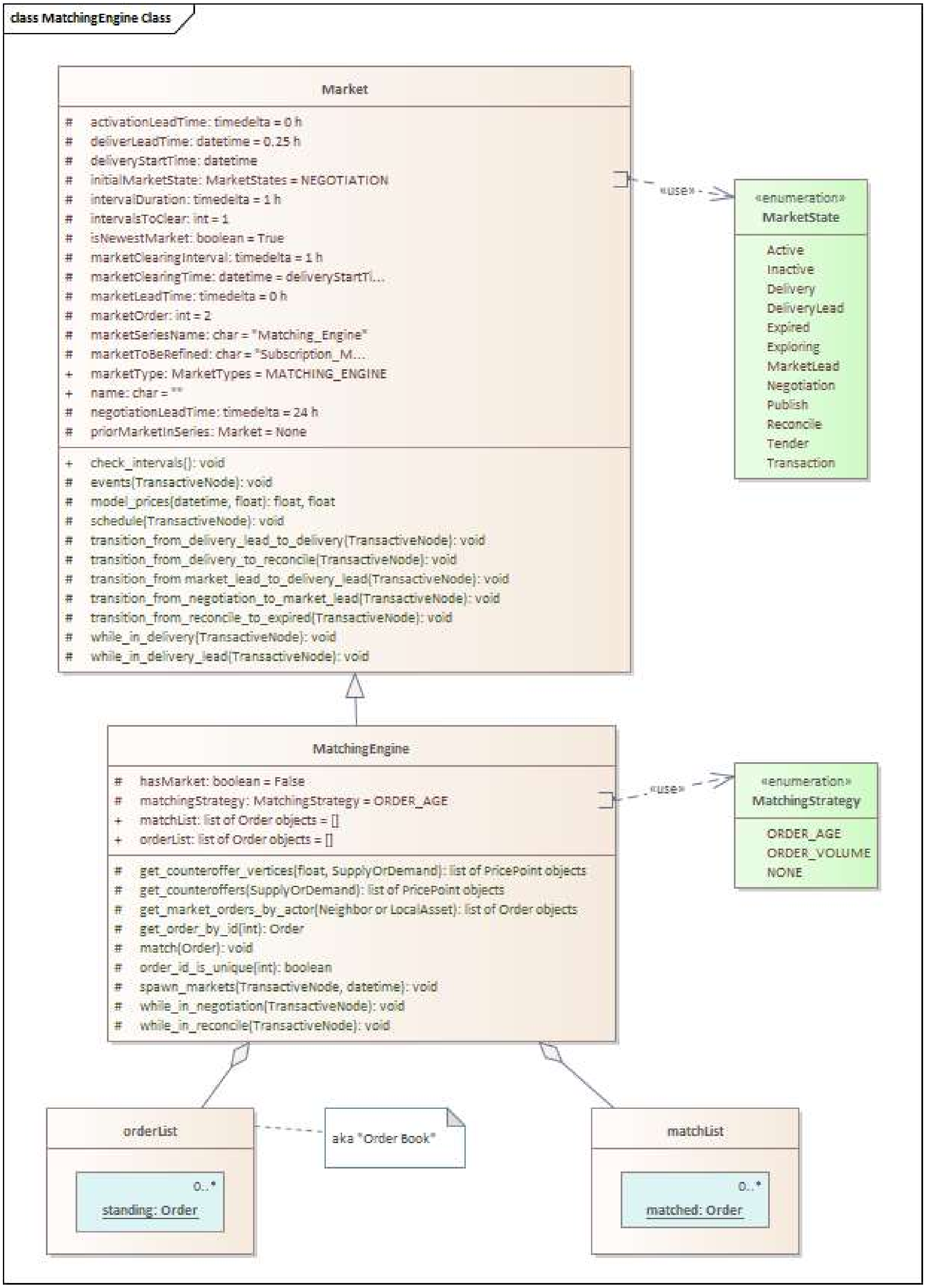

class MatchingEngine Class
Market
# activationLeadTime: timedelta = 0 h
# deliverLeadTime: datetime = 0.25 h
# deliveryStartTime: datetime
# initialMarketState: MarketStates = NEGOTIATION
# intervalDuration: timedelta = 1 h
# intervalsToClear: int = 1
# isNewestMarket: boolean = True
# marketClearingInterval: timedelta = 1 h
# marketClearingTime: datetime = deliveryStartTi…
# marketLeadTime: timedelta = 0 h
# marketOrder: int = 2
# marketSeriesName: char = "Matching_Engine"
# marketToBeRefined: char = "Subscription_M…
+ marketType: MarketTypes = MATCHING_ENGINE
+ name: char = ""
# negotiationLeadTime: timedelta = 24 h
# priorMarketInSeries: Market = None
+ check_intervals(): void
# events(TransactiveNode): void
# model_prices(datetime, float): float, float
# schedule(TransactiveNode): void
# transition_from_delivery_lead_to_delivery(TransactiveNode): void
# transition_from_delivery_to_reconcile(TransactiveNode): void
# transition_from market_lead_to_delivery_lead(TransactiveNode): void
# transition_from_negotiation_to_market_lead(TransactiveNode): void
# transition_from_reconcile_to_expired(TransactiveNode): void
# while_in_delivery(TransactiveNode): void
# while_in_delivery_lead(TransactiveNode): void
«use»
«enumeration»
MarketState
Active
Inactive
Delivery
DeliveryLead
Expired
Exploring
MarketLead
Negotiation
Publish
Reconcile
Tender
Transaction
MatchingEngine
# hasMarket: boolean = False
# matchingStrategy: MatchingStrategy = ORDER_AGE
+ matchList: list of Order objects = []
+ orderList: list of Order objects = []
# get_counteroffer_vertices(float, SupplyOrDemand): list of PricePoint objects
# get_counteroffers(SupplyOrDemand): list of PricePoint objects
# get_market_orders_by_actor(Neighbor or LocalAsset): list of Order objects
# get_order_by_id(int): Order
# match(Order): void
# order_id_is_unique(int): boolean
# spawn_markets(TransactiveNode, datetime): void
# while_in_negotiation(TransactiveNode): void
# while_in_reconcile(TransactiveNode): void
«use»
«enumeration»
MatchingStrategy
ORDER_AGE
ORDER_VOLUME
NONE
orderList
0..*
standing: Order
aka "Order Book"
matchList
0..*
matched: Order

Figure 15. Class MATCHINGENGINE is a child of base class MARKET. Four new attributes and methods are defined. Most important are the *orderList* and *matchList* attributes.

Objects of the new MatchingEngine class are instantiated as follows.

```
> new_matching_engine = MatchingEngine()
```

MatchingEngine methods:

- *get_counteroffer_vertices*: Given an electricity price and the designation of the aggressive order.Order object as either supply or demand, this method creates an order.Vertex object for counteroffers offering nonzero quantity. order.Vertex objects are returned instead of a simple list of quantities because matching engines must keep track of the provenance of counteroffers, including whether the counteroffer supply or demand curve is continuous in this price region.
- *get_counteroffers*: Given the designation of an aggressive order.Order object as either supply or demand, this method collects valid counteroffers from *orderList* (i.e., the order book). Standing counteroffers are "valid" if they are designated as either standing orders or as revised standing orders.
- *get_market_orders_by_actor*: Given a transactive energy system actor (either Neighbor or LocalAsset object), this method gathers and returns a list of orders from among *matchList* and *orderList* objects that are attributed to the actor.
- *get_order_by_id*: Given an order's ID, this method finds the corresponding order.Order object from among the *matchList* and *orderList* orders.
- *match*: Given a new aggressive order, this method attempts to match the order from among eligible counteroffers found in the *orderList*. The matching process may be affected by the current MatchingEngine object's matching strategy (see enumeration MatchingStrategy). If no match is possible, the aggressive offer is moved to *orderList*, where it becomes a standing order for future matches. If one or more matches can be made, the aggressive order and its counteroffers are revised accordingly, and the matched and unmatched order.Order objects are moved into the *matchList* and *orderList* lists.
- *order_id_is_unique*: This method returns True if an ID integer cannot be found among existing order.Order objects in *matchList* or *orderList*.

Important Market methods that are redefined by MatchingEngine:

- *spawn_markets*: This redefined method should create the next MarketEngine.Market object when invoked to do so. The new market object inherits many of its properties from its predecessor, but the delivery time and other times and dates must be updated. For a MarketEngine object, there is a one-to-one relationship between the market object and its TimeInterval object. A new market must be created for every new electricity delivery period.
- *while_in_negotiation*: This redefined method facilitates the asynchronous calculations of new supply and demand curves by LocalAsset and Neighbor objects, the conveyance of signals that represent such supply and demand curves, and the matching of these order.Order objects from among standing counteroffers. Negotiations are permitted until just prior to the delivery period at a time that, for other market types, corresponds to a market clearing time. (No trigger has been determined to stimulate the formulation of new supply or demand curves in this asynchronous manner. Perhaps the creation of new supply or demand order.Order objects should be initiated when predicted supply or demand diverge by some metric from that which has been contracted via prior markets.)
- *while_in_reconcile*: This redefined method should facilitate the gathering of successful match transactions and reconcile the aggregate impact of these transactions with actual electricity usage

during the delivery period. (This process cannot be completed currently because TENT is not itself integrated with metering needed to perform this reconciliation.)

The new MatchingEngine class has properties *orderList* and *matchList* that store collections of order.Order objects. Property *orderList* is for the market's standing orders, and *matchList* is the market's ledger of completed match transactions.

Testing

File *test_order.py* contains unit tests for the extended order.Vertex and order.Order classes. This code may be run to confirm many of the expected behaviors of order.Order methods.

File *test_matching_engine.py* contains tests for new class MatchingEngine and its methods. This code may be run to confirm many of the expected results of MatchingEngine methods.

TENT is currently limited in its ability to support system-level tests, and efforts are underway to address this limitation. Additionally, we have not yet determined the logic by which agents should be induced to create new supply or demand curves for an asynchronous matching-engine market, as was mentioned earlier. For these reasons, TENT's ability to support matching engines is not yet confirmed at the system level.

MatchingEngine.*match()* Logic

```
1.  Gather eligible supply counteroffers.
2.  Get and sort all the unique Vertex prices from the eligible standing orders and aggressive order.
3.  Index through those Vertex prices (order depends on whether aggressive order is supply or demand).
    3.1. Determine effective limit orders for the supply orders and aggressive order at this price.
    3.2. Sum eligible supply order quantities at this price.
        3.2.1. <CASE> Aggressive order quantity is 0 or is undefined at this price
            3.2.1.1. Stop.
        3.2.2. <CASE> Total supply-order quantity = 0
            3.2.2.1. Continue to next price.
        3.2.3. <CASE> Aggressive-order quantity >= total limit supply-order quantity
            3.2.3.1. <CASE> Aggressive-order quantity is not continuous
                3.2.3.1.1. Continue to next price.
            3.2.3.2. <CASE> Aggressive-order quantity is continuous
                3.2.3.2.1. Index through supply effective limit order quantities
                    3.2.3.2.1.1. Create and save match order for supply-order quantity at indexed price
                    3.2.3.2.1.2. Revise standing order
                    3.2.3.2.1.3. Create and save match order for matched aggressive-order quantity
                    3.2.3.2.1.4. Update total matched, transacted aggressive-order quantity and unmatched
                                 aggressive-order quantity remainder
                3.2.3.2.2. Revise aggressive order using total transacted and remainder aggressive-order
                           quantities
        3.2.4. <CASE> Aggressive-order quantity < total limit supply-order quantity
            3.2.4.1. Index through list of matching priorities
                3.2.4.1.1. <CASE> Matching priority is by proportion
                    3.2.4.1.1.1. Continue to next matching priority
                3.2.4.1.2. <CASE> Matching priority is by order age
                    3.2.4.1.2.1. Sort standing limit order quantities by age
                    3.2.4.1.2.2. See Code Logic Block A
                3.2.4.1.3. <CASE> Matching priority is by order volume
                    3.2.4.1.3.1. Sort active supply limit order quantities by volume
                    3.2.4.1.3.2. See Code Logic Block A
```

```
<Begin Code Logic Block A>
1.  Initialize unmatched aggressive-order quantity at aggressive-order quantity for this price
2.  Index through sorted supply-order quantities
    2.1.  <CASE> Aggressive-order remainder quantity < supply-order quantity
        2.1.1.  <CASE> Supply-order quantity is not continuous
            2.1.1.1. Next supply-order quantity
        2.1.2.  <CASE> Supply-order quantity is continuous
            2.1.2.1. Record the tentative supply-order and aggressive-order matches
            2.1.2.2. Update aggressive-order quantity remainder
            2.1.2.3. Stop. There should be no aggressive order remainder.
    2.2.  <CASE> Aggressive-order remainder quantity >= supply-order quantity
        2.2.1. Record the tentative supply-order and aggressive-order matches
        2.2.2. Update aggressive-order quantity remainder
        2.2.3. <CASE> There is an unmatched aggressive-order remainder and aggressive-order continuity is
               false
            2.2.3.1. Next supply-order quantity
        2.2.4.  <CASE> There is no unmatched aggressive-order remainder, or aggressive-order continuity is
               true
            2.2.4.1. Index through the tentative supply-order and aggressive-order quantity matches
                2.2.4.1.1. Create and record match order for matched supply-order quantity
                2.2.4.1.2. Revise order for unmatched supply-order quantity
                2.2.4.1.3. Create and record order for matched aggressive-order quantity
            2.2.4.2. Revise and record a new order for the unmatched aggressive-order quantity
<End Code Logic Block A>
```

Pacific
Northwest
NATIONAL LABORATORY

www.pnnl.gov

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99352
1-888-375-PNNL (7665)

U.S. DEPARTMENT OF
ENERGY