ENGINEERING STATEMENT OF WORK FOR THE DEVELOPMENT AND SUPPLY OF PRODUCTS, SERVICES AND SOFTWARE

This Engineering Statement of Work (“ESOW”), including all schedules and appendices, is governed by the Jaguar Land Rover (“JLR”)Global Terms and Conditions for Engineering Service (including any Supplemental Terms thereto)] referenced on such Purchase Orders (“Terms”) with references to "Purchase Order" construed as referring to this ESOW, except where doing so would produce illogical results. Any work performed by the Supplier in order to meet the requirements of the ESOW shall be governed by the Terms and the JLR shall issue purchase orders for such work

This ESOW (including any schedules, appendices or other attachments) defines the scope of work required to provide the Deliverables specified below.

VEHICLE SIGNAL MANAGER

Version Control

Version	Date	Description
V0.1	Feb 11, 2017	Initial draft
V0.2	Feb 28, 2017	Loosened requirement on generating C-Code, opting instead for a runtime engine. Added Signal Monitor chapter Added Signal Monitor requirements (SM) Harmonised requirements across Signal Monitor and Policy Engine Added Appendix 5 - Policy Engine reference configuration file Added Appendix 6 - Signal Monitor reference configuration file
v0.3	Mar 2, 2017	Added Capture-Replay chapter and requirements.
v1.0	Mar 12, 2017	Collapsed Policy Engine and Policy Monitor into Policy Manager that encompasses both systems. Replaced Policy Engine and Policy Monitor reference configuration files in appendix 5 and 6 with a single Policy Manager reference file. Updated requirements. ~~Deleted items~~ **New items** Deleted Signal Monitor as its own deliverable in Development Schedule. Rearranged Architecture Overview and Development Phases to give better descriptive flow. Added Fig 3 and text to describe Policy Manager operation.
v1.1	Apr 28, 2017	Renamed Policy Manager to Vehicle Signal manager

Table of Contents

Architectural Overview

Below is an illustration with the logical blocks of a signal emitter the signal manager, and three potential targets for the signal.

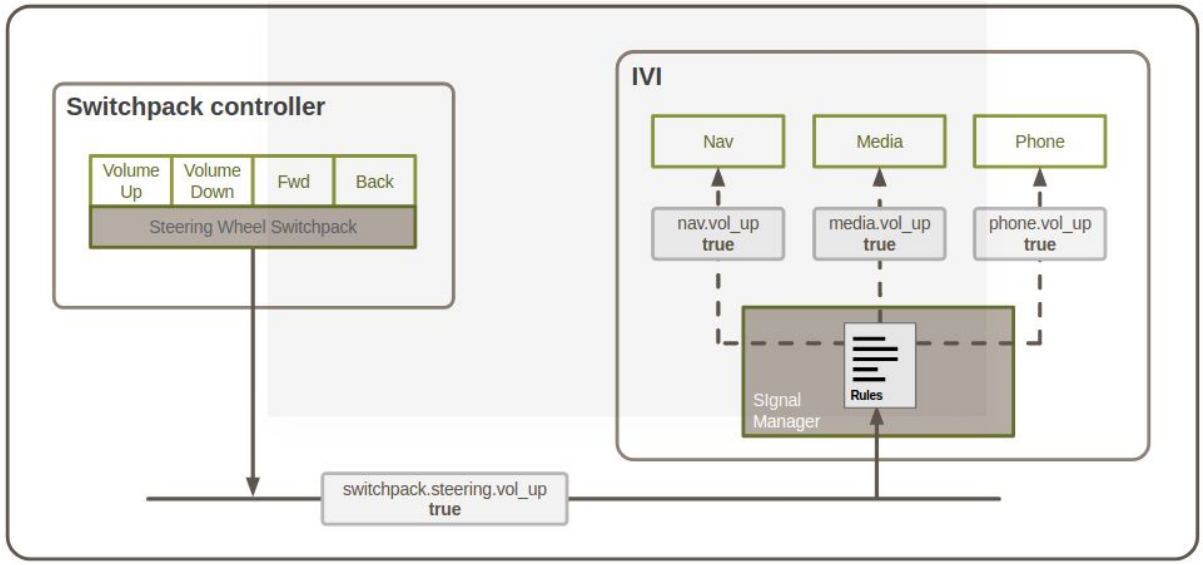

Fig 1. Architectural overview.

When a button is pressed or released, the switchpack controller emits a signal that is consumed by the signal manager.

The signal manager evaluates a number of rules that take the total vehicle state, described by received signals, to determine the target, which can be Navigation, Media, or Phone.

The signal manager emits a single, target-specific, signal that is picked and processed by the receiving component.

The rules are a set of boolean expressions that are evaluated when a specific signal is received by the signal manager, as shown below (in YAML):

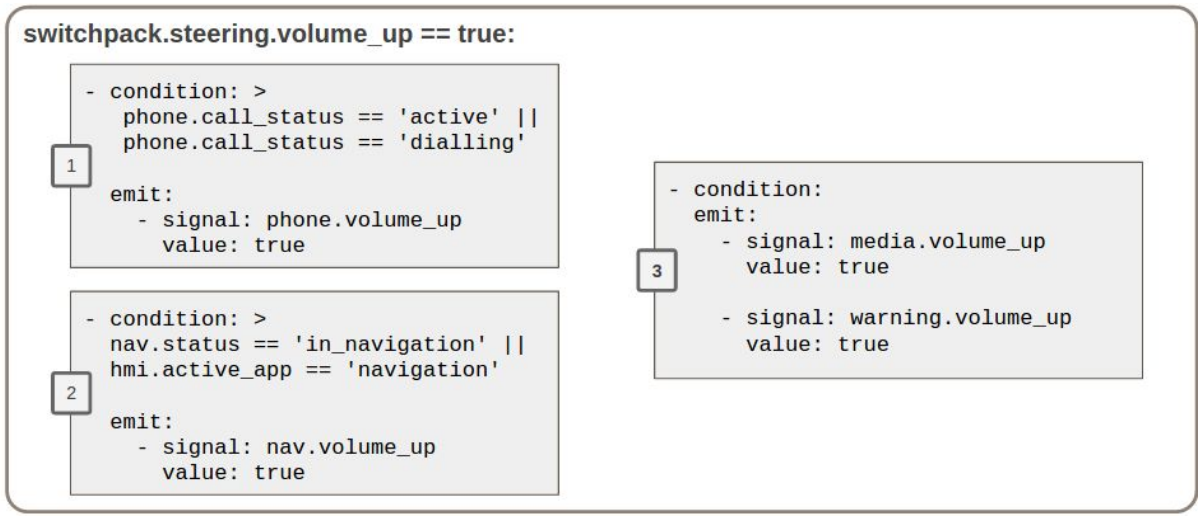

Fig 2. Rules evaluated when the steering wheel's volume up button is pressed.

The condition expression for each rule (1-3) are executed in ascending order. The operands in the expressions are the latest value of the named signals, received from multiple sources in the vehicle (in this case the Infotainment system).

The first expression that evaluates to true will trigger the sending of the signals specified in the emit: clause. The original signal, switchpack.sterring.volume_up in Fig 2, is not retransmitted.

The Signal Manager configuration file is structured as a tree of condition / emit clauses, which in their turn can have condition / emit clauses of their own, effectively forming a tree. Fig 3, below shows an example of a simple configuration file directing the Vehicle Signal Manager

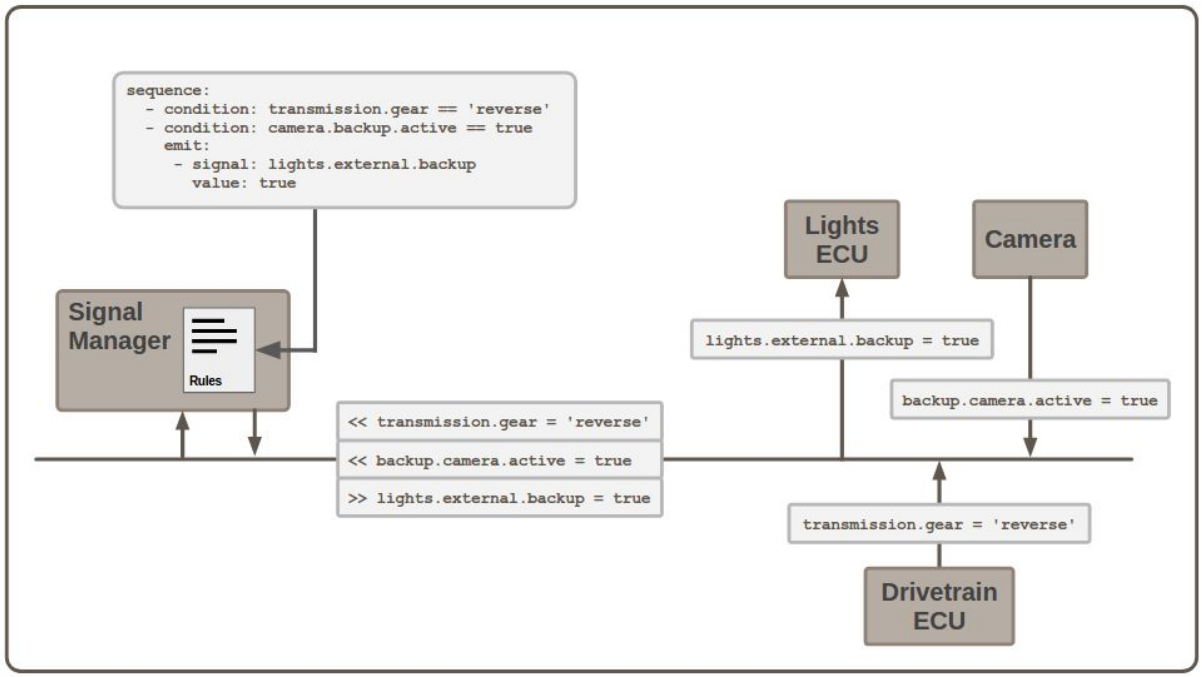

Fig 3. Example of signal monitoring and emittance.

The schematics above shows the signal manager waiting first for transmission gear to be put into reverse, followed by the the backup camera becoming active. Once those two conditions have happened in the correct sequence the Vehicle Signal Manager will emit a signal telling the external backup light to turn on.

Condition to be fulfilled can either be monitored as a sequence of events (as shown above), or as events that can happen in parallel. Please see Appendix 5 for a comprehensive annotated example.

Signal Monitoring

The Signal Manager can optionally also monitor signal sequences and their timing by adding timing specifications to monitored conditions, and generate errors if the sequences do not match the loaded rules. The example given below in fig 4 shows the backup camera not becoming active within the 100 milliseconds allotted by the specification.

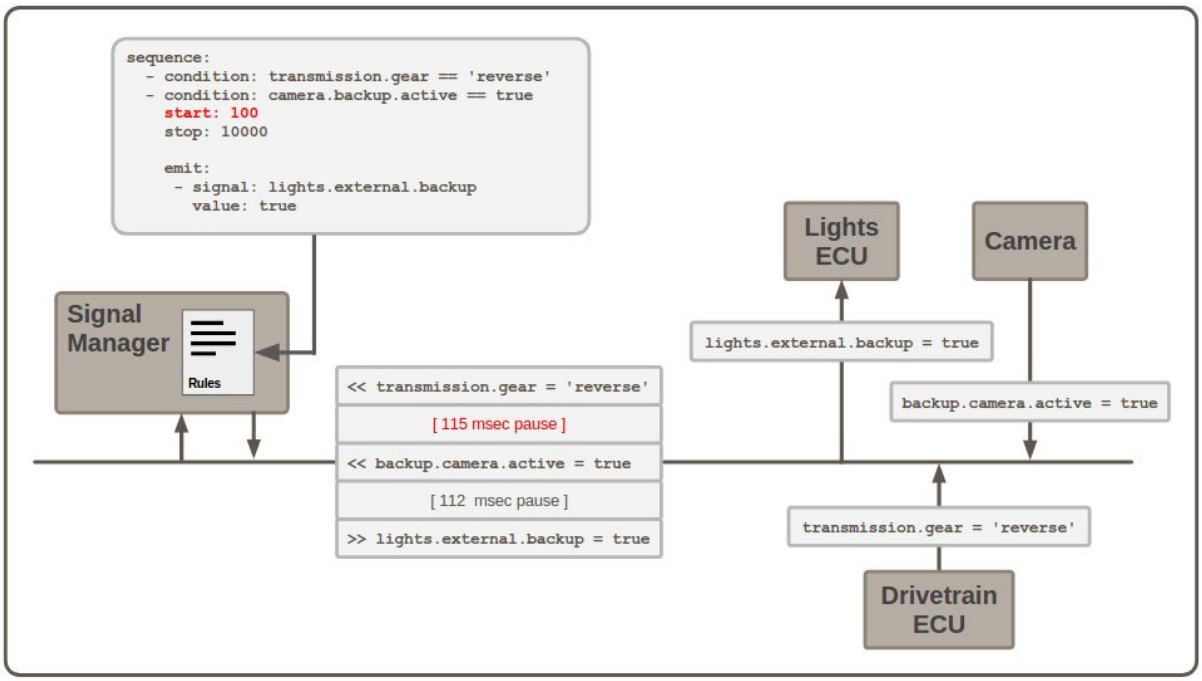

Fig 4. Vehicle Signal Manager acting as a signal monitor

Adding a start and stop specification to a monitor allows it to validate that a given condition is fulfilled. If that does not happen, an error is logged and the monitor is optionally aborted.

The monitored condition is defined as a boolean expression that needs to become true within the specified time interval. A condition that is hosted by a parent monitor will only be actively monitored for as long as the parent condition evaluates to true. In all other cases, the condition is ignored.

When a monitored condition is violated, i.e. an expected signal not being set to the correct value within the specified interval, an error is logged with the following information:

1. Which condition was violated.
2. The value of all signals included in the monitored condition.
3. All parent conditions and their signal values, repeated up to the root condition

The signal manager shall also log all received and emitted signals, with a timestamp, allowing the developer to trace the events leading up to a specific condition.

Signal Capture - Replay capabilities

An important feature of the Vehicle Signal Manager tool is to be able to read all signals seen on a network, timestamp them, and then store them in a file.

At a later stage the captured signals can be read from the storage file and replayed with the same timing onto the network.

This allows a developer to capture live data in the field and then replay the same data in a development environment while testing new software.

Please see Appendix 5 for an annotated reference Vehicle Signal Manager configuration file.

DEVELOPMENT PHASE

Scope of Development

The development effort by the vendor shall deliver a signal manager prototype that can be used to evaluate rule-driven signal routing. There are two main deliverables:

1. ~~Prototyping~~ **Development** tool.
   The tool, written in Python, will read signals, match them against the applicable rules, and emit signals specified by the matched rules.

2. Runtime Code Generator
   The runtime code generator parses rule files and generates runtime code that will read signals, match them against applicable rules (integrated into the generated code), and emits signals specified by the matched rules. The runtime code can either be native C or a lightweight VM such as LUA.

Both variants will log an error if the monitored rules are violated, such a monitored expression evaluating to false when the rule file specifies that it should be true.

Fig 5 and 6, below, illustrates the schematics of both variants

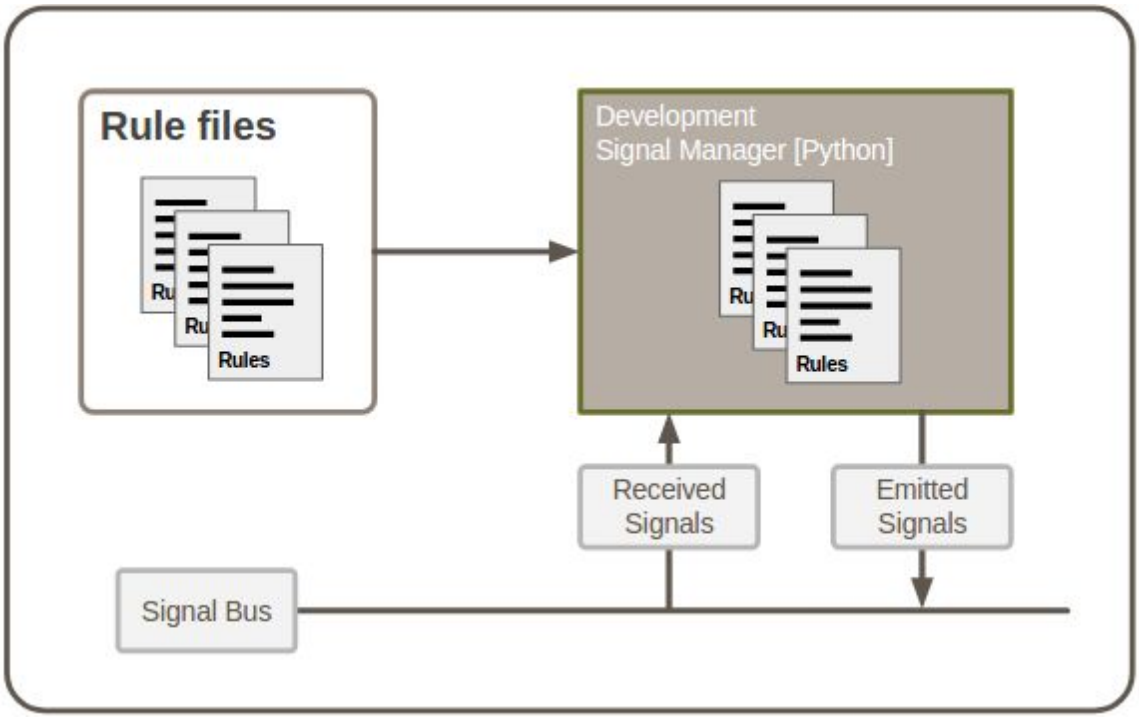

Fig 5. Development Signal Manager schematics

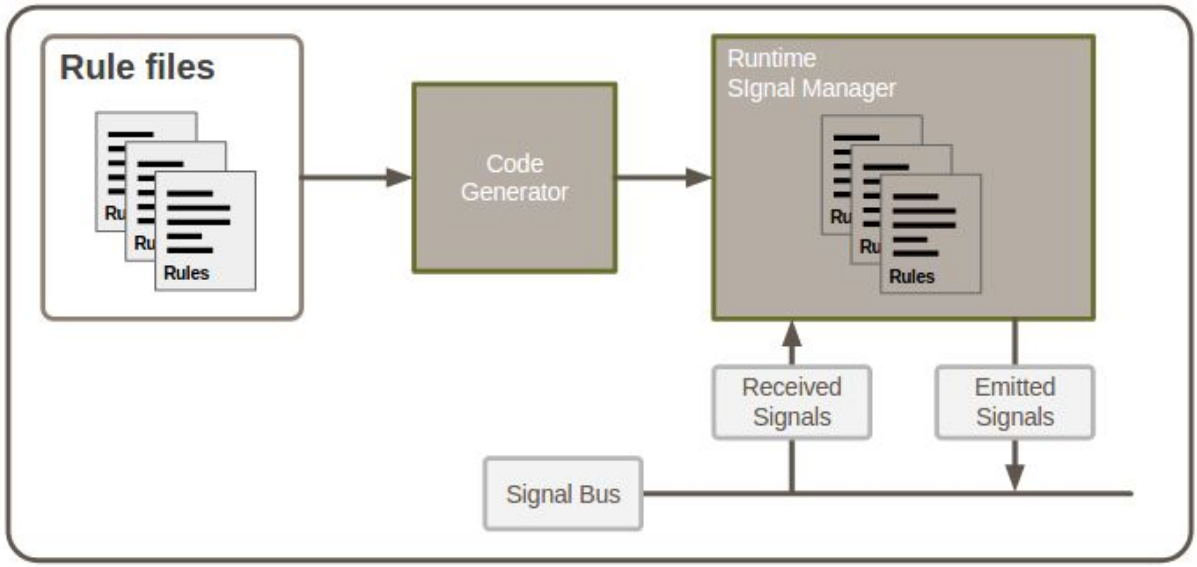

Fig 4.Runtime Vehicle Signal Manager with rules integrated into the generated code.

Development Purpose

On completion of the Development Phase the intent of the Parties is to have a solution that can be used to explore the space of rule-driven signal routing, using the Development Signal Manager, and run performance tests using the runtime code generator.

The Vehicle Signal Manager shall have a pluggable signal reading/writing interface, allowing them to easily integrate with signal sources chosen by JLR.

Both variants shall use the GENIVI Vehicle Signal Specification [https://github.com/genivi/vehicle_signal_specification] as its signal definition source.

Both variants shall, initially, use the GENIVI Vehicle Signal Interface [https://github.com/genivi/vehicle_signal_interface] as the signal reading/writing interface.

JLR Requirements

The Parties agree that the following requirements and features are part of the development phase. Upon final delivery the solution shall include all the requirements listed below. The final hardware deliverable will confirm to the JLR standards listed in Appendix 1

ABORT ON ERROR

ID	Requirements/Features
	General
G1	The Development Signal Manager shall be written in Python
G2	The Runtime Code Generator shall be written in Python
~~G3~~	~~The Signal Monitor shall be written in Python~~
G4	The generated runtime code shall evaluate the integrated rules in its native code
G5	The Development Signal Manager and Runtime Code Generator shall have pluggable signal Interfaces
G6	The generated runtime code can either be native C, or a lightweight VM language such as LUA
	Vehicle Signal Manager
PM1	The Vehicle Signal Manager shall continuously monitor signals for specific conditions
PM2	Conditions shall be specified as Boolean expressions
PM3	Boolean expressions shall **be able to** use signal values as operands
PM4	Boolean expressions shall **be able to** use constant values as operands
PM5	Boolean expressions shall **be able to** support arithmetic expressions as operands
PM6	Boolean expressions shall **be able to** support sub-clauses within parenthesis
PM7	Boolean expressions shall support GTE, GT, LTE, LT, EQ, NEQ, AND, OR, XOR, and NOT as operators
PM8	Conditions shall **be able to** specify a start-stop time interval within which its expression must evaluate to true
PM9	The interval start shall be specified as milliseconds after the monitor becomes active
PM10	The interval stop shall be specified as milliseconds after the interval starts.
~~PM11~~	~~Conditions shall specify a duration for which its expression must remain in a true state after it initially evaluates to that state~~
~~PM12~~	~~The duration shall be specified in milliseconds~~
PM13	A monitored condition shall **be able to** contain child conditions
PM14	Child conditions can be defined to an arbitrary depth, forming a condition tree
PM15	Child conditions shall fulfill the same requirements as the main conditions
PM16	Child conditions shall only be monitored when their hosting parent condition are evaluating to true
PM17	A monitored condition that is violated shall be logged

PM18	The log shall contain a reference to the violated condition
PM19	The log shall contain the values of all signal operands in the violated condition
PM20	The log shall contain a reference to all parent conditions
PM21	The log shall contain the values of all signal operands of all parent conditions
PM22	Conditions that evaluate to true shall **be able to** emit zero or more signals
PM23	Each emitted signal shall have a signal name
PM24	Each emitted signal shall have a signal value
PM25	The signal value ~~can~~ **shall** be an arithmetic expression
PM26	The arithmetic expression shall have constants as operands
PM27	The arithmetic expression shall have signal values as operands
PM28	The emitted signal shall **be able to** have a delay specified before it is sent out
PM29	The delay shall be specified in milliseconds
~~PM30~~	~~The Vehicle Signal Manager shall fulfill the ID requirements (below).~~
PM31	**The Vehicle Signal Manager shall load its rules with monitors and conditions from a configuration file**
PM32	**The configuration file shall be written in YAML**
	Signal ID mapping
ID1	The Vehicle Signal Manager shall receive signals with numerical IDs
ID2	The Vehicle Signal Manager shall be able to translate numerical signal IDs to signal names used by the rule configuration files.
ID3	The signal translation shall be encoded in the code generated by the Code Generator
~~ID4~~	~~The signal translation shall be done by the Prototyping Engine and the Signal Monitor~~
ID5	The signal name-ID mapping shall be read by the Vehicle Signal Manager as an external mapping file
ID6	The external mapping file shall have the same format as the ID file in the Vehicle Signal Specification project
ID7	The Vehicle Signal Manager shall emit an error if a rule contains a signal name not present in the mapping file
	Pluggable Development variant of Manager
PP1	The Development Vehicle Signal Manager shall be able to load signal receive and transmit code through Python extension modules
PP2	The Python extension module shall have a send command to transmit signals
PP3	The send command shall accept a signal ID and a signal value as its arguments
PP4	The Python extension module shall invoke a Python call when a signal is received
PP5	The Python call shall be provided with a signal ID and a signal value as its arguments
PP6	The signal value shall of one of the types supported by the GENIVI Vehicle Signal Specification
PP7	The Python extension module shall integrate with the GENIVI Vehicle Signal Integration module

	Pluggable Generated Code
PN1	The Generated Code shall invoke a predefined function to send a signal
PN2	The predefined function shall be configurable as a command-line argument to the Code Generator
PN3	The predefined function shall accept a signal ID and a signal value as its arguments
PN4	The Generated Code shall have a function to receive a signal
PN5	The generated receive function shall accept a signal ID and a signal value as its arguments
PN6	The name of the generated function shall be configurable as a command-line argument to the Code Generator
PN7	The send and receive function shall have a signature compatible with GENIVI Vehicle Signal Integration project
PN8	The generated code shall be integrated with the GENIVI Vehicle Signal Integration project
	Signal Capture Replay
SR1	The Vehicle Signal Manager shall be able to record all signals received from the network
SR2	The received signals shall be timestamped
SR3	The signals shall be stored, together with its timestamp, in a file.
SR4	The file shall be line-oriented and human readable.
SR7	Each line shall contain comma-separated values, which are escaped.
SR8	The first field shall contain a msec timestamp relative to capture start.
SR9	The second field shall contain the full signal name
SR10	The third field shall contain the numerical signal ID as read from the Vehicle Signal Specification
SR11	The fourth field shall contain the value of the signal
SR12	The value of the signal shall be formatted as python short string literal.
SR13	The short string literal shall handle quotes and escapes as specified by Python 3 Language Reference, Lexical analysis, chapter 2.4.1
SR14	The Vehicle Signal Manager shall be able to replay signals captured during an earlier session.
SR15	The signal replay shall happen with the same timing as the signals were captured with.
SR16	The signal replay shall be able to adjust the speed of the replay, compared to the timing the signals were captured with.
SR17	The speed shall be provided as a decimal value between 0.0% and 10000.0%
SR18	The signals will be replayed at the specified percentage of the speed they were originally received at, with 100.0% being the same speed.
	~~**Rule Engine Rules**~~
~~R1~~	~~Rule files shall be written in YAML or similar data serialisation language~~
~~R2~~	~~The rule configuration file shall organise rules under a received signal~~
~~R3~~	~~Each signal in the configuration file shall have zero or more rules~~
~~R4~~	~~The rules shall be processed in order of appearance~~
~~R5~~	~~Each rule shall have a single trigger Boolean expression~~
~~R6~~	~~The trigger Boolean expression shall fulfill the requirements listed for the Signal Monitor Boolean~~

	~~expressions.~~
~~R7~~	~~Each rule shall have an emit clause~~
~~R8~~	~~The emit clause shall specify signals and their values to send~~
~~R9~~	~~The emit clause shall emit the given signals/values when the associated trigger expression evaluates to true~~
~~R10~~	~~Each rule shall specify if subsequent rules are to be evaluated in case the rule's expression evaluates to true, or if evaluation is to be aborted for the given signal~~

APPENDIX 5 – VEHICLE SIGNAL MANAGER REFERENCE CONFIGURATION

```
# When reverse gear is detected, start two monitors
# to ensure that we have a backup camera and that
# we are not moving forward while in reverse.

# We start the process by unconditionally emitting
# a signal, transmission.gear, set to 'reverse'
emit:
  - signal: transmission.gear
    value: 'reverse'

# We now monitor the incoming signal stream
# for specific conditions to be met.
parallel:


  - condition: transmission.gear == 'reverse'
    # When this monitor's condition (gear in reverse)
    # becomes true, a signal will be emitted to
    # turn on the backup light.
    emit:
      # Emit a signal to turn on the backup light
      # when the hosting condition becomes true.
      #
      # We can emit signals when the condition becomes
      # false, or always (both when true and false).
      - when: always
        # How many msec to wait after condition changes
        # before we emit the signal.
        delay: 100


        # Which signal to emit.
        signal: lights.external.backup

        # The value emitted can be a generic expression
        value: camera.backup.active

    # All monitors under the 'parallel:' element will be
    # activated in parallel.

    # If one monitor condition becomes true it will activate
    # its child monitors and execute its emtis. If, subsequently,
    # a second monitor hosted under the same 'parallel:' element
    # also becomes true, it will also activate its child monitors
    # and emits independently of the first one.
    #
```

```
# If you want to deactivate the other monitors under a 'parallel:'
# element as soon as a specific one evaluates to true, insert
# the 'break:' tag somewhere into the monitor that should
# preempt the others. This will trigger one and only one
# of the monitors under a 'parallel:' element.

parallel:


  # We want to see the backup camera being active
  # within 100 msec of the vehicle being put in reverse.
  #
  # This monitor will be active from 'start' msec after
  # the parent condition (gear == reverse) becomes true.
  #
  # The monitor will terminate 'stop' msec after
  # it becomes active, or when the parent condition
  # becomes false.

  - condition: camera.backup.active == true

    # How many msec do we wait after parent condition
    # becomes true (gear = reverse) until we require
    # our condition (backup camera active) to also be true.
    start: 100

    # How many msec after activation do we keep the
    # monitor active?
    # The monitor condition has to remain true for 'stop'
    # milliseconds after the monitor is started.
    # If those criteria are not fulfilled, an error
    # will be logged
    stop: 1000

    # Use the 'sequence:' element if you want a specific set
    # of conditions to become true in a given sequence.
    # The same effect can be achieved by nesting a number of
    # conditions inside each other, but a 'sequence:' keyword
    # greatly eases readability.

    sequence:

      # We first want the backup radar to become active.

      - condition: radar.backup.active == true
        # Give the backup radar 200 msec to become active
        start: 200

        # The monitor, once started, remains active until
        # the parent condition (backup camera active)
        # becomes false.
        stop: 0
```

```
    # Once the backup radar is active, we then want the
    # backup camera application to be active.
    - condition: ivi.application.backup_camera.active == true
      # Give the backup app 100 msec to become active
      start: 100

      # The monitor, once started, remains active until
      # the parent condition (backup camera active)
      # becomes false.
      stop: 0


# For as long as the gear is in reverse, we want to
# ensure that the vehicle is standing still or
# moving forward.
#
# Since both the backup camera and transmission speed
# condition are hosted under a 'parallel:' element,
# both the backup camera and transmission speed
# will be monitored simultaneously.
#
# If the backup camera becomes active, and that monitor
# in its turn starts monitoring for the backup radar and IVI app
# sequence, the transmission speed will still be monitored
# to ensure that the vehicle is only moving backward.

- condition: transmission.speed <= 0

  # Car may still be rolling forward as reverse is engaged.
  # Wait 1000 msec before activating monitor.
  start: 1000

  # The monitor, once started, remains active until
  # the parent condition (gear = reverse) becomes false.
  stop: 0
```