Adaptive Charging Network Simulator

Zachary Lee
Caltech
1200 E California Blvd.
Pasadena, California 91106
zlee@caltech.edu

Rand Lee
Caltech
1200 E California Blvd.
Pasadena, California 91106
randl@caltech.edu

ABSTRACT

Our goal in this project is to design and implement a system which allows for rapid testing and deployment of theoretical work into production environments. Translation of theories into real-world applications is often difficult and can result in failure when requirements not accounted for during research become evident and critical in an production environment. Furthermore, modeling assumptions may be inappropriate, which does not become evident until it is deployed within production systems. For these reasons, we set out to build a plug-and-play software simulator which allows for expedited testing of new theoretical algorithms within an Adaptive Electric Vehicle Charging Network. This system is built to rapidly test new changes in an environment modeled after real world data and scenarios. This greatly reduces the amount of time required for feedback. To achieve this end, we implement several software models. The first is a model of a battery, which behaves according to data recorded from real electric vehicle charging profiles. We wrap this software-implemented battery into a model electric vehicle (EV), which communicates with the existing software suite as if it were connected to a real charging station (EVSE). It is important to note that in this case, the software knows no more about the vehicle and its battery than it would in the real-world case. We then load an algorithm into the software suite and run a battery of tests, each exposing metrics about performance and stability. These metrics are then checked that they are within acceptable bounds. Usually, such a test would take hours, as that is the time scale for charging EV batteries. To counteract this, we accelerate time within the software. In our tests, we usually run the simulation at 3600x normal time. This means each second counts as one hour. With this optimization, an algorithm can run through an entire week's worth of charging curves in under a minute. The various algorithm performance metrics exposed by the simulator include but are not limited to total energy delivered, service level, and maximum power draw. These metrics can then be fed back into the development of the theory.

KEYWORDS

Electric Vehicles, EV Charging, Adaptive Charging Network, Cyber Physical Systems, CPS Simulators

ACM Reference format:
Zachary Lee and Rand Lee. 2017. Adaptive Charging Network Simulator. In *Proceedings of ACM Conference, Pasadena, CA, USA, June 2017 (CS/EE 148),* 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CS/EE 148, June 2017, Pasadena, CA, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$0.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As electric vehicles (EVs) grow in popularity, there is a growing need to install charging stations known as electric vehicle supply equipment (EVSE) in the parking lots of offices, apartment buildings, airports, and city centers. However, the electrical infrastructure at these facilities was never intended to supply the massive power needs of charging a large number of electric vehicles. As a result, facility owners are forced to choose between installing only a few charging stations or investing in expensive infrastructure upgrades.

In order to mitigate the limits that constrained infrastructure imposes on EV charging, Caltech has developed an Adaptive Charging Network (ACN). The ACN consists of a central controller and a collection of networked and controllable EVSEs. An algorithm running on the central controller takes in data collected from the EVSEs and users as well as information about the constraints of the system and determines a schedule of charging rates for each active EVSE in the network. These rates are then sent over the network to the EVSEs where they enforce an upper bound on the amount of current the connected EV can draw. Our simulator seeks to test the algorithms and associated software on the central controller by simulating the network of EVSEs and EVs and system constraints. For more information about the ACN system see [1].

Our simulation system runs a collection of tests which expose the cost efficiency, completeness, and other metrics regarding the algorithm. Cost efficiency bounds the maximum instantaneous power usage of an algorithm and relates to the demand charge on an ACN user's electricity bill. Completeness represents whether or not sufficient energy was delivered to EVs in the system and is critical for users to trust the system. These are but two metrics the simulator can expose about an algorithm without the actual need for deployment on the physical system. The relative differences in these metrics between algorithms is an indispensable tool in the iterative improvement of theory and its application.

The remainder of this paper is organized as follows. Section 2 introduces our approach to modeling the system architecture, EVSEs, and EVs. In Section 3 we give an overview of our simulator followed by instructions for using the simulator in Section 4. Section 5 presents exemplary results from tests run using our simulator. Finally, Section 6 discusses the conclusions drawn from this work as well as future work on the project.

2 SYSTEM MODEL

In order to simulate the system we first need to model all relevant parts. In all cases we seek to create models which match the behavior of the physical system as closely as possible. We also enforce that our models only communicate with the rest of the code being tested through the same interfaces used when communicating with

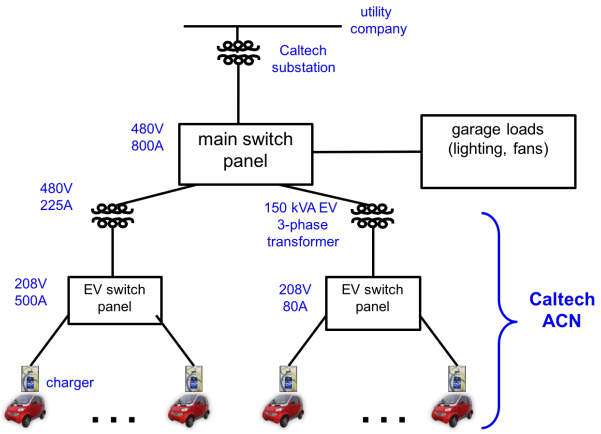

Figure 1: This figure depicts the electricity distribution network at the California Parking Garage on the Caltech campus. Power enters the facility from the grid through a substation. It then passes through the main switch panel. At this point some power goes to lighting, fans, elevators, etc. while the rest is fed through two step-down transformers which feed EVSEs through sub-panels. At all times the current draw through a component must not exceed its rated limit or else a breaker could trip or permanent damage could be done to the component.

the physical system. This ensures the results from our simulator accurately predict the performance of the algorithm and associated code when run on the physical system.

2.1 System Architecture

Fig. 1 depicts the electrical distribution system in a parking facility at Caltech. These systems have a tree structure where power enters the facility at the root node and travels through many intermediate nodes such as transformers and switch panels before reaching the leaf nodes which are the EVSEs. Each intermediate node has an associated current limit. The sum of all currents drawn by EVSEs who are descendants of an intermediate node should be less than this constraint.

In order to model the system architecture, we recreate the tree of constraints in software. We model each constraint as a Load Manager (LM). These load managers form the inner nodes of the tree. The leaf nodes are software interfaces to the physical EVSEs. Fig. 2 shows how the distribution network in Fig. 1 is modeled. Using this model we can simulate an arbitrarily large and complex tree of constraints.

To model these constraints we define S_i to be the set of all EVSEs (leaf nodes) which are descendants of inner node i, $\bar{R}_i$ to be the current constraint of inner node i, and $r_j(t)$ to be the charging rate of EVSE j at time t. We can then express the constraints on the system as:

$$\sum_{j \in S_i} r_j \leq \bar{R}_i$$

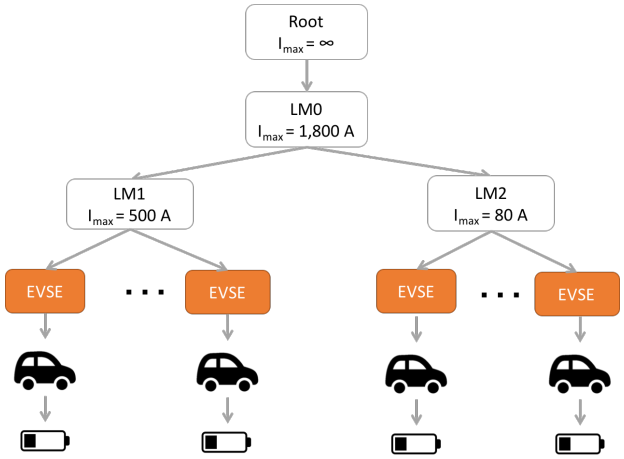

Figure 2: We convert the distribution architecture in Fig. 1 into a tree structure in software. Each of the inner nodes represents a current constraint in the physical system.

2.2 EVSE Model

Within the ACN control code, we have defined a standard interface for communicating with EVSEs from multiple manufactures. These standard commands such as updating the pilot signal (upper current limit) of the station and measuring the actual current draw of the EV are then converted into EVSE specific messages and sent out over a wireless mesh network.

In order to create a simulated EVSE, we implemented this interface. However, instead of communicating to a physical EVSE connected to a real EV, we emulate the functionality of the EVSE software. In addition to implementing the standard interface methods, this simulated EVSE also contains a simulated EV object. The EVSE can then interact with the test EV by setting its pilot signal or querying its current draw. In addition, the EVSE can simulate EVs arriving and departing throughout a test. This is done through our event based testing framework.

2.3 EV Model

Each simulated EVSE contains a test EV object. This test EV can accept new pilot signals from the EVSE, calculate its charge rate and return it to the EVSE, and update it's state of charge based on this charge rate. We use a simplified model of the EV charging circuitry and battery to determine charge rate at any given time. The current that an EV draws from the charging station can be described as:

$$r(t) = \min(r_{chgr}(t),\ r_{pilot}(t),\ r_{batt}(t)) \tag{1}$$

Where $r_{chgr}(t)$ is the current limit of the car's on-board charger, $r_{pilot}(t)$ is the pilot signal received from the EVSE, and $r_{battery}(t)$ is the battery's acceptance current. In a real EV $r_{chgr}(t)$ is a function of the battery voltage, temperature and other factors, however for simplicity, we assume $r_{chgr}(t) := \bar{r}_{chgr}$ which is the upper limit of the on-board charger.

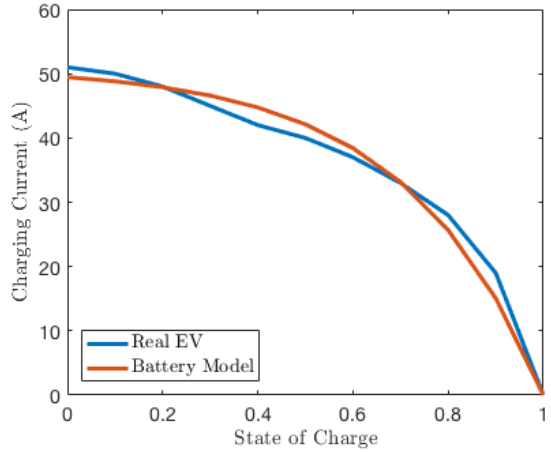

Figure 3: Using real data collected by a Tesla Model S owner, we can fit (2) to the real data. This results in $\bar{r} = 51A$ and $\eta = 3.5$. By inspection we see that the fit is quite good. We can quantify this by taking the relative mean squared error between the actual charging curve and our model which in this case is 8.65e-6. Relative mean square error is given by $\frac{1}{N}\sum_t \left(\frac{e(t)}{r(t)}\right)^2$, where $e(t)$ is the error between the model the the true current and $r(t)$ is the true current.

Furthermore, we add a degree of random error to the charging rate advertised by the EV. We add the error during the current measurement function call to simulate the measurement error in the charger's circuitry.

2.4 Battery Acceptance Model

The acceptance rate of the battery is a function of its state of charge. which we denote as $y(t)$. After examining the charging curves of real EVs we have identified (2) as a good model of the battery charging curve.

$$r_{batt}(y(t)) := \bar{r}(1 - e^{-\eta(1-y(t))}), \quad y(t) \in [0,1], t \geq t_0 \tag{2}$$

where $\bar{r}$ is the max acceptance rate of the battery and η is a tuning parameter.

We can then fit this model to data collected from real electric vehicles. Fig. 3 shows an overlay of the battery model and real data from a Tesla Model S. With correct choices of $\bar{r}$ and η, the model fits the data well.

2.5 Unconstrained Battery Charging Model

We can describe the rate of change in the state of charge of the battery when unconstrained by the pilot signal or on-board charger as:

$$\dot{y}(t) := \alpha(1 - e^{-\eta(1-y(t))}), \quad y(t) \in [0,1)$$

where $\alpha := \frac{\bar{r}}{C}$ and C is the capacity of the battery. Hence, for $y(t) \in [0,1)$ we have

$$\frac{\dot{y}(t)}{\left(1 - e^{-\eta(1-y(t))}\right)} = \alpha, \quad y(t) \in [0,1)$$

$$\underbrace{\frac{e^{\eta(1-y(t))}}{e^{\eta(1-y(t))} - 1}}_{\text{h(y(t))}} \dot{y}(t) = \alpha, \quad y(t) \in [0,1)$$

Let

$$\frac{d}{dy}H(y) = h(y) := \frac{e^{\eta(1-y(t)}}{e^{\eta(1-y(t)} - 1}$$

Using chain rule we have

$$\frac{d}{dt}H(y(t)) = h(y(t))\dot{y} = \alpha$$

giving us

$$H(y(t)) = \alpha t + constant$$

and

$$\begin{aligned} H(y) &= \int h(y)dy \\ &= \int \frac{e^{\eta(1-y)}}{e^{\eta(1-y)} - 1}dy \\ &= -\frac{1}{\eta} ln\left(e^{\eta(1-y)} - 1\right), \quad y \in [0,1) \end{aligned}$$

which yields

$$-\frac{1}{\eta} ln\left(e^{\eta(1-y)} - 1\right) = \alpha t + constant$$

Finally, solving for $y(t)$

$$y(t) = 1 - \frac{1}{\eta} ln(1 + ke^{-\eta\alpha(t-t_0)}), \quad t \geq t_0 \tag{3}$$

where k is found from the initial condition[1]

$$k = e^{\eta(1-y(t_0))} - 1 \tag{4}$$

2.6 Constrained Battery Charging Model

In the real system, the EV charging rate is upper bounded by the on-board charger limit and pilot signal. The actual charging rate of the EV is given by (1). In order to accurately model charging of the EV's battery, we must account for these constraints.

Since we are interested only in the change in SoC over a small time period from t_0 to t_1, we assume that $r_{pilot}(t) = r_{pilot}$ over the interval $[t_0, t_1]$.

Let

$$\beta := min(\bar{r}_{chrg}, r_{pilot})$$

such that

$$r(t) = min(\beta, r_{batt}(t))$$

[1]This derivation was adapted from a memo produced by Dr. Steven Low and is a cleaner version of our original derivation.

We define $y(t_x)$ to be the SoC such that

$$r_{batt}\left(y(t_x)\right) = \beta$$

plugging in 1 we can solve for $y(t_x)$

$$y(t_x) = 1 + \frac{1}{\eta} ln\left(1 - \frac{\beta}{\bar{r}}\right) \tag{5}$$

The time when the EV reaches y_x is denoted t_x. We use the fact that (2) is monotonically decreasing with time to conclude

$$r(t) = \begin{cases} \beta & \text{if } t \leq t_x \\ r_{batt}(t) & \text{if } t > t_x \end{cases} \quad \text{for } t \in [t_0, t_1]$$

Because β is a constant, we can find the time taken to reach y_x using

$$t_x = \frac{y_x - y_0}{\beta} + t_0$$

Note that if $t_x < t_0$ then $r(t) = r_{batt}(t)$ for the entire time interval and if $t_x > t_1$, $r(t) = \beta$ for the entire interval.

The expression for $y(t_1)$ can then be given as

$$y(t_1) = \begin{cases} \frac{\beta}{C}\left(t_1 - t_0\right) + y(t_0) & \text{if } t_x \geq t_1 \\ 1 - \frac{1}{\eta} ln\left(1 + k_x e^{-\eta\alpha(t - t_x)}\right), & \text{if } t_x \in (t_0, t_1) \\ 1 - \frac{1}{\eta} ln\left(1 + k_0 e^{-\eta\alpha(t - t_0)}\right), & \text{if } t_x \leq t_0 \end{cases}$$

where

$$k_x = e^{\eta(1 - y(t_x))} - 1$$
$$k_0 = e^{\eta(1 - y(t_0))} - 1$$

and $y(t_x)$ was found already using (5).

3 SIMULATOR DESIGN

3.1 GoogleTest Framework

The GoogleTest Framework is used for all functional testing in the simulator. It is an industry standard testing suite, and allows for very well-defined setup and cleanup of our testing fixtures. With this framework we are able to construct very specific scenarios in which to run the ACN system.

3.2 Time Acceleration

The time scale for electric vehicle charging is on the order of hours. As a result there is inherent difficulty in testing the behavior of cars over their entire charging cycle within a reasonable amount of time. To accommodate this type of testing in a short time period we developed a Time class which implements an artificial epoch with which it builds an accelerated time scale. The new epoch is set once at the construction of the Time singleton and is reset each time the acceleration of the system changes. We calculate the current artificially accelerated time using

$$t_{now} = t_{epoch} + a \times (t_{system} - t_{epoch})$$

where t_{epoch} is the start time of our artificial epoch and t_{system} is the current system time, both of which are measured relative to the standard Unix epoch. The acceleration constant a determines the linear scaling of time. For example $a = 3600$ means that 1 hour of simulated time takes 1 second of real time. By using t_{now} in all of the system's time-based actions, we effectively can increase the rate at which time passes. Within the Time class, we also implement custom versions of the system sleep commands. These commands divide the desired sleep time by a so that they too are accelerated by the same amount as the rest of the code.

The only caveat here is that processing time is constant per loop regardless of time acceleration. Furthermore, processing time becomes a larger and larger portion of overall time taken by the program the more we increase the acceleration. This means that processing time will begin to skew our results. Consider this example. Our processing time is 5 seconds accelerated 3600x. This is 1.39 milliseconds. If processing time is 0.50 ms, then we have charged cars for 1.89 ms, an increase of 36%, when we really expected to charge for 1.39 ms. In the real-world case, we would have waited 5.0005 seconds with such a processing delay, translating to an increase of 0.01%. To account for this, we measure the actual processing time and subtract it from the desired delay, both of which are measured in accelerated time. This mitigates the problem so long as the processing time is less than the desired delay time. If processing time becomes longer than the desired delay, a must be reduced to maintain testing accuracy. In our experience $a = 3600$ works well and produces sufficiently quick tests without sacrificing test accuracy.

To test that time acceleration works as designed, we have a test fixture to check various timing scenarios. Those can be found in the TestTimeAccleration suite of tests. There, the system runs through various fast and slow acceleration speeds and verifies that those speeds don't break general assertions.

3.3 Event Driven Testing Framework

We have developed an event driven testing framework to make simulations easier to set up and more realistic. In this framework test designers can schedule events to occur throughout the course of the test. Example events include plugging in a new EV, unplugging an existing EV, or testing if the SoC of an EV is near an given value. A complete list of the event types currently offered and their effects are shown in Table 1.

All event types inherit from a common base class, and must contain the address of the EVSE they act on as well as the time when they will occur, measured in seconds since the beginning of the test. In addition, all events must implement an invoke() method which carries out the effect of the event.

All events for a particular test are stored in a priority queue which is sorted by time of occurrence. The use of a priority queue means that the test designer can add events to the queue in any order and the queue will automatically sort events to be in the correct order of occurrence. This is a useful feature when designing tests, as the designer can group events in the test function logically instead of sequentially.

Table 1: Event Types and Effects

Event	Effect
Plug In	Update the EVSE's EV object to have the desired type, SoC, and capacity. Set the EV plugged in flag.
Unplug	Reset the simulated EV to a default state and clear the EV plugged in flag.
Expect Battery Level	Use GoogleTest to assert that the EV's battery level (mAh) is $\geq$, $\leq$, or $\approx$ the given value.
Expect SoC	Use GoogleTest to assert that the EV's SoC (%) is $\geq$, $\leq$, or $\approx$ the given value.
Expect EV Plugged In	Use GoogleTest to assert that the EV plugged in flag is set.

When a test begins, the current time is stored. In each time step, the time duration since the beginning of the test is calculated. The time of the event at the head of the queue is then checked against this duration. If the event time is less than the current time, this event is removed from the queue and its invoke method is called. The time of the new head of the queue is then checked. This process repeats until the time of the event at the head of the queue is greater than the duration of the test thus far.

Fig. 4 shows a simple example of the type of test a designer could implement using the events framework. The framework is flexible enough, however, to support any number of events occurring throughout a test. Currently events are invoked synchronously within the process loop which runs every 5 seconds. This does not pose a problem, since the rest of the ACN software is not aware of any changes in the EVSE or EV until the next process loop anyway. However, the events framework has been designed to be thread safe and can run in a separate thread updating much more quickly if desired in a particular application.

4 USING THE SIMULATOR

4.1 Constructing a Test

(1) Create a new TEST_F function within the SimulationTest file. This file contains test fixtures and utilities which handle boilerplate code so that you can construct tests quickly and easily.
(2) Create the tree of constraints which define your network. Constraints are modeled as load managers and can be added using the network.addLoadManager(...) function. Each load manager should be given a name, a capacity limit, and an algorithm. In addition, a parent load manager can be specified, which allows you to build deeper tree structures like the one in Fig. 2. If no parent is defined, the load manager is automatically made a child of the root node.
(3) Use thebuildNetwork(...) utility function to add EVSEs to the network you created in step 2. The arguments of this function are the number of EVSEs to add, a vector of load manager parents and a vector of rate increments. Element i in each vector corresponds to the parent or rate increment of EVSE i. This function automatically assigns an address to each EVSE. The address for EVSE i can be obtained using the utility function getAddress(i).
(4) If desired, use the initEV(...) utility function to initialize EVs at all EVSEs. The arguments for this function are the number of EVSEs , a vector of EV types, a vector of initial charges, and a vector of battery capacities. As before, element i of each vector corresponds to the EV attached to EVSE i.
(5) Add events to the queue. This is done by using the q.events.add(...) function. The argument to this function is a pointer to an event object. The available events are given in Table 1. Each event constructor takes in the time when you want the event to occur as well as the address of the EVSE it effects. Additional arguments depend on the event type. Events do not need to be added in any particular order.
(6) Call process(N) to run the test. N is the number of iterations through the process loop. Note that the default period of the process loop is 5 seconds.

4.2 Running Tests

Our testing framework outputs tests in a executable which can be run from the command line. Running the executable runs all tests by default, and requires no command line options. Furthermore, the logging output of each test is written in real-time to a log file at: "/var/log/jarvis/tests/<testname>.log". This makes following a test while it is running simple in the case where we need to re-run a failed test.

The binary accepts an option "–gtest_filter=" which allows for singling out a set of tests. For example, to run all simulator tests, one could write: "./tests –gtest_filter=TestSimulator.*". As the command implies, this runs all tests under the TestSimulator fixture, ignoring TestTimeAcceleration and other tests.

4.3 Automated Integration Testing

Test runs need not be done manually. Nightly and on-demand testing runs can be scheduled using the cloud. Since these software-based tests require no specific hardware to run, we have set up a server which builds versions of the binary, snapshotted at specific versions of the code base. For each change which is committed to the repository, we run the entire battery of tests, and report the results. As stated earlier, this is done nightly by default, and on-demand when needed.

The result for this is a per-commit summary of test results, allowing for quick debugging of common bugs because the test results will clearly show which change the failure originated from. This integral simulation testing allows for fewer complex bugfixes in the future.

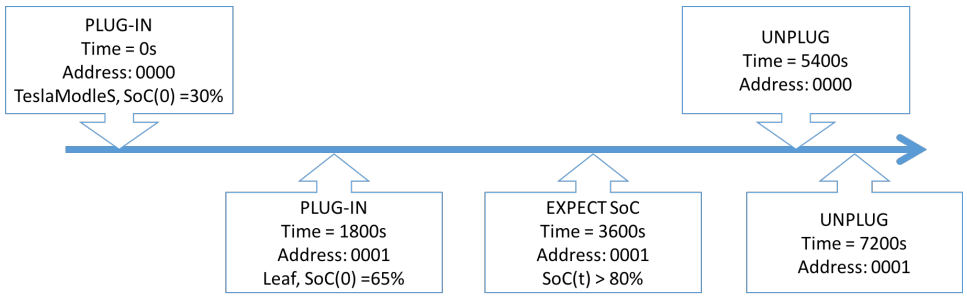

Figure 4: Test designers are able to simulate the arrival and departure of EVs throughout the test as well as test expectations of variables at specific times during the test. In this example, a Tesla Model S arrives at the beginning of the test followed by a Nissan Leaf 30 minutes later. At 1 hour into the test, the SoC of the Leaf is required by the test designer to be above 80%. The Tesla leaves 1.5 hours into the test followed by the Leaf 2 hours after the test began.

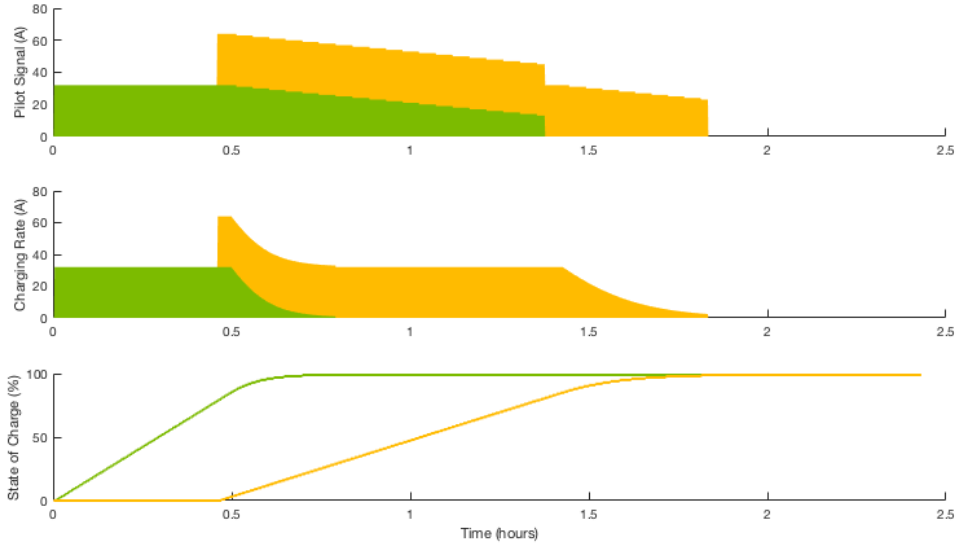

Figure 5: This test corresponds to the events depicted in Fig. 4. In these plots green represents EVSE0 and yellow represents EVSE1. The top plot shows how the pilot signal given by the Round Robin algorithm changes over time. We can see from the top plot that an algorithm called ramp down adjusts the pilot signal of each EVSE so that it follows the actual charging rate of the car as it decreases. This feature helps to free up charging capacity for other cars instead of allocating it to EVs which are not using it. However, because of the design of the ramp down algorithm and the fact that the rate increment of the EVSEs in this test were set to be 1 A, this ramp down still allocates a significant amount of current to the green EVSE even though it had finished charging long before. This is an example of an insight our simulator can give us about the software and algorithms. We can also see that the pilot signal drops to zero when an unplug event occurs. The middle plot shows the charging rate of each EV. We can see that the pilot signal of each EVSE upper bounds its actual charging rate. We can also see that as the battery nears a full charge the charging curve decays which is consistent with what we see in data collected from real EVs. The bottom plot shows how the state of charge of each EV changes over time. We can see that both EVs are fully charged by the time they leave.

5 ILLUSTRATIVE RESULTS

In order to demonstrate the effectiveness of our simulator we have included metrics taken from two tests runs on the simulator.

The first test follows the simple event structure described in Fig. 4.The results of this test are shown in Fig. 5. This test creates a LoadManager with an 80A bottleneck with two EVSEs each with a rate increment of 1 A. After this is set-up, the test simulates the the plug-in event of a Tesla Model S to EVSE 0 at the fifth second, which unplugs 90 minutes later. In addition, the test simulates the

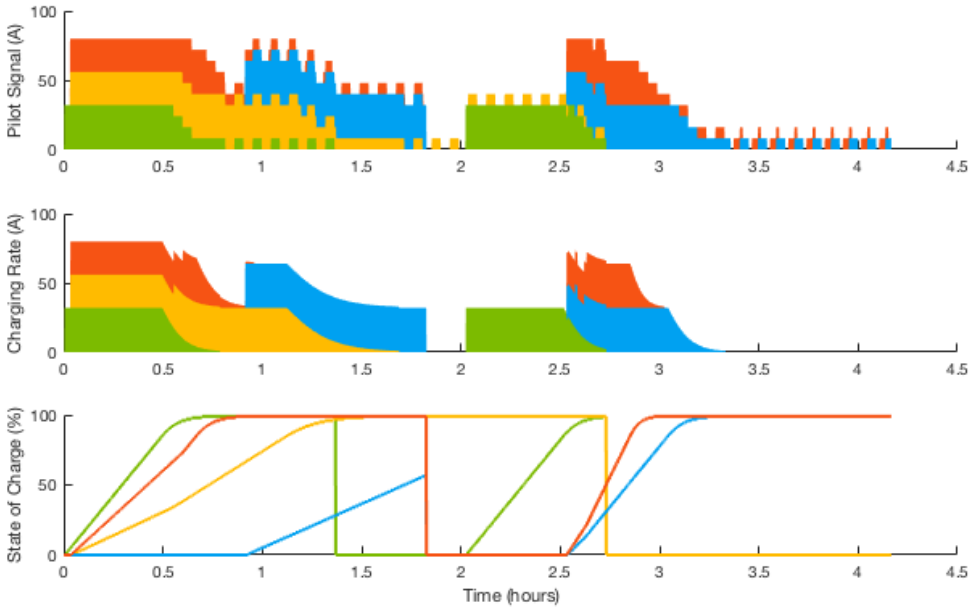

Figure 6: This test depicts the complexity which is possible using our event based framework. Here green, yellow, red, and blue each denote a different EVSE. Throughout the course of a 4+ hour test, a total of 7 EVs charge for varying amounts of time. This plot is purposefully complex and dense as it serves to illustrate how much information can be gained from our simulations. For example, we can see from the top plot that the ramp down algorithm is taking far less time to reduce the pilot signal to 0 after an EV has finished charging. This is because the rate increment in this test has been set to 8 A instead of 1 A. In addition, for the last hour of the test, the blue and red EVSEs have an EV plugged in, but they are done charging. Here we can see how our algorithm attempts to periodically allocate charge to these EVSEs to allow them to wake and continue charging if needed. From the middle plot we can also observe how the actual current draw on the system changes as EVs arrive, charge, and depart. The bottom plot shows how the SoC of the EVs plugged into each EVSE changes over the course of the test. Note that this test includes multiple EVs per EVSE. As such, when an SoC line drops to 0, this means that the EV previously plugged in has unplugged. When the line begins climbing again it indicates that a new EV has arrived and is charging.

plug in of another Tesla Model S 30 minutes into the test which stays for for 2 hours.

The results of a second, more complex test are shown in Fig. 6. This test is designed to show the power of the simulator which can handle simulating an arbitrary number of EVSE and events. In order to make the results of the test more user friendly, we have limited this test to 4 EVSEs and a total of 12 events. However, the simulator can handle far more of each. In addition, for this test we have set the rate increment for each EVSE to 8 A versus 1 A in the previous test. This causes the pilot signals to be much more choppy. Selected metrics from this test are shown in Fig. 6.

6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

This paper describes the design of a simulator for the Adaptive Charging Network in order to facilitate the transfer of theoretical advances to practical code which can be run on the physical ACN. Realistic models for the system architecture, EVSEs, EVs, and batteries allow our simulator to accurately evaluate how software will perform in the physical system. The simulator framework has been designed to allow tests to be designed easily using a simple interface while still giving the test designer the flexibility to test many scenarios and edge cases. In addition the use of time acceleration allows tests to be run in a matter of seconds instead of hours, which increases the number of test that can realistically be performed. Results show that the simulator accurately mimics the behavior of the real system.

All in all, the ACN simulator will allow for faster and more robust translation of theoretical results into practical software. This system will be used to evaluate new algorithms and ACN features before testing these changes on the physical system. This will significantly reduce the time required to test new software. In addition, since we are able to test edge cases which do not occur frequently in the physical system, software tested with the simulator will be more robust.

6.2 Future Work

Future work on this project will include setting up an instance of Influx, the time series database used to store measurements

in the ACN, for storing measurements taken in simulations. This will allow us to utilize the same reporting infrastructure used for monitoring the real system to monitor and evaluate tests. We also plan to model communication delays and dropped packets so that we can test how algorithms and other parts of the software handle network congestion. Finally, we plan to add additional tests as new algorithms are designed for the ACN. These simulations will test stability as well as evaluate key metrics specific to the algorithm being tested.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Steven Low for his mentorship on this project as well as his help in formulating the charging model for EV batteries. We would also like to thank the ANC team whose input and infrastructure contributions have helped make this project a success.

REFERENCES

[1] G. Lee, T. Lee, Z. Low, S. H. Low, and C. Ortega. 2016. Adaptive Charging Network for Electric Vehicles. In *2016 IEEE Global Conference on Signal and Information Processing*.