A Technique to Interconnect and Control Co-Simulation Systems

Stephen Broderick [1*], 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 Networks, Poole, England

[*] corresponding author: srb3g13@soton.ac.uk

Abstract: A "Run Time Scheduler" (RunTS) is presented, a minimalist Discrete Event Simulator able to sequence the operation of multiple simulators, coordinating a simulation of several elements so to form a whole system. Dynamic cause-effect chains may emerge.

In operation the net simulation possesses a known steady-state other than when changed. RunTS controls the application of state-changes, passing these to specific simulators to determine new steady-states. Messages can pass between simulations concerning configuration data, commands and results. Simulators must meet connectability criteria.

The method allows variable periods between simulation stages rather than a fixed tempo, and can undertake specific simulation sequences unique to each event, through use of a novel per-event state-machine.

RunTS is part of INSim, a Smart Grid simulator working with a mix of components including Electric Vehicles, Smart Agents and OpenDSS, a power network analysis tool.

1. Introduction

The Run Time Scheduler (RunTS) is a form of Discrete Event Simulator (DES) [1], part of the Improved Network Simulator (INSim) [2] in development, which intends to model UK power systems with Smart Grid connected EVs. RunTS is a minimalist executive devised to schedule, communicate with and control external simulation components. INSim and RunTS are written in python, an established interpreted language.

INSim is intended to assess the UK's LV infrastructure by combining simulations of:

- Electric Vehicles (modelled directly)
- power-flow (via an external simulator, OpenDSS)
- Smart Grid message passing (modelled directly).

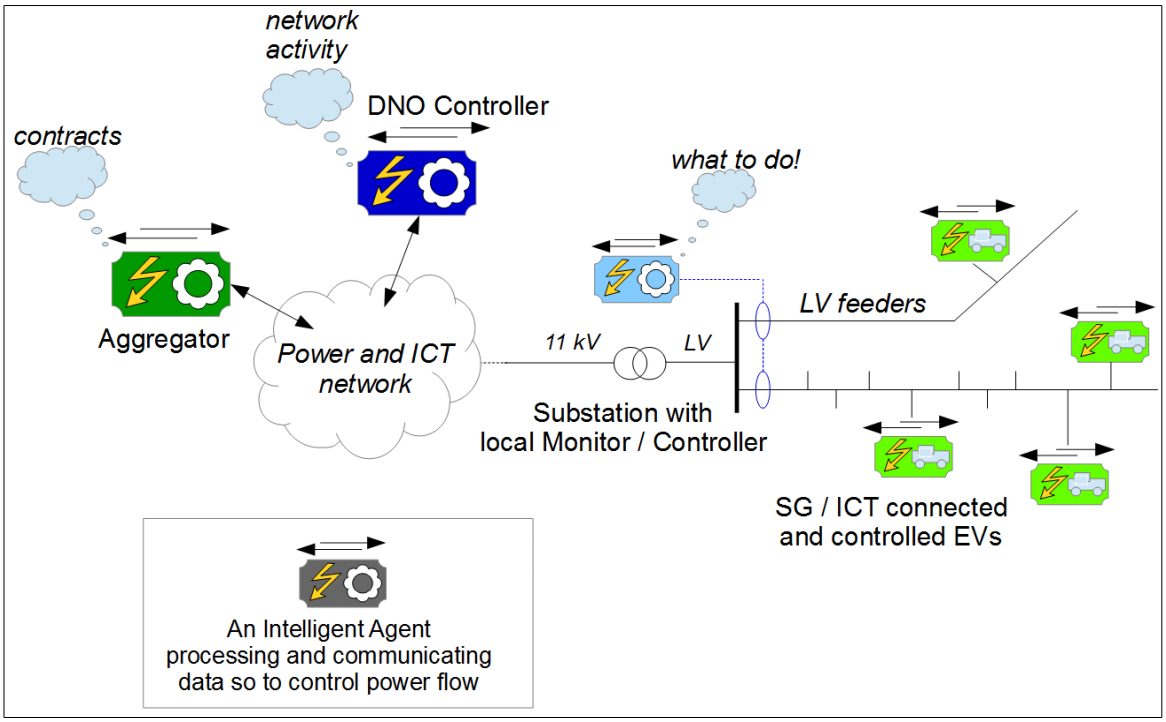


Figure 1: INSim objective: Investigation of EV charging and V2G impacts on LV networks

OpenDSS was selected as it is a mature, fast and capable power systems simulator proven in use. Developed by EPRI, OpenDSS has many features of interest (such as sophisticated transformer models and ability to model Geomagnetically Induced Currents).

A challenge for INSim was to work not only with OpenDSS but other 3rd party systems being developed at Southampton. The approach adopted is general and may use any python feature including sockets, which permit local and remote interworking with other systems. OpenDSS uses a COM interface (a Windows PC feature which allows compatible programs to communicate).

Why is INSim being developed? A recent review of available vehicle, power network and SG / messaging simulators by Mahmud [3] finds:

"Typical applications of the tools include vehicle system analysis and control, renewable energy and vehicle to-grid integration and impact analysis, energy market behavior and charge scheduling, vehicle energy management, and traffic system simulation. No single tool covers all areas of these applications, however sufficient information is provided to enable researchers to select the most appropriate combination of tools to meet specific research objectives."

Mahmud's review covers 125 simulators, described and presented as comparison tables. However searches by both Mahmud and the author did not find a tool suitable for interworking different simulators. Different simulators for renewable energy forecasting, EV traffic and journey modelling (and many others) are available. These address specific modelling needs including power systems and smart grid. Yet although individually useful, there seems no direct means to combine them for they have been written independently and know not of each other.

1.1 A Method to Interconnect Simulators

Any multi-component approach needs a mechanism for interconnection and co-ordination, with co-ordination being the primary goal of RunTS. RunTS is a variation on a DES executive, a method which has been successfully used in power simulation systems [4] and systems working with Agents [5]. By interworking several simulators progressing specific models, RunTS enables co-simulation.

Remote simulators are treated as “Black Boxes”. Optimisations (such as use of parallel-processing) are left to the remote simulator. There may be performance consequences of this; however when connecting arbitrary simulators the priorities are:

- to enable data communication
- to enforce synchronisation
- to permit causality to develop i.e. chains of cause-effect sequences.

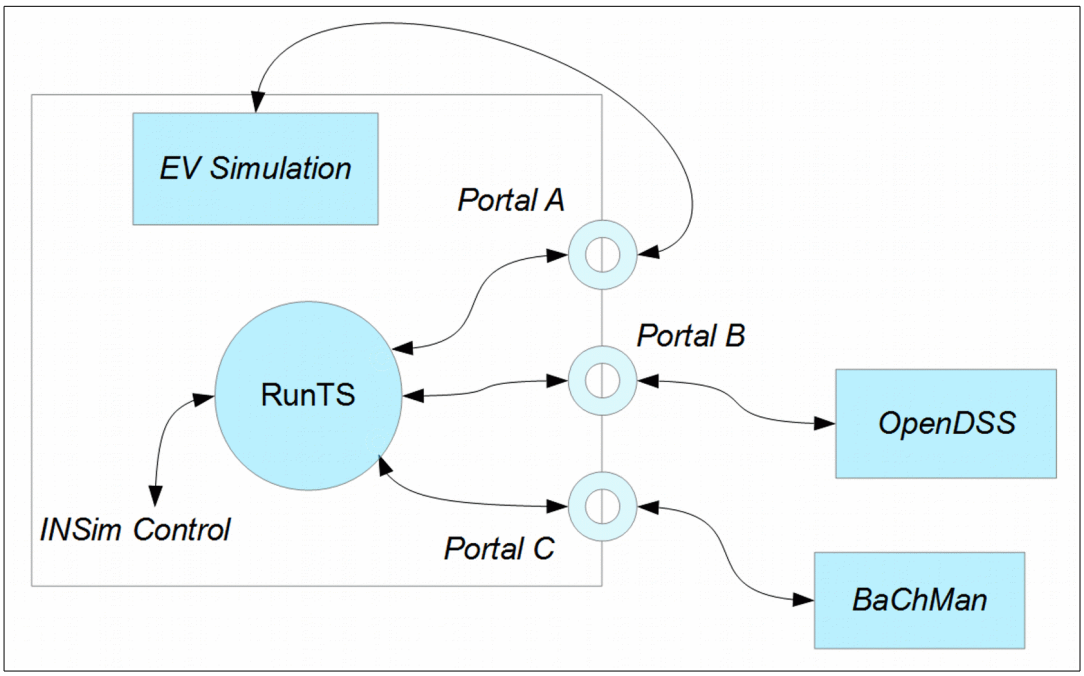


Figure 2: Simplified overview of INSim and RunTS communicating with external simulators

Figure 2 shows the RunTS component of INSim communicating with simulators via logical portals. The portal code performs any necessary connection protocol and data conversions (e.g. of time format). Portals need to be custom written for each connected simulator. BaChMan (part of [6]) is an existing University of Southampton V2G simulator operating at 11 kV, commanding EVs as a dispersed ESS (NB link to BaChMan not yet implemented). In operation, INSim sends a starting schedule to RunTS which dialogues with the simulators.

RunTS enforces synchronisation by commanding state-changing events at specific simulation times ("st"). For example, the classes of events impacting a charging EV include:

- the EV internal battery controller
- the actions of the driver
- the condition of the surrounding power network
- signals and information concerning the power network (i.e. SG inputs)
- which may occur in the schedule or simulator outputs. Simulators may effect each other.

RunTS has a dynamic schedule i.e. scheduled tasks are not fixed; the schedule changes given changing inputs. This allows incorporation of new events from simulations, which in turn can effect other simulators. For example the EV model is affected by charge /

dispatch commands. The EV simulator (part of INSim) determines a resulting socket load and hence the network condition. An example of this is detailed in Section 5 (see also Figure 9).

Each timed event in the RunTS schedule points to a work-package, containing a task to be performed. Data for the task is included in the work-package, together with the subject / target of the simulation. Each state transition may need specific programs to perform the task; this is performed by the called simulator.

The simplest work element is a single state-update at a timed moment:

event at time t_x*:* $S \Rightarrow S'$

However it is possible that states for different simulators are timed for simultaneous update. This is performed by scheduling *M* multiple tasks for the same moment, updating as many states as are needed (the last example in Figure 3, the state of which is driven by results from three simulation tasks):

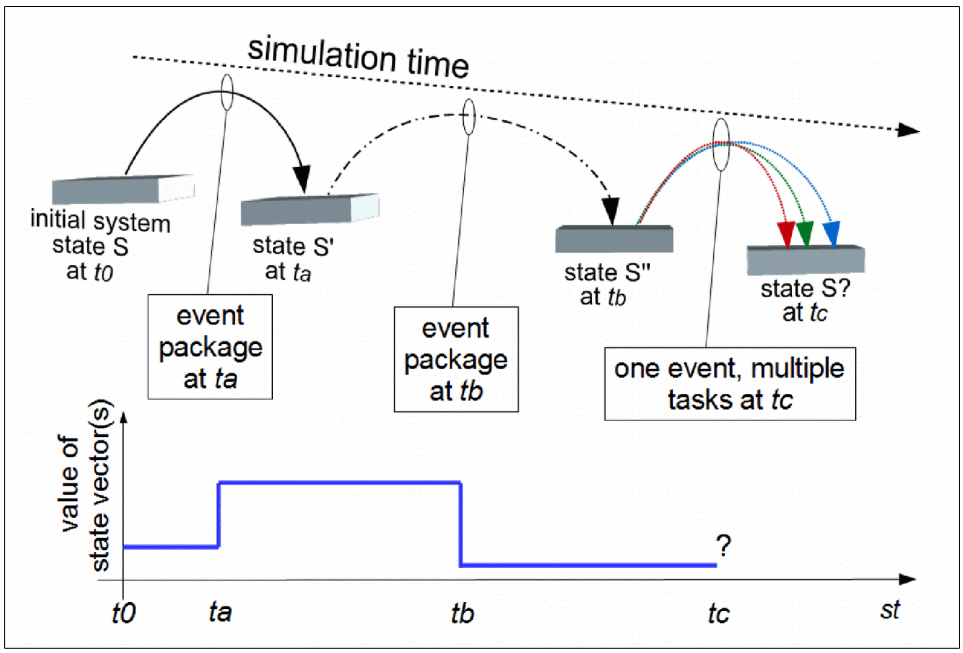


Figure 3: Events occurring at known simulation times change System States

at time t_c*:* $S_1 \Rightarrow S_1'$, $S_2 \Rightarrow S_2'$ ... $S_i \Rightarrow S_i'$

A novel contribution of RunTS is a mechanism to control specific interplays of the *M* multiple tasks occurring at the same instant (described in Section 3).

1.2 Constraints on Simulators which may work with RunTS

Finally, the intended application of RunTS is to interconnect simulators which are:

- time linear
- able to simulate state-changes
- can communicate across an interface
- perform mutiple sets of "single-shot" simulations, retaining their own system state(s) between operations.

Note that RunTS is not intended to be a real-time scheduler.

2. Process Optimisation Rationale

Effort has gone into making RunTS:

- compact - under 500 lines of executable code
- minimalist - basic feature set to suit the interworking problem
- fast - extensive use of 'bisect', a python module performing binary-chop in C.

RunTS is a single-threaded computing process, in contrast to many recent DES which use multiple threads or parallel processing (e.g. PyPDEVS). RunTS does not attempt to organise computing resources; rather, the reverse approach is taken.

The primary options for optimisation are multi-threading and multi-processing. Multiple threads may be considered as extra CPU time-slots made available by the computer operating system, performing sub-tasks for the thread owner. Multiple processes are each high-level applications often with an allocated hardware resource (a processor core, memory, disk-space). Typically, threads can see the parent's resources (e.g. data). Accessing resources between processes may be less straightforward or wasteful (duplication of memory areas).

Figure 4 is a computer with 4 resources. Ideal utilisation uses all resources, at all times.

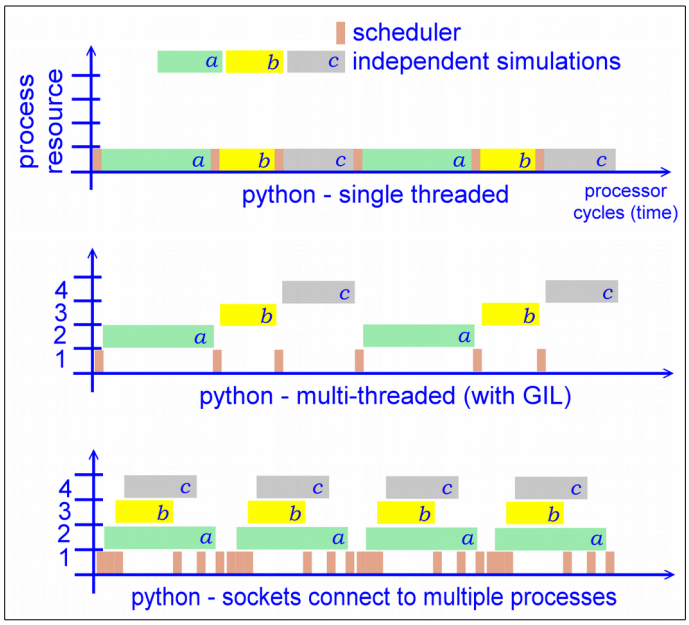


Figure 4: Process resource usage options for python

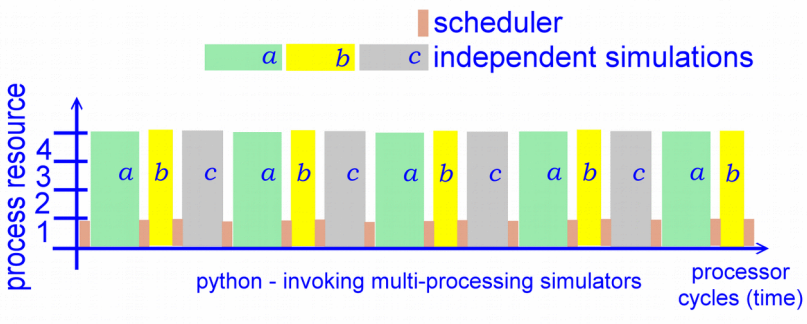
The top option (single-threaded execution) is clearly wasteful; only one resource is used. The second (multi-threaded) system is no better, as python's Global Interpreter Lock (GIL, a defensive measure to protect python's internal states) suspends existing threads when starting new threads.

The third option is more promising. Here, the socket capabilities of python connect to simulation processes running on other CPU resources (cores). The system now possesses a much better resource utilisation - but is still not ideal; slow processes (green) cause inactivity in completed tasks (yellow and grey). However this is an attractive concept for distributed operation. Python's socket system implicitly uses a network TCP/IP stack, meaning that the called simulation can be in any resolvable space - the same machine or a remote machine accessed via a network.

What of splitting RunTS across multiple processors rather than threading? TimeWarp is a popular parallel-processing method used for this and can speed simulation. TimeWarp allows speculative forward simulation, some of which will be wrong (so need to be detected and undone). As RunTS treats external simulations as Black Boxes, RunTS cannot inspect internal states to determine the correctness of speculative calculations.

A method is available to combat this lack of visibility: "Memorization". This involves taking system-state snapshots (implicitly, for all simulations running) and to rollback these to an earlier snapshot, on advice that a simulation is in error. Further, the increased memory footprint may result in contention for memory and potentially swapping (each process needs a separate memory area and likely memory synchronisation, another overhead) so adding complexity and slowing operation. Such systems (e.g. PyPDEVS (described in section 7.2) and can offer advantages.

A minimalist approach is adopted, shown in Figure 5. Here RunTS suspends between calls to external simulators. The autonomous 3rd party simulators can then organise available processing resources as they see fit, acquiring and releasing resources as needed:



Please note that the beige RunTS scheduler activity is shown exaggerated for clarity; in practice each scheduler call is likely c. 10 - 50 micro-seconds. Simulation tasks are likely much longer.

Figure 5: Idealised resource usage with optimising simulators

Such a scheme can only bring benefit when using suitable optimising simulators. Non-optimising simulators will work, however the advantages in Figure 5 will not be gained. Note that the choice of a minimalist solution does not bar the use of parallelised simulators (see also section 3.3).

Given the above, what load does RunTS impose on the computing resource? The performance of RunTS is respectable at:

- insert into schedule (mid-list for worst-case): average < 20 micro-seconds
- advance, find, read and execute a scheduled (null) task: average < 25 micro-seconds

These times were measured using python's "timeit" capability performing 1,000 calls on a 3.4 GHz i7 PC. A variability of c. x3 was seen. Python version 2.7.13 was used with the standard python interpreter. Please compare these with the seconds, minutes or multiple hours needed for complex simulation tasks. These times are quick, which the author attributes to:

- use of hash-tables and 'bisect' in RunTS (a C module performing binary chop)
- automatic processor optimisation, due to repeated execution of code snippets.

The operations RunTS performed for each timed cycle included:

- advance the scheduler to the next known event
- identify the (next) stage to perform
- identify the task required
- access the data-packet to forward to that task
- invoke a discriminator (the "TaskCaller", described in Section 6) which has knowledge of which portal (as in Fig. 2) receives which task
- the discriminator discovers there is no known portal for the test, performs error-handling and returns the dummy task to the scheduler.

Thus RunTS uses DES as:

- DES are compact, simple to code and execute quickly
- DES simulations are intrinsically rapid as periods with no state-changes are skipped
- DES guarantee time and sequence fidelity (significant for cascaded / consequential events)
- the small footprint frees other resources (memory, CPU cores) for use by other simulation engines, which may optimise their own processing as they see fit.

To be used in this manner, the invoked simulators must be able to:

- perform a single-shot calculation to determine their instantaneous state(s), being progressions from an earlier state given inputs and the passage of time
- communicate via a computer-accessible portal (e.g. a COM interface on Windows or a socket interface on the same machine / across a network)
- receive and report state data

- receive and execute commands and return completion states (success, fail etc.)
- and (ideally) include their own process optimisation system.

3. Overview of MASCOT ACP Concept

Elements of RunTS's approach are inspired by MASCOT, a method developed by the UK's Royal Signals and Radar Establishment. The "Modular Approach to Software Construction Operation and Test" (MASCOT) [7], [8]) is a real-time system design methodology. The problem MASCOT addressed was to interconnect discrete real-time systems (hardware and software) - similar to interconnecting and interworking simulators.

MASCOT was developed by electrical and electronic engineers. The diagramming method used is based on data flows and has similarities to circuit diagrams.

The method uses "ACP" components and diagrams, which are:

- Activities (performing or invoking processing)
- Channels (of data via queues or communication layers) and
- Pools (a data repository, perhaps memory or a database).

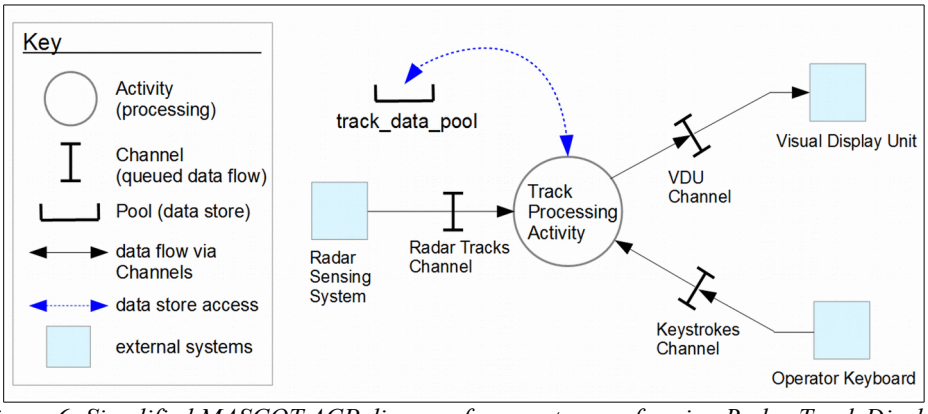


Figure 6: Simplified MASCOT ACP diagram for a system performing Radar Track Display

Figure 6 is a simple ACP diagram for a radar track display, updated once a second. Each component is a discrete entity. Activities may contain other MASCOT systems; stacking allows detail (process function) to be added as needed, whilst keeping the system simple at each abstraction level. Note the Activity connects to external systems via Channels.

The operation of the shown Activity is:

- wait until the start of the next second
- read the Radar Tracks Channel
- process track data and update track_data_pool (contact height, speed, heading)
- read and process the tracks_data_pool data into an image
- send the image to the Display via the VDU Channel
- read keyboard user-inputs ('change display scale'), process as necessary and
- loop to the start (wait for the next second).

The points of interest are:

- the Activity waits on a key event occurring (a new second)
- the Activity actions a sequence of tasks in a set order
- specialist operations are performed by passing tasks and messages to external subsystems. Note the methods used to perform tasks do not concern the Activity
- on each round of operation, the Activity performs a complete set of work.

3.1 Incorporation of Activity concepts

Some concepts are used in RunTS, which operates in the following manner:

- the simulation is defined as being in a known steady-state, other when
- a specific event disturbs the system state. Events are held in
- a schedule of arbitrarily timed events, each of which
- executes a sequence of tasks in an order. Each task is invoked
- by passing messages to remote system components
- resulting in state changes which move the system into a new steady state (this includes known rates of change).

For example, how RunTS might model boiling a kettle of water:

a) the water and kettle form a known system (mass, temperature, heater wattage)

b) the kettle is turned on. This is deemed to be steady-state, as the rates of change are known. A time of boiling is calculated and a new timed task created: “Check-If-Boiled”. This is inserted into RunTS schedule

c) at the due time RunTS calls the Check-If-Boiled task; if boiled the task changes system state (turns the kettle off), otherwise a future time is calculated to repeat the check.

On invoking each task a data-packet (associated in the schedule with the task-call) is passed to the task. On task completion a data packet is returned (this may be a message or a request to create a new task, such as “Check-If-Boiled”).

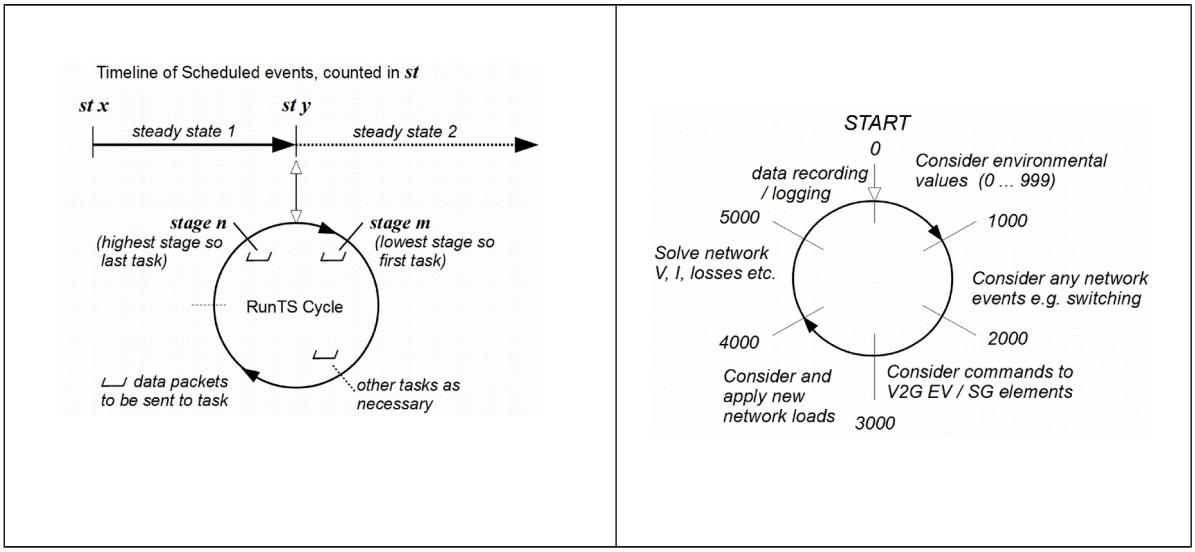
Note that a MASCOT Activity performs a fixed set of functions (e.g. radar image processing) but RunTS invokes tasks which are soft-set (programmed) via data-structures. These tasks are **not** fixed, so can be dynamically updated and may include multiple arbitrary items. This raises the issue of task sequence.

Simultaneous discrete tasks need be logically independent, however their sequence is relevant as tasks which set or condition data must precede those which use that data. For example, an EV may have a long-standing task to start charging - yet a co-incident power-fail may occur. If the EV state-machine begins charging before the power-fail is executed an erroneous state is generated; the power-fail must be processed first.

3.2 Stage Numbering

To control sequence a “stage number" is used. Tasks are executed in low to high stage sequence (the stage value is an arbitrary integer between -2,147,483,647 and 2,147,483,647). Numeric ranges can be used for similar functions, e.g. low stage numbers are for data-setup, high for data recording and output. Note that stage numbers do not need to be unique; tasks may have the same stage number, but will be executed in random order.

The RunTS component is termed a “Cycle” as against "Activity". Like the Activity, the Cycle performs a complete process to transition the system from one steady-state to another. Between Cycles system states do not change. Cycles can be regarded as state-machines.



(a) *(b)*

Figure 7: RunTS Cycle, indicating

*a) at time **st y**, a Cycle calls tasks which modify the system state*

b) use of stage numbers to partition a Cycle into functional bands.

Table 1 A set of Stage Ranges for use with a particular application

Stage Range	**Intended Use**
0...999	determine environmental factors (local temperatures, insolation, wind)
1000...1999	assert network changes (switching, protection etc.)
2000...2999	perform ESS control assessments (Aggregator and EV calculations etc.)
3000...3999	assert other timed load changes (load profiles etc.) at given network points
4000...4999	solve the network
5000...	log results

Why are consistent stage ranges useful? RunTS continually merges same-time Cycles - as real life events can be co-incident. For the example of power-loss coincident with an EV going on charge, the power-loss would have a low stage number. Placing network events in an early stage allows the EV simulation to discover that it has no power.

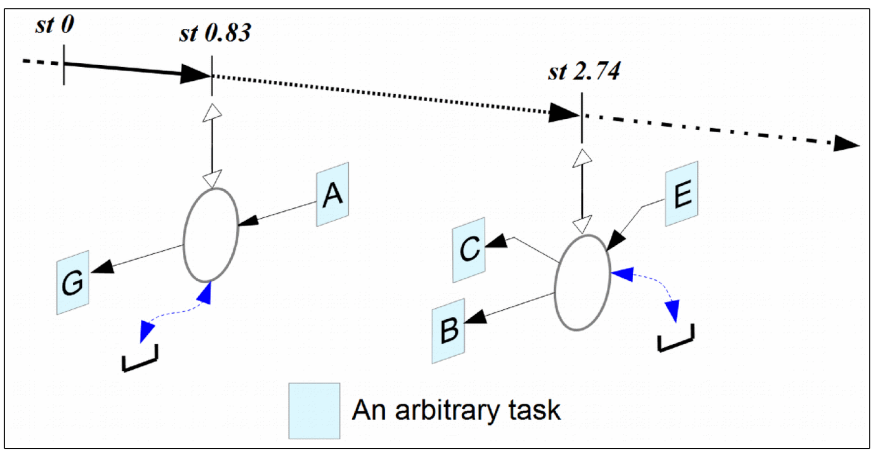


Figure 8: Sample timeline of RunTS execution showing two Cycles. Each Cycle can be unique

A Cycle is a complete logical machine, calling tasks as needed. Summarised, RunTS:

- has no fixed time step, rather a list of sequential events at timed moments
- there are no default tasks within a Cycle - there may be one or thousands,
- each task has a stage number to control execution sequence
- messages are “passed in” on task calls and “passed back” on task completion
- the Cycle is a complete operation moving the system from one known steady-state to another
- simulation time (st) is held as a floating-point number; as a result RunTS can consider events over an extreme dynamic range - close in time or far apart.

An EV may now:

- arrive at a charging station,
- determine a charge rate (thus network load) and to discover
- a “completed charging time”.

This data is is returned to RunTS with a request to add a new timed task to check the EV's State of Charge (SOC) and perhaps ceases charging. As there are no intermediate states the EV remains on charge for the whole period and does not need further assessment during that time (yet power fails will cause updates).

This method permits the simulation of events hours or milliseconds apart e.g. a load applied which causes a cascade of trip events. Such a sequence can be concurrent with EV charging; appropriate use of stages controls timing conflicts.

These provide a key capability:

- dynamically emerging causal chains
- given that the user can construct tasks which are aware of consequences,
    - tasks can create future tasks for entry to the schedule.

In practice, tasks interface to 3rd party software systems (simulators). Calling that task allows RunTS to assess an aspect (state) of interest. A possible simulator is OpenDSS [9], a network load-flow solver. OpenDSS has featured in several similar simulators [10, 11].

Internally RunTS represents simulation time in units of "st", a floating-point number. There is a system-wide ratio to simulation time to st. Specific times are expressed as "st N.N" e.g. st 390.0; time starts at st 0.0 being 00:00 hours of the first day simulated.

The system does impose constraints on the tasks (external simulations) allowable. These tasks must be capable of:

- communicating though a defined and RunTS accessible interface
- possess memory of their prior states
- can update their own condition (state) correctly to mimic a point in time
- perform a selected role or task on demand, updating their states as necessary and for states which have finite duration - the ability to
- project an end-time (anticipated duration) of their current state.

There is no constraint on the time taken by external simulations; RunTS waits for completion. Nothing requires a task to be on a specific computer, or forbids simulation concurrency (although linear simulation time must be maintained lest problems arise [12]).

3.3 Using Stages Numbering to Support Parallelism

Stage numbers can aid support parallel operation. An example: to parallelise the calculation of network impactors, such as PV output (given season and time-of-day), scheduled Aggregation contract commands, determine dynamic SG commands etc.

The code for calling each such task would:

- start the parallel task X and immediately return:
- a request for a new task "check task X complete / returned" prior to Table 1's "assert network changes" in (i.e. "check task X" has stage 999).

As a result the initial schedule, as set by the user, does not change. Sockets may be used for this, as discussed in Section 2.

4. RunTS Implementation

4.1 Python Implementation

The RunTS concept was implemented as a python 2.7 module of the INSim simulation system (described in [2]). Python is a capable multi-platform interpreted language in wide use. With comments, various necessary exception handling (e.g. detection of attempts to create an event in the past), debugging / error reporting, initialisation routines and a test harness, RunTS system occupies c. 900 lines of code of which c 500 are executable.

System st fidelity and range capabilities are set by the nominal floating-point capabilities of python, which for the system used (a Windows PC) has 14 places of accuracy (usually rounded up to 3 significant places). A ratio ("st per hour") sets scale between st size and simulation time. One st per second gives a maximum simulation range of c. 3,170 years with a 1 millisecond resolution. This is suitable for systems simulating circuit-breakers. INSim uses 30 st per hour and has a resolution of 120 milliseconds, adequate for Smart Grid / ESS / EV simulations.

4.2 Major RunTS python Functions

a) init_scheduler() - creates a new schedule data structure with default values
b) prepare_new_task() - called to create a task; this initialises task data objects
c) register_new_task() - inserts complete, verified tasks into the schedule at a nominated st (implies event creation if necessary)
d) create_new_event(), move_event(), remove_event() - manage events
e) run_schedule() - cause RunTS to run.

4.3 Building a Schedule

The first step is creating an empty schedule data structure to which tasks can be added. RunTS includes means to prepare an empty task (b). Once created and populated with appropriate data, the task is incorporated into the schedule by 'registering' it (c). This creates a new event within the schedule. RunTS merges same-timed events into one Cycle; there is no need to separate timing.

Executing the schedule causes an inspection of (first => last) st entry. Within the selected st, stages (also in first => last order) are inspected, then tasks within the stage.

The only complication comes with adding (inserting) new tasks on the fly; however the tiered structure of the schedule limits the regions amended. Sort integrity is tracked and the schedule re-sorted as needed.

Note that much of the work is performed during data setup; execution makes use of python's 'bisect' utility (a binary-chop position finder implemented in C) and is rapid.

5. An Example RunTS Run

Note: The RunTS pseudo-code and the scheduled tasks are detailed in the Appendices, available on the internet [13].

The results below are from a test co-simulating 3 independent EVs, each being variations on a V2G EV model also in development.

EVs are created from templates, then scheduled to move from their base, to arrive elsewhere and recharge. On recharging the EV model estimates a time to cease charging / update that charging mode, adding a "charging update" event 'on the fly'. The network (EVSE socket) load is found for each charging mode.

The example includes:

- operation of RunTS to enter and run a schedule,
- demonstration that the EV model can retain distinct internal states
- message passing to and from tasks
    - creation of consequent future events e.g. EV updates (of SOC, charge rates) and
- determination of network loading.

The EVs are of different types and are given manual instructions to 'Arrive' and 'Depart'. Each EV has three charging rates: standard (to SOC 94%), slow (to 97%) then trickle charge. On assessing present SOC, each EV can change the charging rate and determine (schedule) a future charging rate check (a time at which it may be necessary to adjust charging). The EV simulation calculates a socket load to pass to a network solver.

The following tables relate to a specific EV, “EV_L1”. Table 2 below shows the initial commands passed to EV_L1. Each timed event has a RunTS cycle. Note that the cycle at st 0.01 includes commands to other EVs; as long as the stage numbers are consistently organised, commands to multiple entities can be mixed. For clarity, commands to other EVs are omitted. EV_L1 is of interest, as a power loss and restore is scheduled for this EV.

Table 3 shows a log of events experienced by EV_L1. Starred events are generated "on the fly") including a power fail at st 345. Figure 9 plots the EV socket load and battery SOC.

Table 2 Initial events preloaded into RunTS (st per hr = 30; entries shown for EV_L1 only)

Time	st	Stage	Description
00:00:01	0.01	400	Task: Arrive EV at location 'EVSE_House_01'
08:24:00	252	400	Task: Depart EV
09:06:00	273	400	Task: Arrive EV at location 'EVSE_House_01'
11:30:00	345	200	Task: Power loss 'ZoneTrip'
12:45:00	382.5	200	Task: Power restore 'ZoneUnTrip'
18:24:00	552	400	Task: for Depart EV

Table 3 Output log of EV "EV_L1" events (* items are EV-generated events, scheduled on the fly)

Chart Event	At st	Stop At st	EV SOC%	EV Activity	Load (W)
(1)	0.01	124.83	54.89	'ArriveAt: SOC%_change_per_st: 0.313333'	3,434.5
(2) *	124.83	214.83	94.00	'Charging_Update: SOC%_change_per_st: 0.033333'	696.0
(3) *	214.83	---	97.00	'Charging_Update: SOC%_change_per_st: 0.000333'	62.8
(4)	252.00	0.00	97.00	'Depart: SOC%_change_per_st: 0.000000'	0.0
(5)	273.00	330.82	75.88	'ArriveAt: SOC%_change_per_st: 0.313333'	3,434.5
(6) *	330.82	420.82	94.00	'Charging_Update: SOC%_change_per_st: 0.033333'	696.0
(7)	345.00	---	94.47	'Zone_Trip: SOC%_change_per_st: -0.009600'	0.000
(8)	382.50	469.12	94.11	'Zone_UnTrip: SOC%_change_per_st: 0.313333'	3,434.5
(9)	382.50	---	94.11	'Zone_UnTrip: SOC%_change_per_st: 0.000333'	62.84
(10) *	420.82	507.06	94.13	'Charging_Update: SOC%_change_per_st: 0.033333'	696.0
(11) *	507.06	---	97.00	'Charging_Update: SOC%_change_per_st: 0.000333'	62.84
(12)	552.00	0.00	93.70	'Depart: SOC%_change_per_st: 0.000000'	0.0

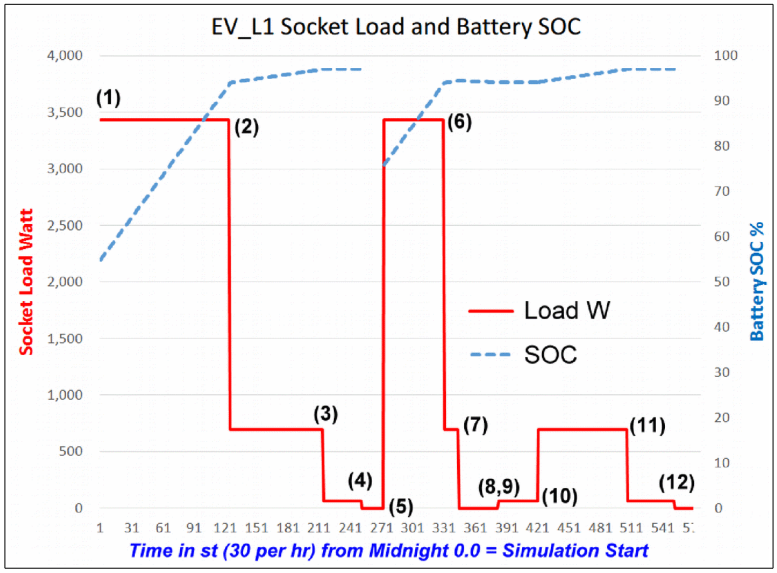


Figure 9: EV SOC and Socket Load over time

5.1 Description of Events

(1) EV_L1 is created ("arrives") at the home EVSE and starts standard charge, calculating the rate of battery SOC change and the end-of-charging time (to reach SOCtarget, 94%) with the socket load. A new event (2) is generated by the EV at the time to reach SOCtarget. RunTS adds this to the schedule.

(2) SOC is assessed and the rate of charging amended, reducing socket load and generating event (3).

(3) The EV enters trickle charge (socket load is now EV system load, not charging).

(4) The EV departs (SOC is not shown as this is now unknown), and at

(5) arrives home, immediately going on charge. A charging-state update at st 330.819 is requested.

(6) The charging update from (5) is actioned and the EV charges in 'slow' mode. A charging update is requested for st 420.82.

(7) The EV suffers loss of power due to a "Zone Trip" and immediately begins to lose SOC as internal systems begin to drain the battery; note that SOC%_change_per_st has become negative (EV system loads are draining the battery).

(8, 9) Service is restored - the trip is removed. But there is an error in the called task code - (9) is a redundant operation. This is though highlights an issue, discussed later.

(10) The charging update requested in (7) is actioned, placing the EV into 'slow' charge. Another charging update is requested for st 507.06.

(11) The charging update requested in (10) is actioned. The EV enters 'trickle' charge.

(12) The EV departs.

Concerning the issue mentioned in (8, 9). Event (9) is spurious - a flaw within the EV ZoneUntrip code; however its presence serves to show two points:

a) The thinking behind why the EV model is calculating charging updates, not applying a commanded change to the charging mode. The EV is connected to a power system, which might fail. A 'blind' command to “stop charging at time X” is not adequate - for power fails might cause the EV to have not reached the correct SOC. A RunTS event is a point in the future at which the system is caused to consider itself; by looking at the SOC means that the system makes an active choice - which might be to continue doing the same thing, or to select a different action. The programmer must be aware of such situations.
b) Event persistence in the schedule. If the defect in the EV "ZoneUntrip" code was removed, event (10) would still occur but be invisible in Fig. 9. Still shown, it is clear scheduled events occur even if upstaged by circumstances. This underscores the need for the programmer to think ahead and code for a choice of actions.

In summary: Events can persist beyond sensible use. Task code needs to be written with defence in mind - i.e. *“Is it reasonable to do this thing I have been asked?”*.

6. Connecting Remote Simulation Tasks

When RunTS calls a task, the request and associated data needs to be sent to a target simulator. This is performed by a "TaskCaller" which holds a list of pre-registered portal modules (as shown in Fig. 2) able to link to the external system.

The invoked portal module performs any link protocol to talk to the external system. Note there may be need for data conversion e.g. of time format etc. On return from the external system, further data conversion may be needed. This code will need to detect any request to create a future task (by using RunTS "create_new_task").

In summary, Tasks are implemented by:

- a dedicated connection module written in python
- during system startup the interface module "registers" itself as being the callable module for that task or set of tasks
- during operation, the module:

- accepts instruction to execute a task from RunTS
- performs any necessary formatting
- calls the external system
- handles return events (results fetch, formatting and saving; new task creation).

The operation of stages are peculiar to specific external simulators, so each connection module needs to be written for that simulator. For instance, when invoking OpenDSS:

- there are two general means to send data to OpenDSS
- there is one method to collect data, but format may vary due to:
  - context-dependant structures for the returned data (e.g. a busbar has no current, but conductors do)
  - specific processing to adapt pointer rationales (OpenDSS maintain a static data window for results; python uses pointers internally; *N* results hence posses the same pointer into the data window. Without adjusting for that, all *N* results show the last value).

Note that the given performance times (section 1.1) includes use of the TaskCaller. For the timing test, RunTS was scheduled with a task which had not been registered. TaskCaller was invoked, found the task had no associated module, performed error handling and discovered a "Test" flag (so threw no error) and returned to RunTS. The times therefore do not include communication with, or operation of, the remote simulation.

7. Simulating or Assessing Other Situations

7.1 Network Events

An event causing a network topology change can be placed in a task, together with any conditional logic wanted. This would then be issued to the network solve engine as a timed network update. Such events can simulate a line fault e.g. by closing a N.O. breaker onto a short-circuit, disconnecting one of a set of Overhead Lines etc.

7.2 Transient Stability Analysis

Transient stability analysis assesses near-time generator recovery (e.g. of slip from synchronism). Implicitly this has temporal duration, over which stability is assessed /

reached. A scheduled task performing transient stability analysis is reasonable given no occurrence of further major destabilising events within the analysis duration. If this condition is not met, the analysis is likely invalid.

However multiple near-simultaneous fault events may be a real analysis task e.g. to analyse a Smart Grid during a storm, during which multiple trips occur as a conductor falls down a tree to contact a metal fence (thus faults occur perhaps 10's of milliseconds apart).

Possible methods include:

- to hand to the analysis tool a list of scheduled events for the forthcoming period
- to modify a transient analysis tool to accept (retrospectively) a network event within the assessment period - and to correspondingly revise its analysis.

7.3 Distributed / Heterogeneous System Operation

RunTS system described was intended to operate on one machine, but there is no impediment to running tasks on remote machines using python's sockets / TCP/IP capability.

It is reasonable to designate one of several computers as an RunTS 'controller' which issues messages across a data network to 3rd party devices which run individual tasks. In this manner multiple dedicated machines can be used. Further, as long as exchanged data is correctly formatted (or so converted) there is no restraint on the machine type.

7.4 Smart Meters, Grids and Actors

Implicitly any task is 'smart' in so far as it can discover states and data and use a process to reach a conclusion. This in turn will likely:

- change local memory states (within the task, e.g. SOC in the EV model)
- issue messages
- create future events.

Smart metering (SM) is simulated by accessing network values at appropriate points. Control is exercised over EV charge / discharge by sending messages to adjust charging rates.

RunTS allows sophistication, but the programmer must be aware of how their system operates. Test cases and verification of operation is always needed.

8. Comparison with Alternatives

8.1 SimPy

SimPy [14] is a comprehensive toolkit for simulators, originally written in python and ported to C, Java and R. The SimPy scheduler has similar timing capabilities to RunTS. SimPy aims to provide the tools needed to construct a complete simulation and include pre-built collections of basic items e.g. common resources pools, logical queues etc.

As a tool-kit to implement a complete simulator, SimPy is not primarily intended to call other simulators. Yet by being based on python, SimPy is notionally capable of undertaking similar tasks to RunTS.

The internal organisation of SimPy uses the “generator” feature of python, resulting in a slightly different conceptual scheme. There is means to observe a central timeline.

There are detail differences. For instance the time counter is an integer, this may complicate (but not prohibit) creating events with a wide dynamic range e.g. moving from hours to milliseconds. Also, the concept comparable to RunTS stages is limited to 8 "priorities" assigned to order coincident events; for many purposes this is adequate but may limit the utility of having stage ranges.

SimPy is popular, has a good reputation for utility and, in its python form, for not being as fast as may be hoped (hence ports to other languages). Popularity has led to the development of tools such as DEVSimPy [15] and Mosaik [16] which are GUI interfaces able to specify SimPy systems. DEVSimPy can accept DEVS formalisms, as next discussed.

8.2 DEVS, PyDEVS and PyPDEVS

The term DEVS here refers to "Discrete EVent Specification". DEVS is a formal mathematical method (conceptual framework) able to express simulator operation / to design simulator implementations ([17], [18]).

PyDEVS [19] and PyPDEVS [20] implement the DEVS framework as python toolkits, able to build simulation suites about DEVS concepts. PyPDEVS is a parallel-processing

version of PyDEVS with many improvements - not just refinements for parallelism (such as TimeWarp and Memoization) but features such as a number of schedulers optimised for particular duties. PyPDEVS is written to connect to distributed systems, which likely will perform other simulations. Execution performance is expected to be good.

The likely users are those needing a wide range of simulation design options, complete with a design philosophy. Such a user would likely craft large portions of their work within Py(P)DEVS. Like SimPy, PyPDEVS uses a specific coding approach.

DEVS formalism can define a simulation system and, although not intended to be part of such an approach, RunTS may be involved in the operation of such a system. Note that RunTS offers opportunities which may need multiple DEVS specifications. DEVS may describe the system in normal operation, yet the use of the Cycle means that the performed operation might diverge.

This can occur when an alternative set of activities are included into a Cycle. Two examples:

1. there might be passive monitoring for a state-change e.g. network component overload, which may trigger a set of Circuit Breakers. On discovering that event, the monitor can insert a new task (or Cycle) to model the operation of the Circuit Breakers and send a network "switch open" command to OpenDSS. These may need to be described as a separate DEVS system.
2. INSim has a "heartbeat" data dump activity, run once per simulated day, which formats and dumps simulation results to file. This is organised via the RunTS as an alternative Cycle. It would not be suggested that this is specified via DEVS.

8.3 Large-Scale Simulation Environments - HLA and FMI

High-level Architecture (HLA, [21]) is a mature standard developed for the US DoD, resulting in a range of compatible simulators. HLA was originally intended to describe the modelling and interworking of wide-scale military elements e.g. simulation of global command and control involving many battlefield assets. Although not similar at first sight to the needs of electrical power simulation, HLA (or similar) may be useful for entire system simulations e.g. a complete, detailed modelling of a National network with a Smart Grid.

HLA consists of a “federation” of interworking simulators. The details of internals are not mandated, rather *"major functional elements, interfaces, and design rules, pertaining as feasible to all simulation applications, and providing a common framework within which specific system architectures can be defined"* [21].

Functional Mock-up Interface (FMI, [22]) is an extensive, generalised co-simulator description and simulation environment developed by the automotive industry. Initially intended to describe vehicle components, FMI now includes dynamic modelling. Such a capability is seen as being useful to describe electrical parts and their operation, such that a description of a component can be downloaded from a supplier, placed into a model and operation simulated - a very useful capability.

Both HLA and FMI offer published standards. Open Source toolsets for these are available, allowing development / composition of various software suites.

Commercial off-the-shelf (COTS) alternatives are available. These are sets of interworking components able to be integrated into a whole simulation; for example Ternion Corporation's FLAMES simulation framework [23]. Here, application expertise is provided not by the system user but the supplier. The intent is for a low-risk, immediate simulation capability. Although standards do exist for some elements (e.g. data exchange via the Common Interface Model / CIM) users are likely to become committed to a particular vendor product. For Operations use, this is preferable.

Where does RunTS sit amongst these? While the options above provide kits from which to assemble a simulator (be it a car panel or a command and control layer), RunTS is a minimalist component aimed at enabling the interworking of existing simulators. There is no present intent for it to form the basis of a standard.

9. Conclusion

This paper presents the INSim Run-Time Scheduler (RunTS), a minimalist DES intended to invoke external simulators.

The envisioned use of RunTS is as a central manager, able to control multiple simulators at specific points in time, hence the sequence of overall execution. The intent of this is to construct a system not otherwise available as one software package, that is, where the whole simulation requires a union of packages, each contributing to the whole.

The method is applicable to models possessing:

- a system state, changed by defined events,
- a state determinable at any time so may be found by knowing an initial state and applying suitable deltas, being known rates of change for the period simulated.

RunTS has been shown simulating the internal states of three autonomous EVs. RunTS is not intended for situations in which an instant in time is either unknown, or the states are diffuse. The method of operation is by state-change events scheduled in an enforced linear time sequence, passing messages between external models (simulations) via a timed schedule. This allows outcomes of state-changes to pass between simulations, influencing forward progress. External systems must possesses a suitable interface to allow communication.

The INSim simulator employs RunTS to connect several simulations, so to enable communication between and control of simulated V2G EVs, local network managers, V2G controllers (Aggregators) and pre-scheduled contextual parameters (such as changing local temperatures, load profiles etc.).

RunTS treats the external simulators as autonomous Black Boxes, and does not attempt to process-optimise their operation. However the tasks used to connect to these must be individually written to suit the target simulator / Black Box.

10. Appendices on Web

The following are available at: http://eprints.soton.ac.uk/id/eprint/405667

Appendix A: Sample Pseudo-Code

Appendix B: A Log of a Completed Schedule

Appendix C: How RunTS invokes Simulation Tasks

11. Acknowledgements

With thanks to sponsors EPSRC, SSEN (SSEPD) and the University of Southampton's Centre for Doctoral Training.

12. References

1. Banks, J., Carson, J.S., Nelson, B.L., Nicol, D.M.: 'Discrete-Event System Simulation: Pearson New International Edition' (Pearson Higher Ed, 2013)

2. Broderick, S., Cruden, A., Sharkh, S., Bessant, N.: 'An improved network simulator for EV/V2G studies', in 'IET CIRED Workshop Proceedings (Paper 0113)' (IET, 2016)

3. Mahmud, K., Town, G.E.: 'A review of computer tools for modeling electric vehicle energy requirements and their impact on power distribution networks', *Applied Energy*, 2016, 172, pp. 337–359.

4. Lin, H., Sambamoorthy, S., Shukla, S., Thorp, J., Mili, L.: 'Power system and communication network co-simulation for smart grid applications', in 'Innovative Smart Grid Technologies (ISGT), 2011 IEEE PES' (IEEE, 2011), pp. 1–6

5. Hybinette, M., Kraemer, E., Xiong, Y., Matthews, G., Ahmed, J.: 'SASSY: a design for a scalable agent-based simulation system using a distributed discrete event infrastructure', in 'Proceedings of the 2006 Winter Simulation Conference' (IEEE, 2006), pp. 926–933

6. Kiaee, M., Cruden, A., Sharkh, S.: 'Estimation of cost savings from participation of electric vehicles in vehicle to grid (V2G) schemes'*Journal of Modern Power Systems and Clean Energy*, 2015, 3, (2), pp. 249–258.

7. Bate, G.: 'The Official Handbook of MASCOT', http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA193178, accessed July 2016

8. Bate, G.: 'Mascot 3: an informal introductory tutorial', *Software Engineering Journal*, 1986, 1, (3), p. 95.

9. Dugan, R.C., McDermott, T.E.: 'An open source platform for collaborating on smart grid research', http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6039829, accessed August 2016

10. Awad, A., Bazan, P., German, R.: 'SGsim: A simulation framework for smart grid applications', in 'Energy Conference (ENERGYCON), 2014 IEEE International' (IEEE, 2014), pp. 730–736

11. Celli, G., Garau, M., Ghiani, E., Pilo, F., Corti, S.: 'Co-Simulation of ICT Technologies for Smart Distribution Networks', 2016.

12. Bagrodia, R.: 'Perils And Pitfalls Of Parallel Discrete-Event Simulation', 1996.

13. Broderick, S., Cruden, A., Sharkh, S., Bessant, N.: 'Pseudo-Code and Data Appendices for Paper: A Technique to Interconnect and Control Co-Simulation Systems', http://eprints.soton.ac.uk/405667/, accessed February 2017

14. 'SimPy: Discrete event simulation for Python', https://simpy.readthedocs.io/en/latest/

15. Capocchi, L., Santucci, J.F., Poggi, B., Nicolai, C.: 'DEVSimPy: A Collaborative Python Software for Modeling and Simulation of DEVS Systems', in (IEEE, 2011), pp. 170–175

16. Lehnhoff, S., Nannen, O., Rohjans, S., *et al.*: 'Exchangeability of power flow simulators in smart grid co-simulations with mosaik', in 'Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), 2015 Workshop on' (IEEE, 2015), pp. 1–6

17. Zeigler, B.P., Praehofer, H., Kim, T.G.: 'Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems' (Academic press, 2000)

18. Vangheluwe, H.L.: 'DEVS as a common denominator for multi-formalism hybrid systems modelling', in 'Computer-Aided Control System Design, 2000. CACSD 2000. IEEE International Symposium on' (IEEE, 2000), pp. 129–134

19. Bolduc, J.-S., Vangheluwe, H.: 'A modeling and simulation package for classic hierarchical DEVS'*MSDL, School of Computer McGill University, Tech. Rep*, 2002.

20. Van Tendeloo, Y., Vangheluwe, H.: 'The modular architecture of the Python (P) DEVS simulation kernel', in 'Proceedings of the 2014 Symposium on Theory of Modeling and Simulation-DEVS' (2014), pp. 387–392

21. Dahmann, J., Fujimoto, R.M., Weatherly, R.M.: 'The DoD high level architecture: an update', in 'Proceedings of the 30th conference on Winter simulation' (IEEE Computer Society Press, 1998), pp. 797–804

22. Bertsch, C., Ahle, E., Schulmeister, U.: 'The Functional Mockup Interface - seen from an industrial perspective', in (2014), pp. 27–33

23. 'Ternion FLAMES Runtime Suite', http://www.ternion.com/framework-architecture/