TECHNICAL UNIVERSITY OF CRETE
ELECTRONIC AND COMPUTER ENGINEERING DEPARTMENT
DIVISION OF COMPUTER SCIENCE

VectorL
A macroprogramming language for testing wireless sensor network applications

by

Asimoglou Aikaterini

A thesis submitted in partial fulfillment of
the requirements for the degree of Master of Science of

ELECTRONIC AND COMPUTER ENGINEERING

July 2015

THESIS COMMITTEE

Assistant Professor Vasilis Samoladas (Supervisor)
Assistant Professor Antonios Deligianakis
Professor Minos Garofalakis

Abstract

Wireless sensor networks (WSNs) have gained world-wide attention in recent years and are expected to find wide applicability and increasing deployment in the near future. These networks are consisted of small sensors, with limited processing and computing resources. Thus, the considerable cost of deploying and maintaining large-scale WSNs for experimental purposes makes simulation a necessary phase of the application development cycle. Sensor node actions are triggered by the information they sense, measure and gather from the environment. Therefore, physical process modeling and simulation is an integral part of a realistic application simulation. Though, sensing is usually neglected in WSN simulators. The usual practice is to feed random numbers to nodes or each node to have a static value. The purpose of this master thesis is to introduce VectorL, a high-level domain-specific language designed to serve the need for an effective and simple way to model and simulate external environment in order to produce realistic sensor readings. Another great advantage of VectorL is that it can be used, not only during simulation phase, but also during wsn testing phase, inside the motes.

Acknowledgements

First of all, I would like express my gratitude to my parents, Vaggelis and Anastasia, for always being there for me to support me, to guide me and, mainly, to love me completely selflessly. They are my only safe shelter for both my greatest and darkest moments.

Secondly, I would like to thank my supervisor Mr. Vasilis Samoladas. He is a superb scientist, willing to generously share his valuable knowledge with his students. His expertise, guidance and advice helped me to accomplish this work. But apart from being an excellent scientist, Mr. Vasilis Samoladas is also a great person, who is treating his students with respect and as equals and who is ready to listen and provide advice to any concern of theirs. To me he is a valuable friend.

Last but not least, I want to thank my boyfriend and all my friends. They are my second family and they have provided me with everything a family provides: love, happiness, support. I am extremely happy for having those specific people in my life and I am grateful that I will leave Chania with a bag full of great memories with them!

Table of Contents

Chapter 1 Introduction

1.1 Wireless Sensor Networks

In recent years, Wireless Sensor Networks (WSNs) have been an area of significant research. These networks are formed by a large number of networked sensing nodes. These nodes are small, with limited processing and computing capabilities. They can sense, measure, and gather information from the environment and, based on some local decision process, they can transmit the sensed data to the user.

A typical node in such networks consists of processing capability (one or more microcontrollers), may contain multiple types of memory, have a radio-frequency (RF) transceiver, have a power source (batteries, solar cells, etc.) and accommodate various sensors and actuators. A variety of mechanical, thermal, biological, chemical, optical, and magnetic sensors may be attached to the sensor node to measure properties of the environment. Since sensor nodes have limited memory and are typically deployed in difficult-to-access locations, a radio is implemented for wireless communication to transfer data to a base station (eg. a laptop, a personal handheld device, or an access point to a fixed infrastructure). Battery is the main power source in a sensor node. Secondary power supply that harvests power from the environment such as solar panels may be added to the node depending on the appropriateness of the environment where the sensor will be deployed.

A complete wireless sensor network usually consists of one or more base stations (or gateway), a number of sensor nodes and the end user. Sensor nodes measure physical quantities and their output is wirelessly transmitted to the base station for data collection, analysis and logging. End users may also be able to receive and manage the data from the sensor via a website from long-distance or applications in console terminal.

There are two types of WSNs: structured and unstructured. An unstructured WSN is one that contains a dense collection of sensor nodes. Once deployed, the network is left unattended to perform monitoring and reporting functions. In an unstructured WSN, network maintenance such as managing connectivity and detecting failures is difficult since there are so many nodes. In a structured WSN, all or some of the sensor nodes are deployed in a pre-planned manner. The advantage of structured network is that fewer nodes can be deployed with lower network maintenance and management cost.

WSNs have great potential for many applications in scenarios such as military target tracking and surveillance, natural disaster relief, biomedical health monitoring and hazardous environment exploration and seismic sensing.

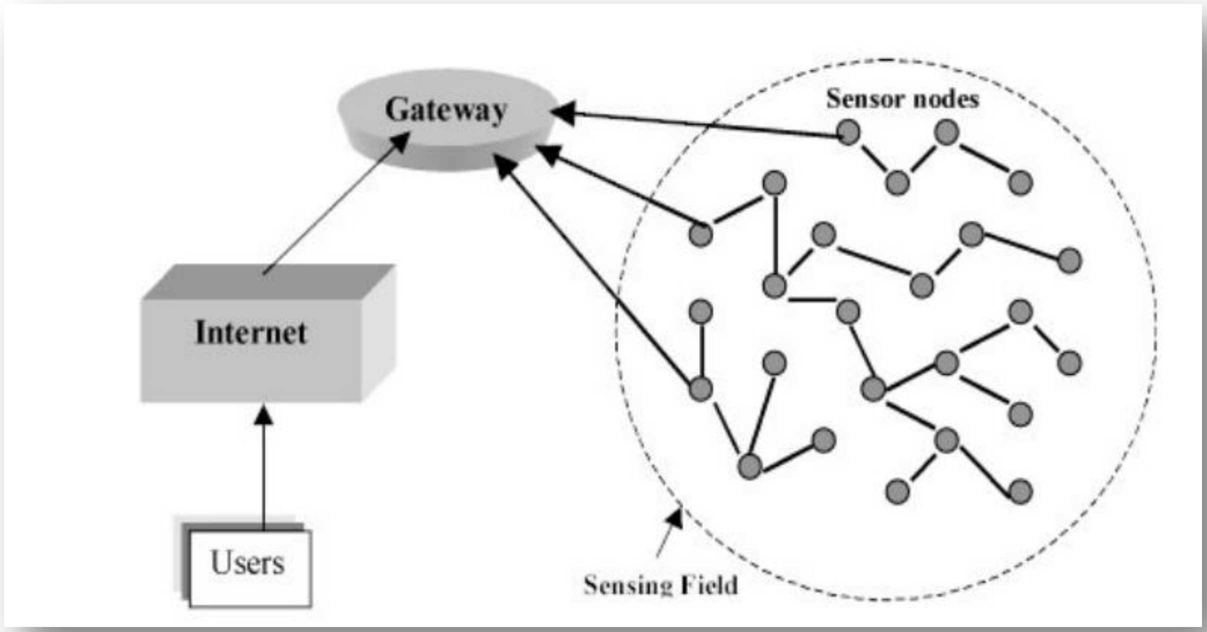


FIGURE 1: A TYPICAL WIRELESS SENSOR NETWORK

1.2 Problem statement

Development of WSN applications with performance guarantees is a very challenging task. The small form factor of sensor nodes imposes severe constraints on the availability of resources such as power, memory, communication range, and sensing capability. The difficulty of providing performance predictions makes it imperative for developers to test applications thoroughly, preferably by using different sets of parameters in a realistic environment at all phases of the application development cycle.

Simulation is a cost-effective choice for prototyping and testing such WSN applications, as the cost, time and complexity involved in deploying and constantly changing actual large-scale WSNs for experimental purposes are prohibitively high.

WSN applications feature tight integration of computation, communication and interaction with the physical environment. Thus, the validity and effectiveness of simulation results depends heavily on how accurately the environment and the sensing process are modeled.

Despite its important role during the simulation phase, sensing is usually neglected in WSN simulators. Issues like sensing device noise or bias are rarely taken into account. The usual practice to sensed data-generation is to feed random numbers to nodes, or each node to have a static value, or at the best case feed the nodes with traces of sensed data. The last case is indeed realistic if we are concerned with a very specific physical process but the data traces rarely lend themselves to every kind of physical process simulation. For early phase algorithm design we need physical process models that are flexible enough yet have some correspondence to real processes (eg. spatial correlation of data, variability over time).

However, this problem is not met only during the simulation phase, but also during WSN testing phase. Fidelity of a simulation cannot always match the physical devices; a field experiment is an ultimate test of operation of a sensor network. Currently, the mostly used approaches record spot measurements and play them back during testing. So imagine how one could test a fire detection WSN's effectiveness.

1.3 Introducing VectorL

This thesis tries to go a couple of steps further than the usual practice and introduces VectorL, a Domain Specific Language (DSL) that is meant to aid the programmer to model and fully simulate the external environment.

VectorL is a small but powerful domain-specific language for creating simple numerical simulations. VectorL programs are executed inside network simulations, but do not interact directly with the sensors or gateways of the system. Instead, a VectorL program is supposed to simulate some natural or artificial system (weather, pollution, car traffic, fires, etc) independent of the WSN network. However, it is possible for VectorL programs to receive input from the nodes, if the latter are equipped with actuators that may affect the environment (eg. lock and unlock doors, turn signal lights on/off, etc).

A design choice for VectorL was to avoid looping constructs and data types that are not needed for simulation. By omitting loops from the language, VectorL programs are much easier to debug and execute safely inside a simulation. On the other hand, loops are important computational components. Therefore, in order to avoid explicit loops, VectorL provides a very powerful syntax for handling arrays, similar to the syntax of mathematical software, such as Octave, Yorick or Python NumPy.

Another impact of the design choices, under which VectorL was constructed, is the possibility of executing VectorL simulation code during application's field testing process, inside the mote.

1.4 WSN-DPCM Project

All the tools presented in this thesis were created in order to contribute to the WSN-DPCM project. WSN-DPCM is a cooperation project of several technical universities and companies from Spain, Italy and Greece. The project is funded by the ARTEMIS Joint Undertaking (the European technology platform representing the field of advanced research and technology for embedded intelligence and systems), national authorities and European partner companies.

WSN-DPCM aim is to develop a full platform to address the main Wireless Sensor Network (WSN) challenges for smart environments that includes the middleware for heterogeneous wireless technologies and an integrated engineering toolset for **D**evelopment, **P**lanning, **C**ommissioning and **M**aintenance activities for expert and non-expert users.

This project offers a real end-to-end integrated tool-chain solution to promote a true model-driven architecture in all design and operational views of a system:

- The Development view, which promotes reusability of software components and guarantees the functional and behavioral portability among different hardware platforms and vendors.

- The Planning view, which assists the network deployment to shorten the deployment time by minimizing the number of the trial-and-error iterations, while at the same time reducing the number of nodes
- The Commissioning and operational Maintenance view, which helps put the whole smart environment into operation and assist the users that will operate and maintain it.

VectorL was created as a part of the Development tool, and specifically of the Network Simulation component.

Chapter 2 Related Work

2.1 Wireless Sensor Network Simulators

In this section we will do a survey on the existing WSN simulators and on how they treat physical process modeling and simulation.

TOSSIM

TOSSIM is a discrete event simulator developed to simulate entire TinyOS applications and works by replacing components with simulation implementations. The greatest feature of TOSSIM is that the same high-level source code can work in both real sensor network testbed and TOSSIM.

TOSSIM does not provide a way to model and simulate the environment of a wireless sensor network. The process of sensing is handled by the application itself; TinyOS 1.x applications used the ADC and ADCControl interfaces to collect sensor data. In TinyOS 2.0, applications use standard data acquisition interfaces Read, ReadStream or Read Now for collecting sensor data. Components written in nesC (like ConstantSensorC – which just returns a constant value as a reading) are responsible for providing those interfaces and feeding the sensors with data.

SIMX

SimX is an integrated simulation and evaluation environment. It is built upon TOSSIM and provides a set of visualized network manipulation and evaluation tools, such as topology manipulation, timing control, variable watch and conditional breakpoints and sensor data input.

SimX allows user to simulate a sensor's data input by various sources, like a local file or math functions. Moreover, it can also simulate different sampling rates in different nodes.

NS-2

NS-2 is a discrete event simulator developed in C++. In NS-2 a network is consisted of sensor nodes which are connected with a data channel for communication with other network stations and with a phenomenon channel for detecting some physical phenomenon. Once a sensor detects a “ping” of a phenomenon in the phenomenon channel, it acts according to the sensor application defined by the NS-2 user. This application defines how a sensor will react once it detects its target phenomenon.

The presence of phenomena in NS-2 is modeled with broadcast packets which are transmitted from the phenomenon nodes through the phenomenon channel. There are specific phenomenon node types that user can use: carbon monoxide, heavy seismic activity, light seismic activity, audible sound and some generic phenomenon.

SENS

Sens is a discrete event simulator implemented in C++. SENS utilizes a simplified sensor model with three layers (application, network and physical) plus an additional combined environment and radio layer. NesC code can be used directly on it.

Sens claims that it provides a mechanism for modeling physical environments but in terms of wave propagation. An environment is simulated as a 2-dimensional grid of interchangeable square tiles. This generally models sensors on the ground outdoors. Tiles use experimentally measured parameters for how radio and sound waves propagate. Sens provides tiles to simulate grass, concrete sidewalks and walls.

Castalia

Castalia is a simulator for Wireless Sensor Networks (WSN), Body Area Networks (BAN) and generally networks of low-power embedded devices. It is based on the OMNET++ platform and can be used by researchers and developers who want to test their distributed algorithms and/or protocols in realistic wireless channel and radio models, with a realistic node behavior especially relating to access of the radio.

Castalia offers a generic physical process model to feed the sensing devices of the nodes with data. The model is based on an arbitrary number of point sources whose “influence” is diffused over space. The equation that determines the value of the physical process at a certain location and at a certain time is:

$$V(p,t) = \sum_{all\ sources\ i} \frac{Vi(t)}{(K * di(p,t) + 1)^{a}} + N(0,\sigma)$$

Where: V(p,t) denotes the value of the physical process at point p, at time t

Vi(t) denotes the value of the i^{th} source at time t

d_i(p,t) denotes the distance of point p from the i^{th} source at time t

K,a are parameters that determine how is the value from a source diffused

N(0,σ) is a zero-mean Gaussian random variable with standard deviation σ

From the survey done, it is concluded that, to our knowledge, no other simulator supports a full environment model simulation feature like the simulator of DPCM platform does with the VectorL feature.

2.2 Wireless System Networks testing

Despite simulators' usefulness, fidelity of a simulation cannot always match the physical devices. To increase simulation fidelity, hardware in the loop simulators execute some of the functions on real sensor nodes. A popular approach to achieve high fidelity in experiments is to use a sensor testbed. Apart from prior to deployment testing, in-field testing of nodes after deployment is crucial in order to maintain a reliable network. In this chapter we will investigate what practices are followed regarding sensor readings during a wireless sensor network testing phase.

In (Beutel, Plessl, & Wohrle, Mar.2007) a wsn application test framework was presented. This framework aimed at a test approach that would allow the same test cases to be used in simulation, on the testbed and in real-world. In this framework, a test driver implements the inputs, so called stimuli that are applied to the software under test during the execution of the test. Each stimulus is characterized by its value (or type) and the time when it is applied to the software under test (SUT). The sequence of stimuli generated by the test driver can be arbitrarily complex. Static drivers apply a single event to SUT or stimulate the application with a predefined sequence of events.

In (Clouser, Thomas, & Nesterenko, 2007) Emuli is described – a method meant to increase the capability of sensor testbeds and other deployments to experiment with environment sensing and monitoring. Emuli emulates sensor stimuli of sensors using pre-loaded data to the mote before the experiment. For example, for light sensor emulation, a simple light-sensor data collection application was created and used to collect light sensor data. On the basis of the experimental data readings, a model (personality) was created for each mote. The probability that Emuli reports a certain value x was made proportional to the number of times x was reported in the actual experiment. Then, for each individual mote the light sensor component was replaced with an Emuli component configured with a unique personality and run experiments on this model.

In (Jia, Krogh, & Wong, 2005) TOSHILT is described, a tool that allows the application to replay previously recorded or synthetic sensor readings. The readings are loaded in advance and stored in sensor node’s EEPROM memory. This approach, is enhanced in (Luo, He, Zhou, Gu, Abdelzaher, & Stakovic, 2006) by allowing the program using annotations inside his code, to specify which particular data points need to be recorded and then replayed.

Based on the existing work it is obvious that VectorL introduces a new way for wsn testing, by providing a way to fully simulate sensing process, inside the mote.

Chapter 3 WSN-DPCM Network Simulation Overview

3.1 Network Simulation in WSN-DPCM

VectorL language was developed as a part of the NetSim (Network Simulation) system, component of the WSN-DPCM Development Tool, used to execute network simulations. A network simulation is an event-driven simulation of the whole WSN application, whose purpose is to help the developer examine the behavior of the application as a whole, both in terms of the hardware and software as well as in terms of the response of the application under different conditions.

In order to execute a network simulation, the NetSim system requires from user to define a Network Simulation Descriptor (NSD). An NSD is a file into which all aspects that determine a network simulation are defined; these aspects can be categorized to the following groups:

- **Network description**. This is the most crucial part of the simulation as here resides the description of the network to be simulated. This description consists of information such as sensor network topology, the types of sensor and gateways nodes and the application being run.
- **Hardware in the loop simulation**. In this part of the NSD user specifies whether he wishes to perform hardware in the loop simulation (HIL), and if so, he provides the NSD with the identities of the hardware nodes, in order to coordinate the hardware and the software simulations.
- **Simulation Parameters**. In this part of the NSD parameters regarding simulation process (e.g. maximum simulation time) are adjusted.
- **Environment Simulation**. This is where VectorL comes into play. In this part of the NSD user can specify a simulation of the environment in the VectorL language. This environment simulation will produce values that will feed the sensors and gateways during the simulation process.
- **Data analysis and visualization**. In this part of the NSD user specifies the ways he wishes to visualize the output of the simulation. The NSD contains a set of definitions

for simple analytic operations, such as filtering, aggregation and transformation. The original data and the data that were produced from these analytic operations can be plotted and visualized in different ways.

In order to ease user during the NSD creation and maintenance phases, a graphical user interface which provides an NSD Editor was developed.

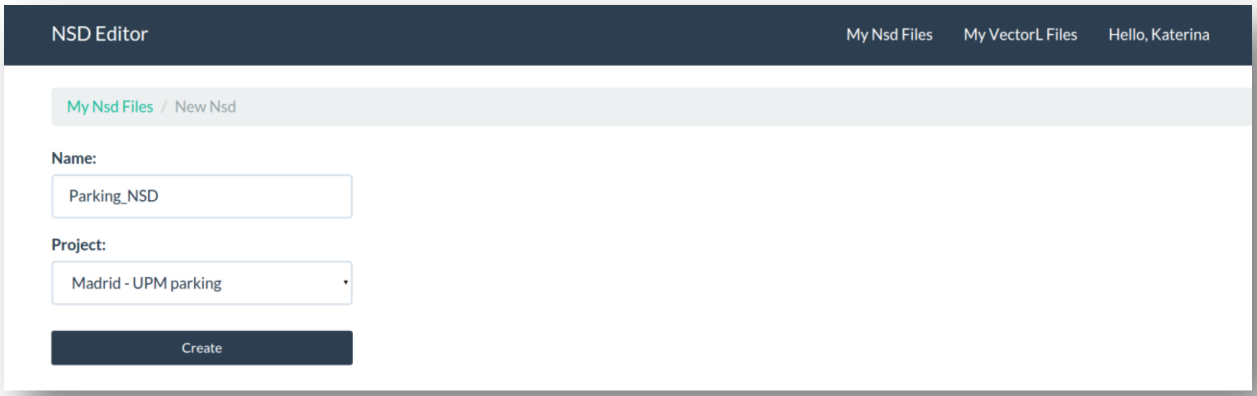
3.2 The NSD Editor

The NSD Editor is an online tool meant to aid DPCM platform users define, access and manipulate NSD files. In this editor there is a place for each one of the five parts of an NSD file mentioned above where user can easily provide the information needed.

In this chapter, NSD Editor graphic user interface is presented.

Defining a Network Simulation Descriptor

In the NSD Editor, user can easily create a new Network Simulation Descriptor by providing a name for it and by specifying the project under which this NSD file will be created. WSN-DPCM projects are created in the Planning Tool of the DPCM platform and NSD Editor fetches them through DPCM Project Repository.



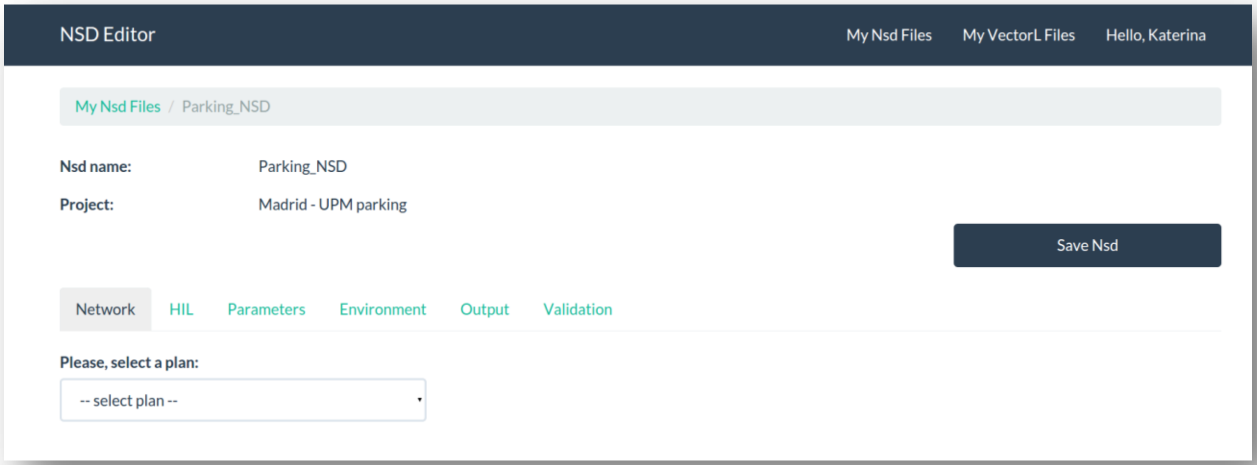
Name: Parking_NSD

Project: Madrid - UPM parking

FIGURE 2 CREATING NEW NSD FOR MADRID-UPM PARKING PROJECT

So, let's say we want to create a simulation for the project Madrid-UPM parking. Actually, this project is a demonstrator application which is built using exclusively the DPCM platform. This WSN application will be an outdoor parking where a WSN will detect free parking slots in the parking area and guide drivers to reach them, park their car and enter automatically in a system all relevant information.

Once user has successfully created a Network Simulator Descriptor, the NSD is immediately opened for further processing. User can see now the five above mentioned categories that need to be filled in order to define completely and correctly a network simulation. We will examine those sections one by one, starting from the Network section.



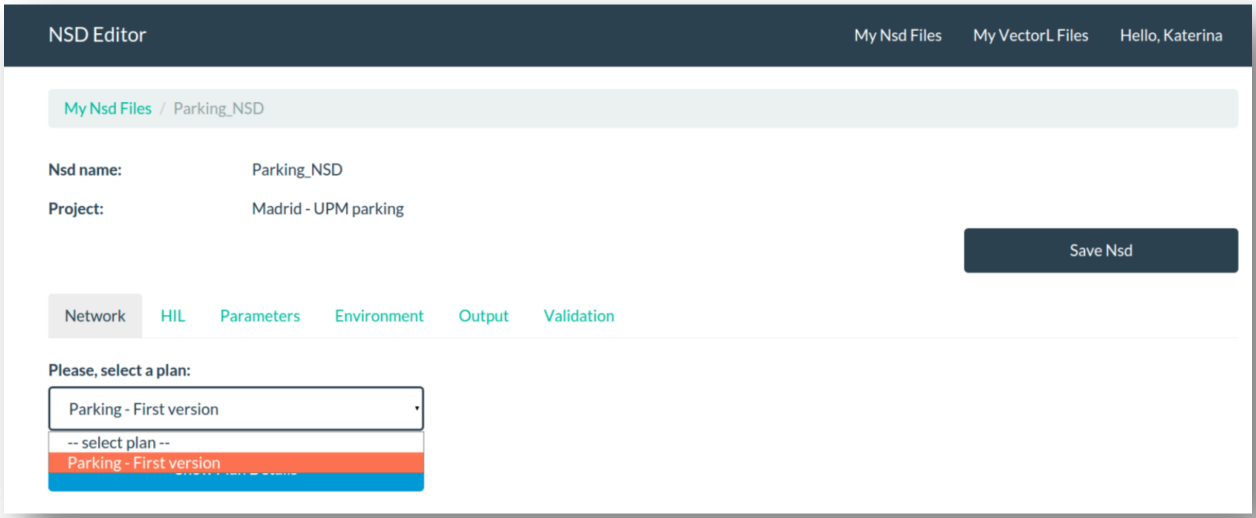
Nsd name: Parking_NSD
Project: Madrid - UPM parking
Please, select a plan:

FIGURE 3 THE FIVE CATEGORIES THAT FORM A NSD

The Network section

As already mentioned, this is the most crucial part of the NSD file. In order to define a simulation for a Wireless Sensor Network it is obvious that we have to specify the network itself, such as what types of sensor and gateways nodes are going to be used, the topology of the network, the application being run, etc.

User has already specified this information during planning phase and using the Planning Tool. The DPCM Planning tool produces a file that contains the types and locations of the sensor and gateway nodes to be used as well as any results of previous Radio Frequency (RF) and Topology simulations. So all user has to do at this step is to provide to the NSD Editor the plan that contains the above data. The NSD Editor provides a list of the different plans that were created under the Madrid-UPM parking project, so all user has to do in this step is to select the plan he wishes to be used in this simulation.



Nsd name: Parking_NSD

Project: Madrid - UPM parking

Please, select a plan:

Parking - First version

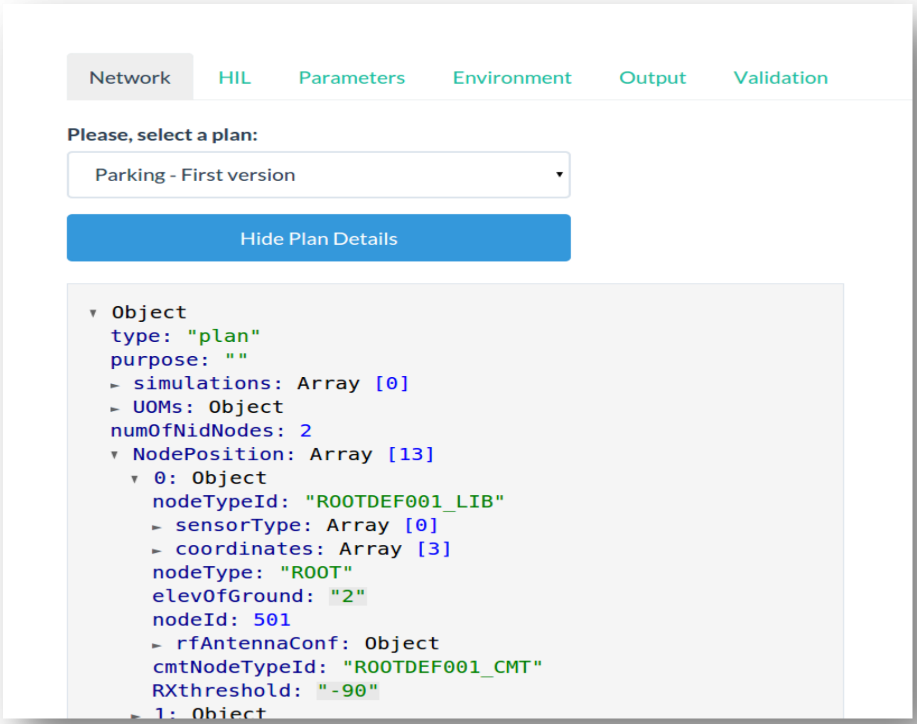
-- select plan --

Parking - First version

FIGURE 4 SELECTING PLAN FOR THE SIMULATION

Once a plan is selected in the Network section, user is able to see the information contained in this file, in JSON format. This capability is very important, as it can help user through making choices in the next sections (for example user has access to node specifications so he can easily choose which nodes are going to participate in Hardware in the Loop simulation).

The application to be simulated is created during the coding phase, using the Development Tool and is specified to the network simulator directly, not to an NSD file through the NSD Editor.



Please, select a plan:

Parking - First version

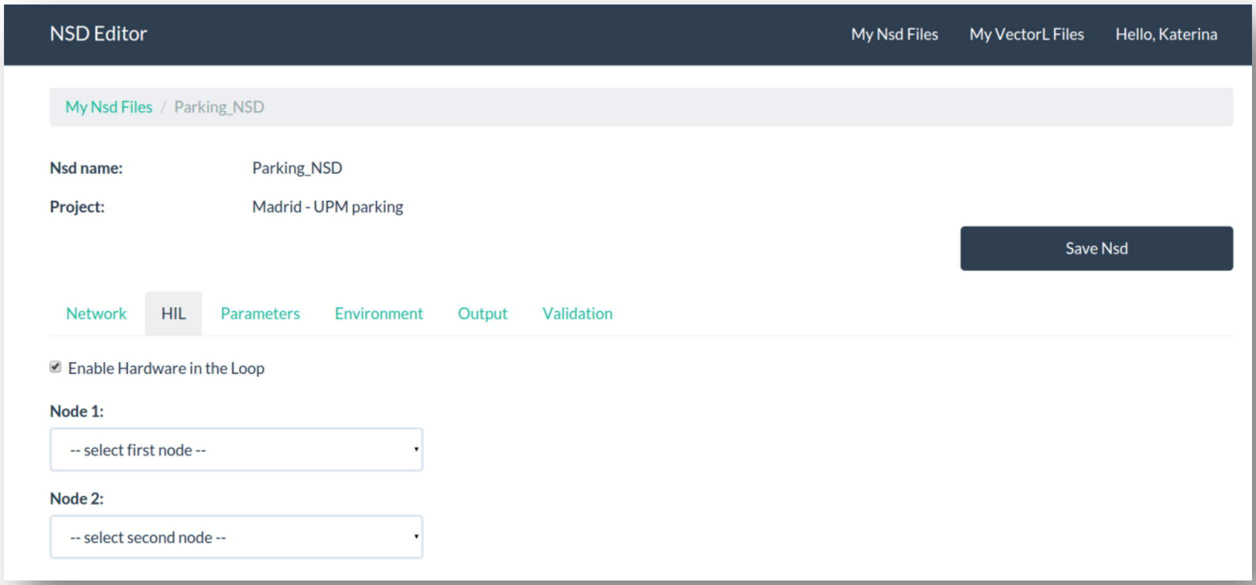
```
▾ Object
  type: "plan"
  purpose: ""
  ▸ simulations: Array [0]
  ▸ UOMs: Object
  numOfNidNodes: 2
  ▾ NodePosition: Array [13]
    ▾ 0: Object
      nodeTypeId: "ROOTDEF001_LIB"
      ▸ sensorType: Array [0]
      ▸ coordinates: Array [3]
      nodeType: "ROOT"
      elevOfGround: "2"
      nodeId: 501
      ▸ rfAntennaConf: Object
      cmtNodeTypeId: "ROOTDEF001_CMT"
      RXthreshold: "-90"
    ▸ 1: Object
```

FIGURE 5 SELECTED PLAN DETAILS

The Hardware in the Loop (HIL) section

Hardware-in-the-loop (HIL) simulation is a technique that is used in the development and test of complex real-time embedded systems. During HIL simulation a mixture of live and simulated network nodes are operating in real time in order to produce highly accurate test data, providing the ability to scale the network size. This approach enables user to test the under development WSN application on real hardware without having to fully live field test the application which is impractical.

DPCM platform supports HIL simulation and user can specify whether he wishes to enable this feature in the HIL section of the NSD Editor.



Nsd name: Parking_NSD

Project: Madrid - UPM parking

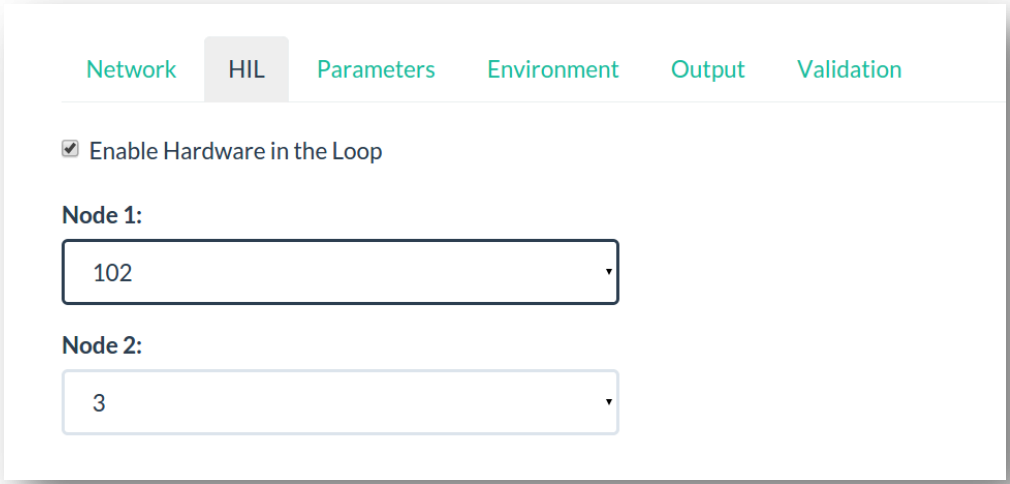
☑ Enable Hardware in the Loop

Node 1: -- select first node --

Node 2: -- select second node --

FIGURE 6 ENABLING HIL SIMULATION

In case user wishes to perform HIL simulation, the NSD must contain the identities of the hardware nodes that will participate to the simulation, in order to coordinate the hardware and the software simulations. NSD Editor takes the list of the node ids that participate to the selected plan and provides them to the user for selection. User can check his selected plan data in the Network section in order to choose the right node ids that will participate to HIL simulation.



☑ Enable Hardware in the Loop

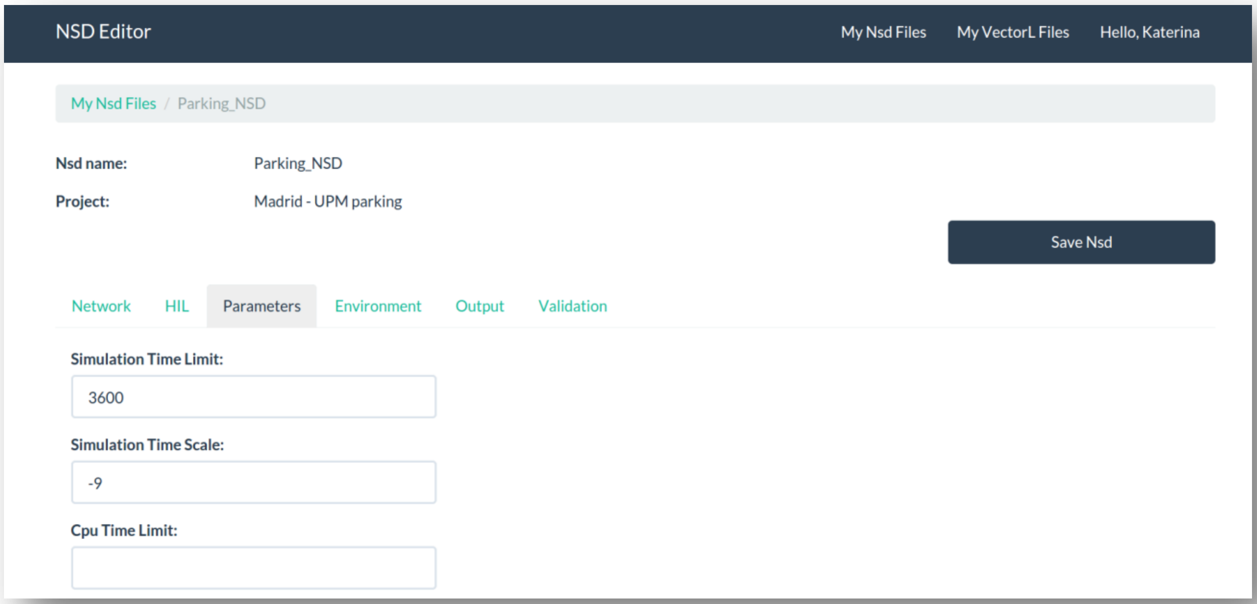
Node 1: 102

Node 2: 3

FIGURE 7 SELECTING NODES TO PARTICIPATE IN HIL SIMULATION

The Parameters section

In this section of the NSD file user can provide values for parameters related to the simulation process. Currently, the parameters taken into account in netSim system are the simulation time limit, the simulation time scale and the cpu time limit.



Nsd name: Parking_NSD

Project: Madrid - UPM parking

Simulation Time Limit: 3600

Simulation Time Scale: -9

Cpu Time Limit:

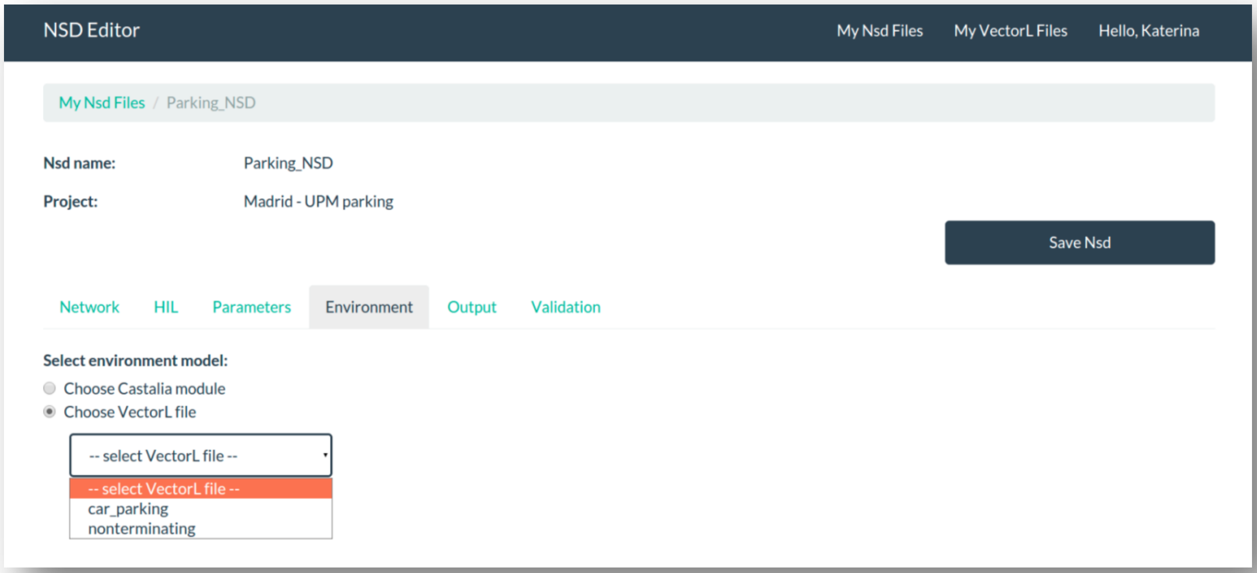
FIGURE 8 ADJUSTING SIMULATION PARAMETERS

The Environment section

In a network application, the correct execution of a WSN application is monitored and the validity of the simulation results depend heavily on how much accurate the sensed data are. In order to produce realistic sensor readings, the physical environment needs to be, as much possible, accurately modeled and simulated.

So in this section user can choose the model to be used for the environment simulation. Environment simulation will feed the sensor nodes with realistic values, fact that will lead to the production of more interesting and accurate results.

The user can choose, either a default module implementation from Castalia library, or to write his own models using VectorL domain specific language. NSD Editor provides a separate graphic user interface for VectorL module editing, as VectorL modules are created separately from NSD files and can be attached to them at a later time. This graphic user interface, called VectorL Editor will be presented in a following section, while VectorL syntax and language characteristics will follow on the next chapter.



Nsd name: Parking_NSD

Project: Madrid - UPM parking

Select environment model:

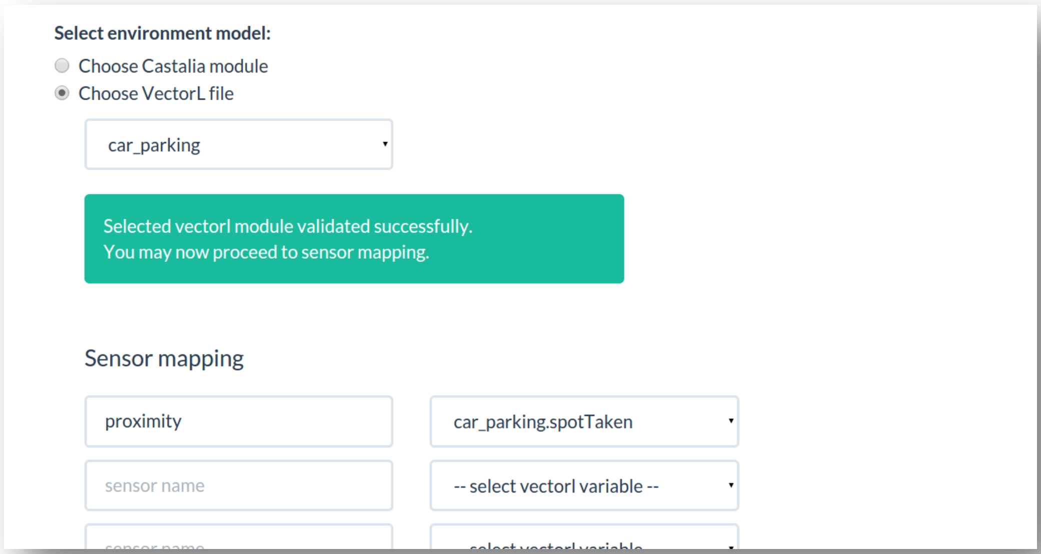
- Choose Castalia module
- Choose VectorL file

-- select VectorL file --
car_parking
nonterminating

FIGURE 9 SELECTING ENVIRONMENT MODEL

Once user has selected an environment model written in VectorL, this model gets instantly validated (compiled). If the compilation of the VectorL code fails, user gets notified and a link is provided to him that opens VectorL Editor and the corresponding model for further processing. If VectorL code is compiled successfully, user is now able to map the sensors used in the application to the VectorL variables that will feed them.

For example, in our parking application motes will contain, among others, proximity sensors that can tell whether a parking spot is free or occupied. So user has to specify the array from his VectorL code that will hold the values produced by the environment simulation and that will represent each proximity sensor's value. This array in our example is the `spotTaken` array which actually keeps the state of each parking spot over time.



Select environment model:

Choose Castalia module [✓ Choose VectorL file]

car_parking

Selected vectorl module validated successfully.
You may now proceed to sensor mapping.

Sensor mapping

proximity	car_parking.spotTaken
sensor name	-- select vectorl variable --

FIGURE 10 MAPPING SENSORS TO VECTORL VARIABLES

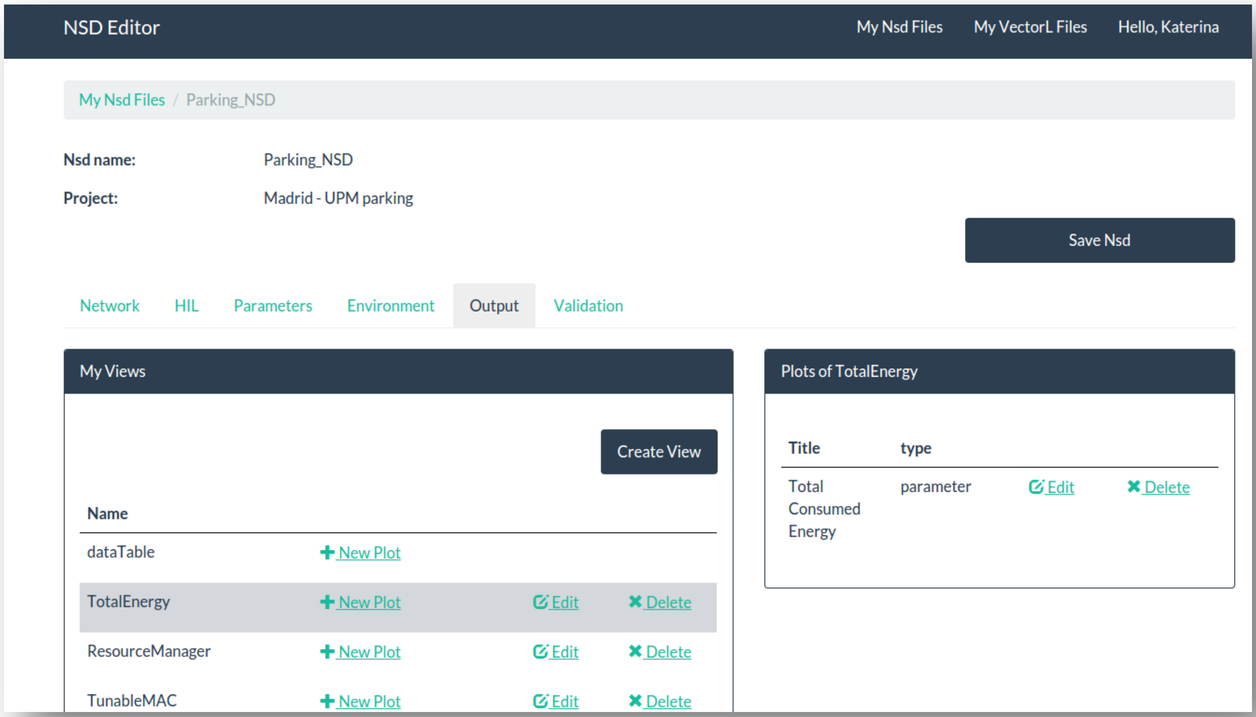
The Output section

In this section of the NSD Editor, the user can specify the processing that he wishes to be made over the data produced by the simulation, as well as define custom ways to visualize these data. Specifically, through the NSD Editor the user can create views and plots which will generate statistics and graphs, accordingly, after the simulation is finished.

Views can be thought of as two dimensional tables where each row represents a single value of a statistic. There is a predefined view called `dataTable` which represents the initial data produced from the simulation. This view cannot be altered or deleted but can be used as a base for defining other views or plots. DataTable view is consisted of the following columns:

- Node: the id of the corresponding node
- Name: the name of the statistic
- Module: the name of the module that this statistic belongs to
- Label: the name of a statistic that is dependent on "name"

- N_index: the id of the node that this statistic is originating from (eg. node 2 sends a packet to node 5 so node 5 will receive a packet with n_index 2)
- Data: the actual value of the statistic.



Nsd name: Parking_NSD

Project: Madrid - UPM parking

Name			
dataTable	New Plot		
TotalEnergy	New Plot	Edit	Delete
ResourceManager	New Plot	Edit	Delete
TunableMAC	New Plot	Edit	Delete

Plots of TotalEnergy

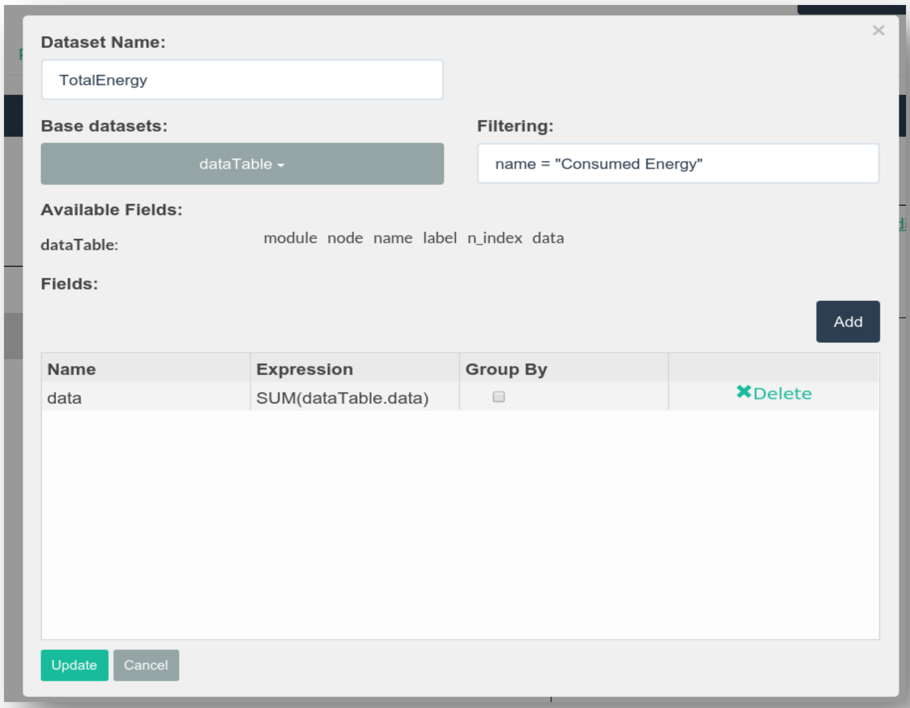
Title	type		
Total Consumed Energy	parameter	Edit	Delete

FIGURE 11 OUTPUT SECTION

The user can further process (aggregate, filter, etc.) the original simulation data by creating new views. For example, let's say that there is a statistic with the name `ConsumedEnergy` produced by the simulation, whose value denotes the energy consumed by each node during the simulation process, and that the user wants to calculate the total energy consumed by all nodes during the simulation.

All the user has to do, is to create a new view and firstly specify the dataset(s) that this view is going to be based on. Here, this dataset is the `dataTable` table. Then, user defines the columns that the currently created view is going to have. In our example it will have only one column whose data will be given by the aggregate `SUM(dataTable.data)`. Of course we do not want all the data rows of `dataTable` to be aggregated, just those that are related to the statistic `ConsumedEnergy` so the user has to specify this filtering expression to his view too. In Figure

12 creating new views, the final state of the newly created view with the name `TotalEnergy` is depicted.


Dataset Name: TotalEnergy

Base datasets: dataTable

Filtering: name = "Consumed Energy"

Available Fields:

dataTable: module node name label n_index data

Fields:

Name	Expression	Group By	
data	SUM(dataTable.data)	☐	✖Delete

FIGURE 12 CREATING NEW VIEWS

Views are closely related to SQL views, actually they are SQL views. The window depicted in Figure 12 creating new views is just a nice way to help the user form an SQL view without the need of actually knowing anything about SQL. The information the user entered in the view `TotalEnergy` will be translated to the following SQL statement:

```
CREATE VIEW TotalEnergy AS
SELECT SUM(dataTable.data) AS data
FROM dataTable
WHERE name= “ConsumedEnergy”
```

Had the user selected the `Group by` checkbox, a GROUP BY statement would have been added for the corresponding column.

Apart from processing original simulation data, user can define plots using the NSD Editor. These plots will produce graphs depicting the selected quantities after the simulation is finished. Plots are created in the context of the view they will get their data from. Following our

`TotalEnergy` example, once user has successfully created his view, he can now create a new plot for this view.

There are three graph types that can be generated:

- Plot
- Node parameter (will generate values for a specific statistic for every node)
- Network parameter (will generate a statistic that has a network wide meaning)
- Node2Node parameter

In our example, we want to create a Network parameter graph, as for the Total Energy statistic calculation the whole network participates.

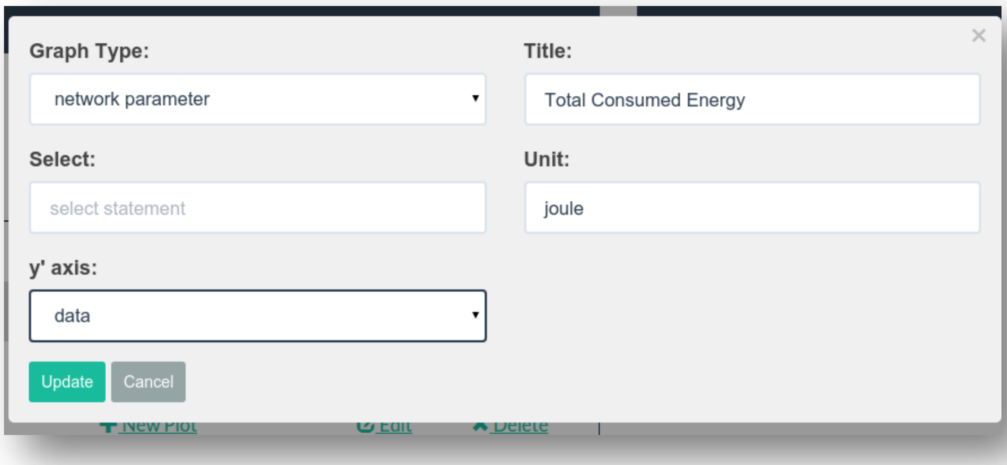

Graph Type: network parameter
Title: Total Consumed Energy
Select: select statement
Unit: joule
y' axis: data

FIGURE 13 CREATING A PLOT

After specifying the graph type, the user can give a title to his plot, choose the y’ axis data (x’ axis in the case of a network parameter will be all nodes), provide optionally a select statement over the selected data and the unit of the statistic. User can select his y’axis data among the columns of the view, in the context of which this plot is created. In our example, there is only one column (`data`) specified in `TotalEnergy` view which corresponds to the SUM aggregation of `ConsumedEnergy` statistic.

The data that are required to be entered are different for each graph type, but this is out of the scope of this thesis as this chapter's purpose is to give a general impression of how NetSim system works, what an NSD is and how it is constructed, rather than to provide a user manual of the NSD Editor.

In Figure 14, the graph produced for the `Total Energy Consumed` plot is depicted.

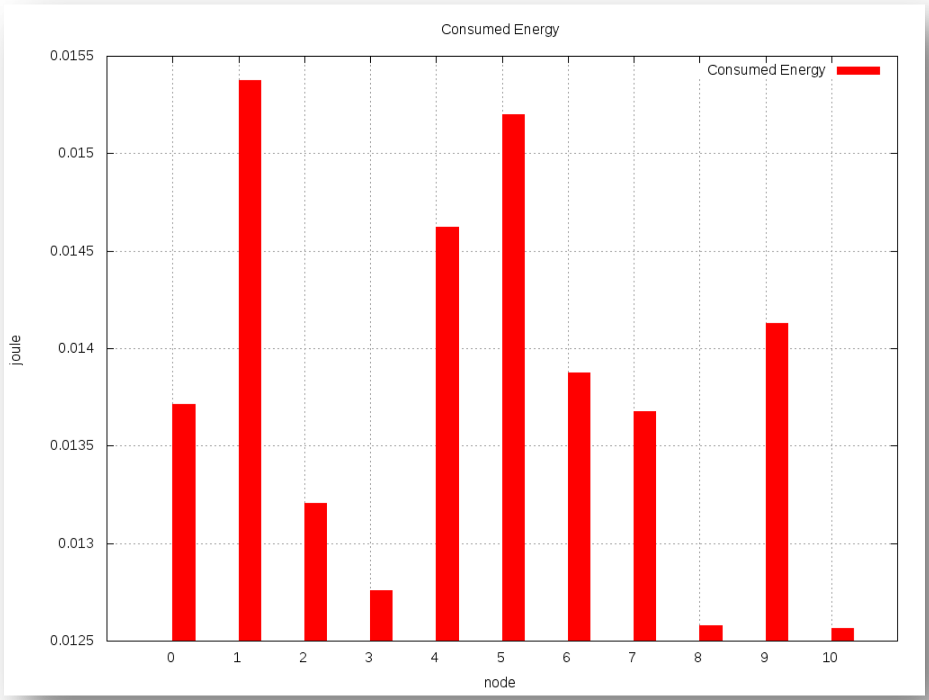
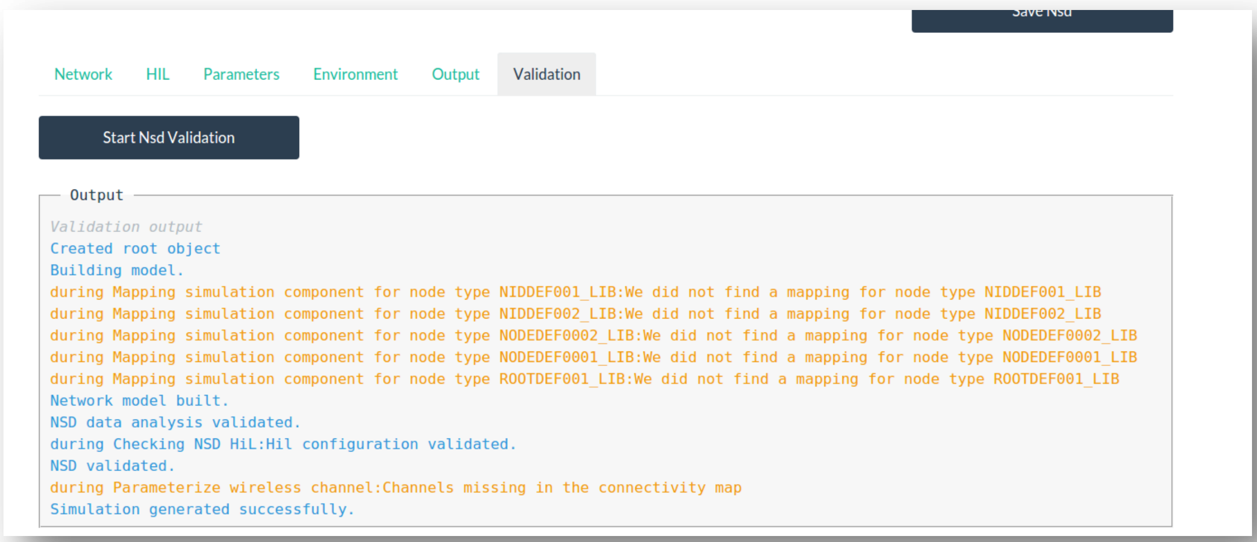


FIGURE 14 NETWORK PARAMETER GRAPH EXAMPLE

NSD Validation

A network simulation contains a lot of information and depends on a large number of parameters. It is not difficult for errors or omissions to enter into the provided data. Although the DPCM framework tries its best to assure that such problems do not arise, it is not feasible to foresee all the possibilities.

Therefore, once a user has edited an NSD to his wishes, he can proceed and check its validity. If there are errors in the NSD file user will be notified. If no problems are found by the validator, the newly created NSD can be used to run successful simulations.



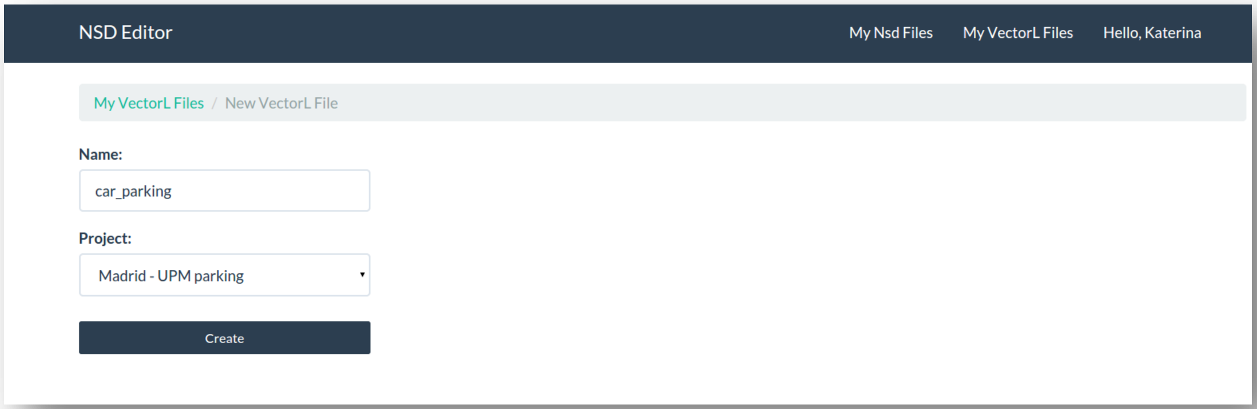
```
Validation output
Created root object
Building model.
during Mapping simulation component for node type NIDDEF001_LIB:We did not find a mapping for node type NIDDEF001_LIB
during Mapping simulation component for node type NIDDEF002_LIB:We did not find a mapping for node type NIDDEF002_LIB
during Mapping simulation component for node type NODEDEF0002_LIB:We did not find a mapping for node type NODEDEF0002_LIB
during Mapping simulation component for node type NODEDEF0001_LIB:We did not find a mapping for node type NODEDEF0001_LIB
during Mapping simulation component for node type ROOTDEF001_LIB:We did not find a mapping for node type ROOTDEF001_LIB
Network model built.
NSD data analysis validated.
during Checking NSD HiL:Hil configuration validated.
NSD validated.
during Parameterize wireless channel:Channels missing in the connectivity map
Simulation generated successfully.
```

FIGURE 15 NSD VALIDATION OUTPUT

3.3 The VectorL Editor

The VectorL Editor is a part of the NSD Editor graphic user interface and is used to create VectorL modules which will later be attached to NSDs and will provide environment models for the network simulations.

In the NSD Editor, user can easily create a new VectorL module by providing a name for it and by specifying the project under which this module will be created. Once user creates a new VectorL module, it is immediately opened in the VectorL Editor for further processing. Following parking demonstrator application, let’s create a VectorL module in order to model the car parking environment.

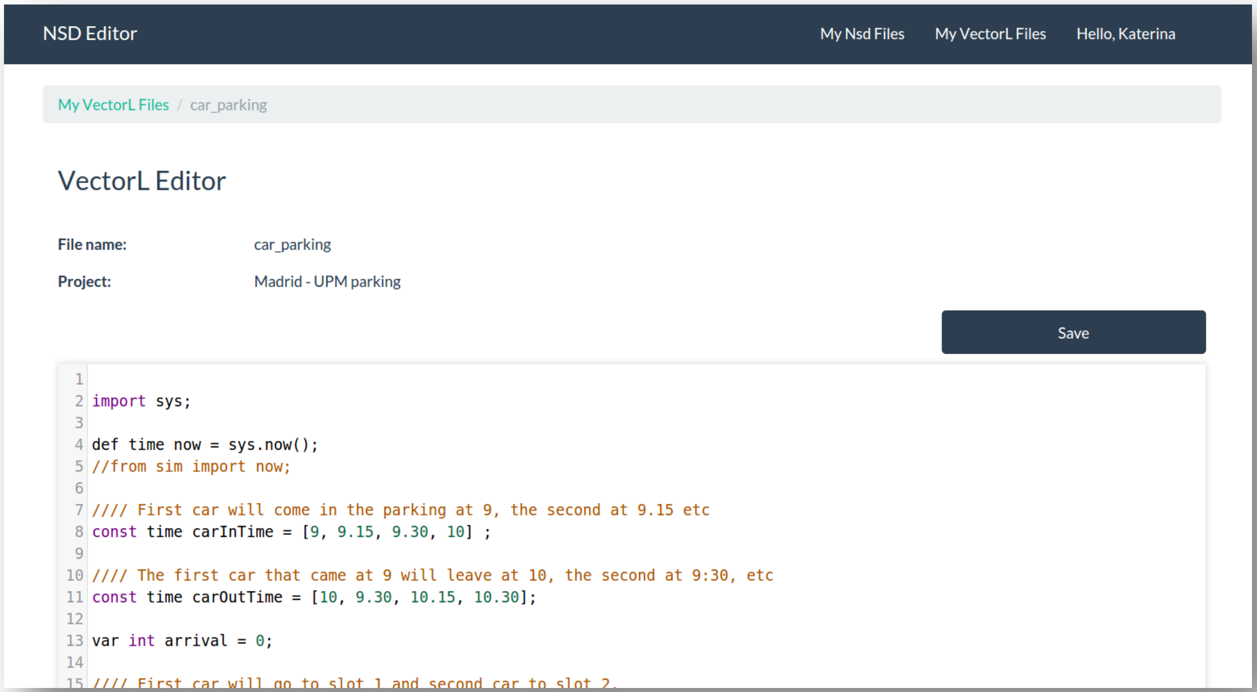


Name: car_parking

Project: Madrid - UPM parking

FIGURE 16 CREATING A NEW VECTORL MODULE

In the VectorL Editor there is a code editor where user can write his VectorL code.



File name: car_parking

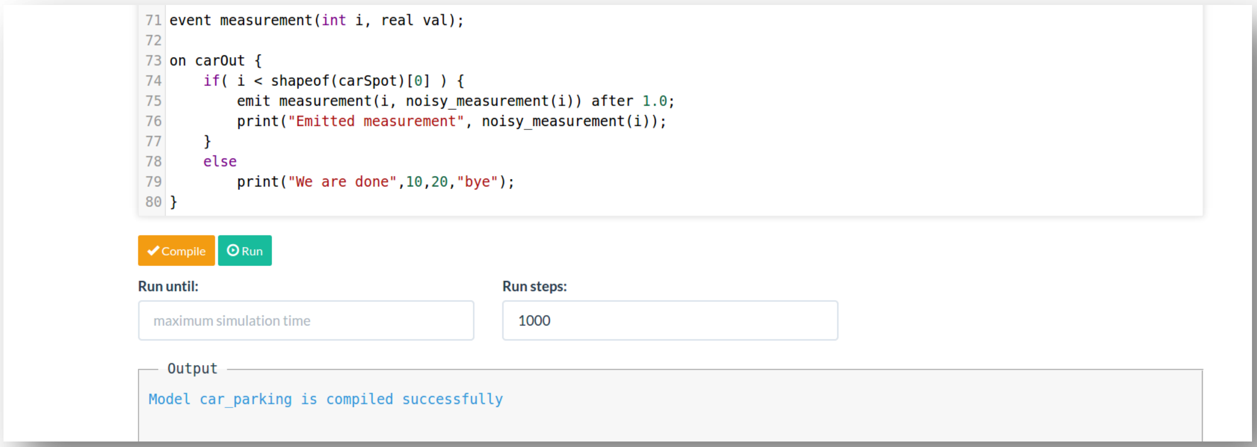
Project: Madrid - UPM parking

```

import sys;

def time now = sys.now();
//from sim import now;

//// First car will come in the parking at 9, the second at 9.15 etc
const time carInTime = [9, 9.15, 9.30, 10] ;

//// The first car that came at 9 will leave at 10, the second at 9:30, etc
const time carOutTime = [10, 9.30, 10.15, 10.30];

var int arrival = 0;

```

FIGURE 17 VECTORL EDITOR

At any time the user can compile his code. Under the hood, the code is lexically and syntactically analyzed and checked for errors and, in case everything is correct, various code generators come into play. From a single VectorL module will be produced both the C++ code needed by

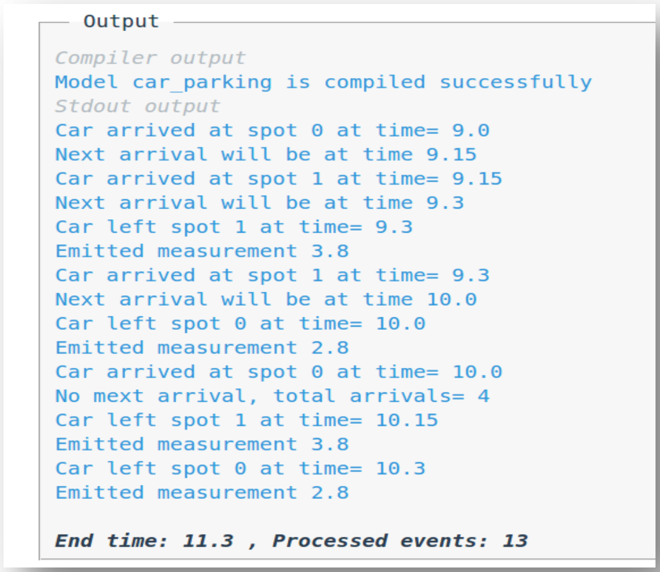
OMNET++ to execute the simulation and the nesC (TinyOS) code that will run inside the real motes during the testing phase. More technical details will follow on subsequent chapters.



```
event measurement(int i, real val);

on carOut {
    if( i < shapeof(carSpot)[0] ) {
        emit measurement(i, noisy_measurement(i)) after 1.0;
        print("Emitted measurement", noisy_measurement(i));
    }
    else
        print("We are done",10,20,"bye");
}
```

```
Model car_parking is compiled successfully
```

FIGURE 18 COMPILING VECTORL



```
Output
Compiler output
Model car_parking is compiled successfully
Stdout output
Car arrived at spot 0 at time= 9.0
Next arrival will be at time 9.15
Car arrived at spot 1 at time= 9.15
Next arrival will be at time 9.3
Car left spot 1 at time= 9.3
Emitted measurement 3.8
Car arrived at spot 1 at time= 9.3
Next arrival will be at time 10.0
Car left spot 0 at time= 10.0
Emitted measurement 2.8
Car arrived at spot 0 at time= 10.0
No mext arrival, total arrivals= 4
Car left spot 1 at time= 10.15
Emitted measurement 3.8
Car left spot 0 at time= 10.3
Emitted measurement 2.8

End time: 11.3 , Processed events: 13
```

FIGURE 19 EXECUTING VECTORL

User can also execute his VectorL code for a specific time or for a specific number of steps. Note that for this execution no generated code is used. Instead, as soon as the Abstract Syntax Tree is constructed right after the syntactical analysis, the model is compiled (translated) into a simple stack machine program and gets executed.

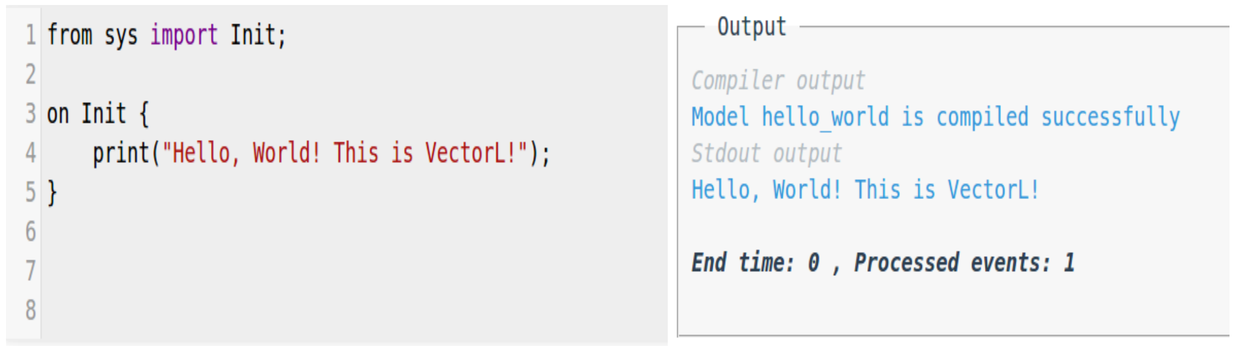
The standard output produced by a successful execution of the VectorL code is displayed to the user. In Figure 19 executing vectorl, the output of a successful execution of the parking model is depicted.

This chapter was an overview of how Network Simulation is treated in the DPCM platform. It introduced NSD notion and demonstrated the role that VectorL plays in it. Now, it is time to take a thorough look at VectorL language, its syntax and the concepts behind it.

Chapter 4 VectorL Language

4.1 Hello VectorL!

Probably the best way to start learning a programming language is by writing a program. And as in every other language, let’s introduce VectorL to the world.



```
1 from sys import Init;
2
3 on Init {
4     print("Hello, World! This is VectorL!");
5 }
6
7
8
```

```
Compiler output
Model hello_world is compiled successfully
Stdout output
Hello, World! This is VectorL!

End time: 0 , Processed events: 1
```

FIGURE 20 HELLO WORLD!

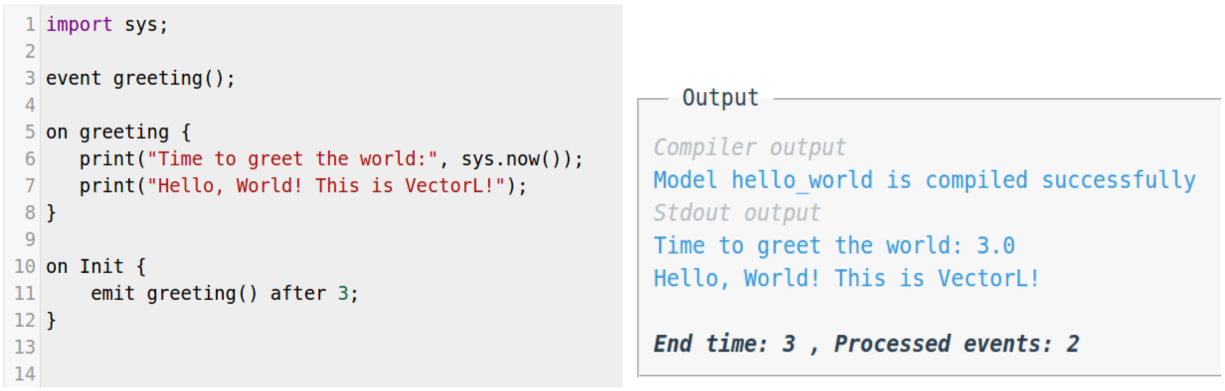
Each VectorL program consists and is consisted of modules. A module is a collection of import statements, which are used to import other modules into the current module so as to add functionality to it, and declarations of variables, events, actions and named expressions.

So, this program consists a module whose name is `hello_world`. This module’s name was declared during module’s creation in the VectorLEditor. `hello_world` module is consisted of the import of `sys` module and the declaration of `Init` action.

`Init` is a name imported from module `sys` and it is a special event that is declared in `sys` module. This special event is emitted at the beginning of the execution of the program. Once an event is emitted in VectorL, the declared actions that handle this event are called. We can see such an action declaration in line 3 of our program. `on Init` is an action, an event handler for the special event `Init`.

So, `Init` event will be emitted (triggered) at the beginning of the execution of the program and its event handler (aka action - from now on, terms `action` and `event handler` will be used interchangeably) will be executed. This is a quite simple event handler that just prints the `Hello, World! This is VectorL!` message to the standard output file. We can see the output of the execution of this program in the right side of Figure 20. Once action `on Init` prints the greeting, it finishes its execution. As long as there are no more events to process, the program's execution will end too.

That was a quite simple example, which introduced the notions of module, module import and action. Yet, we have not seen anything about how we declare events and how they are emitted in VectorL. Let's say that we want to observe the world for a while first, for like 3 seconds, and then say hello to it. Figure 21 shows how we will accomplish this.



```
 1 import sys;
 2
 3 event greeting();
 4
 5 on greeting {
 6    print("Time to greet the world:", sys.now());
 7    print("Hello, World! This is VectorL!");
 8 }
 9
10 on Init {
11     emit greeting() after 3;
12 }
13
14
```

```
Output
Compiler output
Model hello_world is compiled successfully
Stdout output
Time to greet the world: 3.0
Hello, World! This is VectorL!

End time: 3 , Processed events: 2
```

FIGURE 21 DECLARING EVENTS

Let's examine our new program. At line 1, import statement was a little bit changed as we do not want to import only `Init` event from module `sys` but its whole functionality (in line 6 we use built-in function `now` from this module).

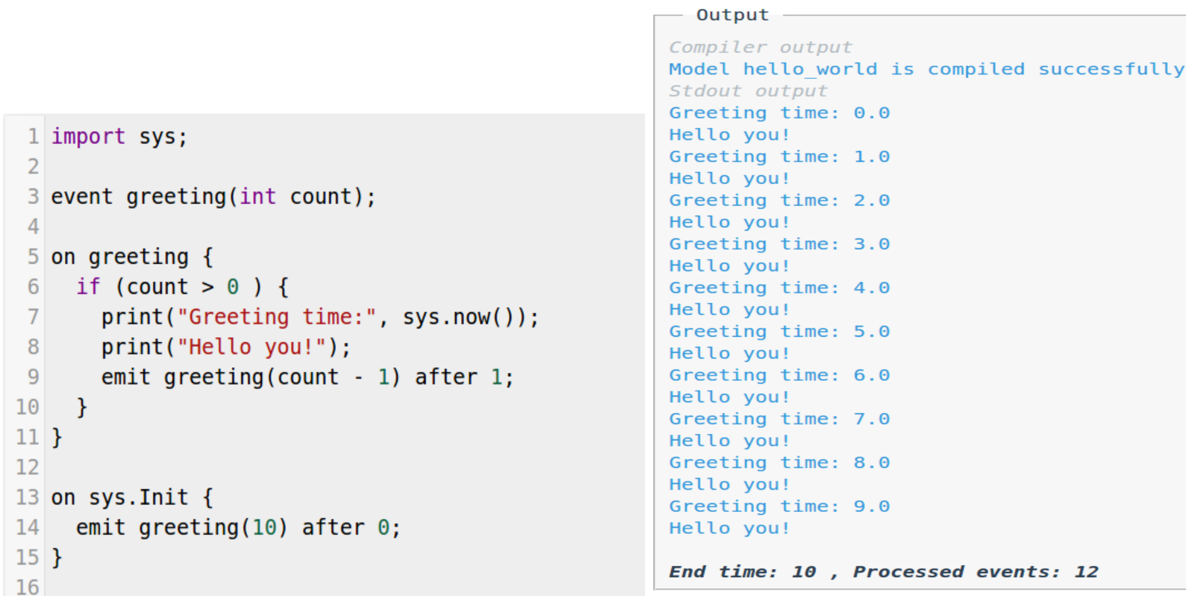
At line 3, we have our first event declaration. We declare an event named `greeting` which takes no arguments. At line 5, there is an action declaration. This action will handle `greeting` event and will print the time at which the action is executed and then the `Hello, World` message.

So, till now we have declared one event and the action that will handle it. Now, let's see how events are emitted. At line 11, we can see how an emit statement looks like. The syntax of the emit statement has two parts: the event (and its parameters) to be emitted, which looks like a function call but is very different from a function call, and an `after` clause. In the `after` clause we specify a delay. This delay is what makes event emission different from function calls.

By specifying the delay, we are introducing into our simulation the concept of time. Indeed, events are processed in order, based on the time of emission, and the `after` clause specifies that the time of emission is 3 time units from now.

So returning to our program the emit statement at line 11, tells `Init` event handler to emit the `greeting` event after 3 seconds (in DPCM simulations time is always measured in seconds). In the right half of Figure 21 we can see the output of the execution of this VectorL program. As output denotes, 2 events were processed: the Init event at first and the greeting event later. Init event handler scheduled greeting event to be emitted after 3 seconds. Once the greeting event was emitted at time 3.0 (simulation time), greeting event handler was executed and said `Hello` to world.

As already was mentioned at the introduction of the VectorL language in Chapter 1, there are no loops in VectorL. So let’s write a last simple program that will help us start thinking in the VectorL way about writing iterative code. Following our “Hello, world” program, let’s say that we met a few (10) nice people out there and we want to say hello to each one of them. Since there are no loops in VectorL how are we going to do this? Hard-coding 10 messages is a viable solution but a really bad practice in programming – what if we had met 1000 nice people? In Figure 22 we can see how we can iterate in VectorL using events.



```
import sys;

event greeting(int count);

on greeting {
  if (count > 0 ) {
    print("Greeting time:", sys.now());
    print("Hello you!");
    emit greeting(count - 1) after 1;
  }
}

on sys.Init {
  emit greeting(10) after 0;
}

```

```
Output
Compiler output
Model hello_world is compiled successfully
Stdout output
Greeting time: 0.0
Hello you!
Greeting time: 1.0
Hello you!
Greeting time: 2.0
Hello you!
Greeting time: 3.0
Hello you!
Greeting time: 4.0
Hello you!
Greeting time: 5.0
Hello you!
Greeting time: 6.0
Hello you!
Greeting time: 7.0
Hello you!
Greeting time: 8.0
Hello you!
Greeting time: 9.0
Hello you!

End time: 10 , Processed events: 12
```

FIGURE 22 ITERATIONS IN VECTORL

There are not much changed to our new program. The only difference is that greeting event now accepts the integer parameter count. In greeting event handler we check this parameter’s value and if it is greater than 0 we print the greeting and emit a new greeting event after 1 second. When emitting the first greeting event we initialize count parameter to 10. Every subsequent greeting event will be called with count parameter reduced by 1. So in this way the program will iteratively print the `Hello` greeting 10 times.

Once we saw a few examples of programs written in VectorL that demonstrated us the basic concepts of VectorL let’s take a thorough look to VectorL’s syntax.

4.2 VectorL Syntax

The syntax of the VectorL is described below in EBNF code. Terminal symbols are represented in bold.

Module = (Import | Declaration)

A module is the top level block of any program in VectorL language. Each program written in VectorL consists a module is consisted of:

- Import statements that import other modules
- Declarations

Import = (**import** ID) | (**from** ID **import** ID {, ID } **“;”**)

As we can see, an import statement can be declared in two ways:

- In the first form import statement imports the module with name ID and allows access to its namespace via the dot notation (remember sys.now())
- An import of the second form binds the provided names from the selected module into current namespace

Declaration = DefExpr | DefFunc | Event | Variable | Action

The declarations allowed in the body of a VectorL program, are the following

- Declarations of named expressions
- Declarations of full-body named expression
- Event declarations
- Variable declarations
- Action declarations

DefExpr = (**const** | **def**) Typename ID “=” Expression “;”

This is the syntax of a named expression declaration. We can have two types of named expressions:

- Def named expressions (e.g. def isquare = i*i) which will be handled by the pre-processor
- Compile time constants (e.g. const pi = 3.14)

DefFunc = **def** Typename ID (ParamList) “{” DefExpr* Expression “}”

This is the syntax of a full-body named expression. These named expressions are defined in a block manner and in their body locally defined named expressions are used in computations.

```
e.g. def int isquare (int i) {
   …
}
```

Event = **event** ID “(” ParamList “)” “;”

We have already seen in the previous section how an event is declared. Its declaration is consisted of the event’s name and an optional parameter list

e.g. event greeting(int count);

Variable = **var** Typename Id “=” Expression “;”
This is a simple variable declaration. Each variable has a type, a symbolic name and is assigned a value. e.g. var int test = 1;

Action = **on** QualId Statement

We have already seen in our example, in the previous section, how actions are declared. They must declare the name of the event they handle and a block of statements that will be executed after the event’s emission.
e.g. on greeting {
  print(“Hello there!”);
}

Statement = ConcatExpression “:=” ConcatExpression “;” | **print**“(” Arguments “)” “;” | **if** “(” Expression “)” Statement { **else** Statement} | “{” (DefExpr | Statement)* “}” | **emit** QualID “(” Arguments “)” **after** Expression “;”
From the grammar above we can figure out that the statements supported in VectorL are: • Concatenation Expressions • Print statements • If statements • Blocks that contain named expression declarations or other statement declarations • Emit statements

QualID = { ID “.” } ID
This rule stands for qualified names.

ConcatExpression = Expression { **“,”** Expression }*

The concatenation expression provides array concatenation.

Expression = UnaryOp Expression
| Expression BinaryOp Expression
| **“[”** Expression { , Expression }* **“]”**
| **“(”** Typename **“)”** Expression
| Expression **“?”** Expression **“:”** Expression
| QualID **“(”** Arguments **“)”**
| Expression **“[”** IndexOp **“]”**
| **“(”** Concat Expression **“)”**
| Literal

This is a classic grammar for Expressions. As it can be seen, VectorL expressions are syntactically similar to C++ expressions. The above grammar denotes that VectorL supports:

- Unary operator (single operand) expressions
- Binary operator expressions
- Array definition expressions
- Type cast expressions
- Ternary operator expressions
- Full-body named expression call expressions
- Array indexing expressions
- Parenthesis
- Literals

Typename = **int | bool | real | time**

The data types that are supported by VectorL are integers, boolean, real and time types. Time is just a subtype of the floating-point type real. Note that VectorL does not support string constants. String constants can be used only in print statements and nowhere else.

ParamList	= [Typename ID { “,” Typename ID }*]
This is a typical parameter list grammar. When declaring a parameter list on an event or a full-body named expression declaration, we have to define each parameter’s type and name.	

Arguments	= [Expression { “,” Expression }*]
This is a typical argument list grammar consisted of comma-separated expressions.	

4.3 VectorL Semantics

Modules

As already has been mentioned, each VectorL program consists a module (*the base module*) and is consisted of imported modules. Module it is the top level block of each program and is a collection of declarations of variables, events, actions and named expressions. User can import other modules to his program using the **import** statement and by doing so he adds the imported module’s functionality to the current module.

The set of the included modules in a program is defined as the set of modules (transitively) imported by the base module (base module is included itself). Cyclical importation (module A imports module B while module B imports module A) is not allowed and will cause a compile-time error.

The inclusion order is determined by a depth-first traversal of the import graph. The order of inclusion is determined according to the following rules:

- If module A imports module B, then module B is included in the program before module A.
- If some module imports module A and then module B in its text, and including A does not cause B to be included, then A will be included before B.

In VectorL there is a special built-in module named `sys` which must always be included in any program as it contains (among others) the special Init event which actually initializes model's execution. `Sys` module is always first in the inclusion order.

Events and Actions

Simulation process is triggered by events which are emitted uisng the emit statement. An action is a statement that is executed when the corresponding event is emitted. Note that the event and the action name must be the same and that user cannot declare an action for a non-existent event.

Any number of actions can be declared for an event and in any module. When an event is triggered, all of its actions are executed. The order of execution is fixed and corresponds to the import relationship between the modules in which the actions are defined. The rules for the action order execution are the following:

- If two actions are defined in the same module, then they are triggered in the order they are defined.
- If module A is included in the program before module B, then any action defined in A is executed before any action defined in B.

However, practically, it is not a good practice to depend too much on the execution order of action. Instead, actions should be logically independent.

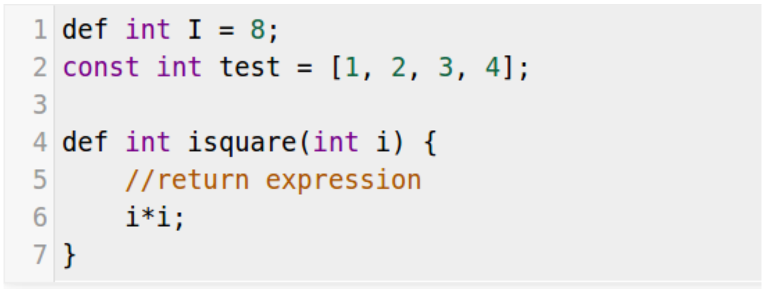
The statement of an action is interpreted inside the scope of the module it is defined in. Though, it does not have direct access to any of the declared expressions or variables of its parent module. We can pass values to the action's statement scope through the parameters of the processed event. As a consequence, in case these parameters have the same name with any declarations of the parent module, parameters will override them inside action statement's scope.

At this moment it is worth to be mentioned that events are objects that can take only scalar parameters. So it is a bad idea to think of events as objects that can carry data to their actions. What event parameters should carry is only information about *which* data to process.

Named expressions

Named expressions can be thought of as macros in C/C++. A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro.

Named expressions in VectorL are constructed with the keywords **def** or **const** and they declare a simple identifier which will be replaced by a code fragment. They are most commonly used to give symbolic names to numeric constants. Additionally in VectorL, named expressions can be declared in a function-like manner – they can accept arguments and have a full body. Note that **const** keyword can be used only to named expressions that do not have parameters. A **const** is semantically identical to a **def**; constants will be handled by the compiler during compile-time.



```
def int I = 8;
const int test = [1, 2, 3, 4];

def int isquare(int i) {
    //return expression
    i*i;
}
```

FIGURE 23 NAMED EXPRESSION EXAMPLES

Expressions

The characteristic of VectorL language is that all its values are array values. Each value has a shape which is determined at compile time. Array shape is a sequence of positive integers; its length denotes the number of the array dimensions and each tuple’s value accords to the length of the corresponding dimension. Those values whose shape is the empty list are called scalars.

In VectorL language there are no loops so in order to ease computations, all operators operate their operands in a point-wise manner. Two operands need not have identical shape in order for an operation between them to make a perfect sense. However, the shapes of the two operands have to be conformable. Two shapes are conformable if their first dimensions match, their second dimensions match, and so on up to the number of dimensions in the array with the fewer dimensions.

Two array dimensions match if either of the following is met:

- The dimensions have the same length
- One of the dimensions has unit length (1 element)

Unit length or missing dimensions are broadcast (by copying the single value) to the length of the corresponding dimension of the other operand. The result of the operation will have the number of dimensions of the higher rank operand, and the length of each dimension is the longer of the lengths in the two operands.

For example, a shape (3,4) is broadcastable with the following shapes: (4), (2, 3, 4), (3, 1), and (). Though, it is not broadcastable with the shapes (3), (2, 4) or (4, 3). Scalars are broadcastable to any shape.

In an expression if operands are not of the same shape nor are they broadcastable, a compile error will occur.

The operators supported from VectorL are the standard C/C++ operators except for the following three newly introduced array-specific operators:

Array definition expressions

The expression "[E1, E2, …, En]", where Ei are all broadcastable to shape s, is a new array with shape equal to prepending n to s. This operator is the standard array constructor.

So, for example, [1, 2, 3] is an array of shape (3), while [1, [1, 2], [3, 4]] is an array of shape (3, 2).

Array concatenation expressions

The concatenation of n arrays is written as "(E1, E2, …, En)". The shapes of the operands have to be compatible in every dimension except the first. Then, the concatenation operator concatenates the operands on the first dimension.

For example, ([1, 2], [3, 4]) concatenation will result to the array [1, 2, 3, 4], while ([[1, 2], [3, 4]] , [[5, 6], [7, 8], [9, 10]]) will result in [[1,2], [3, 4], [5,6], [7,8], [9,10]].

Array indexing expressions

The indexing operator is a powerful way to both alter the shape of an array and obtain a subset of its values. We shall start with a few examples, where we assume that A is a (5, 4) – shaped array.

A[2]	is a (4)-shaped array containing row 2 of A
A[4,3]	contains the bottom-right element of array A
A[0:3, 2]	is a (3)-shaped array containing the first 3 elements of the second column
A[:,:]	is the array A itself
A[…, _]	is a (5,4,1)-shaped array, with contents equal to A

Formally, x[index1, index2, …, indexN] forms a subarray of the array x. Each index corresponds to one dimension of the x array – the only exception is when the index operator is the ellipsis "…" which matches a number of dimensions.

Possibilities of indexI are:

- i: where i is scalar – this index operator selects the corresponding element of the corresponding dimension and deletes the dimension from the result's shape.
- start:end:step : where start, step, end are scalars – this index operator defines a slice of the array on the corresponding dimension. The step may be negative but not zero. An omitted step defaults to 1. If step is not omitted but start and end are, the start defaults to 0 and end to the dimension's length if the step is positive. In the same case, if the step is negative then start defaults to the dimension's length – 1,while the end defaults to -1. So in case A is (5)-shaped array:
    - A[::2] is equal to A[0:5:2]
    - A[::-2] is equal to A[4:-1:-2]
    - A[1:3] is equal to A[1:3:1]
- _: this is the pseudo-index operator which inserts a unit length dimension in the result. This dimension was not present in the original array.
- …: the ellipsis operator forms the rubber-index which matches zero or more dimensions of x. This operator is particularly useful when we are indexing arrays whose shape is unknown.

In some rare cases, the compiler may signal an error to a syntactically legal indexing operator. This happens if the compiler cannot infer the size of the indexing expression for some dimension. For example, A[i,j] where i and j are variables, is syntactically legal, but the compiler cannot infer the shape of the result at compile time. However, A[i: i+4] has a well defined shape that the compiler will infer.

Statements

Emit statement

An **emit** statement emits a new event, which will be triggered (and as a consequence its actions will be executed) with a desired delay, in terms of simulation time. The time specified in the after clause of the emit statement is the trigger delay, that is, the delay after which the event will be dispatched, and must be a scalar value.

In case we want the event to be dispatched immediately, zero or a negative value can be provided.

Assignment statements

Assignments in VectorL are similar in concept to assignments in other languages. However, the array semantics of VectorL make assignments more powerful.

As in other imperative programming languages, an assignment statement assigns the value of the right hand-operand to the storage location named by the left-hand operand. Therefore, the left-hand operand of an assignment must be a modifiable l-value (l-value is a value which points to a place of storage, potentially allowing new values to be assigned).

In VectorL, an l-value is defined as follows:

- A named variable (defined in any module) is an l-value.
- If A is an l-value and I = [I1, I2, …, In] is any indexing operator, then the expression A[I1, I2, …, In] is an l-value.

- If A1, A2, …, An are l-values with equal (not just broadcastable) shapes, then the concatenation expression (A1, A2, …, An) is an l-value.

In other words, the left-hand operand of an assignment must not be an array or a named expression.

Of course, the right hide side of an assignment must be of the same (or castable) type and of the same (or broadcastable) shape, or a compile-error will occur.

If statement

The **if** statement takes a scalar value in the parenthesis and, similar to C or C++ evaluates whether it is true or not by comparing in to zero. Each if statement can be optionally followed by the **else** keyword and another statement. This way user can form easily *if … else if … else* statements.

Print statement

The **print** statement is used to output information to the “standard output file”. It can accept an arbitrary number of operands. Operands can be arrays of any type and shape or they can be string literals.

When we were examining the data types supported by VectorL we noticed that character strings were not included in them. String literals may only appear as arguments in **print** statements.

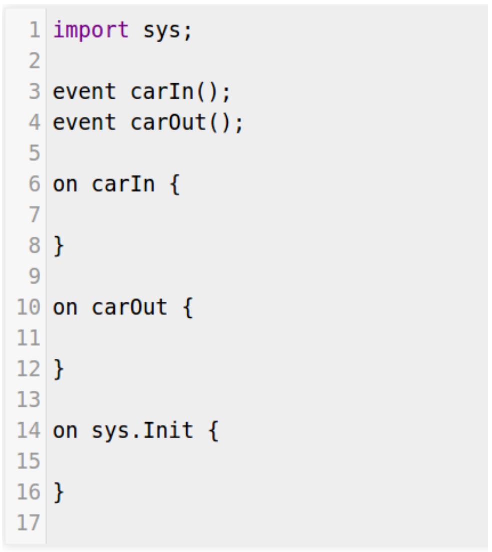
Block statement

A block statement is a possibly empty list of statements and named expression declarations. As in other programming languages, named expressions defined in a block statement are only valid in the scope of the block statement.

4.4 Parking environment model full example

Now that we have examined both the syntax and the semantics of VectorL language it is time to write an environment model for the parking demonstrator application and simulate its operation. To keep things simple we will assume that our parking has only two spots which will be visited by a few cars. The expected result of this simulation is to acquire an array that will hold the measurements of the two sensor nodes (one for each spot) over time.

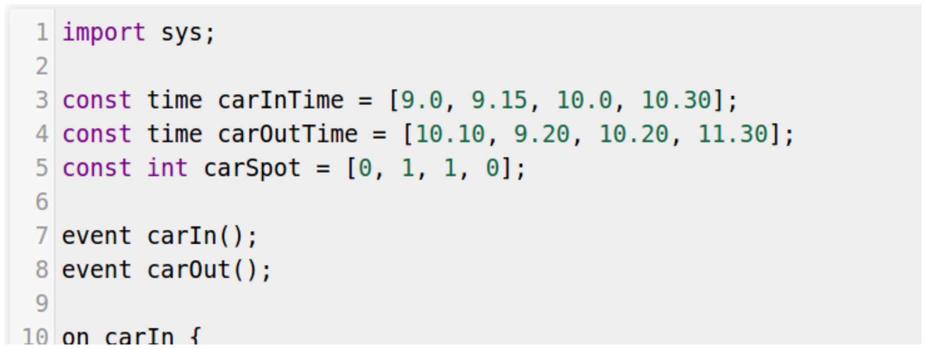
Let's start by defining the events that can take place in an application like this one. The events that affect our sensor readings are the event of a car entering the parking and the event of a car leaving the parking. So, let's declare those events and the corresponding actions (event handlers).



```
import sys;

event carIn();
event carOut();

on carIn {

}

on carOut {

}

on sys.Init {

}

```

FIGURE 24 PARKING EVENTS DECLARATION

In order to simulate our parking environment we need some input: some cars! For this purpose we will define 3 arrays. The size of these arrays will be 4, equal to the number of the cars that we want to visit our parking. The first array will provide us with the time of arrival of each car, the second with the time of departure of each car, while the last one will hold the number of the parking slot occupied by the specified car.

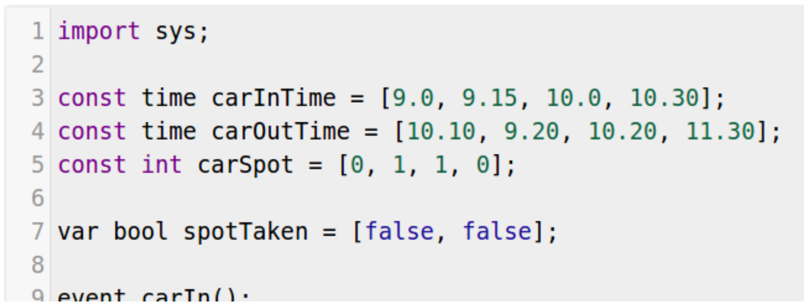


```
import sys;

const time carInTime = [9.0, 9.15, 10.0, 10.30];
const time carOutTime = [10.10, 9.20, 10.20, 11.30];
const int carSpot = [0, 1, 1, 0];

event carIn();
event carOut();

on carIn {
```

FIGURE 25 PARKING INPUT DATA

In Figure 25 we can see those arrays defined, so now we know that the first car came into the parking at time 9, left the parking at time 10.10 and occupied the spot 0, etc. Note that these arrays are declared as const named expressions. This tells to the compiler that the corresponding names are not variables but constant named expressions.

Now we have to define the variable that will hold for each sensor its readings at each time. This variable will be mapped to a sensor and will feed this sensor with values during the simulation (remember The Environment section of NSD Editor in Chapter 3). Back to our example, this variable will be a Boolean array of size 2 (one position for each sensor).

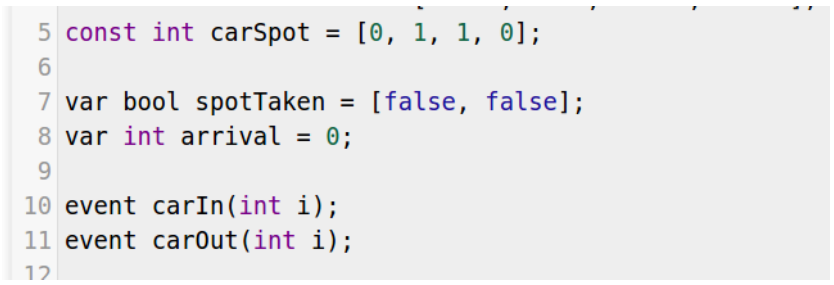


```
import sys;

const time carInTime = [9.0, 9.15, 10.0, 10.30];
const time carOutTime = [10.10, 9.20, 10.20, 11.30];
const int carSpot = [0, 1, 1, 0];

var bool spotTaken = [false, false];

event carIn();
```

FIGURE 26 OUTPUT VARIABLE SPOTTAKEN

spotTaken[0] and spotTaken[1] are false denoting that spots 0 and 1 are currently not occupied. In VectorL, as we already have seen, there is no need to declare the size of the array nor do we need to declare that spotTaken is an array type; remember that everything in VectorL is an array.

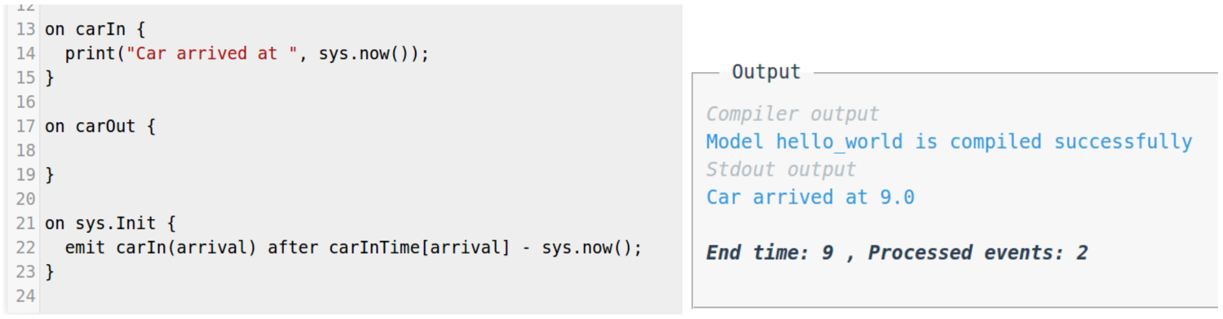
So let’s schedule in Init action our first car arrival! To do this we need to let carIn action know which car has arrived. For this purpose we will alter carIn event a little and make it to accept a parameter that will correspond to the number of the arrival currently processed (i.e. the index of our input arrays to be processed). A global variable named `arrival` that will count the processed events (car arrivals) so far will help us on this.



```
const int carSpot = [0, 1, 1, 0];

var bool spotTaken = [false, false];
var int arrival = 0;

event carIn(int i);
event carOut(int i);
```

FIGURE 27

In order to schedule our first event we need to change Init action and write an emit event in it which will trigger the carIn event when the first car arrives. This delay corresponds to the car arrival time minus current simulation time.



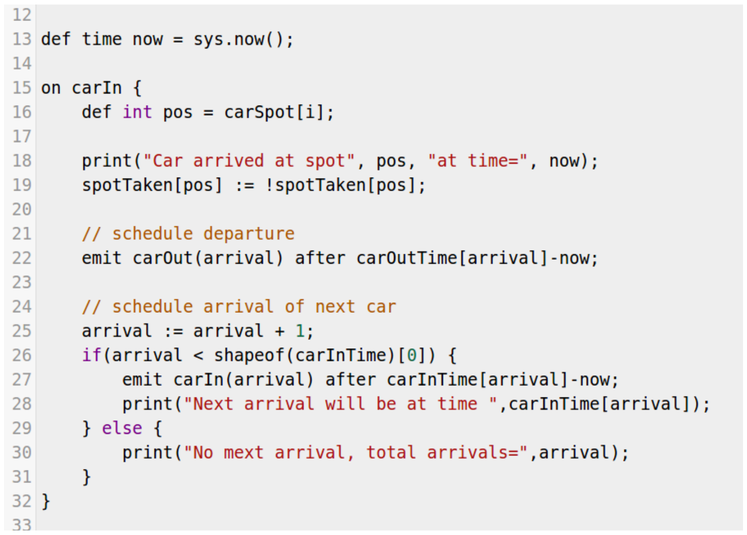
```
on carIn {
  print("Car arrived at ", sys.now());
}

on carOut {

}

on sys.Init {
  emit carIn(arrival) after carInTime[arrival] - sys.now();
}

```

```
Compiler output
Model hello_world is compiled successfully
Stdout output
Car arrived at 9.0

End time: 9 , Processed events: 2
```

FIGURE 28

As we can see in the output (Figure 28), the carIn event was triggered the expected time of the first car’s arrival.

But now let’s complete carIn action’s functionality. When a car arrives at the parking, we want to check to which slot it went, and update the corresponding sensor’s reading. Also we want to schedule this car’s departure event, as well as the arrival of the next car (if there are more cars to arrive).

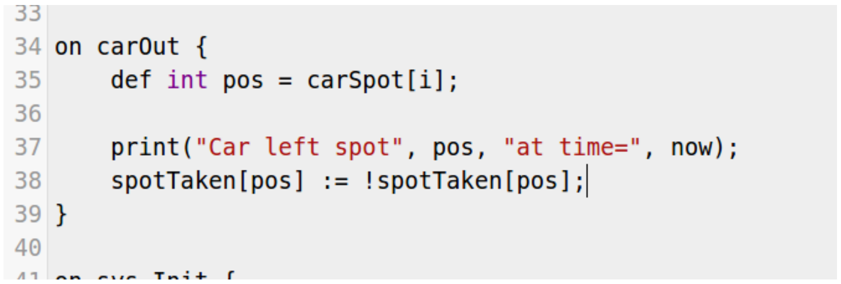


```
12
13 def time now = sys.now();
14
15 on carIn {
16     def int pos = carSpot[i];
17
18     print("Car arrived at spot", pos, "at time=", now);
19     spotTaken[pos] := !spotTaken[pos];
20
21     // schedule departure
22     emit carOut(arrival) after carOutTime[arrival]-now;
23
24     // schedule arrival of next car
25     arrival := arrival + 1;
26     if(arrival < shapeof(carInTime)[0]) {
27         emit carIn(arrival) after carInTime[arrival]-now;
28         print("Next arrival will be at time ",carInTime[arrival]);
29     } else {
30         print("No mext arrival, total arrivals=",arrival);
31     }
32 }
33
```

FIGURE 29 COMPLETED CARIN ACTION

Notice, that in line 13 we added the non-constant named expression `now` which replaces `sys.now()` expression throughout the code. Also, note at line 26 the use of `shapeOf` function. This function is a built-in function provided by the `sys` module, which accepts an array and returns its shape (the sizes of each dimension).

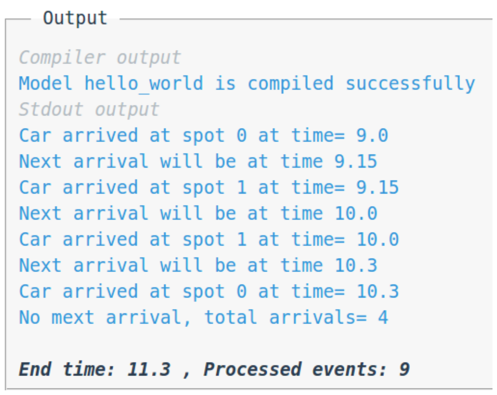
To complete our parking model, we need to implement the carOut action. When a car leaves our parking we want to check which spot it was parked into and update the corresponding sensor’s reading.



```
on carOut {
    def int pos = carSpot[i];

    print("Car left spot", pos, "at time=", now);
    spotTaken[pos] := !spotTaken[pos];
}
```

FIGURE 30 COMPLETED CAROUT ACTION

In Figure 31 the output of the parking model code execution is shown, and as we can see all the events were executed in the right order.



```
Output
Compiler output
Model hello_world is compiled successfully
Stdout output
Car arrived at spot 0 at time= 9.0
Next arrival will be at time 9.15
Car arrived at spot 1 at time= 9.15
Next arrival will be at time 10.0
Car arrived at spot 1 at time= 10.0
Next arrival will be at time 10.3
Car arrived at spot 0 at time= 10.3
No mext arrival, total arrivals= 4

End time: 11.3 , Processed events: 9
```

FIGURE 31 PARKING MODEL EXECUTION OUTPUT

Chapter 5 Implementation

In this thesis, VectorL domain specific language was developed. The development of a domain specific language typically involves the following steps:

- **Analysis**
    - Identify the problem domain
    - Gather all relevant knowledge in this domain
    - Design a DSL that concisely describes applications in the domain.
- **Implementation**
    - Design and implement a compiler that translates DSL programs into another desired programming language.
- **Usage**
    - Write DSL programs for all desired applications and compile them.

In the previous chapters we thoroughly analyzed VectorL's significance, its syntax, its semantics and demonstrated a few examples of VectorL being used in practice. In this chapter we will focus on how VectorL's compiler (vlc) was implemented.

5.1 Compilation process

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another.

The *lexical analysis* is the first phase of a compiler. Its main task is to read input characters and produce as output a sequence of tokens that the parser uses for the next phase, the syntax analysis.

Every programming language has rules that prescribe the syntactic structure of well-formed programs. We examined VectorL's syntactic rules in section 4.2. During *syntax analysis* (or *parsing*) phase the tokens of the source program that were produced from the lexical analysis are grouped into grammatical phrases. Those grammatical phrases formulate the syntax tree or the abstract syntax tree (AST) which is a tree representation of the syntactic structure of the source code. Each node of the tree denotes a construct occurring in the source code.

After the construction of the AST, *semantic analysis* is performed. During this phase the compiler adds semantic information to the parse tree and performs semantic checks, such as type checking, rejecting incorrect programs or issuing warnings.

The last phase of the compilation process is that of *code generation* during which the above intermediate representation of the source code is converted to into a form that can be readily executed by a machine.

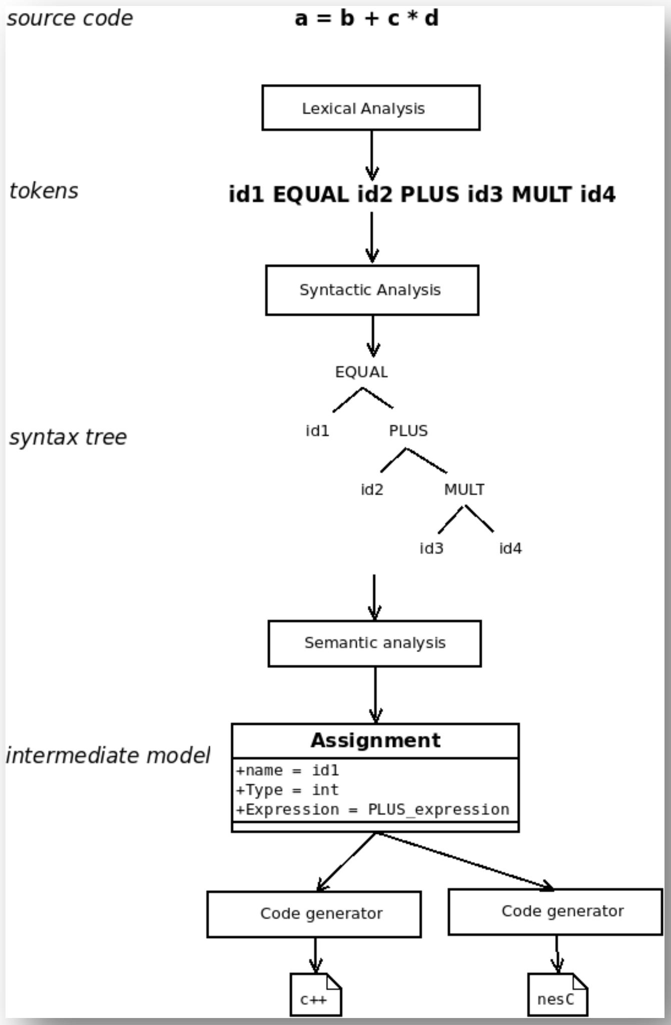


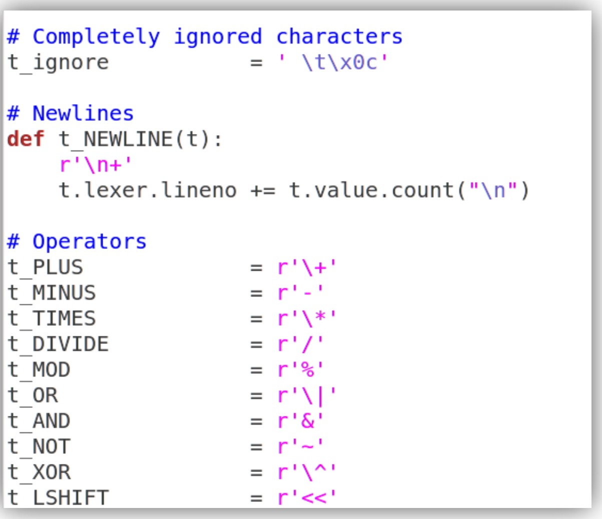
FIGURE 32 COMPILATION PROCESS

5.2 VectorL Lexical, Syntactic and Semantic Analysis

For the lexical and syntactic analysis of VectorL PLY library was used. PLY is a Python implementation of the popular compiler construction tools lex and yacc.

In order the lexical analysis to be performed, lex's lexer must be provided a list of tokens that defines all of the possible token names that can be produced by the lexer. These tokens are described using regular expressions.

So during the lexical analysis, lex initially tokenizes the source code. So, for example the following input string: x = 3 + 43 will be splitted into the following individual tokens: 'x', '=', '3', '+', '43'. Next, lex tries to match those tokens to the regular expression rules we defined so that it can give meaningful names to them. So, the result we be 'ID', 'EQUALS', 'INTEGER', 'PLUS', 'INTEGER'.



```
# Completely ignored characters
t_ignore           = ' \t\x0c'

# Newlines
def t_NEWLINE(t):
    r'\n+'
    t.lexer.lineno += t.value.count("\n")

# Operators
t_PLUS             = r'\+'
t_MINUS            = r'-'
t_TIMES            = r'\*'
t_DIVIDE           = r'/'
t_MOD              = r'%'
t_OR               = r'\|'
t_AND              = r'&'
t_NOT              = r'~'
t_XOR              = r'\^'
t_LSHIFT           = r'<<'
```

FIGURE 33 LEXER RULES EXAMPLE

Next, yacc parser comes into play. Yacc parser accepts lexer's output and a set of grammar rules along with actions which denote what is needed to be done if a particular grammar rule is recognized. Yacc uses LR-parsing which is a bottom up technique that tries to recognize the right-hand-side of various grammar rules. Whenever a valid right-hand-side is found in the input, the appropriate action code is triggered and the grammar symbols are replaced by the grammar symbol on the left-hand-side.

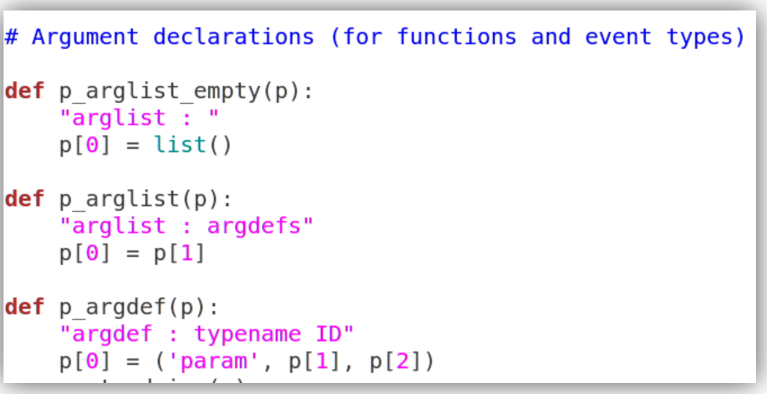

```
# Argument declarations (for functions and event types)

def p_arglist_empty(p):
    "arglist : "
    p[0] = list()

def p_arglist(p):
    "arglist : argdefs"
    p[0] = p[1]

def p_argdef(p):
    "argdef : typename ID"
    p[0] = ('param', p[1], p[2])
```

FIGURE 34 GRAMMAR RULES AND ACTIONS EXAMPLES

Yacc parser returns a list of Abstract Syntax Tree (AST) nodes which contain the clauses that were found during parsing. By default, yacc parser does not return an AST, this is something we formulated through rule's actions. To better understand yacc's output, let's see it in practice.

If we parse the simple **const** declaration `const int generation = 10;` the following AST will be produced: `AstNode('const', 'generation', 'int', AstNode('literal', 10):1):1` which means that we have a root syntax tree node which is a declaration (AST top-level node will always be either import statements or declarations) whose type is 'const', its VectorL data type is int, its name is 'generation', we meet it at line 1 and its value is given from another AstNode.

Once the syntactic analysis is completed, semantic analysis is initiated. The produced AST is parsed in a depth-first manner. For each AST node an equivalent Python object (model) is constructed which contains all the information that will be needed for the program to be executed and by the code generators in order to produce target code. During AST to Model transformation semantic checks, such as type checks, array dimension conformability checks, same variable declaration in a single scope checks, take place. If those semantic checks fail, then the compilation process is terminated and informative messages about these failures are returned.

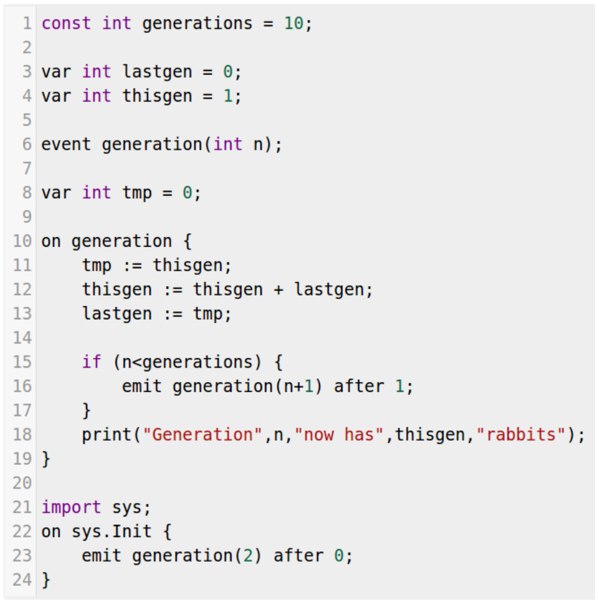
5.3 Code generation

Once the VectorL code is validated and its Model representation is ready, the compiler may proceed to code generation. One model can be translated to several target languages, for each one of which we need to have the corresponding code generator implemented.

VectorL can generate both C++ and nesC code. C++ code will be used in OMNET++ simulations using Castalia WSN simulator, while nesC code can be used for both simulations using TinyOS's simulator TOSSIM (or any other simulator that can simulate TinyOS applications) and for running into the motes during field testing.

C++ and nesC differ as languages but the code produced by VectorL compiler is very similar for both of them. The idea is that each module that participates in a VectorL model is represented by a struct both in C++ and nesC. This struct's members are the variables declared on the corresponding module. The actions of a module are declared as functions inside the corresponding struct. But let's make it more clear with an example.

Let's say that we have a program in VectorL called Rabbits which calculates Fibonacci numbers:



```
const int generations = 10;

var int lastgen = 0;
var int thisgen = 1;

event generation(int n);

var int tmp = 0;

on generation {
    tmp := thisgen;
    thisgen := thisgen + lastgen;
    lastgen := tmp;

    if (n<generations) {
        emit generation(n+1) after 1;
    }
    print("Generation",n,"now has",thisgen,"rabbits");
}

import sys;
on sys.Init {
    emit generation(2) after 0;
}
```

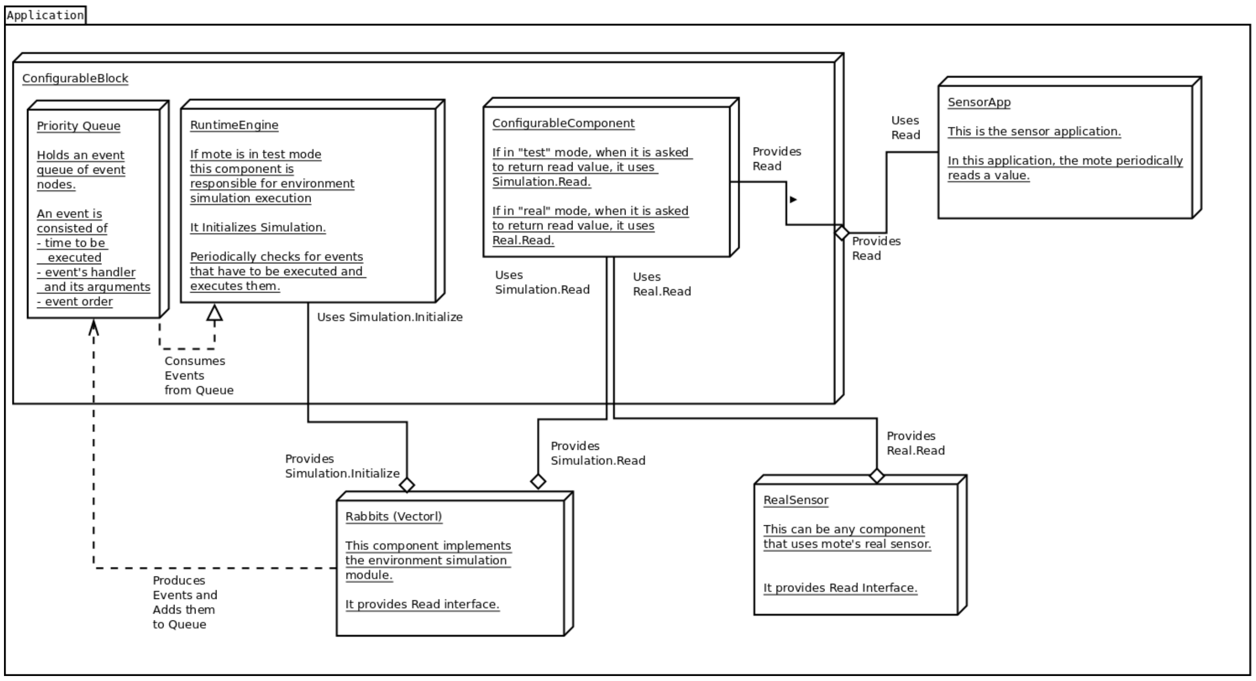
FIGURE 35 RABBITS PROGRAM

This program's representation in both languages will look like the following code:

```c
struct _rabbits_ {
  struct _model_rabbits_ {

    int lastgen;
    int thisgen;
    int temp;
    const int generation;
  }

  struct _model_sys_ {
  }

  void on_rabbits_generation(int n) {
    // do calculations and if n<generation
    // push generation event to event queue
  };

  void on_sys_init() {
    //push generation event to event queue
  }
}

```

FIGURE 36 RABBITS GENERATED CODE

Figure 37 explains how this code can be used inside a TinyOS application, in order to feed a node with measurements (this sensor will count rabbits!) during field testing.



Priority Queue: Holds an event queue of event nodes. An event is consisted of - time to be executed - event's handler and its arguments - event order

RuntimeEngine: If mote is in test mode this component is responsible for environment simulation execution. It Initializes Simulation. Periodically checks for events that have to be executed and executes them.

ConfigurableComponent: If in "test" mode, when it is asked to return read value, it uses Simulation.Read. If in "real" mode, when it is asked to return read value, it uses Real.Read.

SensorApp: This is the sensor application. In this application, the mote periodically reads a value.

Rabbits (VectorI): This component implements the environment simulation module. It provides Read interface.

RealSensor: This can be any component that uses mote's real sensor. It provides Read Interface.

FIGURE 37 TINYOS EXAMPLE

5.4 VectorL Interpreter

As already has been mentioned, when user chooses to run the VectorL code he wrote in VectorL Editor, the output he sees is not the output of a real simulation. His VectorL code is not yet translated to C++ or nesC so no simulator (Castalia/Tossim) is used for its execution.

In order to facilitate user check his program correct execution prior to simulation phase, an interpreter was developed for VectorL. This interpreter accepts the intermediate Model that was constructed by the compiler, compiles it into a stack machine program.

That program will be later loaded and executed by the implemented stack machine. The stack machine holds an operation and a data stack, as well as an event queue. The event queue is a priority queue where priority is the time at which an event must be executed.

First, \`sys.Init\` event is emitted and added in the event queue. From that point, the stack machine starts to iterate over the queue and to process the events in it. When an event is processed its event handler's program instructions are loaded on the operation stack and the stack machine starts to execute them. Any program instructions that correspond to emit statements will add events to the event queue and they will be processed when they become the top events in the priority queue, the stack machine will execute their handler's code and so on.

5.5 Executing VectorL compiler

VectorL's compiler vlc can be used, of course, standalone, outside of DPCM platform's context. User can compile any VectorL program by using `./vlc [options] src` command where `src` is the file path of VectorL's source code. The valid options supported by vlc compiler are the following:

--path / -p	With this option user can provide a list of directories separated by **":"**. Compiler will search in these directories for VectorL files.
--until / -u	Provide the simulation time at which the simulation will end. The default is 'no limit'

--steps / -s	Provide the maximum number of steps (events) to be processed during simulation. The default is ‘no limit’.
--compile / -c	Only compile the given file, do not run it.
--gen /-g	Provide the generator to be used. Compiler will generate code from VectorL model, but it will not run it.

Chapter 6 Conclusion

6.1 Results

In this thesis, VectorL was introduced, a Domain Specific Language that provides a way to easily and safely model physical processes for WSN applications. Environment modeling is of crucial importance during a WSN's simulation as it contributes to the production of more realistic and accurate simulation results. Despite its significance, physical process modeling is neglected by other WSN simulators.

VectorL design choices, such as avoiding looping constructs and data types that are not needed for simulation, make VectorL programs much easier to debug and execute safely inside a simulation.

After the specification of VectorL's syntax and semantics, a compiler was developed that validates and translates VectorL programs into Models which will consist later the input of the code generators. Two code generators were implemented for VectorL: the first one generates C++ code to be used for network simulations by the Castalia simulator and the second one generates NesC code which can be used for network simulations using TOSSIM and for running inside the motes during field testing (in case of course TinyOS platform is used for the application's development).

6.2 Future work

There are several extensions that can be applied to the existing work that will raise VectorL's effectiveness. On the one hand, more code generators can be written in order to support more simulators (other than Castalia) and other platforms (other than TinyOS).

One the other hand, in the case of field testing, where the VectorL code runs inside the motes, it would be extremely effective if each mote contained only the code fragments and values that contribute to its own sensor reading production. For example, let's say that we execute a VectorL program and the sensor readings produced are stored in array A of shape (5). A[0] contains the reading of node 0 at that time, A[1] contains the reading of node 1 at that time and so on. And let's say that this array was constructed by the point-wise addition of arrays B[5] and C[5] (this is a quite simple example). Wouldn't it be extremely effective if during field testing each node contained only the data from B and C that contribute to the production of its own reading? So in case of mote 0, instead of having a program that defines two arrays B and C of shape (5), adds them and returns an array A from which node 0 will use only A[0] value, to have a program that defines two arrays B and C of shape () (scalars) and produce only the reading for the specific sensor. In such a scenario, a lot of memory usage would have been saved which is extremely critical when we are talking about wireless sensor networks.

Table of figures

Bibliography

Aho, A., Lam, M., Sethi, R., & Ullman, J. (2006). *Compilers, Principles, Techniques and Tools.* Addison Wesley.

An Introduction to Numpy and Spicy. (n.d.). Retrieved from http://www.engr.ucsb.edu/~shell/che210d/numpy.pdf

Bakshi, A., Pathak, A., & Prasanna, V. (2005). System-level support for macroprogramming of networked sensing applications.

Beutel, J., Plessl, C., & Wohrle, M. (Mar.2007). Increasing the reliability of wireless sensor networks with a unit testing framework. *Computer Engineering and Networks Lab, Tch. Report 272* .

Boulis, A. (2009). Castalia: A simulator for wireless sensor networks and body area networks.

Cardell-Oliver, R., Smettem, K., Kranz, M., & Mayer, K. (n.d.). Field testing a wireless sensor network for reactive environment monitoring.

Clouser, T., Thomas, R., & Nesterenko, M. (2007). Emuli: Emulated Stimuli for Wireless Sensor Network Experimentation.

Downard, I. (2004). Simulating Sensor Networks in NS-2. *Naval Reasearch Laboratory* .

Gay, D., Levis, P., Von Behren, R., Welsh, M., Brewer, E., & Culler, D. (2003). The nesc language: A holistic approach to networked embedded systems.

Gummadi, R., Gnawali, O., & Govindan, R. (2005). Macro-programming wireless sensor networksa using Kairos.

Hnat, T., Sookoor, T., Hooimeijer, P., Weimer, W., & Whitehouse, K. (2008). Macrolab: A vector-based macroprogramming framework for cyber-physical systems.

Jia, D., Krogh, B., & Wong, C. (2005). TOSHILT: Middleware for hardware in the loop testing of wireless sensor networks.

Levis, P., Lee, N., Welsh, M., & Culler, D. (2003). Tossim: Accurate and scalable simulation of entire tinyos applications. *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems* .

Levis, P., Madden, S., Polastre, J., Szewczyk, R., & Whitehouse, K. (2005). Tinyos: An operating system for sensor networks.

Luo, L., He, T., Zhou, L., Gu, L., Abdelzaher, T., & Stakovic, J. (2006). Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. (In 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)).

Mainwaring, A., Polastre, J., Szewczyk, R., Culler, D., & Anderson, J. (2002). Wireless Sensor Networks for Habitat Monitoring.

Mottola, L., & Picco, G. P. (n.d.). Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art.

Newton, R., & Welsh, M. (2004). Region Streams: Function Macroprogramming for Sensor Networks.

Newton, R., Arvind, & Welsh, M. (2005). Building up to macroprogramming: An intermediate language for sensor networks.

Pathak, A., Mottola, L., Bakshi, A., Prasanna, V., & Picco, G. P. (2007). Expressing Sensor Network Interaction Patterns using Data-Driven Macroprogramming.

Sundani, H., Li, H., Devabhaktuni, V., Alam, M., & Bhattacharya, P. (2011). Wireless Sensor Network Simulators A Survey and Compariosns.

Sundresh, S., Kim, W., & Agha, G. (2004). SENS: A Sensor, Environment and Network Simulator. *Proceedings of the 37th Annual Simulation Symposium (ANSS'04)* .

Van Deursen, A., Klint, P., & Visser, J. (n.d.). Domain-Specific Languages: An Annotated Bibliography.

Yang, X., Xu, M., & Stickney, P. (n.d.). SimX: An Integrated Sensor Network Simulation and Evaluation Environment.

Yorick Language Reference. (n.d.). Retrieved from http://yorick.sourceforge.net/refcard/refs.pdf