A SMART CONTRACTING FRAMEWORK FOR AGGREGATORS OF DEMAND-SIDE RESPONSE

Sergio ELIZONDO
Heriot-Watt University – UK
se31@hw.ac.uk

Stephen WATTAM
Upside Energy Ltd. – UK
stevew@upsideenergy.co.uk

Valentin ROBU
Heriot-Watt University – UK
v.robu@hw.ac.uk

Rachel JONES
Upside Energy Ltd. – UK
rachelj@upsideenergy.co.uk

Graham OAKES
Upside Energy Ltd. – UK
graham@upsideenergy.co.uk

ABSTRACT

This paper proposes a software framework to facilitate the integration of operational flexibility from distributed energy resources (DER) into balancing services for the electricity system. In general, operational flexibility from industrial, commercial, and domestic customers is highly variable, and thus it is hard to characterise it into standardised balancing services. Moreover, customers might only know their flexible capacity a few hours before its realisation, such as having spare battery capacity that could be used for demand response (DR). In the same vein, customers should communicate promptly to their aggregator, so that their spare capacity could be used for balancing services, and thus acquire some revenue. We therefore propose a framework that allows electricity end-users to write their own short-term DR contracts and submit them to the aggregator's digital platform.

INTRODUCTION

The current low-carbon energy transition is making renewable generation an increasingly important part of the generation mix [1]. Although renewable sources of energy are widely available (e.g., wind, sunlight), they are usually stochastic and intermittent [1]. This poses a serious challenge for system operators, such as the UK's National Grid, to balance supply and demand, hence the increasing interest in demand response (DR) services and business models to enable their economic and environmental benefits. Moreover, there is a growing realisation that not just the large industrial consumers or grid-size storage schemes should contribute to providing this demand-side flexibility, but the process should involve all sizes of consumers and owners of grid-connected storage units.

Independent demand-side aggregators, such as Upside Energy [2], are starting to play a crucial role in helping system operators to deal with these challenges. Aggregators coordinate and bundle the responses of several distributed energy resources (DER), including customers' electric load, in a near real-time cost-effective manner, for the purpose of providing low-carbon balancing services as opposed to traditional *peaker* generators [3]. This enables the use of assets already present in the network, deferring the need for expensive network reinforcements or costly investments in new assets, such as grid-scale battery storage schemes [4]. However, aggregators must solve a complex problem of maintaining and controlling a portfolio of heterogeneous flexibility options to be able to provide efficient balancing services.

The use of DR flexibility involves specific agreements with DER owners, which are assessed on a case-by-case basis according to their suitability for balancing services. These agreements are subject to operational constraints that are specific to each case, such as contractually agreed power amounts, energy densities, availability schedules, and limits on number of nominations in a given time period. The aggregator is therefore in charge of evaluating likely scenarios to make use of these flexibility offers in order to provide value through balancing [3]. Flexibility opportunities could be highly variable and may suddenly come and go (e.g., using some unplanned spare DER capacity in the next hours); thus, integrating these into a useful DR on-the-fly is a significant challenge.

As a key step to addressing this problem, we propose a contracting framework to facilitate the process of characterising non-standard flexibility offers and the capacity of aggregators to benefit from them, using a business model that is akin to the "gig economy". This contracting framework, which is the contribution of this paper, consists of: (1) a *domain-specific language* (DSL) for contract specification; (2) a state-machine based representation of contract logic and execution; (3) a predictive analytics module to guide the optimisation of a portfolio of contracts; and (4) a portfolio optimisation module for the dispatch of DR contracts. Figure 1 illustrates this contracting framework and how its major components interact.

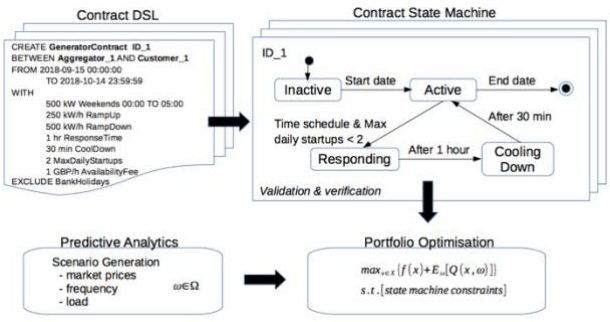

Figure 1: Major components of the proposed contracting framework.

CONTRACTING FRAMEWORK

General Assumptions

It is assumed that customers can only be subscribed to a single aggregator. Similarly, the required technologies for telemetry, control and communication are assumed to be in place and working properly.

DR Contract Types

We aim to characterise two main types of DR contracts:

- **Reactive contracts**. These contracts are executed automatically as soon as their specified trigger expression becomes true. For instance, an agreed contract that provides dynamic frequency response is triggered when the system frequency moves outside its contractual thresholds. These contracts are non-dispatchable, therefore, they are not optimisable. Since this type of contract executes automatically, the reward and penalty fees are previously determined by the aggregator.

- **Optional contracts**. These contracts are dispatchable, and thus, they are scheduled as needed by the aggregator's digital platform to provide balancing services, but without violating the contract terms. A contract of this type could be regarded as a DER's computational model; for instance, a model for a dispatchable generator whose cost function, power and ramping rates are expressed within a contract. Although an optional contract could be served by a collection of DER assets, such as in the case of a virtual power plant (VPP), this paper limits the exposition of this framework to a single DER asset per contract.

Reactive contracts are similar to forward contracts from Finance, provided that there is a triggering expression that becomes true, whilst optional contracts resemble to options from financial derivatives. Similarly, both types of DR contracts are akin to smart contracts from the Blockchain community. However, we prefer reactive and optional DR contracts over the other terms to avoid confusion.

Contract DSL

A domain-specific language (DSL) is a computer language designed to express the requirements of a specific domain and solve particular tasks [6]. For example, the Structured Query Language (SQL) is the *de facto* DSL for managing and querying data from relational databases. In this case the DSL is designed to be able to express the terms and conditions of DR contracts in a way that can be computationally reasoned about. Moreover, the grammar of the proposed DSL does not depend on the host programming language, thus it is an external DSL [6].

We use the extended Backus-Naur form (EBNF) from Computer Science to formally describe the grammar of the contract DSL. This paper only covers basic contract use-cases due to space reasons and unnecessary complexity for the exposition of the proposed framework. Therefore, only a fragment of the grammar is presented. Moreover, the following definition assumes case-insensitive words (i.e., upper- and lowercase letters are treated as being the same).

(*** BEGIN - DSL grammar fragment … ***)

CONTRACT = CREATE_CONTRACT | ALTER_ CONTRACT | DROP_CONTRACT;

(* Main commands *)
CREATE_CONTRACT = 'CREATE', ('REACTIVE' | 'OPTIONAL'), CONTRACT_TYPE, PARTIES, VALIDITY, TERMS_OF_SERVICE;
ALTER_CONTRACT = 'ALTER', CONTRACT_ID
DROP_CONTRACT = 'DROP', CONTRACT_ID

CONTRACT_TYPE = ('GENERATOR' | 'BATTERY' | 'LOAD'), 'CONTRACT';
PARTIES = 'BETWEEN', CUSTOMER_ID, 'AND', AGGREGATOR_ID;
VALIDITY = 'FROM', DATETIME, 'TO', DATETIME;
TERMS_OF_SERVICE = CASE, {CASE}, [OTHER];

(* Terms of service *)
CASE = 'WHEN', TRIGGER_EXPR, TERMS;
OTHER = 'OTHERWISE', TERMS;

TRIGGER_EXPR = ['NOT'] OBSERVABLE, REL_OP, QUANTITY, { BIN_BOOL_OP, TRIGGER_EXPR};
OBSERVABLE = 'FREQUENCY' | 'MARKET PRICE FORECAST' | 'TEMPERATURE';
REL_OP = '<' | '>' | '==' | '!=' | '<=' | '>=';
BIN_BOOL_OP = 'AND' | 'OR';
QUANTITY = ['-'], NONZERO_DIGIT, {DIGIT};

TERMS = {TERM};
TERM = QUANTITY, UNIT, CONCEPT, [INTERVAL];
UNIT = 'kW' | 'MW' | 'kWh' | 'MWh' | 'kW/h' | 'MW/h' | 'GBP' | 'GBP/h' | 'h' | 'hr' | 'minutes' | 's' | ' '; (* etc *)
CONCEPT = 'RESPONSE' | 'RESPONSE TIME' | 'RAMP UP' | 'RAMP DOWN' | 'COOL DOWN' | 'MAX STARTUPS' | 'AVAILABILITY FEE' | 'RESPONSE FEE' | 'STATE OF CHARGE' | 'SOC';
INTERVAL = ['EXCLUDE'], DAY_NAMES, [['FROM'], TIME, 'TO', TIME];
DAY_NAMES = 'BANK HOLIDAYS' | 'WEEKDAYS' | 'WEEKENDS' | 'MONDAYS' | 'TUESDAYS' | 'WEDNESDAYS' | 'THURSDAYS' | 'FRIDAYS' | 'SATURDAYS' | 'SUNDAYS' | 'ALL DAYS' | ' ';

(* Data types *)
CONTRACT_ID = IDENTIFIER (* combination of valid chars *)
CUSTOMER_ID = IDENTIFIER (* combination of valid chars *)
AGGREGATOR_ID = IDENTIFIER (* combination of valid chars *)
DATETIME = DATE, [' '], TIME;
DATE = (YEAR, MONTH, DAY) | (YEAR, '-', MONTH, '-', DAY);
TIME = (HOUR, MINUTE, [SECOND]) | (HOUR, ':', MINUTE, [':', SECOND])

NONZERO_DIGIT = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
DIGIT = '0' | NONZERO_DIGIT;
YEAR = '20', DIGIT, DIGIT;
MONTH = ('0', NONZERO_DIGIT) | ('1', ('0' | '1' | '2'));
DAY = ('0', NONZERO_DIGIT) | ('1', DIGIT) | ('2', DIGIT) | ('3', ('0' | '1'));
HOUR = (('0' | '1'), DIGIT) | ('2', ('0' | '1' | '2' | '3'));
MINUTE = (('0' | '1' | '2' | '3' | '4' | '5'), DIGIT);
SECOND = MINUTE;

(*** END – DSL grammar fragment … ***)

With the previous DSL grammar definition, we can write several contracts for basic DR use-cases. The following examples show two contract scripts, one per each contract type, whose terms and conditions can easily be inferred.

Example of a Reactive Contract

```
Create battery contract
Between customer_1 and aggregator_1
From 2019-01-01 00:00:00 to 2019-01-31 23:59:59
When frequency < 49.7 Hz
        500 kW discharge all days from 13:00 to 19:00
When frequency > 50.3 Hz
        250 kW charge weekends from 00:00 to 06:00
When frequency >= 49.7 Hz and frequency <= 50.3
        50 % state of charge
```

Example of an Optional Contract

```
Create optional generator contract
Between customer_2 and aggregator_1
From 2019-01-01 00:00:00 to 2019-01-03 23:59:59
When market price forecast > 100 GBP
        1 MW response weekends from 00:00 to 16:00
        500 kW response weekdays from 00:00 to 06:00
        750 kW/h ramp up
        500 kW/h ramp down
        5 minutes response time
        15 minutes cool down
        2 max startups
        10 GBP/h response fee
        1 GBP/h availability fee
```

Contract State-Machine

The DSL grammar is not sufficient for being able to reason about contracts. Therefore, we propose a state-machine as a computational model to encode the contract logic and keep track of its execution. More precisely, we use Harel statecharts [5] due to their expressibility and generally more compact representation than canonical finite state-machines, such as Mealy and Moore machines.

We identify a minimum of five states in order to characterise the logic of reactive and optional DR contracts. Therefore, for a DR contract to be valid it must have the following states.

- **Start**: it is the initial pointer to the state-machine.
- **Inactive**: the execution of the contract waits at the Inactive state until the contract's start datetime applies. Then the execution moves to Active state.
- **Active**: at this state the contract is within effective datetime period and waiting for action. The execution moves to Responding when a triggering condition becomes true (including a dispatch schedule).
- **Responding**: this is when the contract execution is set to responding, and thus the DER asset linked to the contract must be responding with the specified terms. There may be more than one responding states depending on the contract conditions.
- **Final**: it denotes that the contract has expired.

Also, as in Figures 2 and 3, it is possible to use auxiliary states in order to simplify the state-machine or improve semantics. For example, in Figure 2 there is a *Managing* state in order to improve the state-machine semantics, as it could have been modelled with a Response state. Similarly, in Figure 3 there is a *Cases* state, which we use for branching conditions; and there is a *Cooling Down* state that is used to improve the semantics, as it could have been placed as a post-condition in the Response state (i.e., on exit event).

As a proof-of-concept, we implemented the DSL in Python using TextX [7] and its parsing facilities. The DSL contract strings are processed and translated into their corresponding Harel statecharts by using *Sismic* library [8, 9]. Figures 2 and 3 are generated by Sismic using PlantUML [10].

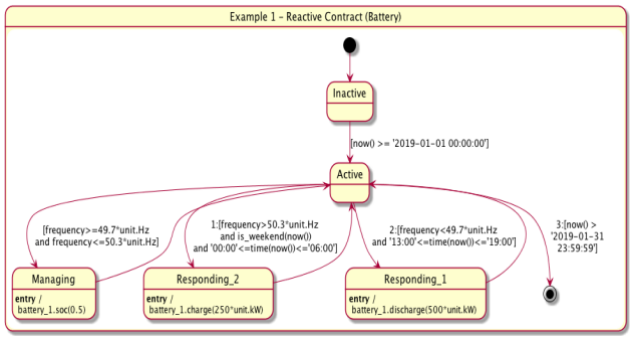

Figure 2: Example of a Reactive Contract state-machine.

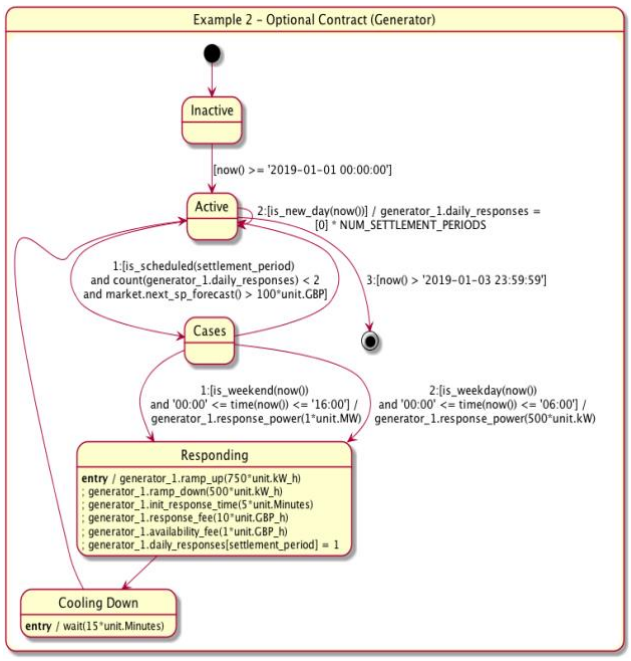

Figure 3: Example of an Optional Contract state machine.

Predictive Analytics of Observables

One of the main sources of uncertainty, within this software framework, comes from the prediction of observable measures (e.g., frequency, temperature, market prices) that are used by DR contracts. Although there are several other sources of uncertainty (e.g., contract reliability, asset failures), these are beyond the scope of this paper. The following procedure is adapted from [11] to generate plausible forecast scenarios for the contracting framework setting:

1) A predictive model is estimated for each observable measure in the context of DR contracts, using historical data. The estimation and selection of predictive models might depend on the attributes (e.g., trend, seasonality) shown in the training dataset. Time-series and machine learning models may both be applicable.

2) A prescribed number of paths (e.g., 100 scenarios) is generated from each predictive model according to the desired horizon (e.g., day-ahead). The error terms are simulated from ~ *N(0, sd)*, where *sd* is the standard deviation in the training data for the target variable. If there are correlated observables, the ones used as explanatory variables should be predicted first and then be included as regressors for the correlated observables [11]. Therefore, each scenario will contain a predicted set of plausible observables.

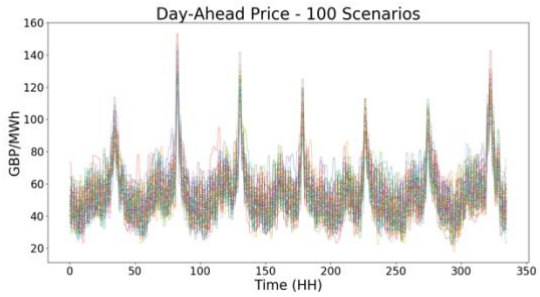

Figure 4: Simulated paths of DA prices for a week in 2007.

3) Scenario reduction techniques, such as backward reduction and (fast) forward selection heuristics [11, 12], are applied so as to get a smaller number of representative scenarios (e.g., 10 out of 100).

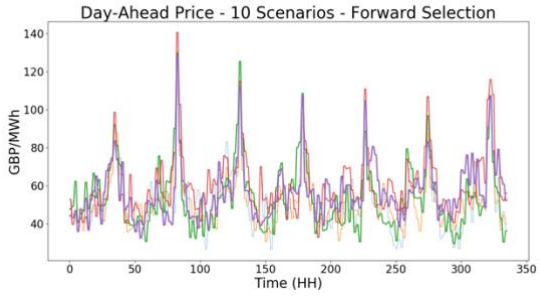

Figure 5: Reduced scenario set from Figure 4.

4) Only the reactive contract state-machines that would become active within the considered horizon are run in simulation mode. That is, the contract effects exist only inside the simulation instance, and thus the actual DR contract does not change. This procedure is used to compute the portfolio position under those scenarios.

Contract Portfolio Optimisation

Since reactive contracts are executed automatically depending on observable measures (e.g., frequency, temperature), they cannot be optimised directly as opposed to the portfolio position. In this case, the main source of flexibility comes from optional contracts, which can be scheduled subject to terms and conditions. Although contract state-machines capture their execution logic, in this contracting framework they are oblivious to other contract state-machines. Therefore, a centralised optimisation module determines the most convenient scheduling for optional contracts, using the following procedure:

1) Only the optional contract state-machines that would become active within the optimisation horizon are run in simulation mode (i.e., without affecting the actual contract outside the simulation). These contracts are simulated for the reduced set of plausible scenarios generated by the predictive analytics module, while collecting technical constraints from the contract state-machine.

2) A mathematical programme, similar to a unit-commitment problem, can be formulated and solved to obtain the most economic schedule for optional contracts under the reduced set of scenarios. Since we restricted the DR contracts to be associated to a single DER asset, it is possible to formulate the management problem as in [13], but for more general DR contracts is still an open question.

3) Finally, the resulting schedule of each optional contract state-machine is updated for only those contracts that are active during the optimisation horizon. Therefore, under actual operation, if the function *is_scheduled(settlement_period)* becomes true and the other conditions hold (Figure 3), the contract moves to a responding state.

CONCLUSIONS

A software framework has been proposed to characterise DR offers into digital contracts, so that it is possible to reason about the contracts' terms and conditions. The use of such a framework provides several advantages to aggregators and DER owners. First, the DSL not only allows a more dynamic creation of flexibility contracts for tailored use-cases (e.g., use 50% of a 1MW battery for the next week 4-6am only), but also allows the delegation of contract specification to DER owners. Second, the state-

machine contract representation, created by the contract DSL, allows for machine reasoning about each use-case, and precise reporting on service status. Third, the computational analysis of the state machines can provide feedback to DER owners who write contracts in order to verify expected contract behaviour under certain simulated conditions. Fourth, the mapping from state machines to a portfolio optimisation problem enables aggregators to automate part of their workflow so as to adapt faster to changing conditions. Fifth, the contract portfolio optimisation problem can be extended with predictive analytics to optimise for predicted conditions, resulting in a more efficient use of DER. Finally, this framework could also be used as a test-bed to generate several thousands of contracts through the DSL in order to research scalable optimisation strategies or inform hypotheses about portfolio composition.

Future Work

Other important aspects of a smart DR contracting framework which are left for future work include:

- *Validation and verification*: contracts must be valid before they are agreed upon and verified for expected behaviour. Validation is not precisely hard, and some initial rules could be defined to determine whether a contract is valid or not (e.g., non-retroactive datetimes, proper semantics, coherent technical constraints, etc). However, verification of expected behaviour is hard, and might require some level of property-based testing and simulation to provide contract writers with some level of feedback.

- *Auditing*: ability to inspect the records so that it is possible to evaluate contract performance, for the purpose of billing (including penalties) and analytics (e.g., reliability).

- *Language expressibility and optimisation*: improvements on the contract DSL syntax to make it more natural and potential ways of reorganising conditions (such as regrouping cases per interval as opposed to conditions over observables, e.g., "When Weekdays 1MW response").

- *Multistage optimisation and risk hedging*: the paper limited to a very simple contract portfolio optimisation, and more realistic optimisations, such as multi-stage stochastic programming with trade-offs of risk measures, are left for future research.

Acknowledgments

The authors gratefully acknowledge the financial support from InnovateUK. In addition, the authors would like to thank Nathan Messer and Anthony Waite, from Upside Energy Ltd, for the engaging discussions about DSLs and state-machine models.

REFERENCES

[1] D. MacKay, 2008, *Sustainable Energy – without the hot air*, UIT Cambridge, UK, 2-18.

[2] Upside Energy Ltd, 2018, https://upsideenergy.co.uk/

[3] H. Ma, V. Robu, N. Li, D.C. Parkes, 2016, "Incentivizing Reliability in Demand-Side Response", *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, NY, USA.

[4] OFGEM, 2016, *Aggregators - Barriers and External Impacts*, London, UK, 1-61.

[5] D. Harel, 1987, "Statecharts: a visual formalism for complex systems", *Science of computer programming*, vol. 8(3), 231-274.

[6] M. Fowler, 2010, *Domain-specific languages*, Pearson Education, NJ, USA, 1-7.

[7] I. Dejanović, R. Vaderna, G. Milosavljević, Ž. Vuković, 2017, "TextX: A Python tool for Domain-Specific Languages implementation", *Knowledge-Based Systems*, vol. 115, 1-4.

[8] A. Decan, 2018, "Sismic Interactive Statechart Model Interpreter and Checker", https://github.com/AlexandreDecan/sismic.

[9] T. Mens, A. Decan, N. Spanoudakis, 2018, "A method for testing and validating executable statechart models", *Software and Systems Modeling*, Springer, 1-27.

[10] PlantUML, 2018, http://plantuml.com/.

[11] A.J. Conejo, M. Carrión, J.M. Morales, 2010, *Decision making under uncertainty in electricity markets*, Springer, NY, USA, 66-115.

[12] J. Dupačová, N. Gröwe-Kuska, W. Römisch, 2003, "Scenario reduction in stochastic programming", *Mathematical programming*, vol. 95(3), 493-511.

[13] J.M. Morales, A.J. Conejo, H. Madsen, P. Pinson, M. Zugno, 2013, *Integrating renewables in electricity markets: operational problems*, Springer, 243-287.