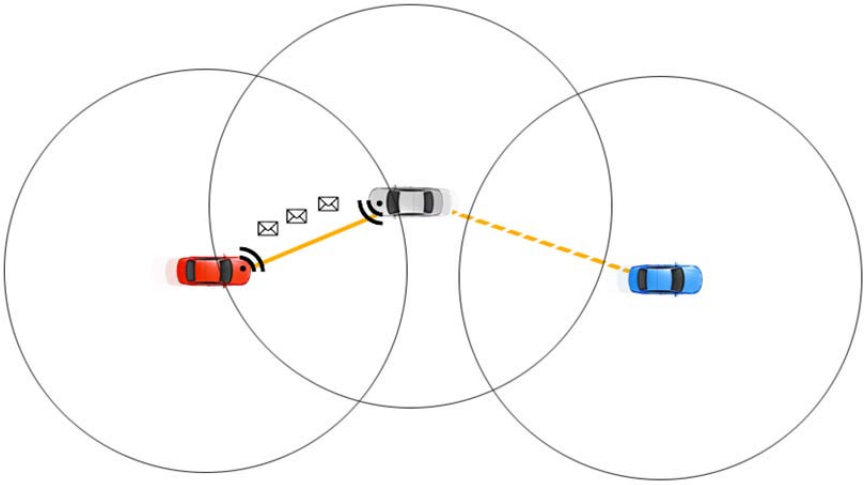
MODELLING DYNAMIC TOPOLOGIES VIA EXTENSIONS OF VDM-RT

Electrical and Computer Engineering
Technical Report ECE-TR-9



AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

DATA SHEET

Title: Modelling Dynamic Topologies via Extensions of VDM-RT
Subtitle: Electrical and Computer Engineering
Series title and no.: Technical report ECE-TR-9

Author: Claus Ballegaard Nielsen
Department of Engineering – Electrical and Computer Engineering,
Aarhus University

Internet version: The report is available in electronic format (pdf) at the Department of Engineering website http://www.eng.au.dk.

Publisher: Aarhus University©
URL: http://www.eng.au.dk

Year of publication: 2012 Pages: 79
Editing completed: July 2012

Abstract: Only a few formal methods include descriptions of the network topology that the modelled system is deployed onto. In VDM Real-Time (VDM-RT) this has been enabled for distributed systems that have a static structure. However, when modelling dynamic systems this fixed topology becomes an issue. Systems with highly distributed and alternating relationships cannot be expressed correctly in a static model. This document describes how VDM-RT can be extended with new language constructs to enable the description of dynamic reconfiguration of the network topology during the runtime execution of a model. The extension is developed on the basis of a case study involving a dynamic system that has a constant changing system topology. With a basis in the case study a model is developed that uses the static version of VDM-RT in order to reveal the limitations of the language. The case study is then revisited where the capabilities of the proposed reconfiguration extension are applied to the model, and the value of having the extension is evaluated. We expect that other notations can benefit from the experiences reported here.

Keywords VDM-RT, Dynamic topologies, Dynamic reconfiguration, software engineering and systems

Please cite as: Claus Ballegaard Nielsen, 2012. Modelling Dynamic Topologies via Extensions of VDM-RT, Department of Engineering, Aarhus University. Denmark. 79 pp. - Technical report ECE-TR-9

Cover image: Claus Ballegaard Nielsen

ISSN: 2245-2087

MODELLING DYNAMIC TOPOLOGIES VIA EXTENTIONS OF VDM-RT

Claus Ballegaard Nielsen

Aarhus University, Department of Engineering

Abstract

Only a few formal methods include descriptions of the network topology that the modelled system is deployed onto. In VDM Real-Time (VDM-RT) this has been enabled for distributed systems that have a static structure. However, when modelling dynamic systems this fixed topology becomes an issue. Systems with highly distributed and alternating relationships cannot be expressed correctly in a static model. This document describes how VDM-RT can be extended with new language constructs to enable the description of dynamic reconfiguration of the network topology during the runtime execution of a model. The extension is developed on the basis of a case study involving a dynamic system that has a constant changing system topology. With a basis in the case study a model is developed that uses the static version of VDM-RT in order to reveal the limitations of the language. The case study is then revisited where the capabilities of the proposed reconfiguration extension are applied to the model, and the value of having the extension is evaluated. We expect that other notations can benefit from the experiences reported here.

Table of Contents

List of Figures

1

Introduction

In Section 1.1 an introduction to the problem domain, scope and goal of the document is given, while Section 1.2 provides an overview of the document structure.

1.1 Primer

The advancement and growing necessity of distributed computing resources [1, 2] have led to the development process of embedded distributed systems becoming increasingly challenging. With the increasing demands the complexity of the specification and the system design increases as well. In order to naturally describe and analyse dynamic distributed systems it is essential that notations used in formal methods include features enabling this.

Formal methods can potentially be utilized at an early development stage and a model of a distributed embedded system can contribute to a greater confidence in the reliability and robustness of the design. By creating a formal model of distributed systems a formal technique, such as VDM-RT, offers the possibility of exploring and validating the system design and behaviour [3].

This document describes how an executable Vienna Development Method (VDM) [4] model of a case study system is developed from a static system to a dynamic system by using a reconfiguration extension of VDM-Real-time (VDM-RT).

VDM-RT is a VDM language dialect that includes the ability to model real-time systems by describing system architecture through executing units knows as **`CPU`**s and a communication medium named **`BUS`**. VDM-RT has quantifiable time with relation to the execution of statements in a model, meaning it is possible to model detailed timing with respect to execution on the individual CPUs defined in the system architecture.

The case study revolves around a system named Vehicle Monitoring (VeMo) which is designed to improve road safety by increasing the traffic information available to motorists. The increased information flow is built on an intelligent traffic infrastructure along with open collaboration between motorists that join co-operative networks in which information can flow. The communication network is established rapidly and the information exchange occurs fast because of the short timespan that the vehicles are in range. The case study contains a very dynamic system where the amount of changes, the number of relations and quantity of data exchange is difficult to predict. This makes it challenging to define the system in a model using the static system architecture topology as defined in VDM-RT, as such an architecture topology does not represent the nature of the system.

Dynamic reconfiguration of a distributed application is the act of changing the configuration of the system during runtime [5] and this paper examines the possibility of enabling the description of this in a model. The main focus of this document is on the dynamic reconfiguration of the distributed system architecture, meaning the alternation of the network topology between nodes during run-time. The intention is to enable the model to capture the fundamentals of dynamic reconfiguration, not to incorporated specific reconfiguration approaches into VDM-RT.

The goal of the VDM-RT extension is to provide the means for creating high-level and abstract models of complex reconfigurable distributed systems. The goal is not to deliver completeness or proof of an entire system functionality or fulfilment of system requirements, but to aid system design decisions by creating an overview and identifying potential functionality and design flaws in the early development phases. The analysis and evaluation of these system properties have thier foundation in the simulation of the system through a VDM-RT model expressed in an executable subset. However, it is the aspiration that other notations can benefit from the experience of the proposed VDM-RT extension.

This work builds and extends work previously done in investigating dynamic reconfiguration capabilities in VDM [6].

1.2 Document Structure

The background and purpose of this document is described in Chapter 2. The case study is presented in Chapter 3 with a focus on the challenging development aspects originating from these kinds of systems and on which properties that are particular interesting to explore in an abstract model of the system. In Chapter 4 a detailed description is given of the VDM-RT model that has been created on the basis of the case study and the results of having the executing model are presented. Based on the results the qualities and limitations of using VDM-RT to model the specific type of system, found in the case study, is highlighted. In Chapter 5 a dynamic reconfiguration extension of VDM-RT is presented with a focus on changing the system architecture as well as adding and removing processing units and communication channels during the runtime execution of the system. The case study is revisited in Chapter 6 with a focus on the capabilities enabled by the dynamic reconfiguration extension and the value of having a more dynamic model is displayed. Finally, a summary of the presented material, the potential future work and concluding remarks are given in Chapter 7.

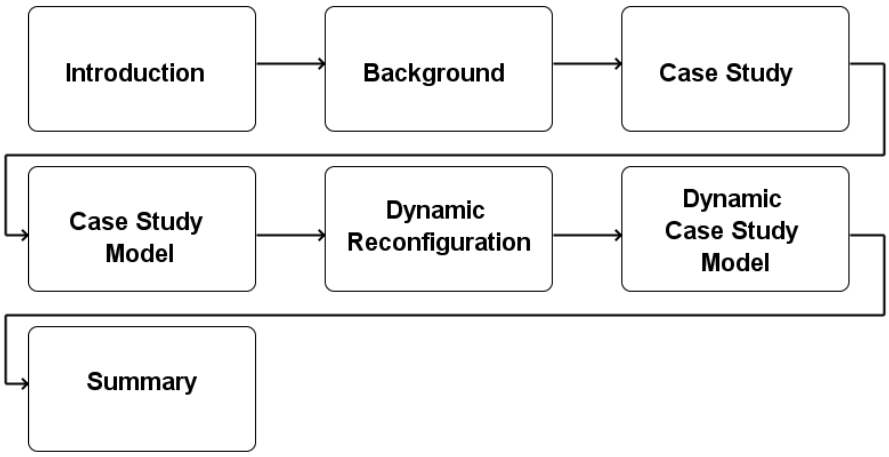


Figure 1.1: Document Overview

2

Background

Section 2.1 describes the motivation for looking into the subject of dynamic reconfiguration. An insight into the formal specification language VDM-RT is given in Section 2.2 and Section 2.3 describes the purpose and anticipated gains of changing the language by adding dynamic reconfiguration capabilities.

2.1 Motivation

To deal with the increase in the complexity of software systems the use of formal techniques for analysing and validating software specifications and designs have been widely encouraged and extensively researched [7, 8]. Formal methods have been practiced for over four decades [9] with the objective of proving correctness of algorithms and system architectures with regard to a formal specification or model.

Systems are becoming more and more complex, partly because their users are expecting an increased amount of functionality and partly because the problems the systems are meant to solve are ever so challenging. This is a time were ubiquitous and mobile computing systems are constantly expected to be able to adapt to a changing environment, and to deliver a near-seamless integration with respect to connectivity and system-to-system interaction. This raises the demands for system adaptability towards coping with systems that have a continually changing number of processing entities for which the number of relations and amount of data exchange is unpredictable.

The view on systems is also changing, with systems being considered more as services that are expected to provide aid to any number of users with specific tasks. Instead of being a committed and established capacity in the system, the users become more of a subscriber to a packaged solution. With terms such as infrastructure-, platform- and software-as-a-service, systems are becoming dynamic pluggable structures that adapt and change based on the number of users, entities and data in the system. They are expected to adapt to diverse environments and requirements that are characterized by constant change.

In order for the formal modelling languages to capture the real behaviour of the systems being developed in the world, they need the ability to express adaptability by allowing dynamic reconfiguration changes of the systems architecture and topology. Dynamic reconfiguration of a distributed application is the act of changing the configuration of the system during runtime [5].

2.2 Vienna Development Method

The Vienna Development Method (VDM) is a formal method technique for specifying, modelling, and evaluating software systems. VDM is one of the oldest formal methods, with its origin dating back to the Vienna Definition Language (VDL) and the early 1970s [10]. With the development of the process, through the addition and combination of multiple techniques, the approach was defined and named as the Vienna Development Method in 1973 [11, 12]. Since then VDM has been applied to a range of industrial projects [13]. To ensure the validity and consistency of a system specification or design, a model of the system is expressed in a formal VDM model which can then be validated, by analytic methods ranging from type checking to execution of the model, through tool support.

The basis of VDM is the ISO standardized VDM-SL notation [14, 15] which is a language for modelling functional specifications of sequential systems [16]. In order to meet new technology and the latest industrial challenges VDM has developed over time by introducing several new language dialects with extended functionality. VDM++ is an object-oriented extension of VDM in which the models consists of collections of classes [4] and the research project "VDM++ In a Constrained Environment" (VICE) introduced a timed extension to VDM++ in order to model real-time systems with respect to time.

Research [17] revealed that neither the existing VDM++ dialect nor the extension made with VICE was sufficient when modelling distributed real-time systems. As a result an extension was proposed [18, 19] to enable the modelling of distributed real-time embedded systems in VDM++. The extension introduced the notions of CPUs, busses, specific time delays and asynchronous operations. The extension made it possible to deploy individual distributed systems on separate CPUs which could by connected by busses. The extension named VDM-RT (as a replacement of the VICE notation) was implemented and validated by multiple case studies [20, 21, 22, 3, 23].

The existing VDM dialect are:

VDM-SL an ISO standardized sequential language for defining software functionality;

VDM++ which includes the fundamental functionality of VDM-SL but extends it with concurrency and object oriented design;

VDM-RT which extends VDM++ by adding timing constraints, CPUs and busses as well as distributed system design and topology.

2.2.1 VDM-RT

In VDM-RT the distributed system architecture is formed on the basis of two predefined classes; `BUS` and `CPU`, which represent communication lines and processing units respectively.

BUS represents a communication bus over which `CPU`s can be connected in order enable remote operation invocations.

To create an instance of a `BUS` three arguments must be supplied. The transfer policy which is either First Come First Served (FCFS) or Carrier Sense Multiple Access / Collision Detection (CSMACD), the speed of the `BUS` and lastly the set of the `CPU`s connected to the `BUS` instance. An example is shown in Listing 2.1.

```
public static bus : BUS := new BUS(<CSMACD>, 60E3,{cpu1, cpu2});
```

Listing 2.1: Instantiation of the BUS

CPU is a single processing unit which is capable of executing parts of a VDM model deployed to it.

An instantiation of a `CPU` requires two arguments. The first argument is the scheduling policy, which is either FCFS policy or Fixed Priority(FP), where FCFS determines a fixed number of operations or expressions that can be executed by a thread before it is swapped out and FP determines a variable number determined by a priority given to specific operations. The second argument is the processing speed in instructions per second. Listing 2.2 shows an example of `CPU` instantiation.

```
public static cpu1 : CPU := new CPU(<FCFS>, 20E6);
```

Listing 2.2: Instantiation of the CPU

The processing power of a `CPU` is employed by deploying an object through the `CPU` class' `deploy` operation. Any execution in the deployed objects member operations will occur on the `CPU`'s processing time. The VDM-RT model has a global notion of time on which all `CPU` and `BUS` processing depends, meaning that `CPU`'s will execute in parallel in relation to global time. For example if two active objects, with a processing thread each, are deployed to two different `CPU`s they will run concurrently, while if the objects were deployed to the same `CPU` they would have to share processing time.

Time is advanced in the model by a penalty being implicitly assigned to each operation call or expressions being executed. However the system developer can specify the penalty the execution of a segment in the model will have and thereby disregard the default number of cycles for each statement being executed. The `duration` statement places a fixed value on the execution time. The `cycles` statement is applied in the same way as a `duration` statement, but it defines the number of CPU cycles for the statement, rather than an absolute duration. The duration therefore depends on the CPU speed.

The distributed system itself is defined in a special **`system`** class, in which the system architecture and topology is constructed by creating relations between instances of the `CPU` and `BUS` classes. The topology configuration must be performed in the **`system`** class, and it remains static once the **`System`** is initialized.

The **system** class is created by the systems developer by using the **system** keyword instead of **class**. This imposes some limitations on the **system** class as only instance variables and a default constructur may be defined. An example of a **system** class is shown in Listing 2.3.

```
system Sys
 instance variables
  cpu1 : CPU := new CPU(<FP>, 22E6);
  cpu2 : CPU := new CPU(<FP>, 22E6);
  cpu3 : CPU := new CPU(<FP>, 22E6);
  bus : BUS := new BUS(<CSMACD>, 72E3,{cpu1,cpu2});

 operations
  public Sys : () ==> Sys
  Sys() ==
  (
   obj1.put({obj2,obj3}); --relate objects
   obj2.put({obj1,obj3}); obj3.put({obj1,obj2});
        --deploy
   cpu1.deploy(obj1);
   cpu2.deploy(obj2);
   cpu2.deploy(obj3);
  );
end Sys
```

Listing 2.3: Example of System class definition

Listing 2.3 creates three CPU's and connects two of them via a BUS. Line 12-13 creates a relation between the three objects, before they are deployed to one CPU each.
The system architecture and topology is laid out in the following way:

- A CPU is connected to another CPU over a BUS,
- the different distributed objects are connected through object references and
- finally these objects are deployed to specific CPUs.

During the execution of the model the integrity of the communication, between the distributed objects, can be validated by comparing the communication (operation calls) processing over object references with the BUS connection between CPUs. If an object deployed on one CPU communicates with an object deployed to another CPU, then the CPUs in question must be connected to the same BUS connection. Meaning that although objects are connected by object references there must be a BUS connection as well to enable communication.

This can be seen from the example in Listing 2.3, which is illustrated in Figure 2.1, where Obj1 will not be able to communicate with Obj3 because of the missing BUS connection; although they are connected by object references. Communication is point-to-point and there is no possibility of broadcasting with VDM-RT.

This construct of double layers of references is important to observe. Given that when the distributed architecture is constructed in the **system** class, the model developer is required to have knowledge of the BUS, CPU and object relationships. However, when creating the remainder of the model, the model developer can to a great extent rely on object to object interconnection without needing knowledge of the association between an object and a specific CPU. This approach provides a layer of transparency to the remote communication, because object references do not need to know how communication is conducted on the BUS. In this way the model developer can to some extent look past the CPUs and concentrate on object to object interaction.

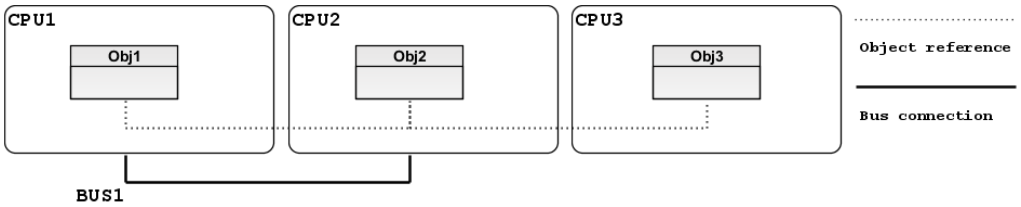


Figure 2.1: Deployment of example in Listing 2.3

When an object is created during the execution of a model it will automatically be deployed to the `CPU` on which the constructor was executed. This ensures that any processing occurring in the new object will have a time penalty and cost processing time on that `CPU`.

In a VDM-RT model a virtual `CPU` and a virtual `BUS` is always implicitly added when the model is initialized. The role of the virtual `CPU` is to enable processing that does not affect the notion of time. This is useful for logic that is not relevant for the system being modelled, but is necessary for the model to function or be realistic. The virtual `CPU` can be used for logging or for the physics in the environment of the system under development that in the real world comes free of charge. Any object that is created during the initialization of a model and is not deployed to a specific `CPU` is considered as deployed to the virtual `CPU`.

Because of the virtual `CPU`'s role in VDM-RT it must have the ability to communicate with all other objects in the model. For that reason the virtual `CPU` and all others `CPU`s in the system is implicitly connected to a virtual `BUS`. Like the virtual `CPU` the virtual `BUS` does not have any time penalty for transferring messages.

Further details on VDM-RT are out of scope for this document; please refer to [24].

2.2.2 VDM Tool Support

Multiple tools exist to support VDM and the language dialects.

VDMTools VDMTools [25, 13] is a commercial tool suite for VDM containing everything from syntax/type checker to interpreter and debugger as well as code generation and test coverage statistics tools for the executed subsets. VDMTools was initially created at IFAD [26], Denmark and is currently maintained and further developed by CSK[1], Japan.

Overture The Overture project [27] is an Eclipse and Java based open-source platform combining a range of tools for constructing and analysing VDM models.

The purpose of this document is not only to construct a language extension for a dynamic reconfiguration in VDM-RT, but also to implement the changes into an interpreter to evaluate the effect. As VDMTools is based on proprietary source code and thereby closed, the open-source alternative Overture will be used.

The Overture Project

The Overture project[2] is aimed at the development of an open-source platform for constructing, executing and analysing VDM models. The project integrates a wide set of tools that include an editor, syntax and type-checking, interpreter, debugger and proof obligation generator.

[1] VDMTools `http://www.vdmtools.jp/en/`

[2] Overture project web portal: `http://www.overturetool.org`

Figure 2.2 gives an overview of the current state of the Overture Architecture.

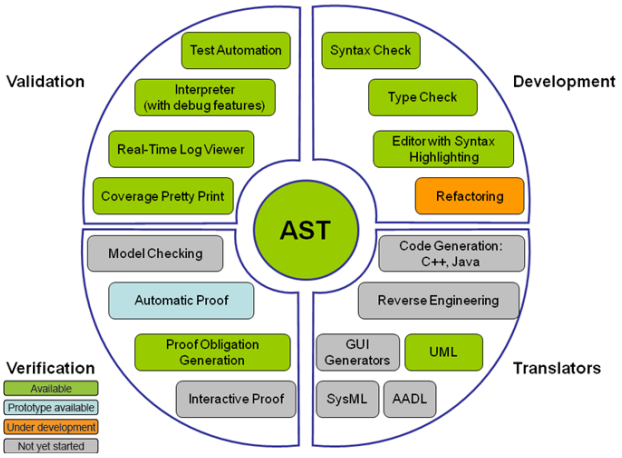


Figure 2.2: Overture Architecture Overview

The goal of the project is to provide an industrial strength tool set for VDM, as well as providing researchers with the option of extending and experimenting with the tools and VDM language dialects [27].
Overture consists of two main parts:

- an IDE based on the extensible Eclipse framework, and
- VDMJ [28] an open-source command-line Java tool that contains a parser, syntax- and type-checker, an interpreter, a debugger and a proof obligation generator for all of the VDM dialects.

For further information on the implementation details for the parts of VDMJ that are particular interesting with regards to system topology and dynamic reconfiguration please refer to [28, 6].

2.3 Purpose

The purpose of this document is to describe a dynamic reconfiguration extension to VDM-RT and to detail an exercise in applying these dynamic reconfiguration capabilities to an existing case study.

The purpose of adding the extension to VDM-RT is to enable the formal specification language to express a type of systems that for multiple years has seen a lot of development in making systems cope with the demands for adaptability, constant change as well as unpredictable size and amount of data exchange. Hopefully, this will enable the creation of more natural models that express the nature of these dynamic and adaptable systems in a better way.

3

Case Study

A brief introduction to the Case Study domain is given in Section 3.1. The concrete Case Study is presented in Section 3.2, while Section 3.3 supplies a more detailed description of the Case Study.

3.1 Introduction

This section describes the case studies which will be used to examine the effect of having dynamic reconfiguration in VDM-RT. The case study has been modelled to be functional in the existing VDM-RT dialect, with the limitations this entails. This establishes a baseline model before the functional changes to VDM-RT, so that the impact of dynamic reconfiguration can be assessed.

This section contains a general description of the case study, while the following section (Chapter 4) describes how the model has been designed and implemented in the existing VDM-RT dialect. The changes made to the models in order to apply the dynamic reconfiguration extension to the case studies are described in Chapter 6.

This case study is based on a major research area within cooperative vehicle communication systems which allow vehicles to communicate with each other and with the nearby roadside infrastructure in order to increase road safety and traffic flow. Within the European Union there are multiple research projects in the area, such as Cooperative vehicle-infrastructure systems [1], the Co-operative system for intelligent Road Safety [2] and the simulation platform iTETRIS [3]. All of these projects focus on autonomous systems and vehicle to vehicle communication in order to increase the information that is available about a vehicle and its environment. The CVIS and Coopers projects include the development of real-life hardware modules and field testing while iTETRIS is an open simulation platform.

These systems function by establishing wireless communication between them and using their interacting to share information concerning the current traffic information in the surrounding area. These systems are highly dynamic in their behaviour as they create relations and interactions as the vehicle moves in their environment. Consequently, they provide the perfect foundation for a case study that can be used to explore the capabilities of express dynamic properties in a formal modelling language.

[1] http://www.cvisproject.org
[2] http://www.coopers-ip.eu/
[3] http://www.ict-itetris.eu

3.2 Vehicle to Vehicle Communication

Given the principles of this research area the case study revolves around a system named Vehicle Monitoring (VeMo) which is designed to improve road safety by increasing the traffic information available to motorists. Presenting relevant information about the surroundings, as well as upcoming traffic events or potential hazards, will aid the driver of a vehicle by providing awareness of undiscovered situations and enable a better reaction and handling of a given situation.

The system could assist in events such as:

- Hazardous road conditions,
- Road maintenance,
- Left turn accidents, crossing of lanes with oncoming traffic,
- Traffic congestion.

If any of these events are identified the system will issue a warning message.

The increased information flow will be built on an intelligent traffic infrastructure along with open collaboration between motorists. Gathering and providing information through the traffic infrastructure will be done by expanding the functionality of existing stationary infrastructure such as traffic lights.

The most important information source will be the associated vehicles that form a co-operative network in which information can flow. The basis of the system is a communication technique described as “car to car” and “car to traffic infrastructure” wireless interfacing [29], in which a wireless network is used for communication as vehicles get physically close to each other. The communication network is established rapidly and the information exchange occurs for as long as possible within the very limited time frame while vehicles are in range.

3.3 Case Study Details

A vehicle becomes an associated member of the VeMo system by having a VeMo controller installed. This means that when the vehicle moves around in traffic it will start to create VeMo networks with other VeMo enabled vehicles in the vicinity and information can then be shared. The vehicles’ course of travel will eventually lead to a breakdown of the network as the distance between them becomes too large, as depicted in Figure 3.1.

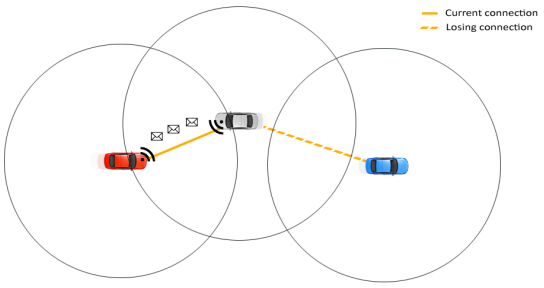


Figure 3.1: Vehicle network and communication

Thus in an area with a lot of traffic a VeMo network is not one large steady network in which everyone joins and communicates constantly; instead it describes very small short-lived networks that merely exist for exchanging information briefly as two vehicles pass each other in opposite directions.

The vehicles are the vital part of the system because it is their movements that make the system come to life; information will be spread out as vehicles pass other vehicles, which in turn will share the information with others. This means that a single source of information will be multiplied each time a vehicle gets in the vicinity of another vehicle.

As the VeMo system does not control where the associated vehicles go, the information originating from a single geographical position would spread out in an unpredictable fashion creating an unrestrained web of relations to a single event. This would cause the system to be flooded with irrelevant information and make it useless. Therefore, one of the main properties of the VeMo system is to control the flow of information by managing what information is shared and which vehicles it is shared between. The VeMo system has no central entity to control communication and information flow; therefore this must be managed by every single entity in the system. Each information holder must be able to determine which vehicles it is going to share information and what information to discard.

The following rules are defined:

- A vehicle will only exchange information with vehicles moving in the opposite direction;
- each vehicle will keep track of which vehicles it has recently communicated with, and refrain from communicating with these; and
- all information will carry a lifetime stamp which decides when it is no longer valid.

A typical scenario could be that a vehicle passes through a slippery piece of road, for example identified, by the traction control being activated; the VeMo controller will identify this and mark it as a potential hazard. Now each time a network is created it will notify other vehicles in the surrounding area of its location and hand over the traffic hazard information. At the same time it may receive traffic information about the upcoming or surrounding area. When vehicles receive new information they become information holders, as they have knowledge to share with others.

A single vehicle might receive information that is not relevant to itself, but it keeps the information to share with others that may benefit from it. Thus a lot of the information that a VeMo controller holds is irrelevant to the driver of that current vehicle and is therefore kept hidden, while the information sharing goes on transparently to the driver. This joining of forces forms the backbone of the system.

This scenario is shown on Figure 3.2 where the blue vehicle has discovered an icy spot on the road and identified this as a hazard. The hazard has been passed along to the black and grey

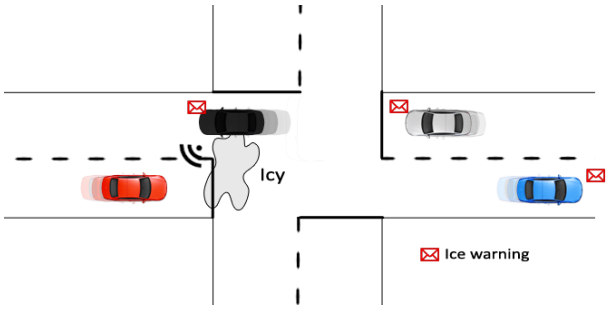


Figure 3.2: Scenario of information sharing

vehicles, but in this scenario the information is irrelevant to them, however they can pass it along to the red car which is about to encounter the icy spot.

The highly autonomous behaviour of the entities in the VeMo system means that the system is strongly based on distributed computing and has a rapid change of topology, meaning it is a perfect case for testing dynamic changes in the network topology.

4

Case Study VDM-RT Model

Section 4.1 presents how the model of the VeMo system has been designed, while Section 4.2 contains the detailed description of the model specification.

4.1 VDM-RT Model Design

A VDM-RT model of the VeMo system has been created which models the movement of vehicles, potential hazards and the exchange of information. A particular point of interest is determining when vehicles are close enough to establish radio communication between them such that they can initiate communication, and reversely when they have to break off communication because the distance between them becomes to great.

Creating a traffic simulator is not the point of the model so various properties, that would be vital to consider in the real world can be abstracted away. To focus on the important aspect, namely the communication and information sharing between autonomous vehicles, the following abstraction has been made to simplify the model:

- Collisions are not considered, meaning that vehicles take up no space and they can simultaneously cross each other's paths without colliding.
- To keep the overall system more controllable the movement of the vehicles has been simplified so they all move parallel or orthogonal to each other. In reality their movement has been limited to north, south, east and west which mean that the vehicles move as though they were tied to a grid, as illustrated on Figure 4.1.
- Each vehicle can only adjust its speed or change direction.
- The system has a centralized controller, the `VeMoController`, which is a big-brother that knows everything and keeps track of all vehicles. The `VeMoController` is responsible for calculating when vehicles are close enough to each other to establish communication and when they are too far apart to maintain the network. Essentially this means that the VeMoController is the representation of the wireless network.

  In the model the wireless connection is based on static BUS connections and object references through which communication is processed when the VeMoController determines it is possible. In reality a wireless network that is to be used between vehicles would be based

on radio communication which is limited by the physics and environment. Therefore a more precise representation of wireless communication would define a network where the topology can be altered on changes in the environment and message transfer is more unreliable than a wired network.

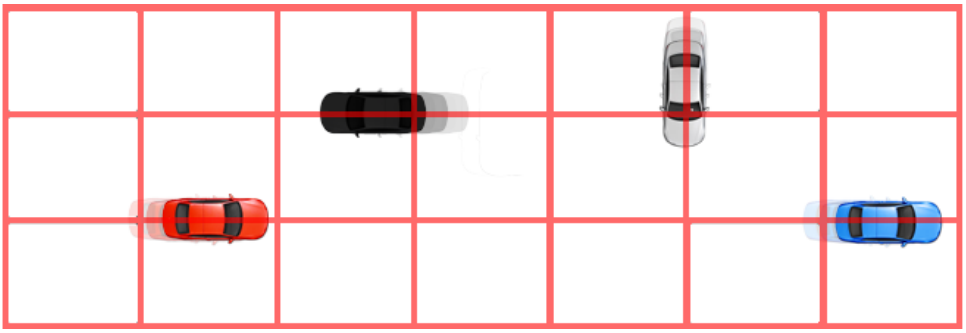


Figure 4.1: Grid of vehicle movement

The class diagram for the VeMo model is shown in Figure 4.2 and consists of the following classes:

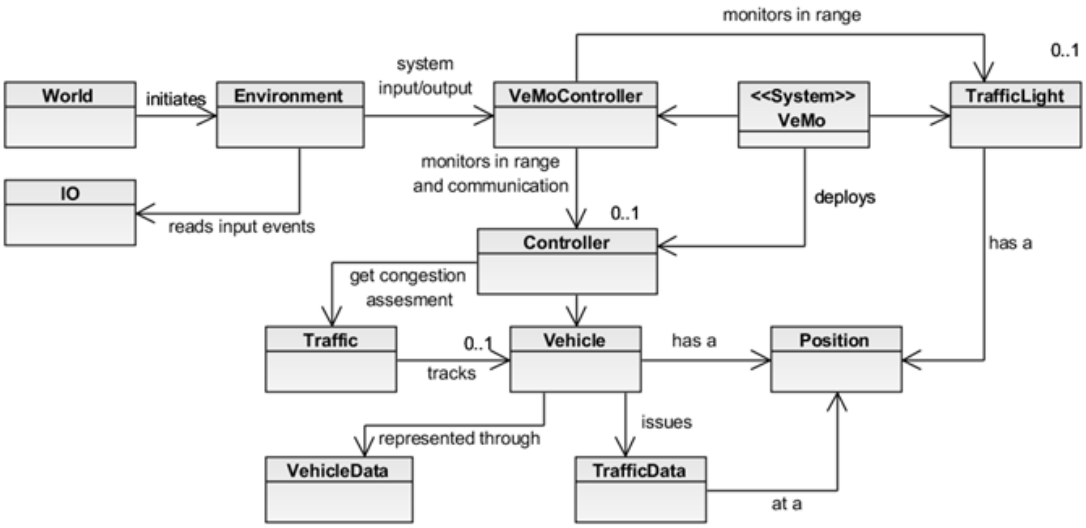


Figure 4.2: Class diagram of the VeMo model

VeMo System class that defines the CPU and BUS relations and the deployment of objects onto CPUs.

World is the main class which initializes the entire model.

Environment represents the input and output of the system.

VeMoController keeps track of all vehicles in a scenario.

TrafficLight is essentially the same as a Controller only a stationary infrastructure.

Controller is the representation of a vehicle in the model and is responsible for exchanging information with passing vehicles and is responsible for issuing warnings about hazards.

Vehicle calculates vehicle movement based on speed and direction.

VehicleData DTO representing a vehicle.

Traffic is responsible for calculating congestion, thus it keeps a sequence of passing Vehicles.

TrafficData contains the information a vehicle has collected about potential hazards.

Position represents an X, Y position in the world of the model.

The environment gets its input from a text file in which events are defined to occur at a specific time. An event is an input given to the system with updates regarding:

- Vehicle start,
- Vehicle speed,
- Vehicle direction,
- Vehicle turn indication, and
- Low grip, (low road traction detected).

The outputs of the system are the warnings issued by the vehicles, based on information from other vehicles. It is important to understand that the input events are not an abstract representation of the expected output; they are merely input to the current actions of the vehicles. Outputs, i.e. traffic warnings, are only generated if the position, direction and speed of vehicles meet the criteria for the potential hazard and the required rules of information flow. Because the input is read from a text file it is possible to define and load different scenarios that can be evaluated in the model.

Two types of messages may flow on the network:

- `TrafficData` is a DTO containing the traffic information shared between vehicles.
- `VehicleData` is a DTO which contains the data needed to create a deep-copy of the vehicle. The vehicle data is used for identifying the vehicle and as input to the congestion calculation.

A vehicle consists of a `Controller`, a `Vehicle` and a `Traffic` object. The `Controller` is the representation of the vehicle towards the rest of the model, in the sense that communication happens between `Controllers`. Each `Controller` is deployed on a **`CPU`**, leading to one **`CPU`** per vehicle.

Deployment

The deployment of a scenario in the model is shown on Figure 4.3, where the `Environment` and the `VeMoController` instances are deployed to the virtual `CPU` and two vehicles are deployed on separate `CPUs`.

As the case study has been modelled in the existing VDM-RT notation there can only be one deployment setup, determined at the creation of the VeMo system class. In the scenario in Figure 4.3 the two `CPUs` have been connected through the `comm` object which is an instance of the `BUS` class. The issue with the existing model is that, because VDM-RT has a static network

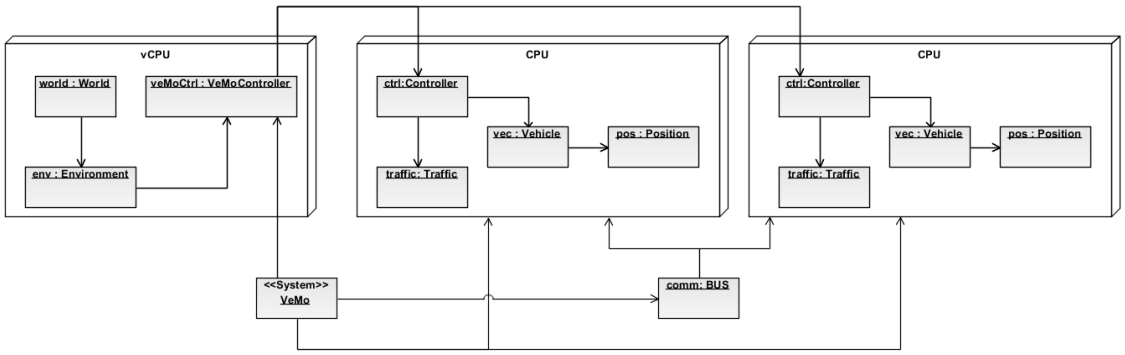


Figure 4.3: Deployment diagram of a VeMo system in the intial deployment setup.

topology, the vehicles are constantly connected to the `BUS`. This can lead to problems because the model may contain an unintentional implementation in which vehicles start to initiate communicate directly, while disregarding the `VeMoController`. This potentially creates scenarios in which vehicles are able to communicate even though they are too far apart to actually establish a network. This is possible because the model is not benefiting from the dynamic check of `BUS` and object connectivity because of the static network layout. By introducing dynamic reconfiguration into the case study, the network topology between `CPU`s, i.e. vehicles, could by adapted to reflect the possible wireless connections determined in the model. With a dynamic reconfigurable extension the topology can change, meaning vehicles can be connected and disconnected as they move around which makes for a closer representation of the real life scenario. By using dynamic reconfiguration functionality the VDM-RT interpreter check on `BUS` to object connectivity, is able to ensure the correctness of the communication and will discover any communication disregarding the `VeMoController`.

4.2 VDM-RT Model Specification

This section describes a selection of the classes contained within the model in greater detail.

VeMoEntity

The `VeMoEntity` class is the base class for all types of entities that can be part of the VeMo system co-operative network, e.g. vehicles or traffic infrastructure. The `VeMoEntity` class contains some instance variables for holding traffic data and for keeping track of the communication history.

```
class VeMoEntity
 instance variables
 -- traffic data issued by this controller, that will be passed on
 -- other controllers.
 protected internalTrafficData : seq of TrafficData := [];
 -- traffic data from other controllers moving in the opposite direction
 protected externalTrafficData : seq of TrafficData := [];
 --keep track of whom we have communicated with.
 protected communicatedWith : seq of nat := [];
 protected id : nat := 0;
 protected pos: Position;
```

Listing 4.1: VeMoEntity Instance Variables

The `internalTrafficData` holds all traffic information that the specific VeMo entity has identified in the surrounding area, and will pass as warnings to entities requesting traffic data for the surrounding area. The `externalTrafficData` sequence stores the traffic information received from other entities moving in the opposite direction. The `communicatedWith` stores a sequence of the last number of other entities that has communicated with the entity holding the list. This sequence is used to prevent the same entities from communicating and exchanging data multiple times. Additionally, the class contains an entity identification stored as a **`nat`**, as well as a `Position` which is used in the range calculation of the wireless communication.

`VeMoEntity` defines a number of abstract operations that have different implementations in the subclasses `Controller` and `TrafficLight`. For instance one difference is that the `TrafficLight` class does not have an implementation for the `GetDirection` operation, as it will communicate with all VeMo entities, unlike vehicle that only communicate with vehicle coming from the opposite direction.

```
-- allow traffic data from other VeMo entities to be added to
-- the specific VeMoEntity
public AddTrafficData: nat * seq of TrafficData ==> ()
AddTrafficData(vemoUnitID, data) == is subclass responsibility;

-- returns the traffic data events issued by this specific VeMoEntity
public GetTrafficData : () ==> [seq of TrafficData]
GetTrafficData() == is subclass responsibility;

-- returns the current position of the VeMoEntity
public GetPosition : () ==> [Position]
GetPosition() == is subclass responsibility;

-- return the unique id of the VeMoEntity
public GetID : () ==> nat
GetID()== is subclass responsibility;

-- returns the direction the vehicle is travelling
public GetDirection: () ==> [Types`Direction]
GetDirection() == is subclass responsibility;

-- used for to establish synchronization mechanism
-- between the environment and each VeMoEntity
public EnvironmentReady : () ==> ()
EnvironmentReady() == is subclass responsibility;
```

Listing 4.2: VeMoEntity Operations

Controller

`Controller` is the main class in every independent VDM unit. This class keeps track of traffic data gathered when passing other VeMo entities. The `Controller` is responsible for issuing traffic warnings, by creating `TrafficData` objects. Finally,`Controller` is the interface between the different VeMo units in the system. All communication with other units goes through

the Controller class. Each Controller is related to a Vehicle and the combination of the two makes for a moveable VeMoEntity.

The Controller contains the traffic instance variable which is used to track vehicles in the surrounding area, and on the basis of their movement determine if there is any risk of congestion. Additionally, the Controller contains a vehicle instance variable which represents the vehicle in which the controller is placed to make the vehicle part of the VeMo cooperative network. The vehicle object is used to manage the position, direction and speed that the Controller use for the calculations of the surrounding traffic.

```
instance variables
-- used to keep track of the surrounding traffic to determine if there is
-- any congestion
private traffic : Traffic;
 -- the vehicle the VeMo system Controller is placed in.
private vemoVehicle : Vehicle;
```

Listing 4.3: Controller - instance variables

The Controller class has a thread that is responsible for monitoring the variables of the vehicle and possibly issues traffic warnings by creating TrafficData objects. This data is added to an internal list, where it can be retrieved by another Controller. Traffic data is added via the private AddInternalTrafficData operation in Listing 4.4. This is implemented by a sequence, where the data is concatenated to the sequence until the sequence reaches a max. TrafficDataKeeptNumber. Afterwards, the data is concatenated to the tail of the sequence, thereby discarding the head.

```
private AddInternalTrafficData: TrafficData ==> ()
AddInternalTrafficData(data) ==
(
 if(len internalTrafficData < Config`TrafficDataKeeptNumber)
 then
  internalTrafficData := internalTrafficData ^ [data]
 else
  internalTrafficData :=  tl internalTrafficData ^ [data];
);
```

Listing 4.4: Controller - AddInternalTrafficData implementation

The traffic data is retrieved via the GetTrafficData operation in Listing 4.5. This perfoms a deep copy of the information. The purpose of this is to ensure that the traffic data is uniquely transferred to the VeMoEntity requsting the data, and to create a new timestamp to indicate the timespan that the reported incident is valid.

```
public GetTrafficData: () ==> [seq of TrafficData]
GetTrafficData() ==
(
-- deep copy
return [ new TrafficData(internalTrafficData(i).GetMessage(),
                internalTrafficData(i).GetPosition(),
                internalTrafficData(i).GetDirection(),
                vemoVehicle.GetID())
                | i in set inds internalTrafficData ];
);
```

Listing 4.5: Controller - GetTrafficData implementation

In order for traffic data to be added to a specific `Controller` the `AddTrafficData` operation in Listing 4.6 is used. This will ensure that we do not receive empty data and it keeps track of the VeMoEntities for which it has already exchanged data.

```
public AddTrafficData:  nat * seq of TrafficData ==> ()
AddTrafficData(vemoUnitID, data) ==
(
 --we can't use empty data
 if data = []
 then
  return;

 --did we already exchange information?
 if vemoUnitID in set elems communicatedWith
 then
  return;

 --keep track of who we have communicated with
 if len communicatedWith < Config`TrafficDataKeeptNumber
  and vemoUnitID not in set elems communicatedWith
 then
   communicatedWith := communicatedWith ^ [vemoUnitID]
 else
   communicatedWith := tl communicatedWith ^ [vemoUnitID];
...
```

Listing 4.6: Controller - AddTrafficData implementation 1/2

Additionally, this operation contains checks to ensure that we ignore any messages that were issued by the entity itself, and otherwise stores the retrieved data in an sequence, that for instance can be displayed to the driver of a VeMo-enabled vehicle.

```
...
let newTrafficData = [ data(i) | i in set inds data & data(i).GetOriginID() <>
 vemoVehicle.GetID()]
 in
 (
  if newTrafficData = []
  then
  return;

  --now add new traffic data
  if(len externalTrafficData < Config`TrafficDataKeeptNumber)
  then
    externalTrafficData := externalTrafficData ^ newTrafficData
  else
    externalTrafficData :=  tl externalTrafficData ^ newTrafficData;
 )
```

Listing 4.7: Controller - AddTrafficData implementation 2/2

The connection between the entities and the movement of data is handled by the `VeMoController` class that is part of the system environment and therefore deployed to a virtual cpu. The `VeMoController` keeps track of the position of the different VeMo entities and initiates communication when they come in range of each other.

The `Controller` contains a thread, listing 4.8, with a constant loop that calls the `Step` operation in listing 4.9.

```
thread
while true do
duration(500)
(
  Step();
)
```

Listing 4.8: Controller - thread

The `Step` operation lets the vehicle move to a new position based on its speed and direction, and it has a check to remove the internal traffic data that is no longer valid based on its age. Finally, the operation contains some logic to determine if any new traffic data should be generated to warn others of any traffic incidents. The latter is not shown in any detail here.

```
public Step: () ==> ()
Step() == (
 -- let the vehicle move to a new position based on its speed and direction.
 vemoVehicle.Move();

 -- check expired internal data
 for all td in set elems internalTrafficData do
 (
  if td.Expired()
  then
  (
  --remove td
  internalTrafficData := [internalTrafficData(i)
                | i in set inds internalTrafficData
                & internalTrafficData(i) <> td];
  )
);

 -- do checks to determine if alerts on new traffic incidents should be issued
 ...
```

Listing 4.9: Controller - Step operation

Vehicle

Represents the vehicle that a `Controller` is mounted in. This class has functionality that makes it possible to move its position based on its speed and direction.

The majority of the operations in the `Vehicle` class are used for getting or setting instance values, therefore the most interesting is the `Move` operation listed in 4.10. This makes the vehicle move to a new position based on its speed and direction.

```
public Move : () ==> ()
```

```
Move() ==
(
 cases dir:
 <NORTH> -> pos.setY(pos.Y() + speed),
 <SOUTH> -> pos.setY(pos.Y() - speed),
 <EAST>  -> pos.setX(pos.X() + speed),
 <WEST>  -> pos.setX(pos.X() - speed)
 end;
);
```

Listing 4.10: Vehicle - Move operation

TrafficLight

The `TrafficLight` has many of the same capabilities as found in the `Controller` class. The major differences are that the `TrafficLight` communicates with entities coming from all directions and obviously that it has a static position.

TrafficData

`TrafficData` is the basis for the different types of messages (traffic data) in the system. When a `Controller` issues a new traffic warning it creates a `TrafficData` object that can then be passed between VeMo entities.

A `TrafficData` object has an expiration time, which is defined by the instance variable `timeToLive`. This value relates to the VDM-RT system time, and is set through the `TrafficData` constructor, as shown in listing 4.11

```
public TrafficData: MessageType * Position * Types`Direction * nat
  ==> TrafficData
TrafficData(m,p,d,origin) ==
(
 pos := p ;
 message := m;
 dir := d;
 originId := origin;
 timeToLive := time + Config`TrafficDataLifeTime;
);
```

Listing 4.11: TrafficData - Constructor

`TrafficData` defines an enumeration of message types, as shown in listing 4.12, that can be issued and used to warn throughout the co-operative system.

```
types
public MessageType = <LowGrip> | <Congestion> | <LeftTurn> | <RedLight>;
```

Listing 4.12: TrafficData - Message types

The message types can deliver traffic information in such events as low traction or traffic congestion, and to inform of common traffic actions such a vehicle crossing other vehicles lane because of a left turn or of vehicles stopping because of a red light ahead.

Traffic

`Traffic` contains the vehicles known by a `Controller`, i.e. keeps a sequence of vehicles that passes in the opposite direction, as shown in listing 4.13

```
public AddVehicle: Vehicle ==> ()
AddVehicle(vehicle) ==
(
 if(len vehicles < Config`TrafficCongestionTrack)
 then
 vehicles := vehicles ^ [vehicle]
 else
 vehicles :=  tl vehicles ^ [vehicle]
)
pre vehicle not in set elems vehicles;
```

Listing 4.13: Traffic - AddVehicle operation

This class is responsible for determining when there is congestion in the surrounding area. This is based on the average speed of vehicles moving in the same direction as shown in listing 4.14

```
public Congestion: () ==> bool
Congestion() ==
(
 dcl inrange : set of Vehicle := {};

 for v in vehicles do
 (
  let vs = FindInRangeWithSameDirection(v,vehicles)
  in
  inrange := inrange union vs;
 );

 if card inrange = 0
 then return false;

 let avgspeed = AverageSpeed(inrange)
 in
 (
  return avgspeed < Config`TrafficCongestionThreshold;
 )
);
```

Listing 4.14: Traffic - Congestion operation

Position

Defines an X, Y position in the world of the VeMo system. The class is involved in the calculation of when VeMo elements are in range of each other based on their position, as shown in listing 4.15

```
public inRange : Position * int ==> bool
inRange(p, range) ==
(
    let xd = x - p.X(), yd = y -p.Y() in
```

```
    (
        let d = MATH`sqrt((xd * xd) + (yd * yd)) in
        (
            return d <= range;
        )
    )
);
```

Listing 4.15: In range calculation

The `Position` class also contains a deep copy operation that is used when a static position is needed for a specific point, as shown in listing 4.16. If there was no deep copy the position would be transferred as an object reference, so if e.g. a vehicle issued a traffic warning at a certain position directly, without using the deep copy, the position of the traffic warning would change as the vehicle moves.

```
public deepCopy : () ==> Position
deepCopy() ==
(
 let newPos = new Position(x,y)
 in
 return newPos;
)
```

Listing 4.16: Deep Copy of Position

VeMoController

The `VeMoController` is the main class for the entire VeMo system. This class represents the communication between vehicles in the system. It is responsible for calculating which vehicles are in range and to ensure the right data is exchange between them.

There are two mappings used to keep track of the entities in the model, one for `Controller` (vehicles) and one for `TrafficLights`, as shown in listing 4.17. These mapping defines the members of the entire VeMo system, and when a new vehicle or traffic light is added to the system during the initial configuration phase it is done via the `VeMoController`.

```
public ctrlUnits : inmap nat to Controller := {|->};
public lights : inmap nat to TrafficLight := {|->};
inv dom ctrlUnits inter dom lights = {};
inv forall id in set dom ctrlUnits & ctrlUnits(id).GetID() = id;
inv forall id in set dom lights & lights(id).GetID() = id;
```

Listing 4.17: VeMoController instance variables

The `VeMoController` contains numerous operations for adding and getting `Controller` and `TrafficLight` objects, however the essential operation is `CalculateInRange`, shown in listing 4.18. The operation is used to determine when entities are in range of each other and thus when they can communicate. This operation is called in an infinite loop in the `VeMoController` thread block. The first part shows the calculation for all vehicles in range of each other. A significant part of these calculations is the `FindInRangeWithOppositeDirection`, as this will supply all vehicles that are currently in range and is moving in the opposite direction of the unit currently being checked. This is the reason why there are different checks for vehicles and traffic

lights, as traffic light will communicate with all vehicles, while a vehicle will only communicate with vehicles approaching from the opposite direction. Once the units or vehicles that are in range of each other have been determined, traffic data will be exchanged between them. The listing also contains lines that are used to update the visualization that is added on top of the running model.

```
-- vehicles/controllers are denoted units in the following
let units = rng ctrlUnits in
 for all unit in set units do
 (
 let pos = unit.GetPosition() in
  graphics.updatePosition(unit.GetID(), pos.X(), pos.Y());

  let dir = unit.GetDirection() in graphics.updateDirection(
                                  unit.GetID(),
                                  Types`DirectionToGraphics(dir));

  let inrange = FindInRangeWithOppositeDirection(unit, units)
  in
  (
   -- only request data, the way the loop is built will ensure that all
   -- units will request data.
   if(card inrange > 0)
   then
   for all oncomingVehicle in set inrange do
   (
    unit.AddTrafficData(oncomingVehicle.GetID()
     ,oncomingVehicle.GetTrafficData());

    let vehicleDTO = unit.getVehicleDTO() in
    oncomingVehicle.AddOncomingVehicle(vehicleDTO);
   );
  )
 );
```

Listing 4.18: VeMoController : CalculateInRange 1/2

Listing 4.19 shows the remaining code segment of the `CalculateInRange` operation where a similar check is performed for traffic lights in order to determine the vehicles that in range of these. Here the `FindInRange` operation is used to include vehicles from all directions. Additionally, the listing contains parts of the synchronization logic used to ensure that the `VeMoController` completes the "in range" check and data communication for all VeMo entities before they are allowed to initiate their next step.

```
for all light in set rng lights do
(
 --find in range
 let inrange = FindInRange(light, rng ctrlUnits) in
 (
  if(card inrange > 0) then
   for all vehicle in set inrange do
   (
    light.AddTrafficData(vehicle.GetID() ,vehicle.GetTrafficData());
    vehicle.AddTrafficData(light.GetID() ,light.GetTrafficData());
   );
 );
);
```

```
-- synchronization, when we have calculated inrange allow vehicles to
--move again
let  vemoUnits = ctrlUnits munion lights in
 for all u in set rng vemoUnits do u.EnvironmentReady();

graphics.sleep();
```

Listing 4.19: VeMoController : CalculateInRange 2/2

The find `FindInRange` and `FindInRangeWithOppositeDirection` operations determines the entities in range based on a simple comparison of the distances between the positions indicated by each entity, and for the latter operation the directions of the entities as well.

Types

The `Types` class defines a range of different types that are used in general throughout the VeMo model. Most important is the `Event` type that defines the different kind of events that can be used as input to the VeMo system. The `Event` type is a union of different record types that represent the individual kind of input events. The record types can then be used to generate an event from an external file that is read by the `Environment` class.

```
public Event = AddVehicle | VehicleRun | RemoveVehicle | TrafficLightRun
  | VehicleUpdateSpeed | VehicleUpdatePosition | VehicleUpdateDirection
  | VehicleLowGrip | VehicleTurnIndication | WasteTime;
```

Listing 4.20: Different kind of events that can be inputted to the system

Environment

Represents the `Environment` of the VeMo system. This class reads input events, defined in the `Types` class, from an event file and thereby affects the system by applying the input. This allows different model scenarios to be defined and executed to see how the VeMo model reacts.

The class contains a thread block that calls the `Event` operation, for which a segment is shown in listing 4.21. The events are read from the inlines sequence, which is based on the inputs from the event file. A `cases` expression is used to determine the type of event and the event timestamp is checked against the current time (`curtime`) to determine if the event is ready to fire. The different kinds of events have different handlers, of which the event kind `VechicleRun` handler is shown in listing 4.21. The loop will keep circling as long as there are still events waiting with the same timestamp; otherwise the loop is stopped to allow the rest of the system to execute.

```
...
while not done do
(
 def event = hd inlines in
 cases event:
  mk_Types`VechicleRun(-,-) ->
  (
   if event.t <= curtime
   then
   (
     let ctrl = vemoCtrl.getController(event.ID) in
```

```
      ctrl.run();

     eventOccurred := true;
    )
   ),
  -- handling of other kinds of events
 ...

 if eventOccurred
 then
 (
  inlines := tl inlines;
  done := len inlines = 0;
 )
 else
  done := true;

 eventOccurred := false;
);
```

Listing 4.21: Environment - Event operation

World

The `World` class is the main entry point of the system and it defines the system environment, the input file containing the events of the modelled scenario, and it initiates the threads running the model.

VeMo

`VeMo` is the **`system`** class of the VeMo system and it defines the topology of the modelled system via `CPU` and `BUS` objects, as shown in listing 4.22. `VeMo` defines a static set of `CPU`s, a `BUS`, as well as the static connections between the `CPU`s via the `BUS` constructor. The `Controllers` (vehicles) and traffic lights in the system are also instantiated and stored as instance variables in the `VeMo` class.

```
instance variables
public  cpu0 : CPU := new CPU (<FP>,1E6);
public  cpu1 : CPU := new CPU (<FCFS>,1E6);
public  cpu2 : CPU := new CPU (<FCFS>,1E6);
public  cpu3 : CPU := new CPU (<FCFS>,1E6);
public  cpu9 : CPU := new CPU (<FCFS>,1E6);
public  cpu10 : CPU := new CPU (<FCFS>,1E6);

public bus1 : BUS := new BUS (<FCFS>,1E6,{cpu0, cpu1, cpu2, cpu3, cpu9, cpu10});

-- Vehicles and traffic lights
public static ctrl1 : Controller := new Controller(
                          new Vehicle(1,new Position(80, -80),
                                                  1, <NORTH>));
-- more instantiations of vehicles and traffic lights.
...
```

Listing 4.22: VeMo - Instance variables

Listing 4.23 shows the VeMo constructor that deploys the defined Controllers to individual CPUs, in order to model them as independent systems with their own processing time and power.

```
operations

public VeMo: () ==> VeMo
VeMo() ==
(
 cpu1.deploy(ctrl1);
 cpu2.deploy(ctrl2);
 cpu3.deploy(ctrl3);
 cpu9.deploy(ctrl9);
 cpu10.deploy(tl1);
);
```

Listing 4.23: VeMo - Constructor

This class illustrates the core issue with the design of the modelled system. Everything is defined statically and inflexible which makes it difficult to express more dynamic and evolving systems within the VDM-RT language. Relations between processing units cannot be created, changed or removed, nor can processing units themselves be created or removed from the system. A way of solving these limitations and instead allow dynamicity and flexibility in the modelled system is the central point of the entire exercise performed in this document.

5

Dynamic Reconfiguration

This chapter describes the dynamic reconfiguration capabilities added to VDM-RT. Section 5.1 supplies a brief introduction to dynamic reconfiguration. Section 5.2 presents the changes made to VDM-RT in order to introduce the dynamic reconfiguration capabilities, while Section 5.3 presents the specific operations and their effect to a system topology.

5.1 Introduction to Dynamic Changes

The goal of dynamic reconfiguration [5] is to change the configuration of a currently running distributed system into a new specified configuration, thereby enabling a system which can dynamical and incrementally evolve without disrupting the system processing.

The configuration of a system is a set of software entities and the relationship between them. Software entities may be components, objects or processing units, while the relationships may be network connections or object references. In a VDM-RT context the entities could be `CPU`'s and the `BUS` class could represent the relationships.

The purpose of having dynamic reconfiguration is to support distributed systems that must provide adaptability and system evolution in order to meet requirements which change throughout the systems lifetime. The run-time alternations that dynamic reconfiguration entails is a disruptive process and the reconfiguration may affect the current interactions between software entities. The reconfiguration mechanism must ensure that the persistency in the system is maintained.

Dynamic reconfiguration mechanisms and approaches can in general be categorized under four common types of capabilities [30, 31]:

Changing Topology changes the relationship between entities by either creating new connections or by removing connections between them.

Replacement involves the substitution of an entity in the system with another entity that realizes the same interface. This can be used to substitute one implementation with a newer during run-time execution of the system.

Addition/Removal is the dynamic introduction or elimination of an entity into the system configuration.

Migration moves an entity from one processing unit to another while preserving the identity of the entity and thereby ensuring referential integrity.

Only “Changing Topology” and “Addition/Removal” are included as part of the dynamic reconfiguration capabilities presented in this document.

5.2 Changes to the VDM-RT

The existing VDM-RT capability of describing static real-time systems must be extended to allow for the dynamic alternation of the overall system architecture during the run-time execution of a model.

In order to open and widen the applicability of VDM-RT, small notational adjustments are introduced into the VDM-RT language. The purpose is to raise the language from merely expressing the fairly low hardware-level notion of `CPU` and `Bus` types, and instead introduce more general terms that apply to a wider range of system types. A `Constituent` type is used to denote an independent processing entity that previously only was represented by the `CPU` only. A `Channel` type is introduced as the link between `Constituents`, in the same way that the `BUS` type connected `CPUs`. This means that the **`system`** class can contain instances of a `Constituent` type that represents constituent systems in the overall architecture and a `Channel` type that is used for establishing communication links between the constituent systems. `Constituent` and `Channels` are essentially extensions of the `CPU` and `Bus` types. They are however different types, meaning that they are not assignable to each other.

An instance of a `Constituent` is defined with a specific processing performance, in terms of executed instructions with relation to time, and multiple applications can be deployed and run in parallel on each `Constituent`. Instantiations of a `Channel` are defined with the specific channels transmission capacity, meaning the max. data amount that can be moved in a time unit. The constituent systems initially connected to a given channel can be defined by supplying a set of constituents during the creation of the channel. An example architecture using these components is illustrated in Figure 5.1.

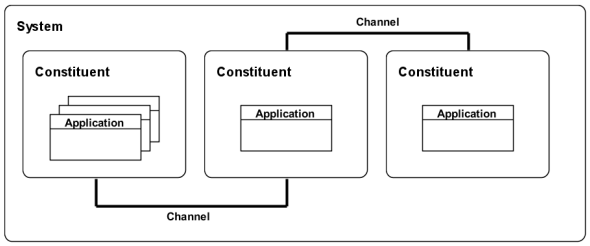


Figure 5.1: Example architecture using the introduced components

Three types of dynamic architectural changes are needed to enable the modelling of highly dynamic systems. The dynamic run-time changes include dynamic run-time: (1) addition and removal of new constituent systems, (2) addition and removal of communication channels, (3) changes of the network topology, meaning changing how constituent systems are connected to channels.

Given that the distributed system architecture is defined in the special **`system`** class and all `Channel` and `Constituent` declarations are contained within this class, it will be used as the heart of the dynamic reconfiguration. Consequently the **`system`** can be considered a type of topology manager.

The changes are added to the **`system`** class as static public operations. It is necessary that the operations are static because the **`system`** class cannot be instantiated. This has the advantage

that the reconfiguration can be performed from wherever the need for reconfiguration arises. A disadvantage of this decentralization is the risk of losing the overview, as it becomes more difficult to control from where reconfigurations are initiated.

The extension does not introduce any new language keywords into VDM-RT since the dynamic reconfiguration is enabled by adding new operations to an existing predefined class. Combined with the existing VDM-RT semantics this will prevent the risk of ambiguity and keyword conflicts with existing VDM models, because in the current VDM-RT semantics the **system** class cannot be neither extended nor inherited. Likewise, despite it being defined by the model developer the implementation is limited by the type checker to only allow the definition of a system constructor. This ensures backward compatibility as conflicts with any operations defined in existing models do not need to be considered, given that they cannot exist in any validated model.

As a whole the overall structure of the system will be defined in the **system** class that will both define the initial architecture of the system and also be used as a type of manager of dynamic changes.

Operations are included for adding and removing **Constituent**s and **Channel**s during run-time execution of the model. Additionally, operations are defined for dynamically altering the network topology by connecting and disconnecting **Constituent**s to and from **Channel**s. The difference between disconnecting a **Constituent** from a **Channel** and removing a **Constituent** is that a disconnected **Constituent** still exists within the architecture and is still processing. Oppositely, a **Constituent** that is removed is permanently deleted from the architecture and will cease to process. This allows the modelling of **Constituent**s that become temporarily isolated, but still function locally. It should also be noted that a **Constituent** can be connected to multiple channels, so being disconnected from one channel does not necessarily entail that the system is isolated.

A clear source of error is the risk of data loss on a `Channel` connection during dynamic reconfiguration changes. A change in the network relationship during execution will cause messages currently on a `Channel` to miss their receiver. Disruptions of on-going communication may occur, either by the connection being torn down or by a constituent leaving the network, as dynamic changes are made to the network topology or constituents are removed.

VDM-RT has exception handling natively, and thus exceptions will be used to report message loss. If a constituent is removed exceptions will be raised at the senders of all messages currently in transit. Any message sent from the removed system is lost without exceptions being raised in the removed system. When a channel is removed all messages in transit will be lost and exceptions will be raised at the senders of the messages. Constituents will not be notified of the channel being removed, unless they are communicating or attempt to communicate over the removed channel. Likewise, constituents will not be notified of other constituents being removed during run-time. This has to be handled manually in the model specification.

5.3 Dynamic Reconfiguration Operations

The reconfiguration operations defined in the **system** class are listed in Table 5.1.

Operation	Description
connect(Constituent, Channel)	Connects a Constituent with a channel.
disconnect(Constituent, Channel)	Disconnects a Constituent from channel.
addConstituent(Performance)	Adds a new Constituent with specific performance capabilities to the system.
removeConstituent(Constituent)	Removes an existing Constituent from the system.
addChannel(Capacity, Constituent)	Adds a new communication channel with a defined transmission capacity, and a set of Constituent's being connected to the channel.
addChannel(Capacity)	Adds a new communication channel with a defined transmission capacity.
removeChannel(Channel)	Removes an existing communication channel.

Table 5.1: Dynamic reconfiguration operations

The functionality for each of these operations is briefly described in the following subsections.

5.3.1 Adding and Removing Constituents

A new constituent is added to the system topology by calling the `addConstituent` with a specific processing performance. The new constituent is added to the topology automatically, and the instance of the new constituent is returned by the operation. Afterwards, this instance can be used to deploy the objects that are to run on the constituent.

```
let constituent = VeMo`addConstituent(1E6) in
  constituent.deploy(new Controller());
  ...
  VeMo`removeConstituent(constituent);
```

Listing 5.1: Adding and Removing Constituents

Besides the `addConstituent` operation, Listing 5.1 also contains the `removeConstituent` operation that removes a specific constituent from the system topology based on the constituent instance passed to the operation. Once a constituent is removed it will stop processing. No objects can be deployed to a removed constituent, this will result in a runtime error.

5.3.2 Adding and Removing Channels

A new channel is created by calling the `addChannel` operation with a transmission capacity and an optional set of constituents that are connected via the channel. The set of constituent can be omitted and the channel is created without having any initial connections.

```
let newChnl = VeMo`addChannel(1E6, {constituent1, constituent2}) in
 VeMo`removeChannel(newChnl)
```

Listing 5.2: Adding and Removing channels

Immediately after the `addChannel` operation in Listing 5.2 the `removeChannel` operation is called that instantly removes the channel again by disconnecting all constituents and dropping any data messages in transit.

5.3.3 Changing Connections in the Topology

The `connect` operation connects a specific constituent to a specific channel, thereby allowing the constituent to communicate with all other constituents connected to that channel.

```
VeMo`connect(constituent1, commChannel);
 -- exchange data
 ...
VeMo`disconnect(constituent1, commChannel);
```

Listing 5.3: Connecting and Disconnecting

Listing 5.3 also shows the `disconnect` operation that is used to tear down an existing connection between a constituent and a channel. The `disconnect` operation is implemented as a disruptive action that will perform a non-graceful disconnection meaning that any messages currently processing will be lost. For all messages lost on the channel an exception will be raised on the initiator of the data transfer.

6

Dynamic Model VDM-RT Model

This chapter details the changes made to the VeMo model in order to utilize the dynamic reconfiguration capabilities introduced in Section 5. The chapter will only show the classes that have been altered in comparison with the model presented in Section 4.

6.1 Introducing Dynamicity in the Case Study Model

The introduction of the dynamic reconfiguration capabilities into the VeMo model has enabled a model that is capable of expressing a system where vehicles can enter and leave the system, and where the connections between vehicles and vehicles and infrastructure can be changed.

Enabling the dynamic reconfiguration capabilities has entailed changes in three classes; `VeMo`, `VeMoController` and `Environment`. These changes are detailed below in Section 6.2.

6.2 Changes to the Case Study Model

The three classes that have been changed in order to incorporate the dynamic reconfiguration capabilities are described in further details below.

VeMo

In the `VeMo` class all of the processing units related to vehicles have been removed, and the `CPU` related to the traffic light has been replaced with a `Constituent` type. This is done to better reflect that vehicle are none static entities of the VeMo system, and only entities that are static in the environment being modelled, such as a traffic light, should be represented initially in the `VeMo` system class.

The use of dynamically added `Constituents` and `Channels` has required the introduction of two new mappings in the `VeMo` class, the `ve2cons` and `ve2channel`. These are used to keep track of the relations between VeMoEntities and the `Channels` they are deployed on, as well as to keep track of which VeMoEntities are connected to which channels. The latter map is a map of maps, because a `VeMoEntity` can be connected to multiple other VeMoEntities. The maps are needed because a `Constituent` does not allow any way of accessing its deployed objects. Likewise, a `Channel` does not supply any information on its connected `Constituents`. This

information cannot be stored in any other object than the VeMo **system** class, e.g. as an instance variable in the VeMoEntity, because of a type limitation in VDM-RT. This type limitation means that CPU, BUS, Constituent, and Channel types can only be defined with in the VeMo class. Because the mappings are created in the VeMo class the Constituent and Channel types can be defined as instance variables. As they are declared **public static** they can be used externally to the VeMo type, thereby the restrictions that are limiting the definition of Constituent and Channel to the **System** class can be circumvented.

```
public static ve2cons : inmap VeMoEntity to Constituent := {|->};
public static ve2channel : map nat to (map nat to Channel) := {|->};

public  consti1 : Constituent := new Constituent (1E6);

--traffic light
public static tl1 : TrafficLight := new TrafficLight(999
                                                     ,new Position(20, -70)
                                                     , 40);
```

Listing 6.1: VeMo - Instance variables

The changes are also reflected in the VeMo constructor in listing 6.2, where only the traffic light is being deployed. The listing also shows how the mapping from VeMo entities (the traffic light) to the constituent system (consti1) is made.

```
operations

public VeMo: () ==> VeMo
VeMo() ==
(
    ve2cons := VeMo`ve2cons munion {tl1 |-> consti1};
    consti1.deploy(tl1);
);
```

Listing 6.2: VeMo - Constructor

Having this mapping makes it possible to keep track of the relations between VeMo entities and constituents.

VeMoController

The VeMoController class is still responsible for determining when vehicles and infrastructure are close enough to establish connections. In the dynamic version of the model the connections and disconnections are represented by dynamically added channels through which the constituents will be connection and disconnected. This is a change from the previous model where the static topology between the constituent systems meant that they were always connected, and the connections and disconnections from vehicles to vehicle and vehicles to infrastructure were simply simulated.

In order to keep track of which vehicles that have been connected to each other, a controllerConnections mapping has been introduced. It maps a single VeMoEntity to the set of VeMoEntities it is connected to.

```
instance variables
...
-- keeps bookkeeping/track of connections from a controller to other controllers
private controllerConnections : map nat to set of VeMoEntity := {|->};
```

Listing 6.3: VeMoController - Events operation

The check used to determine when vehicles are in range of each other is still performed by the `CalculateInRange` operation. It has however been updated such that it creates new channels to which the constituents can be connected and disconnected.

The way the check is performed has also been altered such that at first a check is done to determine if the entities are in range of each other and should be connected or disconnected. Secondly, a check is performed to determine if any data should be exchanged. The first part is shown in listing 6.4 and 6.5. In listing 6.4 all entities "in range" are found and these are diff'ed with the `controllerConnections` mapping, meaning that any difference will indicate new connections. For all new connections a new channel is created and the "in range" units are connected to this channel. The `ve2channel` mapping is updated in order to keep track of the connections between VeMoEntities and channels, as described in the VeMo section above.

```
...
for all unit in set units do
(
 let inrange = FindInRange(unit, units)
 in
 (
  -- find new controllers that has come in range since last check and
  -- create a bus connection.
  let newConnection =  inrange \ controllerConnections(unit.GetID()) in
  for all conn in set newConnection do
  (
    let newChnl = VeMo`addChannel(1E6) in
    (
     VeMo`connect(VeMo`ve2cons(unit), newChnl);
     VeMo`connect(VeMo`ve2cons(conn), newChnl);
     VeMo`ve2channel(unit.GetID()) := (VeMo`ve2channel(unit.GetID()))
                                       ++ { conn.GetID() |-> newChnl};
    );

   graphics.connectVehicles(unit.GetID() ,conn.GetID());
  );
...
```

Listing 6.4: VeMoController - CalculateInRange 1/2

In listing 6.5 the entities that have gone out of range of each other are identified by doing the reverse diff of what was done in listing 6.4 to find the "in range". The `ve2channel` mapping is used to find the correct channel on the basis of the ID's of the involved entities. The entities are disconnected from the channel and the mapping is updated. Finally, when all the range checks with the connections and disconnections have been completed, the `controllerConnections` mapping is updated to keep track of the current connected entitites.

```
 ...
 -- find units that have gone out of range since last check
 -- and tear down the bus connection.
 let lostConnection = controllerConnections(unit.GetID()) \ inrange  in
 for all lost in set lostConnection do
 (
 -- disconnect constituents
 let chnl = VeMo`ve2channel(unit.GetID())(lost.GetID()) in
 VeMo`disconnect(VeMo`ve2cons(unit),chnl);
 -- disconnect in graphics
 graphics.disconnectVehicles(unit.GetID(), lost.GetID());
 -- update our mapping between vehicles and channel
 -- by removing the lost mapping
 VeMo`ve2channel(unit.GetID()) := ({lost.GetID()}
                                   <-: VeMo`ve2channel(unit.GetID()));
 );
-- remember inrange for a specific unit, to enable history/bookkeeping
-- of new and lost connections
controllerConnections(unit.GetID()) := inrange;
)
...
```

Listing 6.5: VeMoController - CalculateInRange 2/2

The entire check is performed for each of the entities in the system.

Environment

The dynamic capabilities are used in the `Environment` class to add and remove vehicles from the modelled scenario throughout the execution of the model. This enables a more elaborate scenario to be played as it is no longer necessary to have everything included in the model from the initial state. Previously, it was a problem with the model that some scenarios were difficult to express because elements that were only to be introduced at a later time would influence the execution of the model from the start of the execution.

The event handlers that are handling the input events from the external text file have been updated to use the dynamic capabilities of the `addConstituent` and `removeConstituent` reconfiguration operations. Listing 6.6 shows how the `addConstituent` is used to add a new vehicle to the system by dynamically added a constituent, deploying an instance of a `Controller` object to it, and then starting the thread in the `Controller` with the `run` command. The listing also shows how the `ve2cons` mapping, keeping track of the relations between VeMoEntities and Constituents, is updated.

```
...
 mk_Types`VehicleRun(-,-) ->
(
   if event.t <= curtime
   then
   (
    let ctrl = vemoCtrl.ctrlUnits(event.ID) in
    (
      --create new Constituent
      let constit = VeMo`addConstituent(1E6) in
      (
       constit.deploy(ctrl);
       VeMo`ve2cons := VeMo`ve2cons munion {ctrl |-> constit};
```

```
      );
    );

    let ctrl = vemoCtrl.getController(event.ID) in
    ctrl.run();
    eventOccurred := true;
   )
 ),
-- handling of other kinds of events
...
```

Listing 6.6: Environment - Events operation

A similar event handler exists for removing vehicles from the modelled scenario.

7

Current Status and Future Plans

This section provides a brief summary of the presented work, it presents some considerations on future work, and ends with concluding remarks.

7.1 Summary of Work

In this document the formal language VDM-RT was used to model a highly dynamic case study. The case study involves a system that has a great number of changes in its system topology, with the relations between the constituents changing swiftly and often. During the creation of the case study model some limitations was found in VDM-RT's ability to express the dynamic behaviour of the case study system. While VDM-RT allows the topology of a system to be expressed via busses and cpus, everything is statically defined and the topology remain its initial state throughout the execution of the model. This is a problem when dealing with a highly dynamic system where the system topology changes frequently. It becomes difficult to express the actual behaviour of the system, meaning that certain workarounds and simulated behaviours need to be added to the model.

Therefore, an extension to VDM-RT has been proposed that allows for the dynamic reconfiguration of the system topology throughout the execution of the model. The proposed extension introduces the more general terms of `Constituents` for describing the individual processing units or systems, and `Channels` to represent the connections between the `Constituents`. To enable the reconfiguration capabilities the extension introduces methods for adding and removing `Constituents` and `Channels` during the execution of the model, as well as methods for changing how `Constituents` are connected to `Channels`, thus changing the topology of the system. The proposed extension was implemented in a branch of the VDM-RT interpreter VDMJ.

The case study model was revisited and the dynamic reconfiguration capabilities was incorporated and utilized in the model. Having the ability to describe dynamic functionality made it easier to express the behaviour of the modelled system and it allowed the model to represent the case study system in a more precise way.

7.2 Future Plans

The presented work can be enhanced by enabling constituents and channels to reveal more about their internal state. Such that constituents enable access to their deployed objects, and channels provide a way of sharing which constituents are connected them. Some type of automated approach for keeping track of the relations between deployed applications and the constituents, as well as the relations between constituents and channels. Having this built directly into the language would avoid the error-prone task of manually creating mappings to keep track of these relations.

7.3 Concluding Remarks

This document has proposed a language extension of VDM-RT which we believe enables the modelling of systems that have dynamic topologies with reconfiguration capabilities. The work presented in this document still needs to be matured further and the use of the extension needs to be investigated further in more case studies. It does however represent a step in the right direction towards enabling the description of dynamic systems within a formal language. Having these capabilities are not only important for VDM-RT but for formal languages in general, in order for them to express the behaviour found in many modern systems. Therefore, it is our hope that it becomes possible to include the important aspects of dynamic reconfigurable systems into more formal languages.

Bibliography

[1] F. Fich and E. Ruppert, "Hundreds of Impossibility Results for Distributed Computing," *Distributed Computing*, pp. 121–163, February 2004.

[2] M. Satyanarayanan, "Pervasive computing: Vision and challenges," *IEEE Personal Communications*, vol. 8, no. 4, pp. 10–17, 2001.

[3] M. Verhoef, *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2009.

[4] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef, *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[5] J. Kramer and J. Magee, "Dynamic Configuration for Distributed Systems," *IEEE Transactions on Software Engineering*, April 1985.

[6] C. B. Nielsen, "Dynamic Reconfiguration of Distributed Systems in VDM-RT," Master's thesis, Aarhus University, December 2010.

[7] E. M. Clarke and J. M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, 1996.

[8] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal Methods: Practice and Experience," *ACM Computing Surveys*, vol. 41, pp. 1–36, October 2009.

[9] R. Floyd, "Assigning Meanings to Programs," in *the Symposium on Applied Mathematics*, vol. 19, pp. 19–32, American Mathematical Society, 1967.

[10] J. S. Fitzgerald, P. G. Larsen, and M. Verhoef, "Vienna Development Method," *Wiley Encyclopedia of Computer Science and Engineering*, 2008. edited by Benjamin Wah, John Wiley & Sons, Inc.

[11] C. B. Jones, "Scientific Decisions which Characterize VDM," in *FM'99 - Formal Methods* (J. Wing, J. Woodcock, and J. Davies, eds.), pp. 28–47, Springer-Verlag, 1999. Lecture Notes in Computer Science 1708.

[12] D. Bjørner, "Pinnacles of software engineering: 25 years of formal methods," *Annals of Software Engineering*, vol. 10, pp. 11–66, 2000.

[13] J. Fitzgerald, P. G. Larsen, and S. Sahara, "VDMTools: Advances in Support for Formal Modeling in VDM," *ACM Sigplan Notices*, vol. 43, pp. 3–11, February 2008.

[14] N. Plat and P. G. Larsen, "An Overview of the ISO/VDM-SL Standard," *Sigplan Notices*, vol. 27, pp. 76–82, August 1992.

[15] P. G. Larsen, B. S. Hansen, *et al.*, "Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language," December 1996. International Standard ISO/IEC 13817-1.

[16] J. Fitzgerald and P. G. Larsen, *Modelling Systems – Practical Tools and Techniques in Software Development*. The Edinburgh Building, Cambridge CB2 2RU, UK: Cambridge University Press, Second ed., 2009. ISBN 0-521-62348-0.

[17] M. Verhoef, "On the use of VDM++ for specifying real-time systems," *Proc. First Overture workshop*, November 2005.

[18] M. Verhoef, P. G. Larsen, and J. Hooman, "Modeling and Validating Distributed Embedded Real-Time Systems with VDM++," in *FM 2006: Formal Methods* (J. Misra, T. Nipkow, and E. Sekerinski, eds.), Lecture Notes in Computer Science 4085, pp. 147–162, Springer-Verlag, 2006.

[19] J. Hooman and M. Verhoef, "Formal semantics of a VDM extension for distributed embedded systems," in *Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever* (D. Dams, U. Hannemann, and M. Steffen, eds.), vol. 5930 of *Lecture Notes in Computer Science*, pp. 142–161, Springer-Verlag, 2010.

[20] J. S. Fitzgerald, P. G. Larsen, S. Tjell, and M. Verhoef, "Validation Support for Real-Time Embedded Systems in VDM++," Tech. Rep. CS-TR-1017, School of Computing Science, Newcastle University, April 2007. Revised version in Proc. 10th IEEE High Assurance Systems Engineering Symposium, November, 2007, Dallas, Texas, IEEE.

[21] M. Verhoef and P. G. Larsen, "Interpreting Distributed System Architectures Using VDM++ – A Case Study," in *5th Annual Conference on Systems Engineering Research* (B. Sauser and G. Muller, eds.), March 2007. Available at http://www.stevens.edu/engineering/cser/.

[22] R. A. Sørensen and J. M. Nygaard, "Evaluating Distributed Architectures using VDM++ Real-Time Modelling with a Proof of Concept Implementation," Master's thesis, Enginering College of Aarhus, December 2007.

[23] S. Wolff, "Universal Multiprotocol Home Automation Framework," Master's thesis, Aarhus University/Engineering College of Aarhus, December 2008.

[24] N. B. J. F. Peter Gorm Larsen, Sune Wolff and K. Pierce, "Development process of distributed embedded systems using vdm," Tech. Rep. TR-2010-02, The Overture Open Source Initiative, April 2010.

[25] CSK, "VDMTools homepage." *http://www.vdmtools.jp/en/*, 2007.

[26] R. Elmstrøm, P. G. Larsen, and P. B. Lassen, "The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications," *ACM Sigplan Notices*, vol. 29, pp. 77–80, September 1994.

[27] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, "The Overture Initiative – Integrating Tools for VDM," *ACM Software Engineering Notes*, vol. 35, January 2010.

[28] N. Battle, "VDMJ User Guide," tech. rep., Fujitsu Services Ltd., UK, 2009.

[29] P. Varaiya, “Smart Cars on Smart Roads. Problems of Control,” *IEEE Transactions on Automatic Control*, vol. 38, pp. 195–207, February 1993.

[30] K. Goudarzi and J. Kramer, “Maintaining node consistency in the face of dynamic change,” in *Configurable Distributed Systems*, pp. 62 –69, 1996.

[31] D. S. Milojíčić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, “Process migration,” *ACM Comput. Surv.*, vol. 32, pp. 241–299, September 2000.

Appendices

Dynamic VDM-RT Specification of VeMo

A.1 Config

```
-----------------------------------------------
-- Class:   Config
-- Description:  Config contains configuration values
-----------------------------------------------
--
-- Values definition section
--
values
--indicates the range in which units in the system can see each other
public static Range : nat = 40;
--indicates the periode for which a TrafficData Message is valid
public static TrafficDataLifeTime : nat = 50000;
--indicates the number of TrafficData Message held by the vdm units
public static TrafficDataKeeptNumber : nat = 10;
--indicates the number of vehicles held for calculation congestion
public static TrafficCongestionTrack : nat = 5;
--indicates the vehicle range for congestion
public static TrafficCongestionRange : nat = 1;
 --indicates the threshold speed for congestion
public static TrafficCongestionThreshold : nat = 2;
end Config
```

A.2 Controller

```
-----------------------------------------------
-- Class:   Controller
-- Description:  Controller is main class in
--     every independent VeMo unit
-----------------------------------------------
```

```
--
-- class definition
--
class Controller is subclass of VeMoEntity

--
-- instance variables
--
instance variables
private traffic : Traffic;
 -- the vehicle the VeMo system Controller is placed in.
private vemoVehicle : Vehicle;

private canRun : bool := true;
--
-- Types definition section
--
types

--
-- Operations definition section
--
operations
public Controller : Vehicle ==> Controller
Controller (vehicle) ==
(
 vemoVehicle := vehicle;
 traffic := new Traffic();
);


async public AddOncomingVehicle: VehicleData ==> ()
AddOncomingVehicle(vd) ==
(
if not traffic.ExistVehicleData(vd)
then
 let v = new Vehicle(vd) in
 traffic.AddVehicle(v);
);


public AddTrafficData:  nat * seq of TrafficData ==> ()
AddTrafficData(vemoUnitID, data) ==
(
 --we can't use empty data
 if data = []
 then
  return;

 --did we already exchange information?
 if vemoUnitID in set elems communicatedWith
 then
 return;

 --keep track of who we have communicated with
 if len communicatedWith < Config`TrafficDataKeeptNumber
 and vemoUnitID not in set elems communicatedWith
 then
  communicatedWith := communicatedWith ^ [vemoUnitID]
```

```
 else
  communicatedWith := tl communicatedWith ^ [vemoUnitID];

-- ignore data we sent out ourselfs
 let newTrafficData = [ data(i) | i in set inds data & data(i).GetOriginID() <>
 vemoVehicle.GetID()] in
 (
  if newTrafficData = []
  then
   return;

  --now add new traffic data
  if(len externalTrafficData < Config`TrafficDataKeeptNumber)
  then
   externalTrafficData := externalTrafficData ^ newTrafficData
  else
   externalTrafficData :=  tl externalTrafficData ^ newTrafficData;

  for d in newTrafficData do
  (
    World`env.handleEvent("Vehicle:  " ^ Printer`natToString(GetID()) ^
        " received " ^  d.ToString());
  );

  VeMoController`graphics.receivedMessage(GetID());
 )
);


private AddInternalTrafficData: TrafficData ==> ()
AddInternalTrafficData(data) ==
(
 if(len internalTrafficData < Config`TrafficDataKeeptNumber)
 then
 internalTrafficData := internalTrafficData ^ [data]
 else
 internalTrafficData :=  tl internalTrafficData ^ [data];
);


public GetTrafficData: () ==> [seq of TrafficData]
GetTrafficData() ==
(
 -- deep copy
 return [ new TrafficData(internalTrafficData(i).GetMessage(),
       internalTrafficData(i).GetPosition(),
        internalTrafficData(i).GetDirection(),
        vemoVehicle.GetID())
      | i in set inds internalTrafficData ];
);

public GetID : () ==> nat
GetID()== return vemoVehicle.GetID();

public GetPosition : () ==> [Position]
GetPosition() ==
return vemoVehicle.GetPosition();

public GetDirection: () ==> [Types`Direction]
GetDirection() ==
```

```
return vemoVehicle.GetDirection();

public getVehicle : () ==> [Vehicle]
getVehicle() ==
return  vemoVehicle;

public getVehicleDTO : () ==> [VehicleData]
getVehicleDTO() == vemoVehicle.getDTO();

public EnvironmentReady: () ==> ()
EnvironmentReady() == if(canRun = false) then canRun := true;

public Step: () ==> ()
Step() ==
(
  vemoVehicle.Move();

 --check expired internal data
 for all td in set elems internalTrafficData do
 (
  if td.Expired()
  then
  (
  --remove td
  internalTrafficData := [internalTrafficData(i)
       | i in set inds internalTrafficData
       & internalTrafficData(i) <> td];
  )
 );

 --check for lowgrip, and check if already set at position.
 if vemoVehicle.getLowGrip() = true
 then
 (
 --The position check will only be relevant if the car has speed 0
 if vemoVehicle.GetSpeed() = 0 =>
 not exists data in set elems internalTrafficData
     & Position`Compare(data.GetPosition(), GetPosition()) and
                 data.GetPosition() <>  GetPosition()and
                 data.GetMessage() = <LowGrip>
 then
  let lowGripMsg = new TrafficData(<LowGrip>, GetPosition().deepCopy()
          , GetDirection(), vemoVehicle.GetID())
 in
 AddInternalTrafficData(lowGripMsg);
 );

 --check for turnindicator, and check if already set at position.
 if vemoVehicle.TurnIndicator() = <LEFT>
 then
 (
 --The position check will only be relevant if the car has speed 0
 if vemoVehicle.GetSpeed() = 0 =>
 not exists data in set elems internalTrafficData
     & Position`Compare(data.GetPosition(), GetPosition())
     and  data.GetMessage() = <LeftTurn>
 then
 let turnMsg = new TrafficData(<LeftTurn>, GetPosition().deepCopy()
         ,GetDirection(), vemoVehicle.GetID())
 in
```

```
   AddInternalTrafficData(turnMsg);
 );

  --check for congestion, and check if already set at position.
 if traffic.Congestion() = true
 then
 (
 --The position check will only be relevant if the car has speed 0
 if vemoVehicle.GetSpeed() = 0 =>
 not exists data in set elems internalTrafficData
     & Position`Compare(data.GetPosition(), GetPosition())
       and  data.GetMessage() = <Congestion>
  then
  (
  let congMsg = new TrafficData(<Congestion>, GetPosition().deepCopy()
          , GetDirection(), vemoVehicle.GetID())
   in
   (
    AddInternalTrafficData(congMsg);
   )
  )
 );
);


async public run : () ==> ()
run() == start(self)

--
-- Functions definition section
--
functions

--
-- Values definition section
--
values


--
-- Thread definition section
--
thread
while true do
duration(500)
(
  Step();
  canRun := false;
)


--
-- sync definition section
--
sync
per Step => canRun;
mutex(AddInternalTrafficData,GetTrafficData);
mutex(AddInternalTrafficData);
mutex(Step)
```

```
end Controller
```

A.3 Environment

```
-----------------------------------------------
-- Class:   Environment
-- Description:  Environment class in the VeMo project
-----------------------------------------------

--
-- class definition
--
class Environment

--
-- instance variables
--
instance variables

private vemoCtrl : VeMoController;
private io : IO := new IO();
private inlines : seq of inline := [];
private outlines : seq of char := [];
private busy : bool := true;

--
-- Types definition section
--
types
inline  = Types`Event;
InputTP   = seq of inline;
--
-- Operations definition section
--
operations

 public Environment: seq of char ==> Environment
 Environment(filename) ==
 (
   Printer`OutWithTS("Environment created: "
        ^ "Some aren't used to an environment"
       ^ " where excellence is expected");

  def mk_(-,input) = io.freadval[InputTP](filename) in
  (
  inlines := input;
  );
 );


 public Events: () ==> ()
 Events() ==
 (
    if inlines <> []
    then
```

```
(
 dcl done : bool := false,
 eventOccurred : bool := false,
 curtime : Types`Time := time;

while not done do
(
 def event = hd inlines in
 cases event:
  mk_Types`AddVehicle(-,-,-,-,-,-) ->
  (
   Printer`OutWithTS("Environment: AddVehicle event "
              ^ Printer`natToString(event.ID));
   VeMo`vemoCtrl.addController(
    new Controller(
     new Vehicle(event.ID,
     new Position(event.posX, event.posY), event.speed, event.direction)));

   eventOccurred := true;
  ),
  mk_Types`VehicleRun(-,-) ->
  (
   if event.t <= curtime
   then
   (
     Printer`OutWithTS("Environment: Start Vehicle event "
              ^ Printer`natToString(event.ID));
     let ctrl = vemoCtrl.ctrlUnits(event.ID) in
     (
      --create new Constituent
      let constit = VeMo`addConstituent(1E6) in
      (
      constit.deploy(ctrl);
      VeMo`ve2cons := VeMo`ve2cons munion {ctrl |-> constit};
      );
     );

     let ctrl = vemoCtrl.getController(event.ID) in
     (
     ctrl.run();
     );

       eventOccurred := true;
    )
  ),
  mk_Types`RemoveVehicle(-,-) ->
 (
   if event.t <= curtime
   then
   (
    Printer`OutWithTS("Environment: Remove Vehicle event "
               ^ Printer`natToString(event.ID));

    VeMo`removeConstituent(VeMo`ve2cons(vemoCtrl.ctrlUnits(event.ID)));
    VeMo`vemoCtrl.removeController(event.ID);
       VeMoController`graphics.removeVehicle(event.ID);

    eventOccurred := true;
   )
  ),
```

```
 mk_Types`TrafficLightRun(-,-) ->
(
  if event.t <= curtime
  then
  (
      Printer`OutWithTS("Environment: "
            ^ " Start TrafficLight event");

        let light = vemoCtrl.getTrafficLight(event.ID) in
        start(light);

       eventOccurred := true;
  )
 ),
 mk_Types`VehicleUpdateSpeed(-,-,-) ->
(
  if event.t <= curtime
  then
  (
    Printer`OutWithTS("Environment: SpeedUpdate event: "
         ^ "For vehicle: "
         ^ Printer`natToString(event.ID)
         ^ " New Speed: "
         ^ Printer`natToString(event.speed));

    let c = vemoCtrl.getController(event.ID) in
        c.getVehicle().SetSpeed(event.speed);

    eventOccurred := true;
  )
 ),
 mk_Types`VehicleUpdatePosition(-,-,-,-) ->
 (
   if event.t <= curtime
   then
   (
     let pos = new Position(event.posX, event.posY) in
      let c = vemoCtrl.getController(event.ID) in
          (
          c.getVehicle().SetPosition(pos);
          Printer`OutWithTS("Environment: PositionUpdate event:
          For vehicle: "
           ^ Printer`natToString(event.ID)
           ^ " New position:"
           ^ pos.toString());
          );

       eventOccurred := true;
   )
  ),
  mk_Types`VehicleLowGrip (-,-,-) ->
    (
     if event.t <= curtime
     then
    (
     Printer`OutWithTS("Environment: LowGrip event: "
        ^ "For vehicle: "
        ^ Printer`natToString(event.ID));
     let c = vemoCtrl.getController(event.ID)  in
      c.getVehicle().setLowGrip(event.lowGrip);
```

```
        eventOccurred := true;
        )
      ),
      mk_Types`VehicleTurnIndication(-,-,-) ->
        (
        if event.t <= curtime
        then
        (
        Printer`OutWithTS("Environment: TurnIndication event: "
           ^ "For vehicle: "
           ^ Printer`natToString(event.ID)
           ^ " New indicator: "
           ^ Vehicle`IndicatorToString(event.turn));
         let c = vemoCtrl.getController(event.ID) in
         c.getVehicle().setTurnIndicator(event.turn);

         eventOccurred := true;
        )
      ),
      mk_Types`VehicleUpdateDirection(-,-,-) ->
     (
        if event.t <= curtime
        then
        (
         Printer`OutWithTS("Environment: DirectionUpdate event: "
           ^ "For vehicle: "
           ^ Printer`natToString(event.ID)
           ^ " New Direction: "
           ^ Types`DirectionToString(event.direction));
          let c = vemoCtrl.getController(event.ID) in
            c.getVehicle().SetDirection(event.direction);

         eventOccurred := true;
         )
      ),
      mk_Types`WasteTime(-) ->
      (
        if event.t <= curtime
        then
        (
         Printer`OutWithTS("Environment: Wasting time");

          eventOccurred := true;
        );
       ),
       others -> Printer`OutWithTS("Environment: No match found")
    end;

   if eventOccurred
   then
   (
    inlines := tl inlines;
    done := len inlines = 0;
   )
    else done := true;

    eventOccurred := false;
  );
 )
```

```
   else busy := false;
 );

 public handleEvent : seq of char ==> ()
 handleEvent(s) ==
 (
 Printer`OutWithTS("#Environment Handled System Event: " ^ s);
 outlines := outlines ^ Printer`natToString(time) ^ ": " ^ s ^ "\n";
 );


 public report : () ==> ()
 report() ==
 (
  Printer`OutAlways("\n\nHowever beautiful the strategy," ^
          "you should occasionally look at the results.");
  Printer`OutAlways("**RESULT***");
  Printer`OutAlways("***********");
  Printer`OutAlways(outlines);
  Printer`OutAlways("\n***********");
  Printer`OutAlways("***********");
 );

 public isFinished : () ==> ()
 isFinished() == skip;

 public goEnvironment : () ==> ()
 goEnvironment() == skip;

 public setVeMoCtrl : VeMoController ==> ()
 setVeMoCtrl(vemoController) == vemoCtrl := vemoController;

 public run : () ==> ()
 run() ==
 (
  start(vemoCtrl);
  start(self);
 )
--
--
-- Functions definition section
--
functions

--
-- Values definition section
--
values


--
-- Threads definition section
--
thread
(
 while busy do
 (
   Events();
   vemoCtrl.EnvironmentReady();
 );
```

```
 Printer`OutAlways("No more events;");
)
--
-- sync definition section
--
sync
 per isFinished => not busy;
 per Events => #fin(Events) = #fin(goEnvironment);
 mutex(handleEvent)

end Environment
```

A.4 Position

```
-----------------------------------------------
-- Class:   Position
-- Description:  Defines a X,Y position
-----------------------------------------------

--
-- class definition
--
class Position

--
-- instance variables
--
instance variables

private x: int;
private y: int;

--
-- Types definition section
--
types

--
-- Operations definition section
--
operations

public Position: int * int ==> Position
Position(x_, y_) ==
(
 x := x_;
 y := y_;
);

public X: () ==> int
X() ==
(
 return x;
);
```

```
public Y: () ==> int
Y() ==
(
 return y;
);

public setX : int ==> ()
setX(newX) ==
(
  x := newX

);

public setY: int ==> ()
setY(newY) ==
(
y := newY

);


public toString : () ==> seq of char
toString() ==
(
 return "position X: "
 ^ VDMUtil`val2seq_of_char[int](x)
  ^ " Y: " ^ VDMUtil`val2seq_of_char[int](y)
);

public inRange : Position * int ==> bool
inRange(p, range) ==
(
   let xd = x - p.X(), yd = y -p.Y() in
   (
     let d = MATH`sqrt((xd * xd) + (yd * yd)) in
     (
       return d <= range;
     )
   )
);

public deepCopy : () ==> Position
deepCopy() ==
(
 let newPos = new Position(x,y)
 in
 return newPos;
)

--
-- Functions definition section
--
functions
public static Compare: Position * Position -> bool
Compare(a,b) ==
a.X() = b.X() and a.Y() = b.Y()


--
```

```
-- Values definition section
--
values

end Position
```

A.5 Printer

```
-----------------------------------------------
-- Class:   Printer
-- Description:  Printes text seq via IO
-----------------------------------------------

--
-- class definition
--
class Printer

instance variables
  private static echo : bool := true


--
-- Operations definition section
--
operations

  public static Echo : bool ==> ()
  Echo(v) ==
  echo := v;

  public static OutAlways: seq of char ==> ()
  OutAlways (pstr) ==
    def - = new IO().echo(pstr ^ "\n") in skip;


  public static OutWithTS: seq of char ==> ()
  OutWithTS (pstr) ==
    def - = new IO().echo(Printer`natToString(time)
           ^ ": " ^ pstr ^ "\n")
           in skip;

  public static natToString : nat ==> seq of char
  natToString(n) ==
  (
    return VDMUtil`val2seq_of_char[nat](n);
  );

  public static intToString : int ==> seq of char
  intToString(i) ==
  (
    return VDMUtil`val2seq_of_char[int](i);
  );

sync
```

```
 mutex(OutWithTS)

end Printer
```

A.6 Traffic

```
-----------------------------------------------
-- Class:   Traffic
-- Description:  Traffic contains the vehicles known by VeMo
-----------------------------------------------

--
-- class definition
--
class Traffic

--
-- instance variables
--
instance variables

private vehicles: seq of Vehicle := [];
inv len vehicles <= 5;
--
-- Types definition section
--
types

--
-- Operations definition section
--
operations

public AddVehicle: Vehicle ==> ()
AddVehicle(vehicle) ==
(
 if(len vehicles < Config`TrafficCongestionTrack)
 then
 vehicles := vehicles ^ [vehicle]
 else
 vehicles :=  tl vehicles ^ [vehicle]
)
pre vehicle not in set elems vehicles;


public ExistVehicle : Vehicle ==> bool
ExistVehicle(v) ==
(
 return {vec | vec in set elems vehicles & v.GetID() = vec.GetID()} <> {};
);


public ExistVehicleData : VehicleData ==> bool
ExistVehicleData(v) ==
(
```

```
 return {vec | vec in set elems vehicles & v.GetID() = vec.GetID()} <> {};
);


public GetVehicles: () ==> seq of Vehicle
GetVehicles() ==
 return vehicles;


public Congestion: () ==> bool
Congestion() ==
(
 dcl inrange : set of Vehicle := {};

 for v in vehicles do
 (
  let vs = FindInRangeWithSameDirection(v,vehicles)
  in
  inrange := inrange union vs;
 );

 if card inrange = 0
 then return false;

 let avgspeed = AverageSpeed(inrange)
 in
 (
 return avgspeed < Config`TrafficCongestionThreshold;
 )
);


private AverageSpeed: set of Vehicle ==> nat
AverageSpeed(vs) ==
(
  dcl sumSpeed: nat := 0;
  for all v in set vs do
   sumSpeed := sumSpeed + v.GetSpeed();
  return (sumSpeed/card vs)
)
pre card vs <> 0;
--
-- Functions definition section
--
functions

 -- compare the range of two vehicles
public InRange : Vehicle * Vehicle -> bool
InRange(v1,v2) ==
 let pos1 = v1.GetPosition(), pos2 = v2.GetPosition()
 in
  pos1.inRange(pos2, Config`TrafficCongestionRange);


-- compare the range of a single vehicle to a set of vehicles moving in
-- the same direction
public FindInRangeWithSameDirection : Vehicle * seq of Vehicle
  -> set of Vehicle
FindInRangeWithSameDirection(v ,vs) ==
 let dir = v.GetDirection() in
```

```
  { ir | ir in set elems vs & v <> ir
          and dir = ir.GetDirection()
          and InRange(v,ir) = true }


--
-- Values definition section
--
values

--
-- sync definition section
--
sync
mutex(Congestion, AddVehicle);
mutex(ExistVehicle, AddVehicle);
mutex(GetVehicles, AddVehicle);
mutex(AddVehicle);

end Traffic
```

A.7 TrafficData

```
-----------------------------------------------
-- Class:   TrafficData
-- Description:  TrafficData is the base for different types of
--     messages in the system.
-----------------------------------------------

--
-- class definition
--
class TrafficData

--
-- instance variables
--
instance variables
private dir: Types`Direction;
private pos: Position;
private message: MessageType;
private timeToLive : nat;
private originId : nat;

--
-- Types definition section
--
types
public MessageType = <LowGrip> | <Congestion> | <LeftTurn> | <RedLight>;

--
-- Operations definition section
--
operations
public TrafficData: MessageType * Position * Types`Direction * nat
```

```
 ==> TrafficData
TrafficData(m,p,d,origin) ==
(
  pos := p ;
  message := m;
  dir := d;
  originId := origin;
  timeToLive := time + Config`TrafficDataLifeTime;
);

public GetPosition: () ==> Position
 GetPosition() ==
 return pos;

public GetMessage: () ==> MessageType
 GetMessage() ==
 return message;

public GetDirection: () ==> Types`Direction
GetDirection() ==
return dir;

public GetOriginID: () ==> nat
GetOriginID() ==
return originId;

public Expired : () ==> bool
Expired() ==
return time >= timeToLive;

public ToString : () ==> seq of char
ToString() ==
return "traffic data reporting "
  ^ MessageTypeToString(message)
  ^ " moved " ^ Types`DirectionToString(dir)
  ^ " at " ^ pos.toString()
  ^ " with lifetime "
  ^ Printer`intToString(timeToLive - time)
  ^ " originally issued by "
  ^ Printer`intToString(originId);

--
-- Functions definition section
--
functions

public static MessageTypeToString : MessageType -> seq of char
MessageTypeToString(m) ==
(
cases m:
<LowGrip>-> "Low Grip",
<Congestion>-> "Congestion ",
<LeftTurn>-> "Left Turn",
<RedLight> -> "Red Light"
end
)

--
-- Values definition section
--
```

```
values

end TrafficData
```

A.8 TrafficLight

```
-----------------------------------------------
-- Class:   TrafficLight
-- Description:  TrafficLight the VeMo project
-----------------------------------------------

--
-- class definition
--
class TrafficLight is subclass of VeMoEntity

--
-- instance variables
--
instance variables
private pos: Position;
private greenLightTime : nat1;
private greenLightCount : nat;
private greenDir: Types`Direction;
private id : nat;

private canRun : bool := true;
--
-- Types definition section
--
types

--
-- Operations definition section
--
operations

public TrafficLight: nat * Position * nat1 ==> TrafficLight
TrafficLight(identifier ,p, t) ==
(
 pos := p ;
 greenLightTime := t;
 greenLightCount := 0;
 id := identifier;

 greenDir := <NORTH>
);

public AddTrafficData: nat * seq of TrafficData ==> ()
AddTrafficData(vemoUnitID, data) ==
(
 --we can't use empty data
 if data = []
 then
```

```
  return;

 --did we already exchange information?
 if vemoUnitID in set elems communicatedWith
 then
  return;

 --keep track of who we have communicated with
 if len communicatedWith < Config`TrafficDataKeeptNumber
 then
  communicatedWith := communicatedWith ^ [vemoUnitID]
 else
  communicatedWith := tl communicatedWith ^ [vemoUnitID];

 for d in data do
 (
   World`env.handleEvent("TrafficLight: " ^ Printer`natToString(id) ^
       " received " ^  d.ToString());

 --add internal data
 if(len internalTrafficData < Config`TrafficDataKeeptNumber)
  then
   internalTrafficData := internalTrafficData ^ data
  else
   internalTrafficData :=  tl internalTrafficData ^ data;
 );
);


async public AddOncomingVehicle: VehicleData ==> ()
AddOncomingVehicle(vd) ==
(
  skip;
);


public GetTrafficData: () ==> [seq of TrafficData]
GetTrafficData() ==
(
Printer`OutWithTS("GetTraffic" ^ Printer`natToString(len internalTrafficData));
-- deep copy
return [ new TrafficData(internalTrafficData(i).GetMessage(),
       internalTrafficData(i).GetPosition(),
        internalTrafficData(i).GetDirection(),
        internalTrafficData(i).GetOriginID())
      | i in set inds internalTrafficData ];
);

public GetPosition: () ==> Position
GetPosition() ==
 return pos;

public GetDirection: () ==> [Types`Direction]
GetDirection() == is not yet specified;

public GreenLightPath: () ==> Types`Direction
GreenLightPath() ==
 return greenDir;

public GetID: () ==> nat
```

```
GetID() ==
 return id;

private Step: () ==> ()
Step() ==
(
    if (greenLightCount > greenLightTime) then
    (
     VeMoController`graphics.changeGreenLightDir(id);
     greenDir := CrossDirection(greenDir);
     greenLightCount := 0;
    );

    greenLightCount := greenLightCount + 1;

 --check expired internal data
   for all td in set elems internalTrafficData do
   (
    if td.Expired() then
    (
     --remove td
     internalTrafficData := [internalTrafficData(i)
                | i in set inds internalTrafficData
                & internalTrafficData(i) <> td];
    )
   );
);

async public run : () ==> ()
run() == start(self);

public EnvironmentReady: () ==> ()
EnvironmentReady() == if(canRun = false) then canRun := true;

--
-- Functions definition section
--
functions

public static CrossDirection : Types`Direction -> Types`Direction
CrossDirection(d) ==
cases d:
<NORTH> -> <EAST>,
<SOUTH> -> <WEST>,
<EAST>  -> <NORTH>,
<WEST>  -> <SOUTH>
end;

--
-- Values definition section
--
values

--
-- Thread definition section
--
thread
while true do
duration(500)
(
```

```
  Step();
  canRun := false;
)

--
-- sync definition section
--
sync
per Step => canRun;
mutex(GreenLightPath);
mutex(Step,GreenLightPath);

end TrafficLight
```

A.9 Types

```
-----------------------------------------------
-- Class:   Types
-- Description:  Defines simple types
-----------------------------------------------

--
-- class definition
--
class Types

types
public Time = nat;
public Direction = <NORTH> | <SOUTH> | <EAST> | <WEST>;

public Event = AddVehicle | VehicleRun | RemoveVehicle | TrafficLightRun
      | VehicleUpdateSpeed | VehicleUpdatePosition | WasteTime
          | VehicleUpdateDirection | VehicleLowGrip | VehicleTurnIndication ;

public AddVehicle ::
 ID : nat
 posX : int
    posY : int
 direction : Direction
 speed : nat
 t : Time;

public RemoveVehicle ::
        ID : nat
        t : Time;

public VehicleRun ::
        ID : nat
        t : Time;

public TrafficLightRun ::
        ID : nat
        t : Time;
```

```
public VehicleUpdateSpeed ::
        ID : nat
        speed : real
        t : Time;

public VehicleUpdatePosition ::
  ID : nat
     posX : nat
        posY : nat
        t : Time;

public VehicleUpdateDirection ::
  ID : nat
  direction : Direction
        t : Time;

public VehicleLowGrip ::
        ID : nat
        lowGrip : bool
        t : Time;

public VehicleTurnIndication ::
        ID : nat
        turn : Vehicle`Indicator
        t : Time;
public WasteTime ::
        t : Time;

functions
  public static DirectionToString : Direction -> seq of char
  DirectionToString(d) ==
  (
   cases d:
   <NORTH>-> "NORTH",
   <SOUTH>-> "SOUTH",
   <EAST>-> "EAST",
   <WEST>-> "WEST"
   end
  );

  public static DirectionToGraphics : Direction -> nat
  DirectionToGraphics(d) ==
  (
    cases d:
    <NORTH>-> 1,
    <SOUTH>-> 5,
    <EAST>->  3,
    <WEST>->  7
    end
  );

end Types
```

A.10 VeMo

```
-----------------------------------------------
-- Class:   VeMo
-- Description:  VeMo is the system class in the VeMo project
-----------------------------------------------

--
-- class definition
--
system VeMo



--
-- instance variables
--
instance variables


--public static bus : Channel := new Channel(1E6, {});
static e : Environment := World`env;
public static ve2cons : inmap VeMoEntity to Constituent := {|->};
public static ve2channel : map nat to (map nat to Channel) := {|->};
public static vemoCtrl : VeMoController := new VeMoController();

public  consti1 : Constituent := new Constituent (1E6);

--traffic light
public static tl1 : TrafficLight := new TrafficLight(999
             ,new Position(20, -70)
             , 40);
--
-- Operations definition section
--
operations

public VeMo: () ==> VeMo
VeMo() ==
(
 ve2cons := VeMo`ve2cons munion {tl1 |-> consti1};
 consti1.deploy(tl1);
);


end VeMo
```

A.11 VeMoController

```
-----------------------------------------------
-- Class:   VeMoController
-- Description:  VeMoController main controller for the VeMo system
```

```
-----------------------------------------------

--
-- class definition
--
class VeMoController

--
-- instance variables
--
instance variables
public ctrlUnits : inmap nat to Controller := {|->};
public lights : inmap nat to TrafficLight := {|->};
inv dom ctrlUnits inter dom lights = {};
inv forall id in set dom ctrlUnits & ctrlUnits(id).GetID() = id;
inv forall id in set dom lights & lights(id).GetID() = id;

--graphics
public static graphics : gui_Graphics := new gui_Graphics();

--keeps bookkeeping/track of connections from a controller to other controllers
private controllerConnections : map nat to set of VeMoEntity := {|->};

static env : Environment := World`env;

--
-- Operations definition section
--
operations

public VeMoController : () ==> VeMoController
VeMoController () ==
(
    graphics.init();
);

public addController: Controller ==> ()
addController(ctrl) ==
(
 -- add to controller map
 ctrlUnits := ctrlUnits munion {ctrl.GetID() |->  ctrl} ;
 -- prep for connection mapping
 controllerConnections(ctrl.GetID()) := {};
 --prep dynamic channel connection
 VeMo`ve2channel := VeMo`ve2channel munion {ctrl.GetID() |-> {|->} };

 --graphics
 let vecID = ctrl.GetID() in
 (
    graphics.addVehicle(vecID);

    let pos = ctrl.GetPosition() in
    graphics.updatePosition(vecID, pos.X(), pos.Y());

    let dir = ctrl.GetDirection() in
    graphics.updateDirection(vecID, Types`DirectionToGraphics(dir));
 )
)
pre ctrl.GetID() not in set (dom ctrlUnits union dom lights);
```

```
public removeController : nat  ==> ()
removeController(ctrlID) ==
(
 ctrlUnits := {ctrlID} <-:  ctrlUnits;
 controllerConnections := {ctrlID} <-: controllerConnections;
);


public addTrafficLight: TrafficLight ==> ()
addTrafficLight(light) ==
(
 let pos = light.GetPosition() in
 graphics.addTrafficLight(light.GetID(), pos.X(), pos.Y());
 -- add to lights map
 lights := lights munion {light.GetID() |-> light};
 -- prep for connection mapping
 controllerConnections(light.GetID()) := {};
 --prep dynamic channel connection
 VeMo`ve2channel := VeMo`ve2channel munion {light.GetID() |-> {|->} };
)
pre light.GetID() not in set dom lights
 and light.GetID() not in set dom ctrlUnits;


public getController : nat ==> Controller
getController(id) ==
  return ctrlUnits(id)
pre id in set dom ctrlUnits;


public getTrafficLight : nat ==> TrafficLight
getTrafficLight(id) ==
(
  return lights(id);
)
pre id in set dom lights;


public EnvironmentReady: () ==> ()
EnvironmentReady() == skip;


public CalculateInRange: () ==> ()
CalculateInRange() ==
(
 -- vehicles/controllers are denoted units in the following
 let units : set of VeMoEntity = rng ctrlUnits union rng lights in
 (
  -- for all units, find the ones in range.
  -- This could be optimized given that if one unit can see another unit,
  -- then they can see each other, no need to calculate the range again
  -- for units seeing each other. However this will be complex, given that
  -- one unit might have serveral units in its range that aren't in range
  -- of each other.
  for all unit in set units do
  (
    let inrange = FindInRange(unit, units)
    in
    (
```

```
  -- find new controllers that has come in range since last check and
  --create a channel connection.
  let newConnection =  inrange \ controllerConnections(unit.GetID()) in
  for all conn in set newConnection do
  (
   let newChnl = VeMo`addChannel(1E6) in
   (
    VeMo`connect(VeMo`ve2cons(unit), newChnl);
    VeMo`connect(VeMo`ve2cons(conn), newChnl);
    VeMo`ve2channel(unit.GetID()) := (VeMo`ve2channel(unit.GetID()))
                                ++ { conn.GetID() |-> newChnl};
   );

   graphics.connectVehicles(unit.GetID() ,conn.GetID());
   Printer`OutWithTS("+" ^ Printer`natToString(conn.GetID()));
  );

   -- find controllers that has gone out of range since last check
   -- and tear down the channel connection.
   let lostConnection = controllerConnections(unit.GetID()) \ inrange  in
   for all lost in set lostConnection do
   (
    VeMo`disconnect(VeMo`ve2cons(unit),
                    VeMo`ve2channel(unit.GetID())(lost.GetID()));
    graphics.disconnectVehicles(unit.GetID(), lost.GetID());
    VeMo`ve2channel(unit.GetID()) :=
                   ({lost.GetID()} <-: VeMo`ve2channel(unit.GetID()));
    Printer`OutWithTS("-" ^ Printer`natToString(lost.GetID()));
   );
    -- remember inrange for a specific unit, to enable history/bookkeeping
    -- of new and lost connections
     controllerConnections(unit.GetID()) := inrange;
 )
);

--the communication approach is different for trafficlights and vehicles.
--So the communication is split. Lights communicate with everything from
--all directions
for all light in set rng lights do
(
  --find in range
  let inrange = FindInRange(light, rng ctrlUnits) in
  (
    if(card inrange > 0) then
     for all vehicle in set inrange do
     (
       light.AddTrafficData(vehicle.GetID() ,vehicle.GetTrafficData());
       vehicle.AddTrafficData(light.GetID() ,light.GetTrafficData());
     );
   );
 );

 graphics.sleep();

 -- vehicles only communicate with vehicle coming from the opposite
     -- direction.
 -- only request data, the way the loop is built will ensure that all
 -- units will request data.
 for all unit in set rng ctrlUnits do
 (
```

```
      let pos = unit.GetPosition() in
       graphics.updatePosition(unit.GetID(), pos.X(), pos.Y());

      let dir = unit.GetDirection() in
       graphics.updateDirection(unit.GetID(), Types`DirectionToGraphics(dir));

      let communicateWith = controllerConnections(unit.GetID()) in
        if(card communicateWith > 0)
         then
          for all oncomingVehicle
          in set FindInRangeWithOppositeDirection(unit, communicateWith)
          do
          (
           Printer`OutWithTS("2nd AddTrafficData called. "
            ^ Printer`natToString(card communicateWith)
            ^ " Oncoming: "
            ^ Printer`natToString(oncomingVehicle.GetID())
            ^ " from " ^ Printer`natToString(unit.GetID()));
            unit.AddTrafficData(oncomingVehicle.GetID(),
                                oncomingVehicle.GetTrafficData());

           let vehicleDTO = unit.getVehicleDTO() in
            oncomingVehicle.AddOncomingVehicle(vehicleDTO);
          );
     );
 );

 -- synchronization, when we have calculated inrange allow vehicles to
 --move again
 let  vemoUnits = ctrlUnits munion lights in
    for all u in set rng vemoUnits do u.EnvironmentReady();
);

--
-- Functions definition section
--
functions
public static OppositeDirection : Types`Direction -> Types`Direction
OppositeDirection(d) ==
cases d:
<NORTH> -> <SOUTH>,
<SOUTH> -> <NORTH>,
<EAST>  -> <WEST>,
<WEST>  -> <EAST>
end;


-- compare the range of a single vehicle/controller to a
-- set of vehicles/controllers
public FindInRange : VeMoEntity * set of VeMoEntity -> set of VeMoEntity
FindInRange(v, vs) ==
   let inrange = { ir | ir in set vs & v <> ir and InRange(v,ir) = true }
   in
   inrange;


-- compare the range of two vehicles/controllers
public InRange : VeMoEntity * VeMoEntity -> bool
InRange(u1,u2) ==
 let pos1 = u1.GetPosition(), pos2 = u2.GetPosition()
```

```
 in
 pos1.inRange(pos2, Config`Range);


-- compare the range of a single vehicle/controller to a set of
-- vehicles/controllers moving in the opposite direction
public FindInRangeWithOppositeDirection : VeMoEntity * set of VeMoEntity
 -> set of Controller
FindInRangeWithOppositeDirection(u ,us) ==
 let dir = OppositeDirection(u.GetDirection()) in
  let inrange = { ir | ir in set FindInRange(u, us) &
       isofclass(Controller,ir)  and dir = ir.GetDirection()} in inrange;


--
-- Thread definition section
--
thread
(
 while true do
 (
  CalculateInRange();
  env.goEnvironment();
 )
)

--
-- sync definition section
--
sync
per CalculateInRange => #fin(EnvironmentReady) > #fin(CalculateInRange);

mutex (CalculateInRange, addController, getController);
mutex (addController);
mutex (getController);
mutex (CalculateInRange)
end VeMoController
```

A.12 VeMoEntity

```
class VeMoEntity

instance variables
 -- traffic data issued by this controller, that will be passed on
 -- other controllers.
 protected internalTrafficData : seq of TrafficData := [];
 inv len internalTrafficData <=  Config`TrafficDataKeeptNumber;
 -- traffic data from other controllers moving in the opposite direction,
 protected externalTrafficData : seq of TrafficData := [];
 -- this will not be passed on as it makes no sense with the current
 -- warning types.
 inv len externalTrafficData <= Config`TrafficDataKeeptNumber;
 --keep track of whom we have communicated with.
 protected communicatedWith : seq of nat := [];
```

```
 inv len communicatedWith <= Config`TrafficDataKeeptNumber;
operations

 public AddTrafficData: nat * seq of TrafficData ==> ()
 AddTrafficData(vemoUnitID, data) == is subclass responsibility;

 async public AddOncomingVehicle: VehicleData ==> ()
 AddOncomingVehicle(vd) == is subclass responsibility;

 public GetPosition : () ==> [Position]
 GetPosition() == is subclass responsibility;

 public GetDirection: () ==> [Types`Direction]
 GetDirection() == is subclass responsibility;

 public EnvironmentReady : () ==> ()
 EnvironmentReady() == is subclass responsibility;

 public GetID : () ==> nat
 GetID()== is subclass responsibility;

 public GetTrafficData : () ==> [seq of TrafficData]
 GetTrafficData() == is subclass responsibility;

end VeMoEntity
```

A.13 Vehicle

```
-----------------------------------------------
-- Class:   Vehicle
-- Description:  Vehicle class describes the physical moving
--     elements in the system
-----------------------------------------------

--
-- class definition
--
class Vehicle

--
-- instance variables
--
instance variables

private dir: Types`Direction;
private speed : nat;
private lowgrip : bool;
private turnIndicator : Indicator := <NONE>;
private pos : Position;
private id : nat;
--
-- Types definition section
--
types
public Indicator = <LEFT> | <RIGHT> | <NONE>;
--
-- Operations definition section
--
operations

public Vehicle:  nat * Position * nat * Types`Direction ==> Vehicle
Vehicle(identifier, p, s, d) ==
(
  pos := p;
  speed := s;
  dir := d;
  id := identifier;
  lowgrip := false;
);


public Vehicle:  VehicleData ==> Vehicle
Vehicle(vdDTO) ==
(
  pos := vdDTO.GetPosition();
  speed := vdDTO.GetSpeed();
  dir := vdDTO.GetDirection();
  id := vdDTO.GetID();
  lowgrip := vdDTO.getLowGrip();
);


public GetDirection: () ==> Types`Direction
```

```
GetDirection() ==
 return dir;

async public SetDirection: Types`Direction  ==> ()
SetDirection(d) ==
(
 dir := d;
);

public GetSpeed: () ==> nat
GetSpeed() ==
 return speed;

async public SetSpeed: nat ==> ()
SetSpeed(s) ==
 speed := s;

public getLowGrip: () ==> bool
getLowGrip() ==
(
 return lowgrip
);

async public setLowGrip: bool ==> ()
setLowGrip(lg) ==
(
 lowgrip := lg;
);

public TurnIndicator: () ==> Indicator
TurnIndicator() ==
 return turnIndicator;

async public setTurnIndicator: Indicator ==> ()
setTurnIndicator(indicator) ==
(
 turnIndicator := indicator;
);

public GetPosition: () ==> Position
GetPosition() ==
 return pos;

async public SetPosition: Position ==> ()
SetPosition(p) ==
 pos := p;

public GetID: () ==> nat
GetID() ==
trap <MessageLost> with
skip -- introduce error lgo
in
(
 exit <MessageLost>;
 return id;
);

public Move : () ==> ()
Move() ==
(
```

```
 cases dir:
 <NORTH> -> pos.setY(pos.Y() + speed),
 <SOUTH> -> pos.setY(pos.Y() - speed),
 <EAST>  -> pos.setX(pos.X() + speed),
 <WEST>  -> pos.setX(pos.X() - speed)
 end;

Printer`OutWithTS("Vehicle "
   ^ Printer`natToString(id)
   ^ " moved "
   ^ Types`DirectionToString(dir)
   ^ " to  " ^ pos.toString()
   ^ " with speed "
   ^ Printer`natToString(speed));
);

public getDTO : () ==> VehicleData
getDTO() ==
(
 return new VehicleData(id, pos.deepCopy(), speed, dir, lowgrip);
)

--
-- Functions definition section
--
functions

public static IndicatorToString : Indicator -> seq of char
IndicatorToString(i) ==
(
cases i:
<LEFT>-> "LEFT",
<RIGHT>-> "RIGHT",
<NONE>-> "NONE"
end
)
--
-- sync definition section
--
sync
 mutex(Move);
 mutex(Move, SetPosition, GetPosition);
 mutex(SetPosition);
 mutex(SetDirection);
 mutex(GetDirection, SetDirection);
 mutex(SetSpeed);
 mutex(GetSpeed, SetSpeed);
 mutex(setLowGrip);
 mutex(getLowGrip, setLowGrip);
 mutex(setTurnIndicator);
 mutex(TurnIndicator,setTurnIndicator);

end Vehicle
```

A.14 VehicleData

```
-----------------------------------------------
-- Class:   Vehicle
-- Description:  DTO representing the data in the Vehicle class
-----------------------------------------------

--
-- class definition
--
class VehicleData

--
-- instance variables
--
instance variables

private dir: Types`Direction;
private speed : nat;
private lowgrip : bool;
private turnIndicator : Indicator := <NONE>;
private pos : Position;
private id : nat;
--
-- Types definition section
--
types
public Indicator = <LEFT> | <RIGHT> | <NONE>;
--
-- Operations definition section
--
operations

public VehicleData : nat * Position * nat * Types`Direction * bool
 ==> VehicleData
VehicleData(identifier, p, s, d, grip) ==
(
  pos := p;
  speed := s;
  dir := d;
  id := identifier;
  lowgrip := grip;
);

public GetDirection: () ==> Types`Direction
GetDirection() ==
return dir;

public GetSpeed: () ==> nat
GetSpeed() ==
return speed;

public getLowGrip: () ==> bool
getLowGrip() ==
(
return lowgrip
);

public TurnIndicator: () ==> Indicator
TurnIndicator() ==
return turnIndicator;
```

```
public GetPosition: () ==> Position
GetPosition() ==
return pos.deepCopy();

public GetID: () ==> nat
GetID() ==
return id;


--
-- Values definition section
--
values

--
-- sync definition section
--

end VehicleData
```

A.15 World

```
-----------------------------------------------
-- Class:   World
-- Description:  World class in the VeMo project
-----------------------------------------------

--
-- class definition
--
class World

--
-- instance variables
--
instance variables

public static env : [Environment] := new Environment("inputvalues.txt");

--
-- Types definition section
--
types

--
-- Operations definition section
--
operations

public World: () ==> World
World() ==
(
 Printer`OutAlways("Creating World");
```

```
 VeMo`vemoCtrl.addTrafficLight(VeMo`tl1);
 env.setVeMoCtrl(VeMo`vemoCtrl);

 Printer`OutAlways("World created: "
     ^ " Maybe this world is another planet's hell.");
 Printer`OutAlways("----------------------------------------\n");
);

public Run: () ==> ()
Run() ==
(
  env.run();
  env.isFinished();
  duration(1000)
  env.report();
  Printer`OutWithTS("End of this world");
);

public static Verbose : bool ==> ()
Verbose(v) == Printer`Echo(v);

--
-- Functions definition section
--
functions

--
-- Values definition section
--
values

end World
```

A.16 gui_Graphics

```
class gui_Graphics

operations

public init : () ==> ()
init()== is not yet specified;

public sleep: () ==> ()
sleep()== is not yet specified;

public addVehicle: int ==> ()
addVehicle(vecID)== is not yet specified;

public removeVehicle : int ==> ()
removeVehicle(vecID) == is not yet specified;

public addTrafficLight : int * int * int ==> ()
addTrafficLight(id, posx, posy) == is not yet specified;
```

```
public changeGreenLightDir : int ==> ()
changeGreenLightDir(id) == is not yet specified;

public connectVehicles: int * int ==> ()
connectVehicles(vecID, vecID2)== is not yet specified;

public disconnectVehicles: int * int ==> ()
disconnectVehicles(vecID, vecID2)== is not yet specified;

public updatePosition: int * int * int ==> ()
updatePosition(vecID, x, y)== is not yet specified;

public updateDirection: int * int ==> ()
updateDirection(vecID, dir)== is not yet specified;

public receivedMessage : int ==> ()
receivedMessage(vecID) == is not yet specified;

end gui_Graphics
```

Claus Ballegaard Nielsen, Modelling Dynamic Topologies via Extensions of VDM-RT, 2012.

Department of Engineering
Aarhus University
Edison, Finlandsgade 22
8200 Aarhus N
Denmark

Tel.: +45 4189 3000