Pseudo-Code and Data Appendices for Paper:

A Technique to Interconnect and Control Co-Simulation Systems

Stephen Broderick *, Andrew Cruden ¹, Suleiman Sharkh ¹, Nigel Bessant ²

¹ Energy Technology Research Group, Faculty of Engineering and the Environment, University of Southampton, Southampton, England
² Scottish & Southern Energy Power Distribution, Bournemouth, England
*corresponding author: srb3g13@soton.ac.uk

February 2017

Table of Contents

Appendix A: Sample Pseudo-Code...............................................................2
Appendix B: A Log of a Completed Schedule................................................6
Appendix C: How RunTS invokes Simulation Tasks.........................................12
Appendix A: Sample Pseudo-Code

Core concepts only are shown; pragmatic aspects (e.g. error handling) are omitted.

RunTS System Schedule

The key capabilities of RunTS code are: AddNewTask, ExecuteSchedule. These require a Schedule, which consists of nested data objects:

```plaintext
{ key: at_time =>
    { key: at_stage =>
        (set of [Task, data_packet] objects)
    }
}
```

e.g.

```plaintext
{ 12:
    { 100: ( [Task_2, data_packet], [Task_16, data..], [Task_30, data..] )
    120: ( [Task_8, data_packet], [Task_26, data..], [Task_14, data..] )
}
16:
    { 100: ( [Task_1, data_packet] )
    500: ( [Task_109, data_packet] )
}
}
```

AddNewTask

DataIn: Task, task_type, schedule_time, to_task_data_packet,
DataOut: Success / Fail

LookUp tables: task_type => stage_number

**Table A.1** task_type to stage_number table

<table>
<thead>
<tr>
<th>Task Type</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>data setup</td>
<td>100</td>
</tr>
<tr>
<td>network changes</td>
<td>200</td>
</tr>
<tr>
<td>EV assessments</td>
<td>500</td>
</tr>
<tr>
<td>network solves</td>
<td>1000</td>
</tr>
<tr>
<td>data logging and reporting</td>
<td>10000</td>
</tr>
</tbody>
</table>

begin:

```plaintext
if RunTScycle does not exist at this stage_time:
    create new RunTScycle at this time
    sort RunTS Schedule

find stage_number from task_type table

if RunTScycle does not contain this stage_number:
    insert stage_number
    sort stage_number list
```
insert into this stage_number's tasks-cache: Task, task_data_packet

**ExecuteSchedule**
The coded implementation uses binary-chop and hashed table lookups for speed.

DataIn:   Starting_time, ending_time
DataOut:  Success / Fail

begin:

current_time = starting_time - fractional time i.e. \(10^{\text{-20}}\)

if (count of events remaining > 0) and (current time < simulation_end_time):

    current time = time of next RunTS Schedule event

    for all stage entries:
        find lowest numbered unprocessed stage
        for each task within the stage:
            get a task_response:
                call the task, attaching the current time and the task data_packet
                if task_response includes a new task to schedule:
                    AddNewTask
            else:
                return Schedule terminated with Outcome code

Tasks.
It is assumed memory is private to each EV (Tasks see data set by other Tasks for the same EV only).

EV Tasks: ArriveAt, DetermineChargingRate, Depart

**ArriveAt**
Data In: EV_ID, present time, location_name
DataOut: EVSE_ID, EVSE_load, charge_rate, end_of_charging_at, present_SOC

begin:

for EV_ID at present_time:
    find free EVSE at this location
    connect to EVSE
    determine present battery SOC from travel-time
    DetermineChargingRate()

return EVSE_ID, EVSE_load, charge_rate, end_of_charging_at, present_SOC

**DetermineChargingRate**
Data In:   EV_ID, present time
DataOut:  new_charge_rate, new_socket_load, end_of_charging_at, present_SOC

FarOffFuture = max_reals - \(10^{\text{-20}}\)      // likely thousands of years in the future
Table A.2 SOC to charge_rate

<table>
<thead>
<tr>
<th>SOC</th>
<th>Charge Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 94%</td>
<td>standard charge</td>
</tr>
<tr>
<td>94 ~ 97%</td>
<td>slow charge</td>
</tr>
<tr>
<td>&gt; 97%</td>
<td>trickle charge</td>
</tr>
</tbody>
</table>

begin:
for EV_ID at present_time:

    SOC = last_known EV SOC + charge_rate . (present_time - last_assessed_time)
find charge_rate from SOC to Charge Rate table

    if not performing trickle charge:
        end_of_charging_period = present_time + time to SOC of next charging band
    else:
        end_of_charging_period = FarOffFuture

    system_losses = fn (battery losses, inverter losses, EV ICT load, battery heating / cooling)

    socket_load = (system_losses + charge_rate) / socket_volts
return socket_load, SOC,
    NewTask(DetermineChargingRate, EV: EV_ID, end_of_charging_period)

Depart
Data In:   EV_ID, present time
DataOut: charge_rate, socket_load, present_SOC

EV_SOCmin = 20%
begin:
for EV_ID at present_time:

    SOC = last_known SOC + charge_rate . (present_time - last_assessed_time)

    if SOC > EV_SOCmin:
        disconnect from EVSE
        charging_rate = 0
        socket_load = 0
    else:
        throw error “EV cannot depart, insufficient SOC”

return charge_rate, socket_load, SOC

Network Solver
Key tasks are: DefineNetwork, AmendLoadAt, SolveNetwork

DefineNetwork
DataIn:   network_definition_files
DataOut:  Success / Fail
begin:
    connect to NetworkSolver // OpenDSS via COM interface
    command NetworkSolver: clear network memory
    command NetworkSolver: read and process network_definition_files
    return Success / Fail status

**AmendLoadAt**

DataIn: network_location, new_load // negative load means generation
DataOut: Success / Fail

begin:
    connect to NetworkSolver
    command NetworkSolver: set new_load at network_location

    return Success / Fail status

**SolveNetwork**

DataIn: <none>
DataOut: Success / Fail, new_network_states // electrical status of each network point

begin:
    connect to NetworkSolver // electrical status of each network point
    command NetworkSolver: perform static load-flow solve, update new_network_states with results

    if successful:
        return Success and new_network_states
    else:
        return Fail and any error codes
Appendix B: A Log of a Completed Schedule

1. Each task on being entered to the schedule is given an incrementing task_number. The first task is ‘SIM_STOP’ placed at the far future.

2. 3 EVs are used: ‘EV_M1’, ‘EV_M2’ (twins) and ‘EV_L1’ (which has a larger battery).

3. The scenario (EV journeys) is then added, plus a power loss effecting EV_L1.

4. The schedule is retained post-run to evidence operation.

5. At st 0.01 all EVs are instantiated (brought into existence) / arrive at their respective home locations with the same battery SOC and begin to charge. The EV model discovers the need to cease charging and these are created as auto-generated tasks at st 102.043.

Observe that RunTS has merged multiple tasks into the same Cycle. This happens in several places.

The Post-Run Schedule:

'active_st_Do': {
    # st: {stage: {task_number: target, task_number: target ... } }

    124.83: {900: {24: 'EV_L1'}},
    192.043: {900: {25: 'EV_M1', 26: 'EV_M2'}},
    214.83: {900: {27: 'EV_L1'}},
    216.0: {400: {9: 'EV_M2'}},
    225.0: {400: {8: 'EV_M1'}},
    246.0: {400: {12: 'EV_M2'}},
    252.0: {400: {10: 'EV_L1', 11: 'EV_M1'}},
The Tasks:

'all_tasks_Do': {
    1: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02547940>,
        'activity_task': 'SIM_STOP',
        'stage': 1,
2: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02547968>,
   'activity_task': 'ArriveAt',
   'stage': 400,
3: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x025478C8>,
   'activity_task': 'Start_DataDump',
   'stage': 100,
4: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02547990>,
   'activity_task': 'ArriveAt',
   'stage': 400,
5: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x025479B8>,
   'activity_task': 'Start_DataDump',
   'stage': 100,
6: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x025479E0>,
   'activity_task': 'ArriveAt',
   'stage': 400,
7: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02547A80>,
   'activity_task': 'Start_DataDump',
   'stage': 100,
8: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02547AF8>,
   'activity_task': 'Depart',
   'stage': 400,
9: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02547CD8>,
   'activity_task': 'Depart',
   'stage': 400,
10: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02547DF0>,
    'activity_task': 'Depart',
    'stage': 400,
11: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02547FD0>,
    'activity_task': 'ArriveAt',
    'stage': 400,
...
22: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02556170>,
   'activity_task': 'Charging_Update',
   'stage': 900,
23: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02556300>,
   'activity_task': 'Charging_Update',
   'stage': 900,
24: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02556468>,
   'activity_task': 'Charging_Update',
   'stage': 900,
25: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02556620>,
   'activity_task': 'Charging_Update',
   'stage': 900,
26: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02556698>,
   'activity_task': 'Charging_Update',
   'stage': 900,
27: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02556800>,
   'activity_task': 'Charging_Update',
   'stage': 900,
28: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02556A80>,
   'activity_task': 'Charging_Update',
   'stage': 900,
29: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02556A08>,
   'activity_task': 'Charging_Update',
   'stage': 900,
30: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02556D00>,
   'activity_task': 'Charging_Update',
   'stage': 900,
31: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02556F08>,
   'activity_task': 'Charging_Update',
   'stage': 900,
32: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02556F80>,
    'activity_task': 'Charging_Update',
    'stage': 900,
33: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x025621C0>,
    'activity_task': 'Charging_Update',
    'stage': 900,
34: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x025623F0>,
    'activity_task': 'Charging_Update',
    'stage': 900,
35: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02562468>,
    'activity_task': 'Charging_Update',
    'stage': 900,
36: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02562878>,
    'activity_task': 'Charging_Update',
    'stage': 900,
37: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02562940>,
    'activity_task': 'Charging_Update',
    'stage': 900,
38: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02562A30>,
    'activity_task': 'Charging_Update',
    'stage': 900,
39: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02562B48>,
    'activity_task': 'Charging_Update',
    'stage': 900,
40: {'RTdata': <RunTimeClasses.TaskRT_Class instance at 0x02562C10>,
    'activity_task': 'Charging_Update',
    'stage': 900,
Appendix C: How RunTS invokes Simulation Tasks

This section is in two parts:

- General Capabilities
- Method of Operation / execution.

General Capabilities - the ways in which RunTS might invoke a simulator

1. via python modules. The programmer constructs a simulator in python which receives data packets and / or pointers to large data structures. The simulator is included into INSim as a module;

2. via a Windows COM API which allows direct connection between two programs. This is the mechanism used by OpenDSS. The COM protocol defined by OpenDSS is the same as that used by the console / HMI i.e. are the same commands. As python has a COM interface connector then the programs can dialogue. Data (simulation results) can also be returned via the COM interface;

3. .csv files i.e. data as text can be written to or read from a file by python; the partner simulator can read it and respond;

4. via a sockets connection. python has a capable sockets module, permitting remote interworking with other machines. This is normally used for internet and web services; however there is nothing to prohibit sockets being used with a simulator as a partner program. This allows dialogue across local networks, access to machines across the internet or connection to supercomputers.

These would be invoked via a simulator interface or connection of some type, written in python.

Method of Operation - How RunTS Invokes a Simulation Task

Running a Task requires initial set-up. Setup consists of adding the simulator module address to a database, keyed by Actor Category and Target_ID. For example, an EV may be a member of the “V2G” category and have an ID for a unique vehicle; a simulation of an Aggregator would use a different Category etc.

Also needed is a pointer or object address for the interface. If the item is a python object then the address is the object name. These are determined when the Target is created. The database of { [Category, Target_ID] => object address / interface module } is part of the INSim “Task Manager”.

Example: Make 10 new EVs

```python
for new_EV_ID in ['EV_001', .. 'EV_010']:
    # now create an object with the Target_ID 'EV_001' etc.
    EV_object = EV_Class('V2G, new_EV')
    # now register the Category, Target_ID and object address
    Task_Manager.Register('V2G_EV', new_EV_ID, EV_object)
```

Executing tasks is a straightforward process:

- RunTS has determined a Task is to be performed - this includes the Target (Category, Target_ID pair) to which the task is applied plus any data packet. These are passed to the Task Manager;
- the Task Manager uses the Category and Target_ID to determine the correct interface module - which is now invoked, passing to it:
  - the current simulation time “st_now”
  - the Task the simulator code is to perform, and
  - the data packet
- the simulator (or interface) processes this data i.e performs the designated Task. There may be outputs from this (success (no further action), success (new Task needed), an error code)

To deal with these eventualities the following is returned by the task to the Task Manager:

```
(outcome code, data packet format code, [data packet])
```

The Task Manager will take any appropriate action e.g. request the creation of a future Task (an addition to the Schedule). Given that the system has not experienced a failure and that all outstanding activities are complete, the Task Manager returns execution back to RunTS so it may continue with the next activity.

Example: (these activities occur within the Task_Manager)

```python
DataIn:  Category, Target_ID, st_now, Task, datapacket

begin:
    try:  target_object = all_objects[Category, Target_ID]
    except: raise error: Unknown object; quit
    response = Invoke(target_object, Category, Target_ID, st_now, Task, datapacket)
```
if error in response: throw error
if new_task in response: Create new task(response data )
<finished this task / hand back to RunTS>