DYNAMIC RECONFIGURATION OF DISTRIBUTED SYSTEMS IN VDM-RT

BY

CLAUS BALLEGÅRD NIELSEN

20086014

MASTER'S THESIS
IN
TECHNICAL INFORMATION TECHNOLOGY
DISTRIBUTED REAL-TIME SYSTEMS

SUPERVISOR: PROF. PETER GORM LARSEN

Aarhus School of Engineering

December 17, 2010

Claus Ballegård Nielsen
20086014

Peter Gorm Larsen
Supervisor

FACULTY OF SCIENCE
AARHUS UNIVERSITY

This Thesis is printed with Computer Modern 11pt
Layout and typography by the author using LaTeX

The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer

Edsger W. Dijkstra

Abstract

Nowadays computers are found everywhere and with an ever increasing functionality. More and more of them exist in cooperative networks with other computers in order to form distributed systems. These systems often have high demands for adaptability and scalability which means that they are expected to adapt to requirements that are characterized by constant change. This can be achieved through a dynamic behavior in which the system has the ability to adapt to system modification by the means of dynamic reconfiguration. However as the possibilities in software increase, so does the complexity of system specifications and designs, making development of distributed systems increasingly challenging.
VDM-RT is a modeling language used to validate and verify software through the entire software development process, with a particular focus on distributed systems. In the existing VDM-RT language the architecture of the distributed systems is preconfigured in the model and it stays static throughout the execution of the model.
This thesis has evaluated the feasibility and value of extending VDM-RT with dynamic reconfiguration in order to facilitate the modeling of dynamic distributed systems and the validation of subsequent implementations. Common principles of dynamic reconfiguration have been examined and the existing literature has been evaluated to form a foundation for an extension of VDM-RT.
The result is a well-founded language extension, with defined semantics, for expressing dynamic reconfiguration in VDM-RT, as well as the development of a interpreter prototype and the successful application of the extension into two case studies.

Resume

Dynamisk Rekonfigurering af Distribuerede Systemer i VDM-RT

Nu om dage kan computere findes alle steder og med en altid stigende funktionalitet. Flere og flere computere forekommer i kooperative netværk med andre computere med det formål at skabe distribuerede systemer. Disse systemer har ofte høje krav til tilpasning og skalerbarhed hvilket betyder at de forventes at kunne tilpasse sig systemkrav som er karakteriseret ved konstant forandring. Dette kan opnås gennem en dynamisk opførelse hvor systemet er i stand til at tilpasse sig ændringer ved hjælp af dynamisk rekonfigurering. Som mulighederne i software forøges; øges kompleksiteten af system specifikationer og designs også, hvilket gør udviklingen af distribuerede systemer tiltagende udfordrende.

VDM-RT er et modellerings sprog som anvendes til at validere og verificere software gennem hele udviklingsprocessen, med et specifikt fokus på distribuerede systemer. I det eksisterende VDM-RT sprog er arkitekturen af det distribuerede system forudkonfigureret i modellen og den forbliver statisk gennem hele eksekveringen af modellen.

Dette speciale evaluerer muligheden og værdien i en dynamisk rekonfigureringsudvidelse af VDM-RT, for at fremme modelleringen af dynamiske distribuerede systemer og valideringen af efterfølgende implementeringer. Generelle principper i dynamisk rekonfigurering er blevet undersøgt og eksisterende litteratur er blevet evalueret for at skabe et grundlag for en udvidelse af VDM-RT.

Resultatet er en velbegrundet sprogudvidelse, inklusiv semantik, til at udtrykke dynamisk rekonfigurering i VDM-RT, samt udviklingen af en fortolker prototype og en succesfuld anvendelse af udvidelse i to eksempler/cases.

Acknowledgements

First of all I would like to thank my supervisor Peter Gorm Larsen for his invaluable inputs and guidance. His unrestrained dedication is remarkable and in the role as motivator he is outstanding.
Secondly I would like thank Nick Battle for his valuable comments, observations and willingness in answering questions about VDMJ, which without doubt has increased the quality of this thesis immensely.
Thirdly I would like to thank Kenneth Lausdahl and Augusto Ribeiro for their support with the Overture development environment, which made the initial development phase so much easier.
Further thanks go to John Fitzgerald, Einar Broch Johnsen and Olaf Owe for their inputs which helped to shape this thesis.
Finally gratitude goes to my family and friends for their never ending support.

Table of Contents

List of Figures

Chapter 1

Introduction

This chapter presents the background, motivation as wells as thesis scope and the goals for the thesis, in the first three sections. Section 1.4 provides a brief description and reasoning for the choice of case studies and related work is supplied. Lastly the chapter contains a reading guide in Section 1.6 and an overview of the thesis structure in Section 1.7.

1.1. Background

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.

Douglas Adams
(1952–2001)
Mostly Harmless

Through the evolution of the computer industry the computer have taken many forms; from giant mainframes with terminals, over standalone desktop machines to private computers connected in massive networks. Nowadays computers are everywhere and even the smallest units can contain multiple CPUs and be part of a network [Benini&02]. The advancement and growing necessity of distributed computing resources [Fich&04, Satyanarayanan01] has lead to the development process of embedded distributed systems becoming increasingly challenging. The distribution of computing resources as well as demands for adaptability and dependability has become more fundamental requirements of embedded distributed systems. With these increasing demands the complexity of the specification and system design increases as well.

Analysis of the design at an early development stage enables the evaluation of the compositions between the different software parts composing the system, and contributes to a greater confidence in the system functionality meeting the requirements. This can enable the software engineers to challenge those "*things that might go wrong*" and at the same time be assured that the "*things that cannot possibly go wrong*" actually will not.

The purpose of formal modelling is to selectively focus on specific aspects of requirements and designs in order to obtain a clearer understanding and reasoning of a system. In a model the central requirements and desirable properties of a system can be analyzed in detail, to reveal the essential underlying behavior and errors. Errors which may otherwise remain hidden until late in the development process or even worse remain in a deployed system.

Modelling has the aim of directing the thought process to ponder, challenge and analyze the perception of the desired behavior through the use of abstraction. Abstraction is the intentional disregard of certain aspects of a system or functionality in order to extract the underlying essence. Abstraction allows for the omission of details which are not relevant to the purpose of the system, with the objective of describing the *what* and not the *how* of a certain behavior [Kramer07].
The challenge of using modelling is to determine the right level of abstraction. If the model includes so many details that the model actually is a complete implementation, the model is not abstract enough to contract the desired analysis and the detailed focus has been lost. Oppositely if the model is too abstract, it might not supply the intended insight into the system as there is no properties or behavior to focus on at all.

Finding the right level of abstraction is therefore the challenge of removing details that are not relevant for the purpose of the anlysis desired. This is not only a task for the System Developer$^{\mathcal{T}}$ creating a model, but also the responsibility of the modeling language itself. The modeling language must be able to reflect a proper relationship between the properties of the the formal model and possible implementations of the final system. The language should be able to express the most important properties needed to ensure the validity of the model in a subsequent implementation, while keeping a distance to implementation specific details.

In the development of critical systems, formal methods has been used to model and analyze the requirements in reference to the system design. Formal methods can 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 explore and validate the system design and behavior [Verhoef08].

Many embedded distributed systems are designed assuming a static set of hardware and software configurations. However newer computing systems have high demands for adaptability, scalability and availability which can be made available through a flexible design. These systems often have high quality of service demands or have an autonomic [Ganek&03] or ubiquitous [Weiser91] behavior in which they are expected to adapt to diverse environments and requirements that are characterized by constant change.
This can be achieved through a dynamic behavior in which the system has the ability to adapt to system modification and extensions by the means of dynamic reconfiguration and migration mechanisms. Dynamic reconfiguration of a distributed application is the act of changing the configuration of the system during runtime [Kramer&85].

1.2. Motivation

The grounds for this thesis was laid in a course on VDM-RT. During this course a model was created of a system that has a highly autonomous behavior and in which wireless networks is a central factor in the validation of the system. The model use the primitives supplied by VDM-RT, such as `BUS` connections and `CPU`s to model the wireless network
In VDM-RT the deployment of functionality on `CPU`s as well as the network topology between `CPU`s is preconfigured in the model and it stays static throughout the execution of the model. This restraint means that it was very difficult to create a representation of the wireless network in the model, as disconnections and new connections cannot occur.

The need for modelling this kind of behavior, exemplifies the classic dilemma in modelling languages; weighing the aspiration of keeping the level of abstraction, against the need of keeping a relation to the real implementation.
The limitation of the static system architecture of VDM-RT makes it very difficult to model systems that require a dynamic behavior, whether it being needed for QoS$^{\mathcal{T}}$ demands or because of the nature of the system, like in ubiquitous computing systems.
The level of abstraction present in VDM-RT may possibly be too great, as it excludes natural models of specific types of systems.
An investigation is needed to determine if a dynamic reconfiguration extension to VDM-RT will be valuable in the modeling of certain types of dynamic distributed systems.

The model which initiated the thesis topic, is the VeMo case study which is described in detail in Section 4.1

1.3. Thesis Goal, Approach and Scope

1.3.1 Goal

The goal of this thesis can be devided into 3 subgoals:

1. **To evaluated the feasibility and value of extending VDM-RT with dynamic reconfiguration in order to facilitate the modeling of dynamic distributed systems.**
2. **To expand the language in such a way that the changes to the existing VDM-RT language and methodology is kept to a minimum.**
3. **To ensure that the dynamic reconfiguration extension maintains an appropriate relationship between the modeling level and the implementation level, such that dynamic reconfiguration models can be used for validation of subsequent implementations.**

The thesis will seek to answer if it is possible to integrate dynamic reconfiguration into the VDM-RT language and tools, and if the possibilities attained by having dynamic reconfiguration are beneficial. This depends on the properties of the modeling language having a relationship to the properties of the real systems, while still keeping the desired level of abstraction.
It should be possible to make the model elaborate enough to establish an adequate representation of the modelled system without having to decide on final implementation details; otherwise the value of modelling is lost.

Oppositely the modeling language must be extensive enough to allow for the important aspects to be modeled. If constraints in the modeling language, such as a static topology, forces the model to become inaccurate for certain types of system, then the value of the model is deminished. It is important to note that abstraction is the deliberate disregard of certain irrelevant aspects of a system, and not imprecision in relevant details.
If the distance between the properties used in model and the properties of the real system becomes too great, the model cannot be used to validate the system. This is the challenge in creating the extension and at the same time the measure for which the dynamic reconfiguration will be evaluated.

1.3.2 Approach

The approach taken to carry out the thesis work can be divided into following parts:

- An investigation into the common principles of implementing dynamic reconfiguration;
- The exploration of the existing approaches in middelware and formally based modelling languages supporting dynamic distributed system and an evaluation of these approaches in relation to VDM-RT;
- The creation of semantically well-founded language extensions to VDM-RT enabling the description of dynamic reconfiguration in VDM-RT;
- Development of a working implementation of the dynamic reconfiguration extension into a branch of the Overture Project and the VDMJ interpreter; and
- Application of the extension to a couple of case studies in order to validate the functionality and to evaluate the value of having dynamic reconfiguration in VDM-RT.

The approach is reflected in the structure of this thesis.

1.3.3 Scope

Dynamic Reconfiguration is a large area and it includes many types of operations and mechanisms, therefore this thesis will focus on two types of dynamic reconfiguration:

- *Changing Topology* is alternation of the communication relationships between entities.
- *Migration* moves an entity from one processing unit to another while preserving the identity of the entity.

Not having the possibility of *changing topology* in the existing VDM-RT is the main motivate for the subject of this thesis, while the possibility of *migration* is included to investigate the possibilities of having this type of reconfiguration as a native part of a modelling language, as it is a very common operation in implementations of dynamic reconfiguration.

These two concepts of reconfiguration differs from each other in relation to VDM-RT. *Changing the topology* performs an alternation of network relationships which is impossible in current VDM-RT, while *Migration* performs an action that to a large extent can be implemented in existing VDM-RT with a little effort.

1.4. Case Studies

In order to examine the effect of having dynamic reconfiguration in VDM-RT the possibilities of the extension is applied to a number of case studies which have been modeled in VDM-RT.
Based on the goal and scope of the thesis, presented above, two case studies have been created in which the dynamic reconfiguration operations can be applied.
There is one primary case study focusing on changing topology, and one secondary targeted at migration.

VeMo: The **Ve**hicle **Mo**nitoring system is designed to improve road safety by increasing the traffic information available to motorists through cooperative vehicle to vehicle communication.

The system has a highly autonomous behavior which means it has rapid exchange of information between the different vehicles in the system, meaning it is an apparent case for testing dynamic changes in the network topology.

ConPlay: **Con**tinuous **Play** is a distributed music system that continuously will deliver music without disruption, even when the user changes physical location. This is achieved by moving the current playlist between distributed music players on the fly.

Given the functionality of the system, the ConPlay model might benefit from having a migration functionality.

1.5. Related Work

In order to establish a wider understanding of dynamic reconfiguration, different approaches from the litterature is examined and evaluated in relation to VDM-RT. The purpose is to establish a broader perspective of how dynamic reconfiguration is employed in existing approaches and how these can be related to a dynamic reconfiguration extension of VDM-RT. The related work includes approaches developed in other formal methods as well as in middelware technology.
A brief introduction of the related work is presented, while a more detailed evaluation is presented in chapter 5.

An object-oriented component model for heterogeneous nets
In [Johnsen&08] a high-level modeling of distributed systems with diverse quality networks is proposed with the purpose of analyzing properties of heterogeneous nets. The work extends the executable object-oriented modeling language Creol [Johnsen&07] to create a formal framework which makes it possible to model dynamic reconfiguration.
This approach is interesting because it defines a system class which is used to set up the overall system and network topology, which is very similar to VDM-RT, and this system class is used for the management of the reconfiguration.
The extension offers some other interesting properties in relation to dynamic reconfiguration as it describes timing, message processing, error handling as well as language abstractions for specifying dynamically evolving network and component relationships.

Coordinating object oriented components using data-flow networks
[Jaghoori08] proposes a framework for component-based modeling of distributed systems in which the network connections between components are dynamic reconfigurable.

This work also seeks to introduce dynamic reconfiguration into the Creol modelling, but the approach differs from [Johnsen&08]. It has its focus on enabling reconfiguration through components and dataflow networks, inspired from Reo [Arbab06].

Dynamic Reconfiguration for Middleware-Based Applications
[Wegdam&03] introduce an approach for giving middleware applications support for dynamic reconfiguration. This approach differs from the other approaches above, as it is intended to be embedded into concrete software that use real distributed system technology, such as CORBA (See Section 3.5), and not as an extension to a formal method. This entail an approach that does have the benefit in the level of abstraction found in formal models, which means it is closely tied to the constrains of a real implementation.

1.6. Reading Guide

The following conventions are used in the thesis:

External references and citations:
References to primary and secondary literature are place in square brackets containing the surname of the primary author and the year of publication, e.g. [Doe08]. If the literature has multiple authors this is indicated by an ampersand, e.g. [Doe&08].
References to tertiary literature are placed in footnotes.

Emphasis:
Phrases or words which are particular significant in the given context are emphasized by an *italic text*.

Terminology:
An elevated tau: $^{\tau}$ behind an abbreviation, ancronym, or phrase indicates that an explanation can be found in Appendix A Terminology.
The terminology is referenced the first time the term is used in each chapter.

VDM highlighting and model examples:
VDM Keywords are shown in **`boldface`**.

References to element or identifiers in a VDM model are shown as `italic typewriter`.

VDM model examples are placed in Listings:

```
class MyClass
operations
MyClass : () ==> MyClass
MyClass == skip;

end MyClass
```

Listing 1.1: Example of VDM class

1.7. Structure

This thesis is divided into eight chapters:

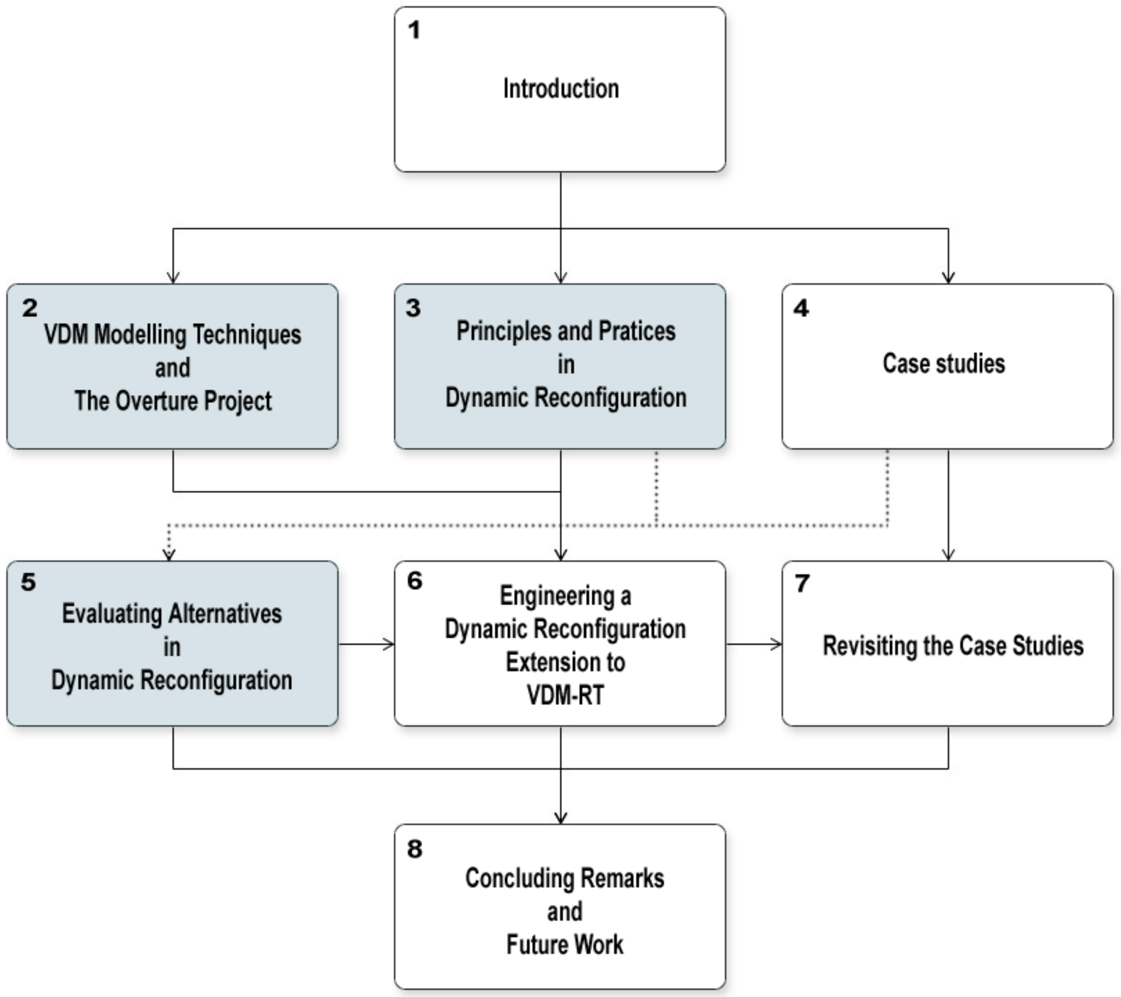

1
Introduction
2
VDM Modelling Techniques and The Overture Project
3
Principles and Pratices in Dynamic Reconfiguration
4
Case studies
5
Evaluating Alternatives in Dynamic Reconfiguration
6
Engineering a Dynamic Reconfiguration Extension to VDM-RT
7
Revisiting the Case Studies
8
Concluding Remarks and Future Work

Figure 1.1: Thesis Structure

Following the introductory chapter three chapters follow which provide the essential elements of the thesis: VDM-RT, dynamic reconfiguration and the case studies.

2. Chapter 2 supplies an overview of VDM-RT and introduces the Overture tool suite in which the dynamic reconfiguration extension is to be implemented.

3. Chapter 3 lays the foundation for dynamic reconfiguration and topics related to diffirent ways of implementing distributed systems.

4. Chapter 4 introduces the case studies which will be used to evaluated the effect of having dynamic reconfiguration in VDM-RT.

The following three chapters contain the development and evaluation of the dynamic reconfiguration extension.

5. Chapter 5 explores existing approaches in the literature of dynamic reconfiguration in distributed systems, in order to gain inspiration as to dynamic reconfiguration can be employed in VDM-RT. Chapter 5 make use of the case studies in Chapter 4 as well as the fundamentals from Chapter 3, as indicated by the dotted lines in Figure 1.1.

6. Chapter 6 is the main chapter in which the dynamic reconfiguration extension to VDM-RT is designed and implemented. The chapters marked with a color, in Figure 1.1 is the foundation on which the extension will be build.

7. Chapter 7 evaluates the effect of applying the dynamic reconfiguration extension to the case studies presented in Chapter 4.

The thesis is summarized in Chapter 8, where the achieved results are presented and assessed. Finally future directions in which this work could be extended is considered.

Chapter 2

VDM Modelling Techniques and The Overture Project

This section supplies an overview of the VDM modeling language and the tools which exists to support the language.
Because VDM-RT's ability to model distributed systems is the main focus of this thesis, the details of VDM-RT's deployment model is explained in the Section 2.2. Section 2.3 describes the existing tool support for VDM, with a focus on the Overture project that is used for the prototype implementation of the proposed dynamic reconfiguration extension. A brief overview of the Overture Project and the Overture IDE is provided, followed by an introduction to Overture's core: VDMJ. Finally essential implementation details of VDMJ, which are highly relevant in relation to dynamic reconfiguration, will be described in further detail in Section 2.3.4.

To deal with the increase in the complexity of software systems the use of formal techniques for analyzing and validating software specifications and designs has been widely encourage and extensively researched [Clarke&96, Woodcock&09]. Formal methods has been practiced for over four decades [Floyd67] with the objective of proving correctness of algorithms and system architectures with regard to a formal specification or model.

2.1. The Vienna Development Method in General

The Vienna Development Method (VDM) is a formal method technique for specifying, modeling, 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 1970 [Fitzgerald&08a]. 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 [Jones99, Bjørner00]. Since then VDM have been applied to a range of industrial projects [Fitzgerald&08b].

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 tools support.

The basis of VDM is the ISO standardized VDM-SL notation [Plat&92, ISOVDM96short] which

is a language for modeling functional specifications of sequential systems [Fitzgerald&09]. 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 [Fitzgerald&05] 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 [Verhoef05] revealed that neither the existing VDM++ dialect nor the extension made with VICE was sufficient when modeling distributed real-time systems. As a result an extension was proposed [Verhoef&06, Hooman&10] to enable the modeling 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 [Fitzgerald&07d, Verhoef&07, Sørensen&07, Verhoef08, Wolff08].

The exisiting 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. VDM-RT Static Deployment Model

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$^{\mathcal{T}}$) or Carrier Sense Multiple Access / Collision Detection (CSMACD$^{\mathcal{T}}$), 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$^{\mathcal{T}}$), 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` the 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^{T} 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 keywSystem is initialized.

The **system** class is created by System Developer by using the **system** keyword instead of **class**. This imposes some limitations on the **system** class as only instance variables and a default constructure 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 object 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 `CPU`s. If an object deployed on one `CPU` communicates with an object deployed to another `CPU`, then the `CPU`s 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.
f

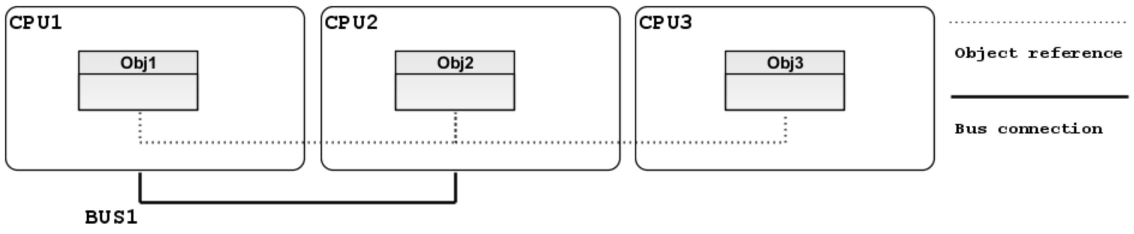


Figure 2.1: Deployment of example in Listing 2.3

This construct of double layers of references is important to observe. Given that when the distributed architecture is constructed in the **`system`** class, the System Developer is required to have knowledge of the `BUS`, `CPU` and object relationships. However when creating the remainder of the model, the System 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 System Developer can to some extent look past the `CPUs` and concentrate on object to object interaction.

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 occuring 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 a 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.

The topology of the virtual `CPU` and virtual `BUS` in a model is illustrated in Figure 2.2.

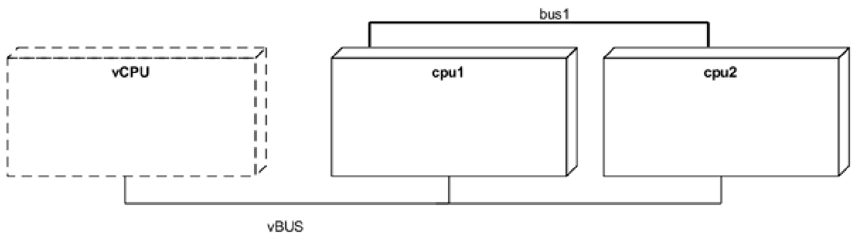


Figure 2.2: Role of the virtual `CPU` and the virtual `BUS` in a system.

Further details of VDM-RT are out of scope for this thesis, please refer to [Larsen&10a].

2.3. Tool support

Multiple tools exist to support VDM and the language dialects.

VDMTools VDMTools [VDMTools, Fitzgerald&08b] is a commercial tool suite for VDM containing everything from syntax/type checker to tnterpreter and debugger as well as code generation and test coverage statistics tools for the executed subsets. VDMTools was initially created at IFAD [Elmstrøm&94], Denmark and is currently maintained and further developed by CSK[1], Japan.

Overture The Overture project [Larsen&10b] is an Eclipse and Java based open-source platform combining a range of tools for constructing and analyzing VDM models.

The purpose of this thesis is not only to construct a language extension and define the semantics for a dynamic reconfiguration extension to 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 the focus of this thesis.

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

2.3.1 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.

Figure 2.3 gives an overview of the current state of the Overture Architecture.

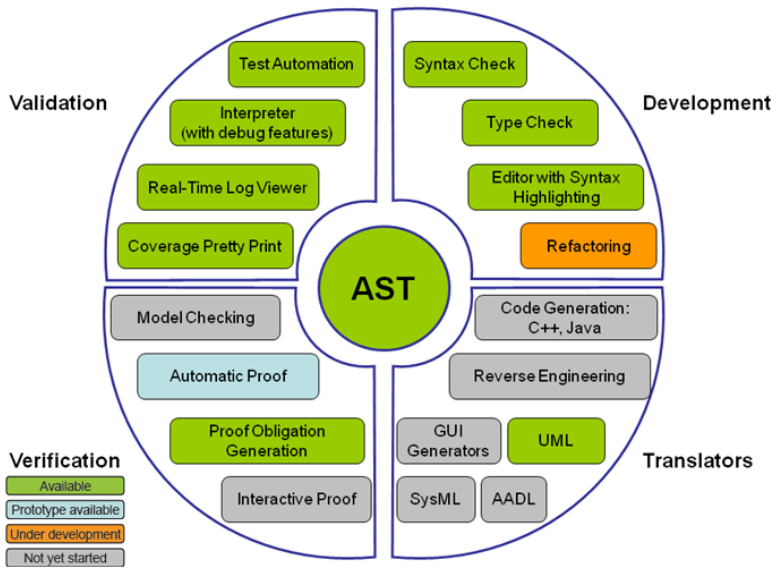


Figure 2.3: Overture Architecture Overview

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 [Larsen&10b].

Overture consists of two main parts:

- an IDE^T based on the extensible Eclipse framework and
- VDMJ [Battle09] 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.

2.3.2 Overture IDE

The Overture IDE uses the plug-in architecture of Eclipse[3] to enable the System Developer to interact directly with VDMJ through a Graphical User Interface. This involves both type and syntactically errors as well as running and debugging models through the runtime interpreter which is part of VDMJ for an executable subset.
Eclipse-based IDE's predefine different perspectives, which are task oriented visual containers in which the most task relevant information and resources are displayed when the specific perspec-

[2]Overture project webportal: `http://www.overturetool.org`
[3]Eclipse Documentation is available from `http://help.eclipse.org/`

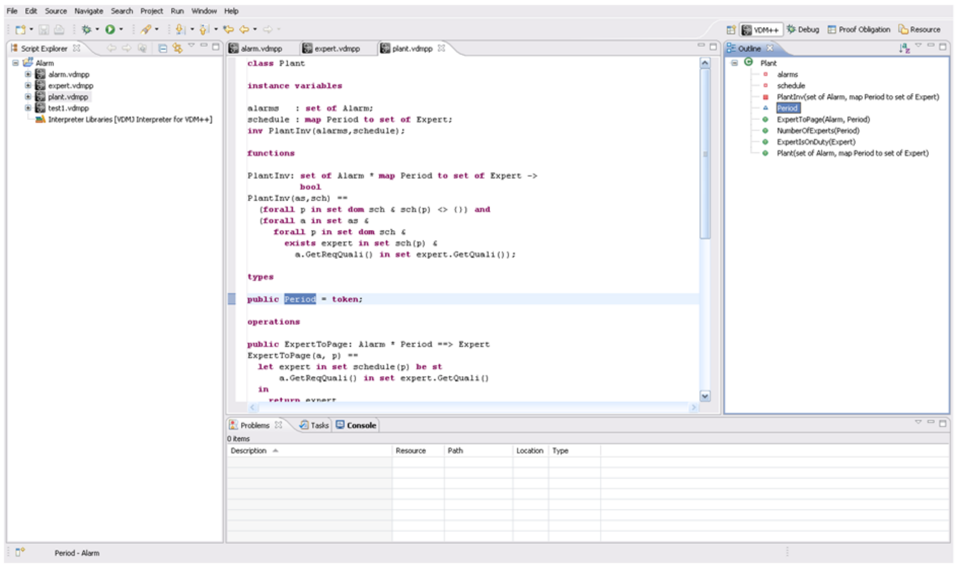
tive is selected.

In the Overture IDE there are two primary perspectives:

Overture perspective is the default perspective which contains the project explorer and the editor with syntax highlighting as the primary views.

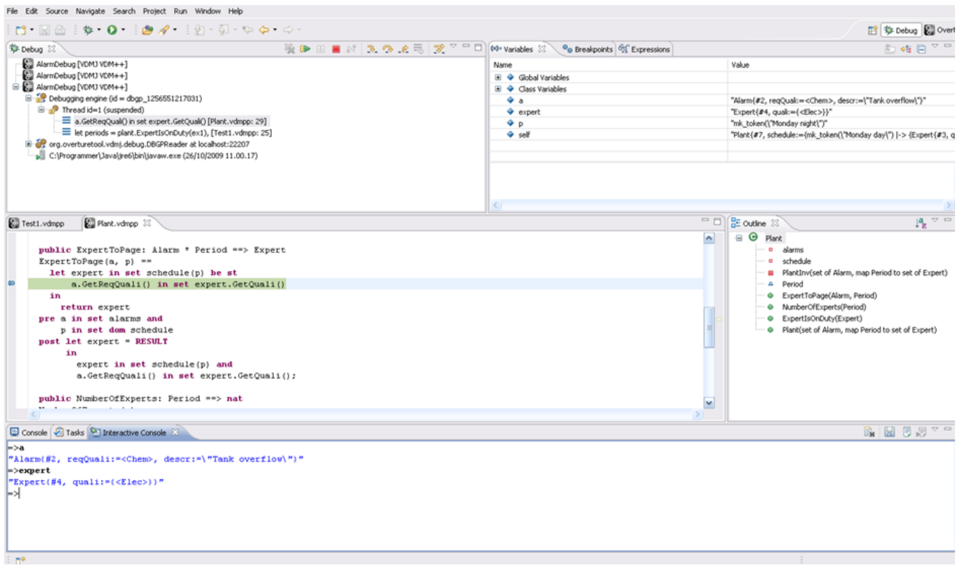
Debug perspective provides access to the debugging functionality which includes the possibility of setting break-points, single stepping, inspecting local and state variables and inspecting the call stack.

Screenshots of the two perspectives are shown in Figures 2.4 and 2.5.



```
class Plant

instance variables

alarms   : set of Alarm;
schedule : map Period to set of Expert;
inv PlantInv(alarms,schedule);

functions

PlantInv: set of Alarm * map Period to set of Expert ->
          bool
PlantInv(as,sch) ==
  (forall p in set dom sch & sch(p) <> {}) and
  (forall a in set as &
     forall p in set dom sch &
       exists expert in set sch(p) &
         a.GetReqQuali() in set expert.GetQuali());

types

public Period = token;

operations

public ExpertToPage: Alarm * Period ==> Expert
ExpertToPage(a, p) ==
  let expert in set schedule(p) be st
      a.GetReqQuali() in set expert.GetQuali()
  in
    return expert
```

Figure 2.4: Overture IDE with the Overture Perspective


File Edit Source Navigate Search Project Run Window Help
Debug
Variables
Breakpoints
Expressions
Test1.vdmpp
Plant.vdmpp
Outline
Plant
alarms
schedule
PlantInv(set of Alarm, map Period to set of Expert)
Period
ExpertToPage(Alarm, Period)
NumberOfExperts(Period)
ExpertIsOnDuty(Expert)
Plant(set of Alarm, map Period to set of Expert)
Console
Tasks
Interactive Console

```
AlarmDebug [VDMJ VDM++]
AlarmDebug [VDMJ VDM++]
AlarmDebug [VDMJ VDM++]
  Debugging engine (id = dbgp_1256551217031)
    Thread id=1 (suspended)
      a.GetReqQuali() in set expert.GetQuali() [Plant.vdmpp: 29]
      let periods = plant.ExpertIsOnDuty(ex1), [Test1.vdmpp: 25]
  org.overturetool.vdmj.debug.DBGPReader at localhost:22207
  C:\Programmer\Java\jre6\bin\javaw.exe (26/10/2009 11.00.17)
```

Name	Value
Global Variables	
Class Variables	
a	"Alarm{#2, reqQuali:=<Chem>, descr:=\"Tank overflow\"}"
expert	"Expert{#4, quali:={<Elec>}}"
p	"mk_token(\"Monday night\")"
self	"Plant{#7, schedule:={mk_token(\"Monday day\") |-> {Expert{#3, q

```
public ExpertToPage: Alarm * Period ==> Expert
ExpertToPage(a, p) ==
  let expert in set schedule(p) be st
      a.GetReqQuali() in set expert.GetQuali()
  in
    return expert
pre a in set alarms and
    p in set dom schedule
post let expert = RESULT
     in
       expert in set schedule(p) and
       a.GetReqQuali() in set expert.GetQuali();

public NumberOfExperts: Period ==> nat
```

```
=>a
"Alarm(#2, reqQuali:=<Chem>, descr:=\"Tank overflow\")"
=>expert
"Expert(#4, quali:=(<Elec>))"
=>
```

Figure 2.5: Overture IDE with the Debug Perspective

2.3.3 VDMJ in general

VDMJ[4] is the core tool suite for the Overture project and it provides tool support for the parser, syntax and type checker, interpreter, debugger and proof obligation generator. VDMJ supports all the VDM dialects: VDM-SL, VDM++ and VDM-RT.

VDMJ is developed as a command line tool only, but it implements the DBGp$^{\mathcal{T}}$[5] remote debugging protocol to allow e.g. a graphical user interfaces to be attached to VDMJ so that a model can be loaded as well as interpreted or debugged by remote control.
VDMJ is written purely in Java and it consists of roughly 500 classes containing approximately 62000 lines of code.

Figure 2.6 shows the architecture of VDMJ divided into logic elements, with each representing a key internal mechanism of VDMJ.

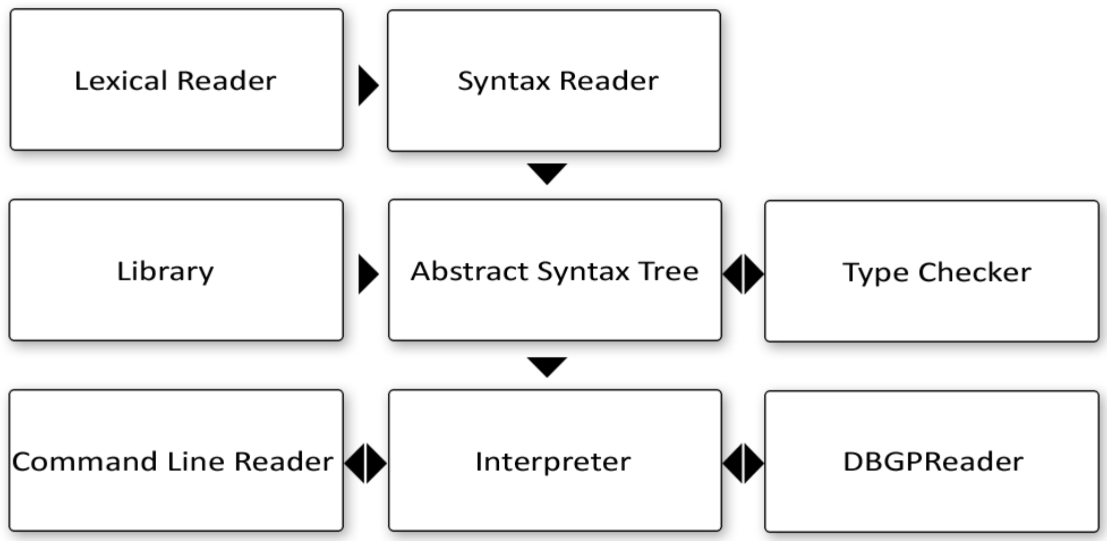


Figure 2.6: VDMJ logical elements and relations

Lexical Reader scans the VDM source files and converts the input sequences to tokens.

Syntax Reader receives the stream of lexical tokens, which are checked and used to construct the Abstract Syntax Tree.

Abstract Syntax Tree (AST) is a syntactic structure of the parsed model, which is constructed from VDMJ types that are equivalent to the types and structure found in the VDM dialect used.

Type checker is passed the AST, which is examined for static type errors as well as being enhanced with additional type information.

Library contains standard VDM libraries, such as IO and MATH, which can be added to a model.

[4]VDMJ on SourceForge: `https://sourceforge.net/projects/vdmj/`

[5]A common debugger protocol for languages and debugger UI communication `http://xdebug.org/docs-dbgp.php`

Interpreter creates a runtime environment, on the basis of the AST, in which the expressions of the model can be evaluated. The Interpreter also provides debugging functionality by enabling breakpoints to be placed in specific expressions.

Command Line Reader can give input to – and receive output from – the interpreter to a console.

DBGPReader allows the interpreter to be controlled by the DBGp remote debugging protocol.

A more detailed description of VDMJ can be found in the VDMJ Design Specification [Battle09].

2.3.4 Implementation specifics of VDMJ

Implementation details for parts of VDMJ which are particular interesting for this thesis, are explained in this section.

2.3.4.1 System Topology

The `BUSValue` class is the Java representation of an instance of the VDM-RT `BUS` class and accordingly one `BUSValue` object exists for every `BUS` in the model. Each `BUSValue` object has a composition relationship to an BUSResource object, which contains a list of the `CPU`s connected to the `BUS` as defined in the **`system`** class.
This list could be used for validating the BUS connection between two `CPU`s during run-time execution, but actually the list is only used during the initialisation of the model. Because a remote operation call between objects does not specify an exact BUS, the use of these lists would require the interpreter to run through all busses in the model to check for connectivity. This is necessary since a `CPU` has no knowledge of which busses it is connected to.

During execution of a model a centralized mapping is created which gathers all the lists from the busses in the model. For performance reasons VDMJ actually use a two way mapping for determining the connection between `CPU`s. The mapping is constructed following the initialization of the **`system`** class and is placed in the `BUSValue` class as a static *cpumap* field. The *cpumap* maps `CPU`s pairwise to a `BUS` to indicate a connection between them.

Figure 2.7 and Table 2.1 illustrates an example of a simple system topology. *Bus1* forms a single connection between *cpu1* and *cpu2*, which is reflected in Table 2.1 by the cpu pairs *cpu1 <-> cpu2*, and oppositely *cpu2 <-> cpu1*, mapping to *bus1*.

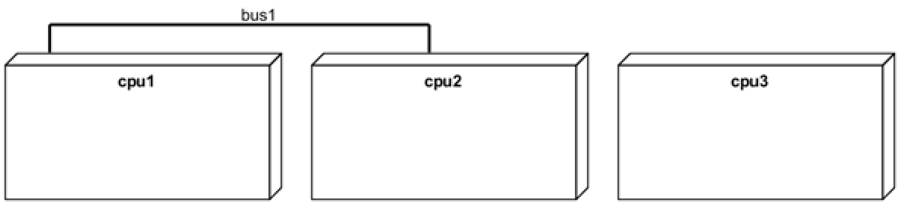


Figure 2.7: Simple system topology

CPU pair	BUS
cpu1 <-> cpu2	bus1
cpu1 <-> cpu3	null
cpu2 <-> cpu1	bus1
cpu2 <-> cpu3	null
cpu3 <-> cpu1	null
cpu3 <-> cpu2	null

Table 2.1: Example of CPU mapping based on the toplogy in Figure 2.7

By using the map the `BUS` connection between two `CPU`s, if any, can be determined by a single lookup.

2.3.4.2 Remote Invocation

The purpose of relating `CPU`s through `BUS` connections is to enable remote operation invocation between objects deployed on separate `CPU`s. This process includes messages being transfer on the `BUS`, either a request or a response; it involves asynchronous threads being started on remote `CPU`s and it involves synchronization and blocking operations between `CPU`s and `BUS` connections.

There are two message types in VDM-RT, `MessageRequest` and `MessageResponse`, which both are subclasses of the abstract MessagePacket, as shown in Figure 2.8. The relevant classes have the following usage:

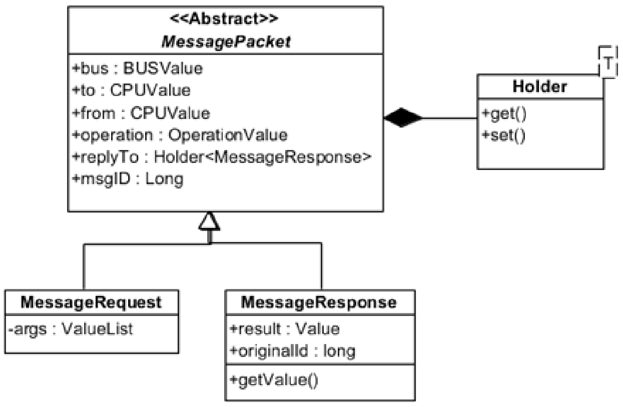


Figure 2.8: Class diagram of Message hierachy used in VDMJ

MessagePacket besides a unique message id, a reference to the `BUS` it is carried on and a reference to the remote operation, the `MessagePacket` holds a reference to the `CPU` that the message is to be transferred to and a reference to the initiating `CPU` that put the message on the `BUS`. If the remote operation call is synchronous, meaning the calling `CPU` will wait for a response, the `replyto` attribute will hold a reference to a generic `Holder` object. The `Holder` object is a simply data container, which resembles a blocking mailbox with the size of 1, as it contains a `get` operation that blocks until notified by its `set` operation. Thus the calling thread will wait for the result to be placed in the holder.

MessageRequest expands the `MessagePacket` by carrying the arguments for the operation

MessageResponse has a value field to hold the result of the request which it is the response to. Because a new message id is generated at the construction of a new message, the `MessageResponse` also contains the `originalid` with is the message id of the request which initated the remote operation.

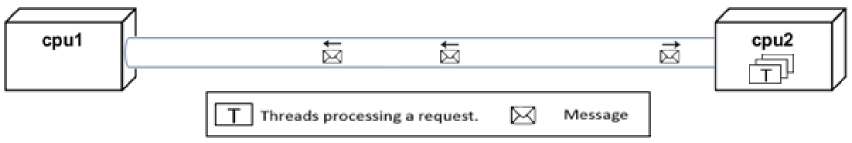


Figure 2.9: BUS connection between two CPUs

Figure 2.9 shows a `BUS` connection between two `CPUs` with messages indicated as envelopes, currently passing on the `BUS` in the direction of the arrow. Here the `CPU`, to which the arrow is pointing, is held in the `to` attribute and oppositely the other `CPU` is held in the `from` attribute. The `from` attribute is also denoted the *request initiator* and the `to` attribute the *request receiver*. Figure 2.9 reflects a static `BUS` topology or a topology which has not been reconfigured. An envelope indicates a message in transit and "T" indicates threads processing a remote request.

The process of making a remote synchronous invocation is shown in Figure 2.10.

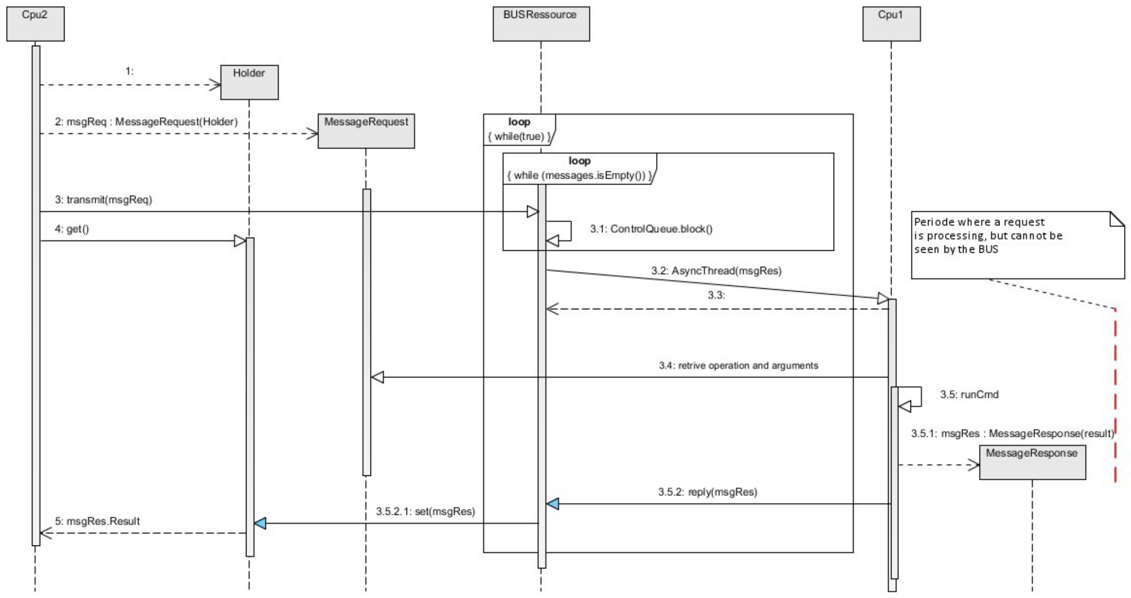

Cpu2
BUSRessource
Cpu1
1:
Holder
2: msgReq : MessageRequest(Holder)
MessageRequest
loop
{ while(true) }
loop
{ while (messages.isEmpty()) }
3: transmit(msgReq)
4: get()
3.1: ControlQueue.block()
Periode where a request is processing, but cannot be seen by the BUS
3.2: AsyncThread(msgRes)
3.3:
3.4: retrive operation and arguments
3.5: runCmd
3.5.1: msgRes : MessageResponse(result)
MessageResponse
3.5.2: reply(msgRes)
3.5.2.1: set(msgRes)
5: msgRes.Result

Figure 2.10: Sequence diagram of a remote synchronous invocation in VDMJ

A synchronous request results in the calling cpu/thread putting a `MessageRequest` on the `BUS` and then making a blocking call to the `Holder` object to wait for a reply. The `Bus` has its own thread which continuously checks its internal message queue for new messages to process. When a `MessageRequest` is transmitted to the `BUS`, it will be pulled out of the queue and a new thread will be started on the target `CPU` while the `BUS` returns immediately to process other messages in its queue. Once the thread has finished processing the operation invocation in the request it will return a `MessageResponse` with the processed result to the `BUS` queue. Eventually the `MessageResponse` will be pulled out of the `BUS` queue and the result will be put in the `Holder`, which releases the blocked thread that made the initial synchronous request.

Chapter 3

Principles and Pratices in Dynamic Reconfiguration

In order to lay the theoretical foundation of dynamic reconfiguration an overview of the different types of dynamic reconfiguration is provided in Section 3.2. Mechanisms used to ensure system persistency is presented in Section 3.3, followed by a section on different type of network and network protocols. Finally Sections 3.5 and 3.6 introduces two commonly used middleware technologies in order to supply a reference to real world technologies.

3.1. Introduction

The goal of dynamic reconfiguration [Kramer&85] 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.

These distributed systems are often safety critical and have high requirements for extensibility, portability, load balancing and availability [Bates98] which are enforced by recovery and failover safety measures that require alternation of the system architecture as well as data migration between running systems.

The run-time alternations which a dynamic reconfiguration entail 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.

3.2. Types of dynamic reconfiguration

Dynamic reconfiguration mechanisms and approaches can in general be categorized under four common types of capabilites [Goudarzi&96, Milojicic&00]:

- Changing Topology
- Replacement
- Addition/Removal
- Migration

3.2.1 Changing Topology

This reconfiguration capability changes the relationship between entities by either creating new connections or by removing connections to a specific entity, as illustrated in Figure 3.1. In a VDM-RT context this could be changing the `BUS` connection between `CPU`'s.

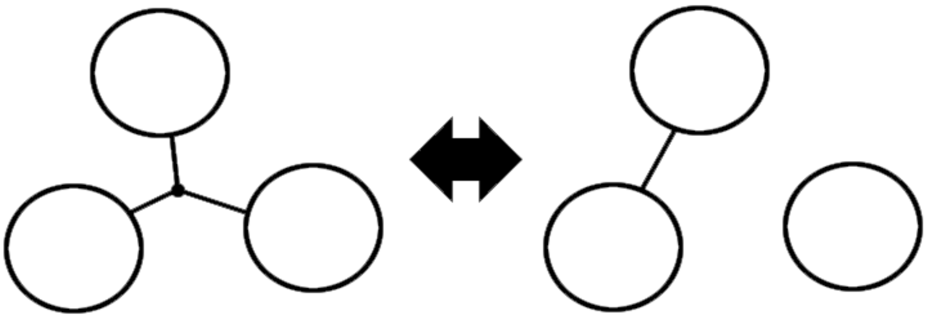


Figure 3.1: Changing Topology.

3.2.2 Replacement

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.
This dynamic reconfiguration operation is out of the scope for this thesis, and is not considered further (see Section 1.3).

3.2.3 Addition/Removal

Addition/Removal an is the dynamic introduction or elimination of an entity into the system configuration. This dynamic reconfiguration operation is out of the scope for this thesis, and is not considered further (see Section 1.3).

3.2.4 Migration

Migration moves an entity from one processing unit to another while preserving the identity of the entity and thereby ensuring referential integrity. In a VDM-RT context this could be moving an object from one `CPU` to another, as illustrated in Figure 3.2.

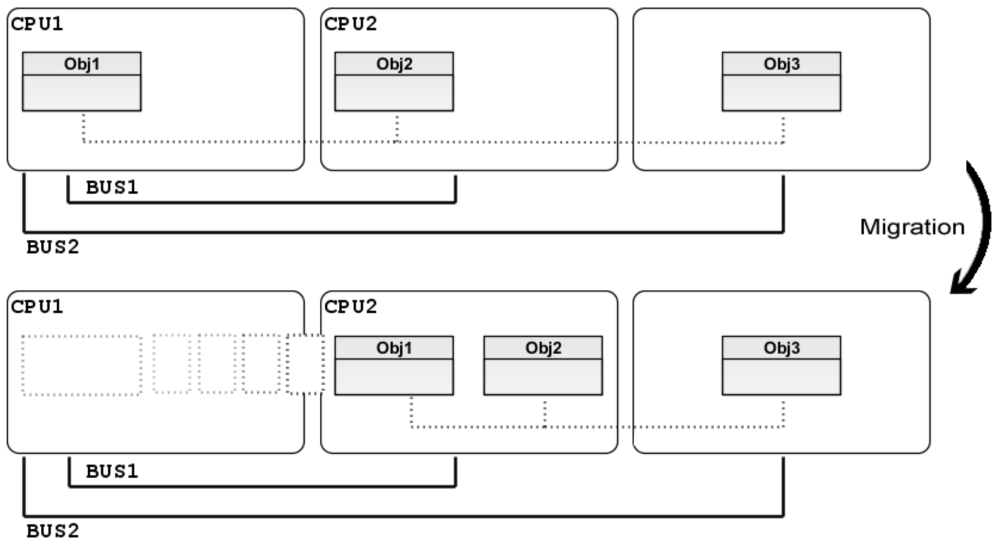


Figure 3.2: Object migration.

3.3. Characteristics of system correctness

To keep confidence in the model and to ensure the correctness of the system three characteristics are identified [Goudarzi99, Wegdam03b] which are to be met and assured by the dynamic reconfiguration extension to guarantee a reliable system state.

- **Structural integrity** is to ensure the relationships between entities in the system,
- **Mutually consistent state** is the assurance of a mutual perception of state changes between entities, and
- **Application state invariants** guarantees invariants in the model following reconfiguration.

3.3.1 Structural integrity

Structural integrity concerns the assurance of the relationships between entities in the system being reliable and correct, after a dynamic reconfiguration. In distributed systems these entities may be physical processing units or objects in middleware based object-oriented systems.

In a VDM-RT context a reconfiguration change cannot directly cause changes between object references, because of the double layer of references (see Section 2.2). Instead the reconfiguration affects the relationship between `CPU`'s and `BUS` connections which might have a significance on the communication.
The structural integrity characteristic relates to ensuring that only the `CPU`s which are explicitly involved in the reconfiguration are affected and that the structure of the network between the involved `CPU`s is correct, in relation to the reconfiguration changes applied.
Thus the structural integrity does not concern ensuring a relationship between all entities in the system, but to ensure that the structure of the affected entities actually reflect the architecture the System Developer$^{\mathcal{T}}$ configures. The structural integrity is met when the relationship between the involved entities are consistent and unanimous between all entities in the system.

3.3.2 Mutually consistency

Dynamic reconfiguration entails that changes occur during run-time execution, which means that state changes and interactions will potentially be occurring at the time of reconfiguration. A reconfiguration may result in disruptions of the ongoing remote invocations occurring between entities in the system, leading to the risk of entities being left in inconsistent states.

In a VDM-RT context the communication between two objects *A* and *B* deployed on separate `CPU`'s can be used as an example. *A* reconfiguration change might cause a remote invocation to never arrive at the receiver *B*, or the response of an remote invocation to never reach the initiator *A*, as illustrated in Figure 3.3. This leads to inconsistent states if *A* assumes that an invocation has reached *B*, or *B* expects that *A* has received the response of an invocation.

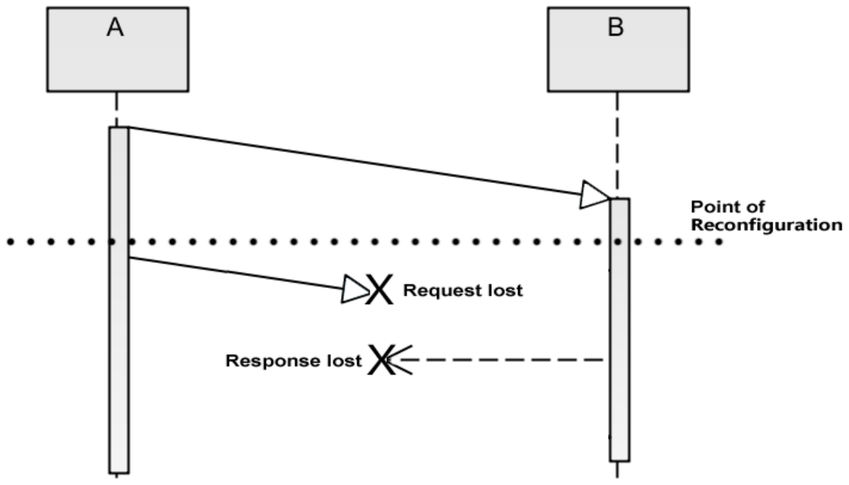


Figure 3.3: Sequence diagram of invocations disrupted by reconfiguration.

Mutually consistency is an agreement on the common perception of the state changes occurring between entities, based on of their remote interactions. Entities are in mutually consistent states if they have a shared understanding of the remote invocations that has occurred between them following a reconfiguration.
This entails that either both entities must perceive invocations as being successful or as failed; or one of the entities must maintain the overview over the invocations and be responsible for managing and establishing consistency following a reconfiguration.

To achieve mutual consistency in a system several approaches can be utilized, which are presented in Figure 3.4. The goal of the approaches is to reach a safe-state in which the behavior of the system is well defined. The definition of the safe-state is not the same for all approaches, e.g. some is based on abortion others on completion. However the goal is to place the system in a controlled state in which the behavior can be anticipated, which essential entail a method for managing invocations.

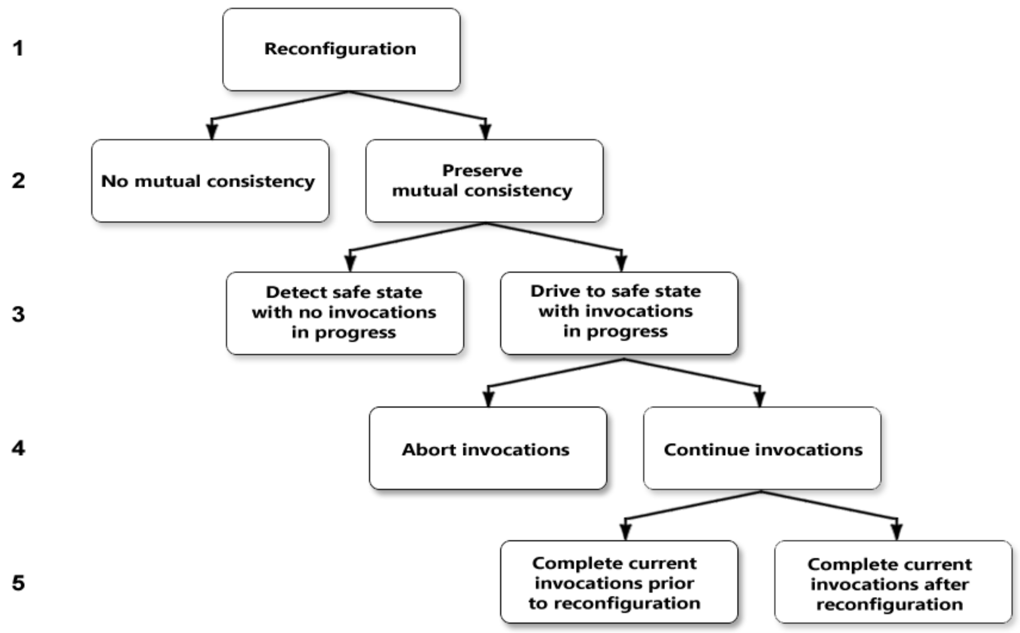

1
2
3
4
5
Reconfiguration
No mutual consistency
Preserve mutual consistency
Detect safe state with no invocations in progress
Drive to safe state with invocations in progress
Abort invocations
Continue invocations
Complete current invocations prior to reconfiguration
Complete current invocations after reconfiguration

Figure 3.4: Approaches for achieving mutually consistency. (Illustration based on [Almeida&01])

For each horizontal level in Figure 3.4 two different alternatives are possible, either a defined approach (left branch) or a refinement of the upper approach (right branch).

2. The *second* level is the choice between preserving mutual consistency and doing nothing at all. Evidently the latter is not really applicable in preserving mutual consistency, but is included to show the option of doing nothing.

3. The *third* level contains the option between *driving* the system safe-state and *detecting* safe-state of the system. *Detection* of the safe state involves an approach where a monitoring of the systems remote invocation is initiated on a reconfiguration request and the reconfiguration change is put on hold, as long as there are remote invocations processing. This approach ensures that no remote invocations will be affected by a reconfiguration, however the period for a reconfiguration to completed is unpredictable and in a system with continuous communication the reconfiguration may never occur.

   Driving the system into the safe sate involves the application a reconfiguration mechanism to ensure that the state is reached.

4. The *driving* reconfiguration mechanisms are divided into two at the *fourth* level, either invocations are *aborted* or they are allowed to *continue* and another approach is used to reach the safe state. If the invocations are *aborted* some kind of error reporting is needed to allow for error handling and recovery, such that the state is preserved and the system is able to continue following reconfiguration. The advantage is that the state can be reached instantly; the downside is that it may increase the workload of the System Developer when using the dynamic reconfiguration mechanisms. This approach is used by [Johnsen&08] as described in Section 5.1.

5. The *fifth* and final level defines two approaches for driving the system to a safe state while allowing them to complete before or following the reconfiguration.

5. In the first approach all invocations are to complete before the reconfiguration occurs. This resembles the approach of detecting the safe-state, the difference is that only the current invocations which are affected by the reconfiguration are allow to complete, while all other invocations are blocked and queued until after the reconfiguration. This approach has the advantage that invocations are not removed from the system, however the approach do require a lot of management and detailed insight into the system in order to determine which invocations are in the group being affected by the reconfiguraiton and therefore should be allowed to complete. Likewise the time needed for the current operations to complete is unknown and thereby the time cost of the reconfiguration is unpredictable. This approach is used by [Wegdam&03] as described in Section 5.3.

   The second approach involves the suspension of all invocations, both current invocation and invocation occurring during the reconfiguration. Following the reconfiguration all invocations are resumed. This approach requires an in-depth control of the entities in the system as both stack, program counter and thread context information is needed to perform the suspend and resume action. Therefore this approach is quite complex to apply to a system.

3.3.3 Application state invariants

Application state invariants, or model state invariants in the context of VDM-RT, implicate the system's ability to ensure invariant definitions following reconfiguration. An invariant could be an incremental id number that must be ensured throughout the life time of the model.
This characteristic is not significant for the dynamic reconfiguration extension considered for VDM-RT in this thesis, as the extension cannot affect the invariants of the modeled system. A reconfiguration mechanism such as object replacement could affect the model state invariants, but this mechanism is not considered in this thesis.

3.4. Network types and protocols

Characteristics of different network types and protocols are important when modeling certain types of distributed system, as these systems depend on the behavior of the network.

3.4.1 Network types

Distributed systems have traditionally used wired connections between the entities composing the system, because of its properties for timing and reliability. However with the evident increasing use of wireless technologies in consumer electronics, the need for distributed wireless systems is becoming just as applicable. Wireless networks are becoming gradually more relevant in industrial environments as well [Willig&05], where the reliability and timing requirements are measured up against cost, installation and maintenance demands. Therefore it is relevant to consider both network types in relation to modeling distributed systems.

From an application point of view there should be no difference between using the two network types as this should be handled by the underlying network layers and protocols. However from a modeling perspective some of the underlying limitations might be interesting to investigate. Wired connections and wireless connections are two very different kinds of transportation mediums. The disturbance of the radio communication results in a greater loss than occurs in cabled networks, therefore wireless networks are more sensible to noise and environmental changes than wired connections. Because the range and noise is a such an significant factor, wireless networks behave differently from wired connections because data loss as well as connections or disconnections occur more frequently. Another factor in wireless networks is that the physical locations of the connected entities are unknown and the possible range of the wireless network is more unpredictable.

3.4.2 Protocols

In order to handle the unpredictability in the communication and the risk of alternations in the network relations, the different network types utilize communication protocols to establish precise behaviors. The different communication protocols can essentially be categorized as being either *connectionless* or *connection-oriented*. The difference between the two is basically if a logical connection is established or not.
The *connection-oriented* protocol will negotiate a logical connection between the two entities to establish a steady network on which a reliability technique is used to ensure data transfer. Reliability is normally ensured by the protocol adding control or acknowledgement message to the communication stream. Arrival of data can never be guaranteed, but certainty of the success or failure of a transmission can.
The *connectionless* protocol on the other hand does not have any notion of a logical connection nor does it utilize any form of reliability technique. Data is simply transmitted without checking to see if the messages arrived or if there were any receivers at all.

Both protocol types can be employed in the different network types, however due to the unpredictability of wireless networks, they often implement a type of connection-oriented behavior at a low level to deal with communication issues and radio disturbance.

In relation to VDM-RT the only possibility of making a connection between entities is the BUS class, which only allows for static connections and must be considered to represent a wired type of network. The `BUS` does not allow for any type of protocol to be utilized, as the notion of protocols have been abstracted away. However combing the static topology of VDM-RT with the fact that there cannot be any data loss on a `BUS`, it could be considered to have the characteristics of connection-oriented protocol.

As a result the notion of the BUS might be too simplistic to handle all types of distributed systems, as it does not cover all the relevant properties of the different networks types.

3.5. CORBA - Common Object Request Broker Architecture

CORBA is a long-lasting middleware technology that has a widespread use, which makes it interesting in relation to distributed systems.

3.5.1 CORBA in general

The Common Object Request Broker Architecture (CORBA) [OMG&96] is a vendor and platform independent middleware architecture which is standardized by the Object Management Group[1]. One of the fundamental features of CORBA is its ability to function across a variety of operating systems, programming languages and network types. Specialized specifications enable CORBA to run on real-time systems[2] and small embedded systems[3] as well.

In CORBA remote objects are defined in the interface definition language (IDL) which is a specification language for defining interfaces in a language-neutral manner. CORBA specifies mappings from an IDL specification into a native implementation in various languages, including C++, Java and Python.

The heart of CORBA is the Object Request Broker (ORB) which provides the interoperability between distributed objects including remote invocation, instantiation policies, marshalling, validation of objects as well as resources management.
CORBA defines the abstract General Inter ORB-Protocol (GIOP) as the standard communication protocol. A widely used concrete implementation of the GIOP is the Internet Inter-ORB Protocol (IIOP) which is build on top of TCP/IP.
One of the message types of the GIOP is the Inter Object Reference (IOR) which is an unique object reference that identifies CORBA objects. The IOR of an object is initially attained from a either a stringified URI^T string, passed as a method argument during an invocation or through a nameservice lookup such as the Common Object Services (COS) Naming Service [OMG&04].

3.5.2 Dynamic Reconfiguration

Changing topology:
As middleware is placed above the physical level, the ability to handle a physical change of the network topology is outside the scope of CORBA. The relationships in CORBA are purely based on object references (IORs) and not on the relationship between the processing units and the underlying network.

CORBA only has a connect/disconnect operation in the sense that objects are in a network as long as they have a valid object reference to the remote object.

Migration:
CORBA has no migration service, but multiple approaches [Pellegrini99, Wegdam03b] exist to enable object migration in CORBA, by building a reconfiguration framework on top of existing CORBA mechanisms and services.

[1] Object Management Group `http://www.omg.org/`
[2] Real-time CORBA `http://www.omg.org/spec/RT/`
[3] CORBA/e `http://www.omg.org/spec/CORBAe/`

3.5.3 Error handling

The error reporting in CORBA is based on distributed CORBA exceptions which the ORB will raise as native exceptions to enable exception handling in the native implementation.

CORBA defines two exceptions types:

System Exception are a standard set of errors which are raised by the ORB or server, such as initialization or communication problems.

User Exception are user defined in the IDL and are raised from within CORBA objects.

CORBA System Exceptions contain a completion status which may indicate if the remote invocation completed before the error occurred. However the completion status may return an undeterministic *maybe* status.

3.6. Java Remote Method Invocation

Java RMI is well established middleware technology and is employed in vast types of distributed systems.

3.6.1 Java RMI in general

Java Remote Method Invocation(RMI) [JavaRMI] is an architecture for enabling distributed objects within the homogeneous Java virtual machine (JVM^T) environment. Java RMI enables remote invocation between objects in distributed virtual machines, with the RMI core, RMI Runtime, supplying mechanisms such as invocation, marshalling, distributed garbage collection.
Java RMI has no interface definition language, instead native Java interfaces are used to define the remote objects. The remote objects are native Java objects which implement or extend RMI specified interfaces or classes.
Objects can be registered with the Remote Object Registry, a type of name server from which the remote objects can be obtained.

To carry the communication between objects, Java RMI primarily uses the Java Remote Method Protocol (JRMP) which runs over TCP/IP. It is also possible to use the Remote Method Invocation Internet Inter-Orb Protocol (RMI-IIOP) which delivers CORBA capabilities in collaboration with RMI.

3.6.2 Dynamic Reconfiguration

Changing topology:
Like CORBA, Java RMI is at a higher level than the physical changes of the network topology. The relationships are based on object references, which essentially are tied to TCP/IP.

Migration:
Toolkits [Avvenuti&01] exists to enable an Java object to be moved across different JVMs by adding a mobility layer on top of Java RMI. The toolkit explicitly writes the remote object to a stream and sends the serialized object over a normal RMI invocation. The migrated object is replaced by a proxy object which forwards messages to the migrating objects new location.

3.6.3 Error handling

Java RMI use native Java exceptions for error reporting. The possible exceptions cover a variety of error types ranging from connection and marshalling errors to server and security errors.

The exceptions can be distinguished by their type, to determine if they originate from the clients RMI call or from errors in the server or during the RMI return. It is however not possible to determine if the server completed the method invocation prior to the error.

Chapter 4

Case studies

This section describes the case studies which will be used to examine the effect of having dynamic reconfiguration in VDM-RT. The systems which the case studies revolve around are introduced, followed by a description of how the systems are modelled in VDM-RT.

In order to examine the effect of having dynamic reconfiguration in VDM-RT the possibilities of the extension will be applied to a number of case studies which have been modelled in VDM-RT. Each case study has been chosen to explore different features of the extension. For instance one case study revolves around a type of system that relates to dynamic changes of network topology. There is one primary case study focusing on changing topology, and one secondary case study targeted at migration.

The case studies have been designed and 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 studies and how the model is 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 is described in Chapter 7. The benefits, consequences and impact of the changes are also evaluated in Chapter 7.

4.1. VeMo Case Study - Changing Topology

4.1.1 Introduction

A major research area is 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-infrastucture systems `http://www.cvisproject.org`, the Co-operative system for intelligent Road Safety `http://www.coopers-ip.eu/` and the simulation platform iTETRIS `http://www.ict-itetris.eu`.
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 a open simulation platform.

4.1.2 Overview

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 as well as enabling 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 [Varaiya93], 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.

4.1.3 The System

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 4.1.

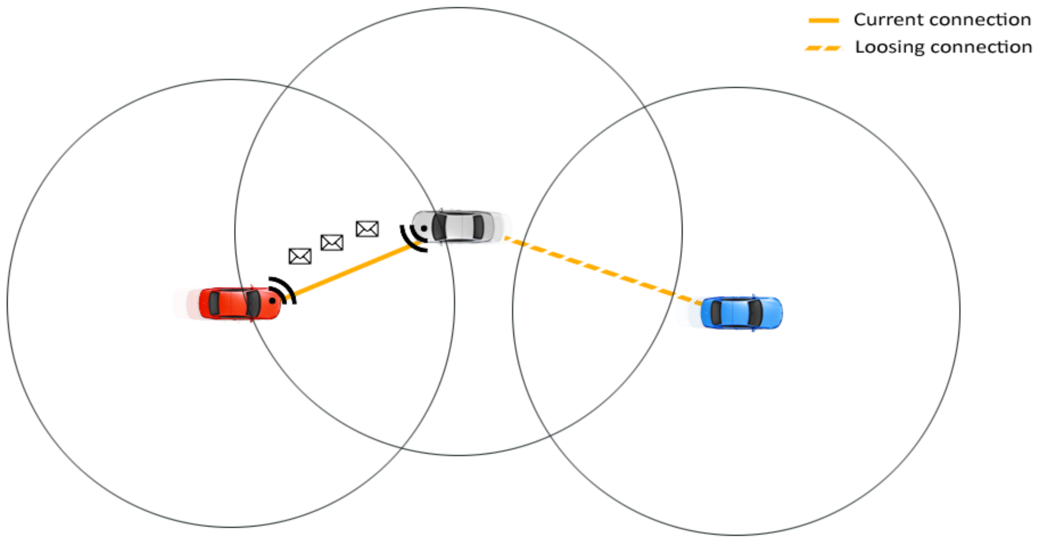


Figure 4.1: Vehicle network and communication

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 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 with which vehicles 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 becomes 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 4.2 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 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.

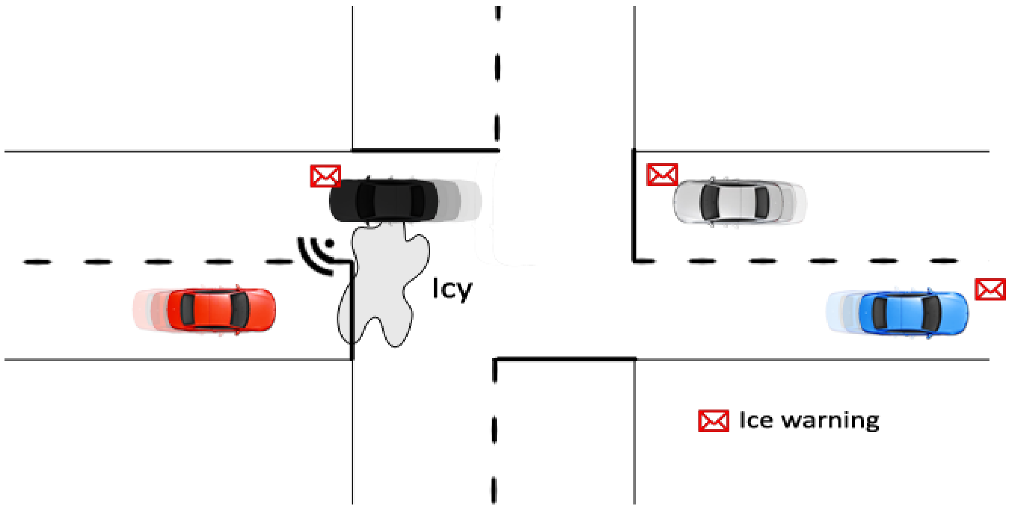


Figure 4.2: Scenario of information sharing

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

4.1.4 The Model

A VDM-RT model of the VeMo system has been created which models the movement of vehicles, potential hazards and the exchange of information. 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 others path without colliding.

- To keep the over all 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 4.3.

- 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.

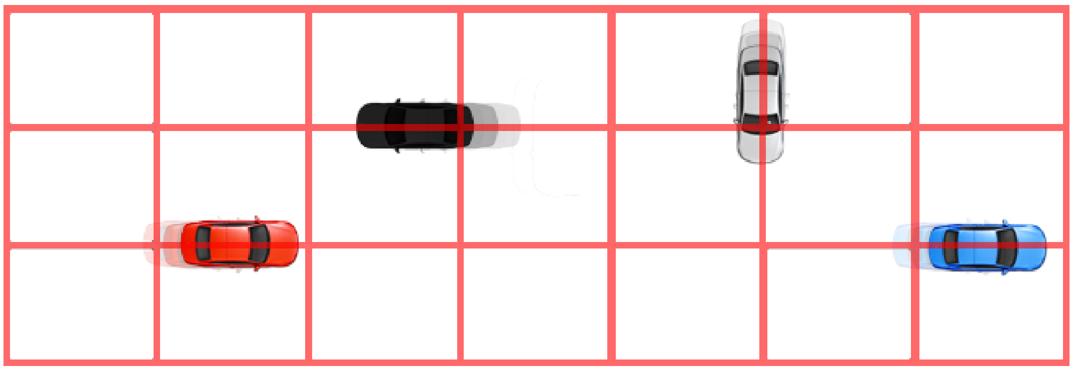


Figure 4.3: Grid of vehicle movement

The class diagram for the VeMo model is shown in Figure 4.4 and consists of the following classes:

VeMo System class that defines the CPU and BUS relations and the deployment of object 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. This is not incorporated in the model.

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^T 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.

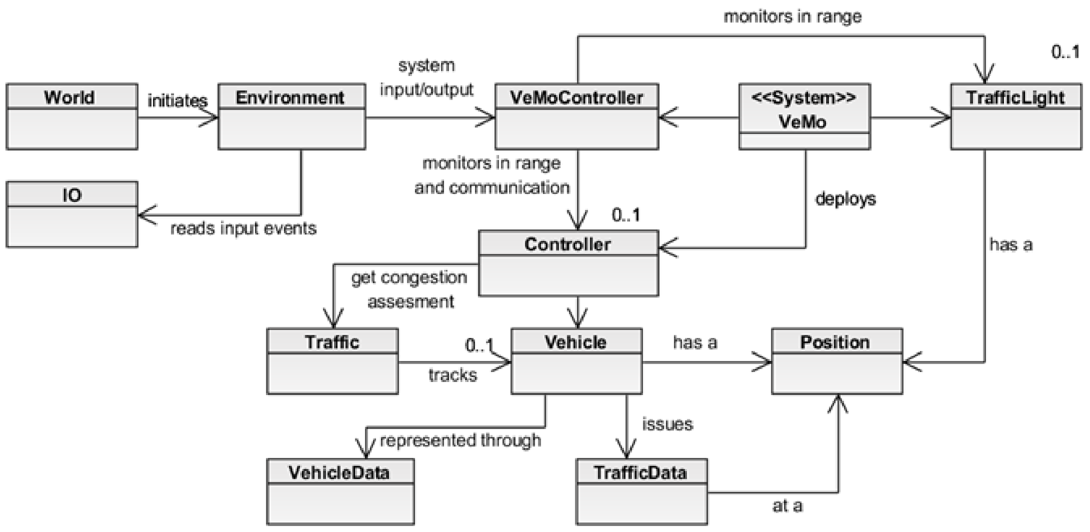


Figure 4.4: Class diagram of the VeMo model

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 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. Output, 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[7] 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.

4.1.5 Deployment

The deployment of a scenario in the model is shown on Figure 4.5, where the Environment and the VeMoController instances are deployed to the virtual `CPU` and two vehicles are deployed on separate `CPUs`.

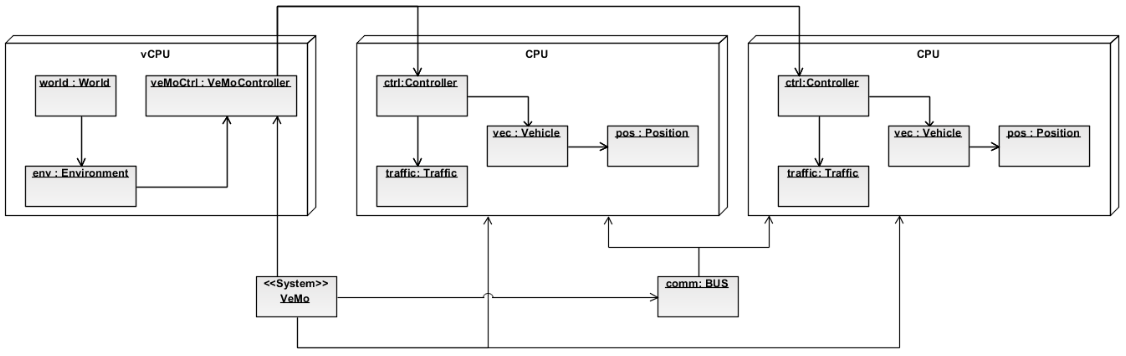


Figure 4.5: Deployment diagram of an VeMo system in the intial deployment setup.

As the case study has been modelled in existing VDM-RT there can only be one deployment setup, determined at the creation of the VeMo system class. In the scenario in Figure 4.5 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 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 `CPUs`, 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. ConPlay- System Migration

4.2.1 Introduction

The ConPlay system is a distributed music player, in which the music always follows the user of the system, as long as a ConPlay music player is available

4.2.2 The System

The Conplay music players are physical units which have been distributed before the system is used. The user of the systems defines a playlist of tracks, which the system should play. When the user moves to another location the playlist, along with the currently playing track, will be moved with the user. A scenario could be that the user is listening to the playlist in a car which is equipped with a ConPlay music player. Upon returning home, the ConPlay system will automatically move the playlist and the progress of the current track to the music player in the house thereby ensuring a continuous flow of music.

4.2.3 The Model

A VDM-RT model of the ConPlay system has been created which models the ConPlay music players and the movement of music between the players.
The class diagram for the Conplay system is shown on Figure 4.6.

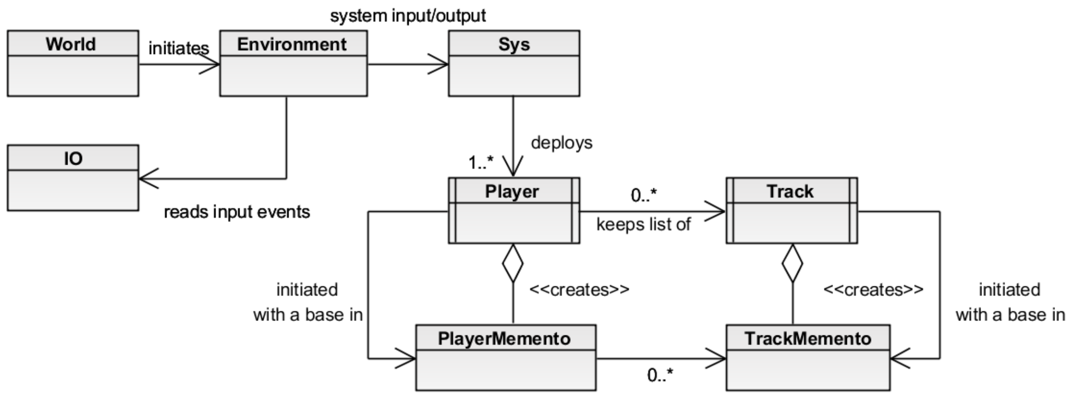


Figure 4.6: Class diagram of the ConPlay system.

The Conplay system consists of the following classes:

World is the main class which initializes the entire model.

Environment represents the input and output of the system.

Player is the representation of the music players which contains a playlist of tracks and keeps track of the currently playing track.

Track is the representation of a piece of music and contains the length of the number as well played time.

PlayerMemento makes is possible to store the current state of a Player.

TrackMemento stores the current state of a track.

A Track object contains a thread that plays the track by moving the playtime forward until the length of the track has been reached. The Player object contains a thread that runs through its contained playlist, by taking the head of a playlist. The thread in the Track is started by the **start** statement, after which the Player thread calls a Finished() operation on the Track, which is bound to permission predicate. This creates synchronization between the *Player* and the Track objects.

The environment is capable of supplying two types of inputs. *Start* which initiated the music, and *Migrate* which moves the music.
The migration is the movement of the Player and Track state from one Player to another Player. A limitation in the system is that only one Player can play at a time, and neither the Player nor the Track can be stopped once started.
The migration is performed via the Migrate operation which exist in both the Player and the Track objects. Once called the operations are able to suspend the threads and store the state of the objects through the *Memento* design pattern [Gamma&95]. The *Memento* objects can be used to reinitialize the objects on another Player and resume both the playlist and the current track.

4.2.4 Deployment

The system used in the case study is deployed as shown in Figure 4.7.

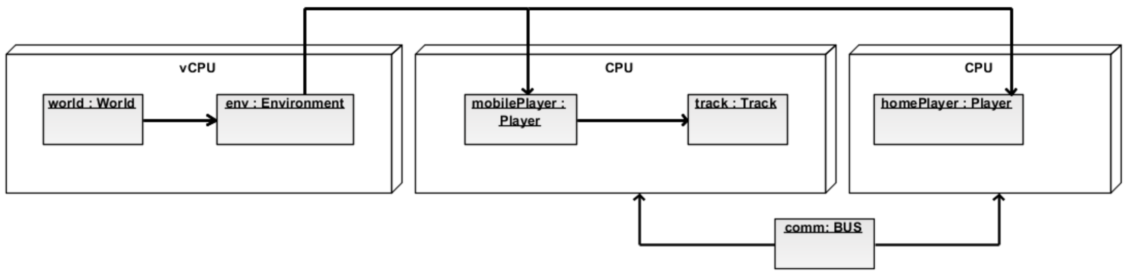


Figure 4.7: Deployment diagram of the case study ConPlay system setup.

Each Player is deployed to its own CPU, which is then connected by a BUS connection to enable the migration.

Chapter 5

Evaluating Alternatives in Dynamic Reconfiguration

This chapter explores existing approaches in literature in relation to VDM-RT. Section 5.1 examines an approach of introducing the modelling of heterogeneous networks and their relations in to the modeling language Creol. Section 5.2 considers a proposal for integrating dynamic data flow network modeling into the modelling language Creol. Section 5.3 looks into an approach for giving middleware applications support for dynamic reconfiguration. The chapter is concluded in Section 5.4 by a summary and assessment of the examined approaches in relation to VDM-RT.

In order to better comprehend the area of dynamic reconfiguration in distributed systems, existing approaches in literature is explored with special language constructs and system designs as a focus area. With the exploration of the state of the art approaches in the field, a broader perspective is established on different methodologies for characterizing and designing a dynamic reconfiguration mechanism for VDM-RT.

When considering the design of a dynamic reconfiguration extension for VDM-RT there are two central aspects that need to be investigated in order to derive the design.
The *first* aspect involves the creation of a language construct for describing reconfiguration which may require semantical adjustments of VDM-RT. Such an alternation is nessesary to provide a way of expressing the dynamic changes that needs to be carried out the network topology of the model and the deployed objects during run-time execution.
The *second* aspect concerns a design for the technical part of having dynamic reconfiguration in VDM-RT. A design needs to be engineered, which determines how the dynamic reconfiguration is actually performed in the interpreter and what the impact on the model will be when a change occurs. This concerns low level matters such as execution disruption, state integrity and mutual consistency.
We define the former as the *language construct aspect* and the latter as the *technical design aspect*.

The literature can be categorized into the research emerging from formal methods and the research with a background in distributed systems. In the formal method research the central point is on the semantics aspect and the value of a having a generalized solution to the analysis and verification of complex distributed systems. In contrast the research with a basis in distributed systems is more concerned with technical solutions to aspects such as performance, robustness and preservation of consistency. The different focus areas of the two research categories is reflected in the relevance of the literature in relation to the two main aspects considered for the VDM-RT extension; the language construct aspect and the technical design aspect. Consequently some parts of the literature will only be relevant in relation to one aspect while others are applicable for both aspects.

For the *language construct aspect* the evaluation criteria essentially comes down to what makes sense? What is the most reasonable and practical way of describing and applying changes in the network toppology and object migration. What will be the most sensible way for the system developer$^{\mathcal{T}}$ to describe the desired reconfiguration?
As to the *technical design aspect* the important element is to find a technical solution that conforms to the current run-time model of VDM-RT and does not affect the run-time execution except for the necessary changes when an actual reconfiguration takes place.

The different approaches will be put in the context of VDM-RT by applying them to the VeMo case study as a pseudo implementation. The actual approaches will not be implemented in VDM-RT, but the model in the case study will be altered as if they were. This makes it possible to make an experimental evaluation of the pros and cons for the different approaches in relation to VDM-RT and thereby aid in determining the most feasible approaches for a dynamic reconfiguration extension.
When applying the approaches to the case study, the exercise is to maintain the central facets of the approach, such as the language primitives, while still keeping the language structure and basic philosophy of VDM-RT intact.

The overall goal of this chapter is to determine the best possible design for a dynamic reconfiguration extension to VDM-RT, by exploring different alternatives.

5.1. An object-oriented component model for heterogeneous nets

In [Johnsen&08] the possibility of modeling communication in networks of varying quality in relation to e.g. reliability, is proposed by extending the executable modeling language Creol [Johnsen&07]. The framework is aimed at creating a practical modeling of heterogeneous system connected by different networks. With a basis in formal methods the framework can act as source for analyzing and validating system properties and structures.

5.1.1 Basic Concepts

Creol is an object-oriented programming language which provides constructs for modeling distributed concurrent objects that interact by asynchronous method calls. An active concurrent object represents a separate computational unit by having a single thread of execution which control flow is managed by processor release points, i.e. essentially guarded statements [Dijkstra75]. By using asynchronous method calls and asynchronous message passing the transfer of control between distributed concurrent objects does not occur, as synchronization and response blocking is avoided. Instead the concurrent object can continue its execution during a remote method call and thereby the processing is separated from the communication. Objects are strongly typed by interfaces and classes may use cointerfaces for specifying the type of objects allowed to call as specific method on the class.
In order to model the different entities of heterogeneous networks and their relations Creol is being extended with the representations of network components, networks types, global time, network

message processing and network error handling. A Network component encapsulates multiple concurrent objects into a single element in which the contained objects communicate directly between each other, while external communication to the contained objects is done through the component.
The framework considers three types of network:

tight: Reliable connection with guaranteed FIFO ordering of messages;

loose: Reliable connection with no ordering of message, and

wireless: Unreliable connection with the risk of transmission collision and loss.

The networks are created by linking different components together through a specific network type which then determines the quality of the communication. Components may have multiple connections consisting of different types of network.
The distributed communication is based on messages which contain information about the network type as well as sender and destination identity. Three different kinds of messages are defined:

Invocation: Includes the method and parameter for the remote method invocation;

Completion: Contains the result value of a remote method call, thereby giving an indication of successful communication, and

Error: Represents network errors such as collision or communication failures which are being reported to the caller.

These messagetypes support a central safety property of the Creol model; that a sender is aware of result of sent messages. Meaning that either the communication was a success or a failure. The involved objects are responsible for handling potential errors themselves.

The extension of Creol introduces the special class **`system`** which is used to set up the overall system and network topology by creating components and establishing links between the components. This is in an identical approach to the system class of the existing VDM-RT language. The **`system`** object can then later be used to modify these links
A central entity in the Creol extension is the network component. VDM-RT does not have components but the Creol description of components as entities able to interconnect over defined networks and being encapsulation of multiple active concurrent objects bear a close resemblance to the VDM-RT `CPU` class. The similarity is so close that the notion of a component will not be introduced into VDM-RT as it already exist through the `CPU` class.
The network glue is called **`link`** and like the `BUS` class of VDM-RT they define the relation between components and allows data to be moved across them but this is where the resemblance stops.
Unlike a VDM-RT `BUS` a link does not have the serving policy nor the indication of bus speed. They do however have the ability to model different network types and the property that the links are implicit created, they do not need to be defined.
In Creol the active objects are added to their component when the object is created, with the command:

```
new object in component
```

This is very different from the way objects are deployed to `CPU`s in VDM-RT where object creation and deployment is separated. By binding the object deployment directly with the creation of the object a strong and lifelong relation between the object and the component is indicated, which might not be the desired behavior under all circumstances. For instance the direct binding may create an unnatural mindset when considering a system which relies on object migration. Consequently this style is disregarded and the current deployment method of VDM-RT is unchanged, as shown in Listing 5.1.

```
cpu1.deploy(ctrl1)
```

Listing 5.1: Deployment of objects on CPUs

5.1.2 Changing network topology

As mentioned above the extension introduces language abstractions for specifying dynamic reconfiguration of the network between components. A network is established by using the **link** statement which specifies that one set of components should be connected to another set of components through a certain type of network. A connection between components is torn down by using the statement **unlink** that specifies a subset that should be disconnected from the superset of components which are connected over a specific network type.

```
link A wless B
unlink A wless B
```

A specific property of linking components through a wireless (**wless**) network is that the connection is one way with the components on the left hand side being linked to the right hand side component but not the other way around. Meaning that A can communicate with B, but B cannot communicate with A. The network types **tight**, **loose** are linked in both directions.

5.1.3 Migration

[Johnsen&08] is centred around a framework for modeling various types of heterogeneous networks, consequently migration is not considered.

5.1.4 Performing reconfiguration

Delving into the dynamic reconfiguration functionality of the Creol extension, it uses the processor release points in the form of the guard statement **await**, to control when a reconfiguration of the network occurs. The dynamic reconfiguration of the network is initiated by some predicate being evaluated to true.
For example unlinking cpu0 from its wireless connection with cpu1 at the global time 500 would be expressed:

```
await clock > 500; unlink cpu0 wless cpu1.
```

In order to move this into VDM-RT, the predicate statement in **sync** member declaration group seems as the closest concept with its use of permission predicates. However placing reconfiguration expressions in the sync declaration group does not seem quite fitting as it does not have anything to do with synchronization.

Instead two approaches are to be considered. One is to let the **system** class become active by allowing the thread declaration within the system class and thereby use the thread to monitor the relevant reconfiguration criteria. Another is to create a new declaration group for the network configuration where reconfiguration statements can be placed.
As the point of the exercise is to apply the Creol approach, when it does not conflict directly with existing VDM-RT methodology, the latter approach is chosen by introducing a reconfiguration group, as shown in Listing 5.2.

```
reconfiguration
reconf unlink cpu0 wless cpu1  => time > 500;
```

Listing 5.2: Reconfiguration declaration group

5.1.5 Integrity and Consistency

The approach is an extension to a modeling language which entail that the environment is more manageable than a real-life system and hereby changes occuring during reconfiguration can be controlled.
Message transfer over a network connection takes time meaning that message can be lost on the network following a reconfiguraiton. The extension has an error messagetype which the interpreter use to inform the communicating objects of any loss. This ensures that mutual consistency is kept.

5.1.6 Applied to the VeMo case study

In Creol the network can be configured in the constructor of the **system** class, which gives the possibility of setting up an intial network as well as defining static network setups which may not be part of the dynamic functionality. This approach can be adopted into the existing **system** class of VDM-RT, although it not being applicable for the case study as it has no need for the initial network setup.
In relation to the case study the central point is that `CPU`s are only connected when they are in the proximity of each other, meaning that the vehicles, in which they are placed, are near by each other. Meaning that a construct is needed in the reconfiguration declaration that makes it possible to specified changes on the wireless connections whenever specific vehicles e.g. `vehicleX` are close by `vehicleY`. From this an in issue emerges as the system class, in the case study, does not have the information needed to evaluate the point of reconfiguration.
This information can be made available by modifying the `VDMcontroller` class with a `Get` operation. As there may be multiple vechicles in range of each other the reconfiguration expression needs to be modified to support multiple values. This is an expansion from the Creol approach of determining when a reconfiguration occurs, as it proposes reconfiguration on the basis of more static criteria, such as time, as shown in Listing 5.2.

```
reconfiguration
reconf link vehicleX wless vehicleY  =>
vehicleX, vehicleY in set vdmCtrl.GetInRange();
```

Listing 5.3: Reconfiguration connecting vehicle in range

In Listing 5.3 a quantified expressions is used to ensure that all possible bindings of vehicles which are in range of each other will be linked over a wireless connection. The unlinking is not considered in the exploration of the language concept aspect, as it essentially would be the same; only using unlink with a `GetOutOfRange` operation instead. How the `GetInRange` and `GetOutOfRange` operations actually are to be implemented is a different matter. Their functionality requires extensive knowledge of the system state and a quite a lot of book keeping, this is however not the main point of this current exercise.
Instead another matter arises in relation to applying the Creol approach to the case study. The `VdmController`, which keeps track of all vehicles, only knows which vehicles are in range of each other or to be more precise it is only aware of the vehicle and controller objects but not the `CPU`s. So despite it knowing the correct objects it has no knowledge as to which CPU the object is deployed on. This leads to an issue, as Creols dynamic reconfiguration expressions are based on connecting components/CPUs and not on objects, which makes it impossible to build or tear down the network with the information available.
One solution could be for the system class to keep track of which objects are deployed on which CPU. Thereby creating a type of registry from which a deployment mapping could be found. Another would be to create a `LookupObject` operation on the CPU class in order to identify the right CPU by querying all of them. A third approach could be to add a general predefined variable in each object which contains a reference of the concrete CPU the object is deployed on.

5.2. Coordinating object oriented components using data-flow networks

In [Jaghoori08] Jaghoori proposes *"a framework for component-based modeling"* of distributed systems in which the network connections between components are dynamic reconfigurable. The presented approach concentrates on integrating data flow network modeling, inspired from Reo [Arbab06], into the object oriented modeling language Creol [Johnsen&07].
Creol delivers the modeling of distributed concurrent objects with asynchronous remote calls and Reo provides a model for structuring the network composition of components through channels and component connectors. The combination of having both data flow networks and distributed objects enables the modeling of a complete distributed system.

5.2.1 Basic Concepts

The main element in Reo is the *component connector* used for reasoning about data-flow between components using a graphical notation. A *component connector* is a variable type which is build in a compositionally manner from other connectors. The most basic connector type is the *channel* that represents a connection with exactly two *connector ends* which can be a source, sink or the combination of the two. Here source and sink denotes outgoing and incoming data flow to and from the network, respectively.

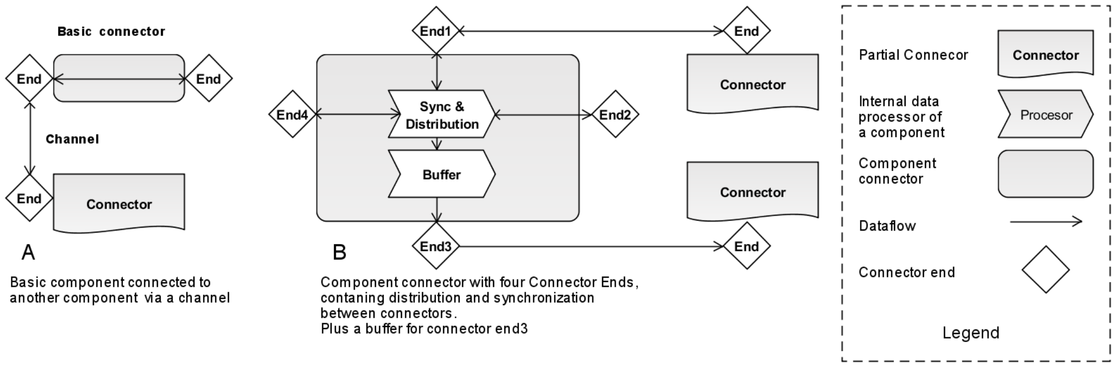

Basic connector
End
End
Channel
End
Connector
A
Basic component connected to another component via a channel
End1
End
Connector
End4
Sync & Distribution
End2
Buffer
Connector
End3
End
B
Component connector with four Connector Ends, contaning distribution and synchronization between connectors.
Plus a buffer for connector end3
Partial Connecor
Connector
Internal data processor of a component
Procesor
Component connector
Dataflow
Connector end
Legend

Figure 5.1: Basics of Reo Connectors

This is illustrated in Figure 5.1A where a channel is created between a basic *component connector* and another connector, that is shown as partial as the internals are irrelevant, which each has a source/sink connector end. Internally the basic connector is pure data flow from one end to another.
By grouping multiple *connectors ends* together, i.e. the possibility of connecting multiple channels, a more advanced *component connector* can be constructed which not only controls the distribution of data between its connected channels but may also add functionality such as buffering, synchronization and retransmission. Any given *component connector* has a fixed number of connector-ends which cannot be changed dynamically.

In Figure 5.1B an example of a complex *component connector* type that has three souce/sink and one source *connector end* (End3) is depicted. In the given example the *component connector* has a data processor, which handles both synchronization and data distributed for all *connector ends*, as well as data buffer on the connector end End3. Two of the *connector ends* are connected to other connectors in order to form at simple network.
This outlines the essentials of the Reo elements used by Jaghoori. An overview of Creol has previously been given in Section 5.1.

With Creol and Reo as the foundation Jaghoori proposes a framework for modeling distributed systems which consists of three elements.

Components encapsulate objects and define interfaces for interacting with the component,

Mobile Connectors establishes dynamic connections between component,

NetworkManager controls dynamic reconfiguration.

The ***first*** element is the component which is characterized by having **`events`**, a façade (interface) and the ability to contain objects. **`Events`** are special constructs that a component may raise in order to alert event subscribers of a given event, such as a service request or a time out. The part of the event construct that is particular interesting for this thesis is the special **`raise_event`** keyword.By using the **`raise_event`** combined with an event operation, predefined in its façade, a component has the possibility to raise system wide event that will be handled by the *NetworkManager*. This enables a component to trigger a network reconfiguration, depending on the implementation of the event handler in the *NetworkManager*.

The façade encapsulates the internals of the component by acting as an interface which describes the component events and the ports of the component. A port is a reference from the component to a *connector end* on a *component connector*. In reality this enables a component to call a remote method on another component. How one port on a component is connected with a port on another component, through a component connector, is transparent to the components themselves as this is handled by a third party. The basics of a network between components through a *component connector* is illustrated in Figure 5.2.

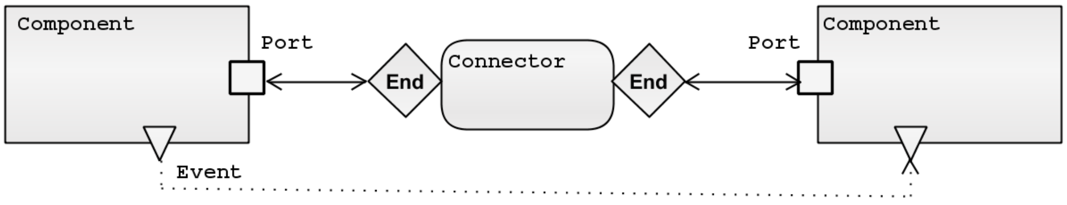


Figure 5.2: Network between two components through a component connector

The ***second*** element is a Creol implementation of the Reo *component connectors*, called the *mobile connector*; connector because it connects different component ports with each other and mobile because the connector-end can bind to different component ports during execution, thereby changing the network topology. Combined with the dynamic creation of new connectors and components these two elements provide the basis for the dynamic reconfigurable network.
The actual dynamic reconfigurations are defined by the ***third*** element; a *NetworkManager*. The *NetworkManager* has knowledge of all components in the system and furthermore defines and controls the network relations between these components by connecting them via *mobile connectors*.

5.2.2 Changing network topology

Reconfiguration of the network topology can be accomplished by the *NetworkManager* changing the binding between components and mobile connectors by rearranging the channels between connector ends. Because the Network Manager is implicitly aware of all components, it is able to reconfigure the network by changing the binding of component ports to *mobile connectors* which themselves are created by the *NetworkManager*. An example of establising a new network connection is shown in Section 5.2.4.

5.2.3 Migration

As [Jaghoori08] has the main focus on component-based modeling using data-flow networks migration is not considered.

5.2.4 Performing reconfiguration

The component-specific **`raise_event`**, mentioned above, are handled by the *NetworkManager* when triggered in a given component that requests a network reconfiguration. The decision of performing a reconfiguration is done on the basis of handled events and the reconfiguration policies defined in the *NetworkManager*. A configuration on the basis of a **`raise_event`** is illustrated in Figure 5.3 where two components are connected with a basic *mobile connector* by the **`NetworkManager`**.

In Figure 5.3A the *NetworkManager* has a reference to two components, and it creates a *mobile connector* upon handling a **`raise_event`** from one component. In Figure 5.3B the *Network-Manager* has a reference to both the component and the *mobile connector*and a data network has been created between the two components through the *mobile connector*.

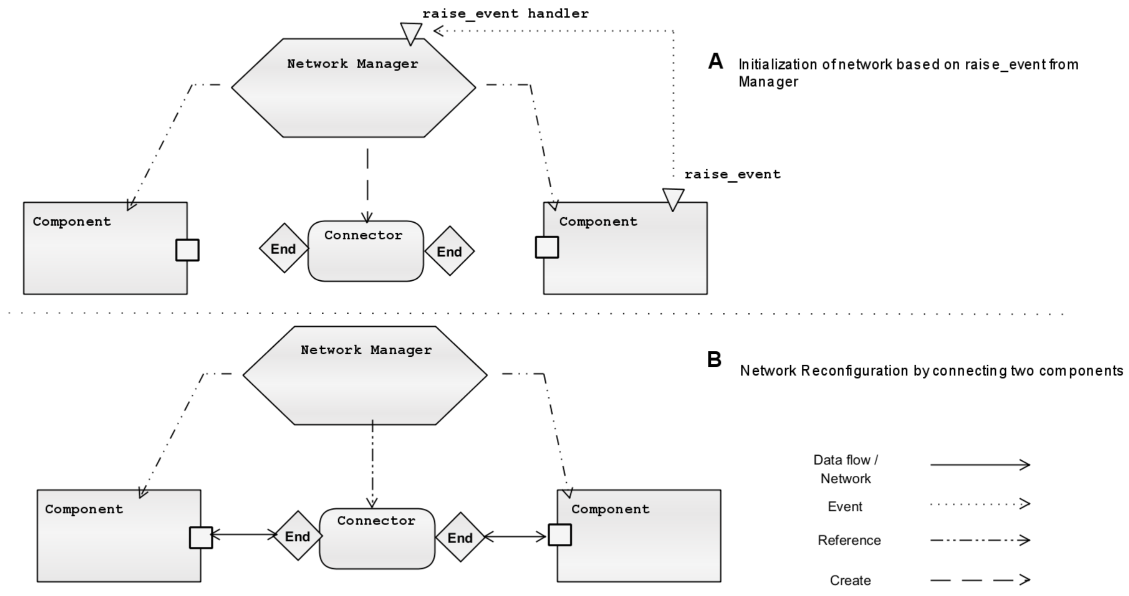


Figure 5.3: Initialization and reconfiguration of network based on raise_event

5.2.5 Integrity and Consistency

As the approach builds upon a modeling language time is under the complete control of the model, which mean that the reconfiguration can occur without any other processing taking place. Hereby integrity and consistency concerns are easier attainable as it eliminate object state changes and remote methods invocations occurring during the reconfiguration. Likewise all connections are considered uninterruptible and remote calls happen instantaneously so no messages can be lost on the network.
The system behaviour when a remote method is unable to return its result to the caller before a disconnect is not considered. Does it drop the data or wait forever?

5.2.6 Applied to the VeMo case study

In Jaghoori' s approach the fundamental aspect is that a network is build between components through mobile connectors which is controlled by a Network Manager, therefore these three elements needs to be introduced in the context of VDM-RT to make the approach applicable to the case study.

In the approach proposed by [Johnsen&08], which was explored in Section 5.1, the VDM-RT `CPU` class was used as a component because of the similarity between the VDM-RT `CPU` and Johnsens definition of a component. Jaghoori's approach needs a more concrete notion of components than can be delivered by the CPU class, because interfaces and events need to be declared for the specific type of component. Jaghoori uses façades for defining components, which not only define events and ports for communication, but also generalize the class that realize the façade, as being a component by adding an implicit event. In a components constructor an `initPorts` event is implicitly called by the interpreter to notify the *NetworkManager* of the components creation. This is how the *NetworkManager* is always aware of all components in the system. Façades and interfaces are not directly supported in VDM-RT, it is however possible to get the concept of an interface by using abstract classes.

In order to keep the object oriented principles and basic philosophy of VDM-RT both façade and component types will be classes and not a new language construct. This is in accordance with other extension previously applied to VDM-RT, such as the `CPU` and `BUS` class [Verhoef&06]. Due to the lack of an explicit interface keyword the **`is subclass of`** keyword will be used for defining new types of facades and components. This is despite the term subclass implies implementation-inheritance, instead of an incremental definition of a new type based on existing types (subtyping). This means that subclassing an abstract class is to be considered a realization (subtype) of a defined prototype interface instead of inheritance. Using this approach limits the changes on the VMD-RT language primitives and it is not uncommon as is in accordance with the UML notation used for interfaces in languages such as C++ [Fowler&03, p. 71]. Consequently a façade type will be defined as a subclass of an abstract Façade class that needs to be introduced in VDM-RT. Likewise a component type will be defined as a class subclassing a façade type as illustrated in Figure 5.4A.

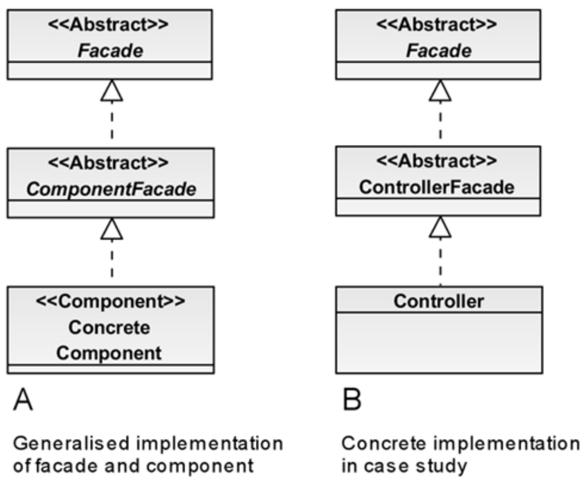


Figure 5.4: Facade and Component realization

In the case study the `VeMoController` class needs to be one kind of component and the `Controller` class another kind. The VeMoController object needs to issue the events that trigger reconfiguration while the `Controller` objects and the network between them are the ones being affected by the reconfiguration.
A new class called the `ControllerFacade` is introduced for defining the facade of the Controller class. In relation to the original implementation of the `Controller` from the case study the `ControllerFacade` does not define any new operations nor does it introduce any events to be implemented by the `Controller`. Instead it merely generalize instances of the `Controller` as being components so that the implicit `initPorts` event is fired to make the *NetworkManager* aware of new components. The definition of the `Controller` class as a com-

ponent is illustrated in Figure 5.4B and in the implementation in Listing 5.4.

```
class ControllerFacade is subclass of Façade

operations
  {prototype operations of original Controller operations}

end ControllerFacade
......
class Controller  is subclass of ControllerFacade

operations
  {original Controller operations}
end Controller
```

Listing 5.4: Facade and Component classes

The `VeMoControllerFacade` is realized by the `VeMoController` which enables it to raise the `InRange` event defined in the VeMoControllerFacade, as shown in Listing 5.5.

```
class VeMoControllerFacade subclass of Façade
operations

event public InRange: ControllerFacade * ControllerFacade ==> ()
InRange(ctrl, ctrl2) ==
(
 is subclass responsibility
);

end VeMoControllerFacade
```

Listing 5.5: VeMoControllerFacade defining a prototype event handler for the InRange event

This is put into the original case study operation in Listing 5.6, which calculates when vehicles are in range.

```
public CalculateInRange: () ==> ()
CalculateInRange() ==
(
 let units = rng ctrlUnits in
  for all unit in set units do
  (
   let inrange = FindInRangeWithOppositeDirection(unit, units)
   in
   (
     if(card inrange > 0)
     then
     for all oncomingVehicle in set inrange do
13        (
14          raise_event InRange(unit, oncomingVehicle);
15
16          {original implementation of data communication...}
17        );
18      )
19    )
20  );
```

Listing 5.6: VeMoController raiseing the InRange event

On line 14, in Listing 5.6, the **raise_event** is used to trigger a network reconfiguration in order to establish a network between two vehicles before communication takes place. Again for the sake of simplicity only the establishment of a network when the vehicles are in range is considered, while the challenge of determining out of range and disconnection is pushed aside for this current exercise.
The proposed solution **mobile connectors** that can be considered a type of `BUS` in relation to VDM-RT, as it is the element which makes the connection between components. Unlike the `BUS` the mobile connectors have the ability to connect and disconnect its attached components which enables reconfiguration. The mobile connectors are conversely limited by their static size in relation to the number of components they can handle. However for the case study a combination between the two is needed as both the reconfiguration and a dynamic amount of network attachments are needed. Therefore a `Network` class is added that allows the connection/disconnection of set of `ControllerFacades` meaning that the number of components that can be on a single network is made dynamic, which is not possible in Jaghoori's approach. Listing 5.7 shows the prototype operations of the `Network` class. The `Network` class could either be implemented as simple connections through a type of `BUS` or to do more advanced processing such as buffering or retransmission. The `Network` class is not a general solution for all VDM-RT models as it relies on specific kinds of facades when creating a network.

```
class Network

operations

 public Connect: set of ControllerFacade ==> ()
 Connect(ctrls) ==
   is not yet specified;


 public Disconnect: set of ControllerFacade ==> ()
 Disconnect(ctrls) ==
   is not yet specified;
end Network
```

Listing 5.7: Network class

The actual reconfiguration is done by the Network Manager, which is also implemented as a class. The event declaration in a facade is essentially a contract between the Network Manager and the component realizing the facade, which entails that the Network Manager must implement handlers for all events of all components. This is a achievable task in case study as it in a full implementation merely has a couple of events (connect/disconnect of controllers), but it should be noted that this can be quite a substantial task for the system developer of a system with a large number of components.

Listing 5.8 show the implementation of the `NetworkManager` with the network being created in the constructor. In order obtain the prototype of the event handler the `NetworkManager` realizes the `VDMControllerFacade` that defines the event. In the `InRange` event handler the two controllers in range of each other are simply connected on the network. Alternatively more advanced reconfiguration strategies could be implemented.
The Network class also contains `Disconnect` operations for tearing down a network between two `Controllers`. When the vehicles move around their `Controllers` eventually go out of range and the network between them are to be torn down. In order to know which `Controllers` go from a state of being in range to an out of range state some managed of state history is necessary. This is however unrelated to extension of the VDM-RT language which is explored in this section.

```
class NetworkManager is subclass of VDMControllerFacade

instance variables
networkChannel : Network;

operations
  public NetworkManager: () ==> NetworkManager
  NetworkManager() ==
  (
    networkChannel := new Network();
  );

  public InRange: ControllerFacade * ControllerFacade ==> ()
  InRange(ctrl , ctrl2) ==
  (
    networkChannel.Connect(ctrl, ctrl2);
  );

end NetworkManager
```

Listing 5.8: The NetworkNanager class creating a network and implementing the event handler of the VDMControllerFacade

An overview class diagram of changes made in comparison to the original case study design is shown in Figure 5.5. As the traffic light is also part of the network, the `TrafficLight` class should have a facade in order to define it as a component and it should be reference by the `Network` class. However the necessary changes bare so close resemblance with the changes already applied to the `Controller` class, that the implementation for the `TrafficLight` component is left out and is stippled out in Figure 5.5.

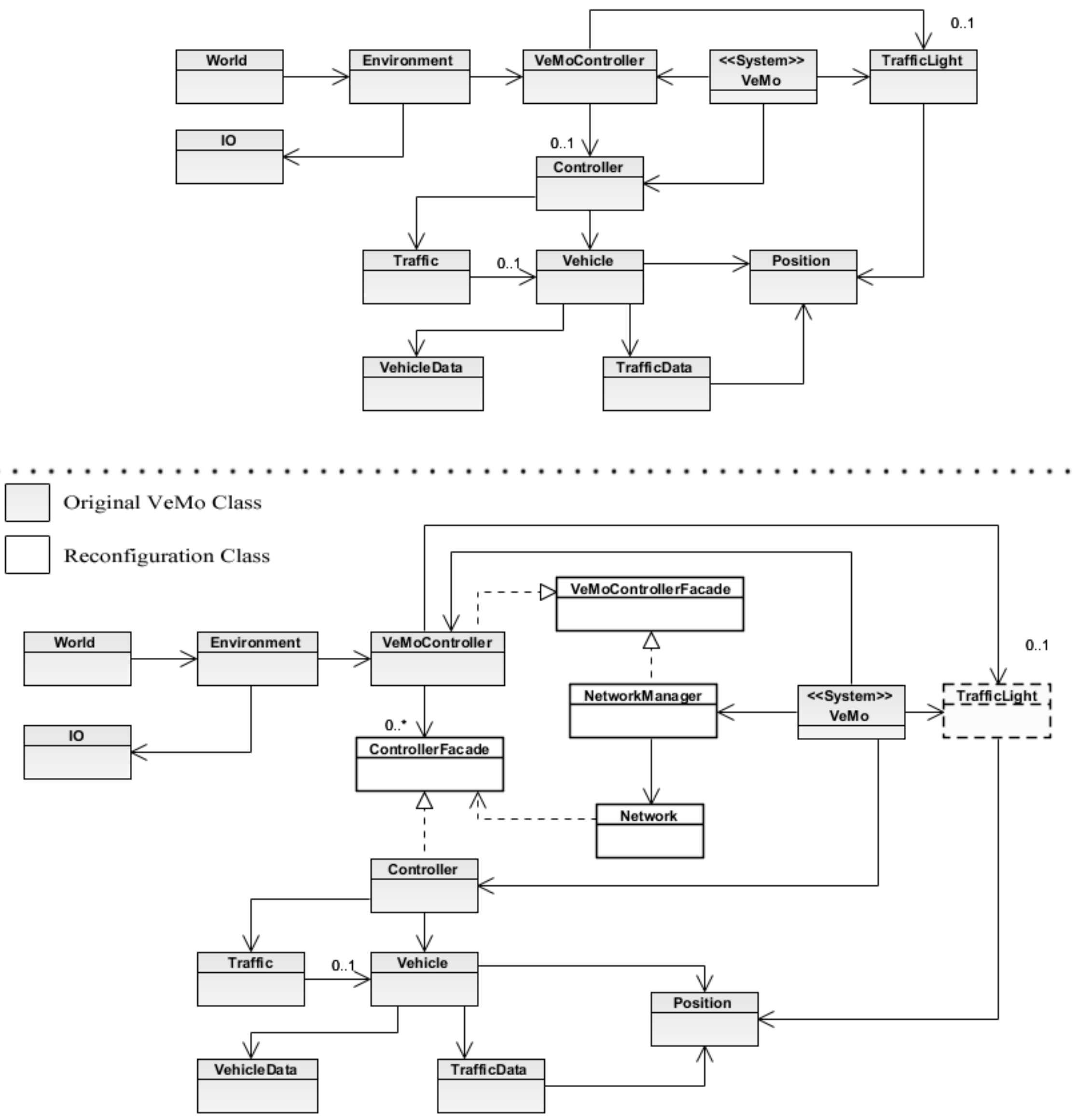


Figure 5.5: Overview class diagram compared to class diagram of case study

5.3. Dynamic Reconfiguration for Middleware-Based Applications

An approach for giving middleware applications support for dynamic reconfiguration is introduced by [Wegdam&03]. The term middelware-based applications covers distributed software systems which are based on existing middleware technologies, such as CORBA [OMG&96]. This approach differs from the other approaches above, as it is intended to be embedded into concrete software that use real distributed system technology and not as an extension to a formal method.

5.3.1 Basic Concepts

In fact there is no formal representation of the system or the reconfiguration capabilities, as everything is implemented in the middleware technology while state conversion and control of the dynamic reconfiguration are placed in the application source. Hereby the reconfiguration functionality and the management of the configuration changes are to some extent separated into two different modules. The reconfiguration functionality is not completely independent from the application implementation. The reconfiguration part requires application specific state translation and thus it is not a fully generalized solution. Nonetheless the separation and the focus on using middelware technology provide an interesting and alternative view on the description and implementation of dynamic reconfiguration.

In [Wegdam&03] the approach has been realized as a CORBA service in which the dynamic reconfiguration functionality is used for object migration in a load distribution service.
The proposed approach does not necessary apply for the entire system in which it is used. Objects cannot automatically be considered as reconfigurable; instead the reconfiguration is enabled by a special Reconfigurable Object type. These objects contain an application specific implementation, but at the same time they are the basic building blocks of the reconfiguration.

To enable the reconfiguration mechanisms a *Dynamic Reconfiguration Service* is introduced in to the systems that need to be reconfigurable.

The service consists of the following objects:

Reconfiguration Manager Is the central entity of the Dynamic Reconfiguration Service and is the main controller of the reconfiguration. The Reconfiguration Manager has a relation to all other reconfiguration-related objects in the system. It passes on object creation and removal to the Reconfigurable Object Factories, it registers and deregisters objects through the Location Agent and it drives the system into a safe-state in coordination with the Reconfiguration Agents.

Location Agent Is a centralized entity in the system which maps location-independent object references to real object references of Reconfigurable Objects.

Reconfiguration Agents Are created for each process and has a direct relationship with all the Reconfigurable Objects in the process as well as to the Reconfiguration Manager. The agent plays a central part in reaching the safe-state (see Section 5.3.5) as its main task is to filter and manage requests to its related Reconfigurable Objects.

Besides the service object some application specific objects are needed:

Reconfigurable Object Are the basic building blocks and they contain application specific implementation.

Reconfigurable-Object Factory Implements a Factory [Gamma&95] which create instances of reconfigurable objects on request of the Reconfiguration Manager.

State Translator Is used to add additional application specific change to the object state during reconfiguration, and can be seen as a sort of state Decorator [Gamma&95].

An overview of the reconfiguration objects and the relationship between them can be seen on Figure 5.6.

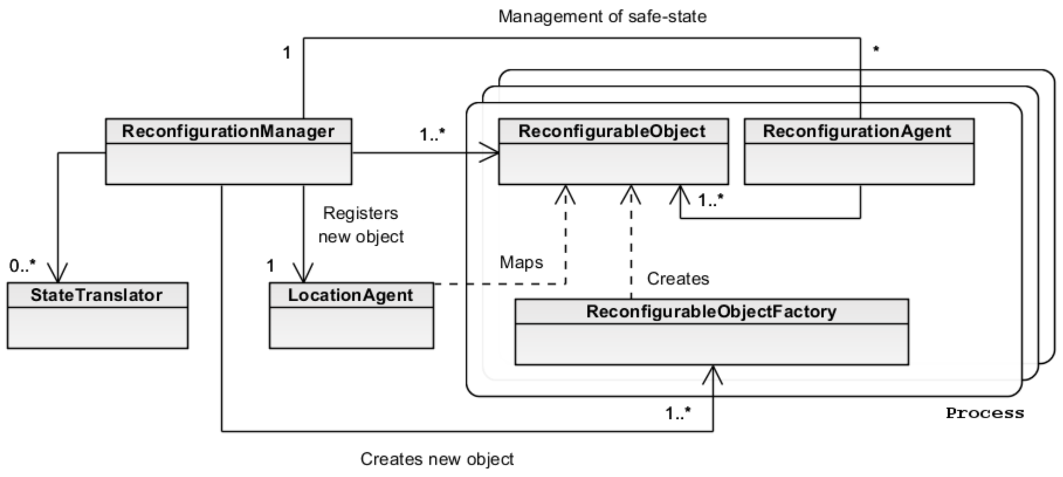


Figure 5.6: Class diagram of Reconfiguration classes overview

5.3.2 Changing network topology

As the above presentation indicates there are no functionality for changing the network topology and connections between entities. The reason is that most middleware technologies use the Internet Protocol as their primary communications protocol and have location transparency as a central property. Therefore object references are Uniform Resource Identifiers URI[7] that can be received by performing a lookup through name services, such as CORBAs COS Naming Service [OMG&04]. Accordingly most middleware technologies have a sense of a connect/disconnect operation natively, in a CORBA context one could say that objects are in a network as long as they have an valid object reference to the remote object.

The use of URIs are not relevant in a VDM-RT context, as one can take advantage of the fact that the whole “world” is contained within the model and it is therefore possible to use object references instead of URIs, as VDM-RT object references actually are unique across `CPU`s. However getting the notion of a shared communication channel, like the wireless connections of the VeMo case study, is difficult to achieve in middleware as communication is normally directly between objects over a network shared with the rest of the world (i.e. the internet). If the communication between objects merely depends on object references, it can already be modelled in VDM-RT where it would be equivalent to having all `CPU`s connected to a single `BUS`, leaving only the relations of object references to decide if communication is possible. This can however been seen as circumventing the current VDM-RT methodology where `CPU` and `BUS` relationships are the basis for communication. The use of `CPU` to `BUS` relations and the fact that the communication protocol is being abstracted away in VDM-RT entail that the principles of network topology found middleware are not directly transferable to VDM-RT.

5.3.3 Migration

The approach has a mechanism called *replacement* which allows for one Reconfigurable Object to be replaced by another Reconfigurable Object instance and thereby updating or changing the implementation at runtime. In Wegdam's definition of *replacement* the new object may run from a different location than the replaced object, meaning that it is actually a reference to the new object that takes the place of the replaced object. As *migration* is the movement of an object from one location to a new location, the definition of the replacement operation entail that *migration* is considered a special type of replacement where the object stays the same but the location is changed.

The migration of an object entail a change in location and thereby a change in the object reference. To handle this challenge location-independent object references are used, which are a type of static global references that lay on top of the real object references. Instead of referencing the real object, the location-independent object references refer to the Location Agent which maps a unique object identifier to the real object reference. All Reconfigurable Objects contains a unique object identifier which is implicitly passed as a hidden parameter to all remote operation calls, making the lookup in the Location Agent possible.

5.3.4 Performing reconfiguration

The Reconfiguration Manager implements the *ReconfigurationStep* interface through which the reconfiguration is managed. The "step" indicates the process of going from one configuration into another during an atomic reconfiguration "step".
The interface defines four different reconfiguration operations, specifically *creation*, *replacement*, *migration* and *removal*. *Creation* and *removal* is fairly self-explanatory as they allow for the creation and removal of Reconfigurable Objects in the system during runtime. *Replacement* allows for one Reconfigurable Object to be replaced by another Reconfigurable Object instance, thereby updating or changing the implementation at runtime. Although the object implementation is replaced, its state may be preserved. While the type compatibility is ensured by interfaces, it is the system developers' responsibility to ensure that the objects are satisfactory in regard to functionality and quality of service demands. *Migration* is described in Section 5.3.3.

5.3.5 Integrity and Consistency

When a reconfiguration step is performed, the system will be brought to a safe-state, prior to the actual configuration change, in which the state and mutual consistency of the affected object are ensured. The system is said to be in a safe state when the involved objects are not currently involved in any invocations and they will not be involved in any invocations until after reconfiguration. Basically this means that none of the involved objects will be processing any requests or waiting for any replies.
To reach the safe state, the mechanism freezes all interactions of the involved objects in a complex succession in which both distributed reentrance and invocations whose processing is necessary for the system to reach the safe-state are allowed to pass through the initial freeze. As each object freezes any ingoing message requests will be queued and the queue will be processed after the reconfiguration step has finished.

The objects which are allowed to pass through are the objects in the “laissez-passer set”. All remote invocations carry with them an invocation path, which is the set of objects that the invocation has passed through nested calls. Likewise the objects that will be affected by the reconfiguration, e.g. objects migrating, will be placed in an “affected set”. Any objects that are in the intersection between the “invocation path set” and the “affected set” will be the “laissez-passer” set.

The safe-state has the advantage that object state as well as mutual consistency is ensured and that there is no message loss on the network as they will be queued and passed on following reconfiguration. The disadvantage is first of all that it can take time to reach the safe-state which might result in the reconfiguration occurring in an unpredictable way. Secondly under some particular circumstances following a specific sequence of interactions, which details are out of scope for this thesis, the safe-state might never be reached.
The `Reconfigurable Object` defines some methods for entering and exiting the safe-state as well as some state-access operations to enable access to the objects internal state during reconfiguration. This is done to allow for state to be transferred during object replacement.

5.3.6 Applied to the VeMo case study

Figure 5.7 illustrates how this approch can be applied to the VeMo case study.

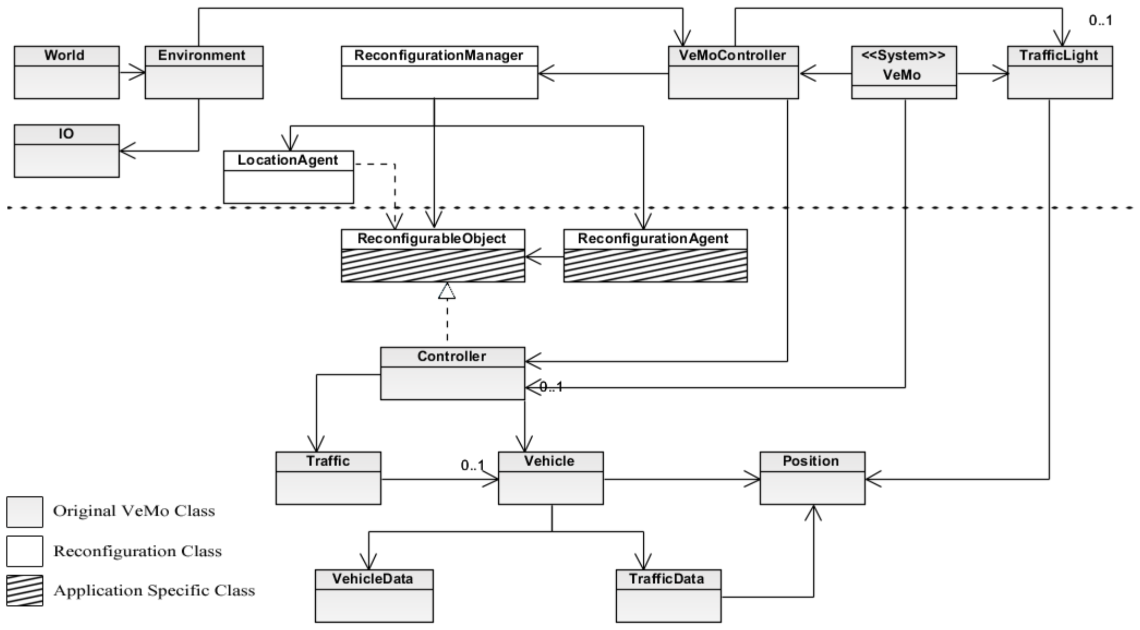


Figure 5.7: Class diagram for VeMo with Reconfiguration Objects

The class with the slashed lines indicates the classes that are application specific and must be implemented for the particular system. The other two classes added for the reconfiguration are generic.
Every class that is beneath the vertical dashed line can be considered as a group of objects that will be deployed to each `CPU` in the system, oppositely there will only be one instance of the classes above the line.A simplified version of this setting is depicted in Figure 5.8.

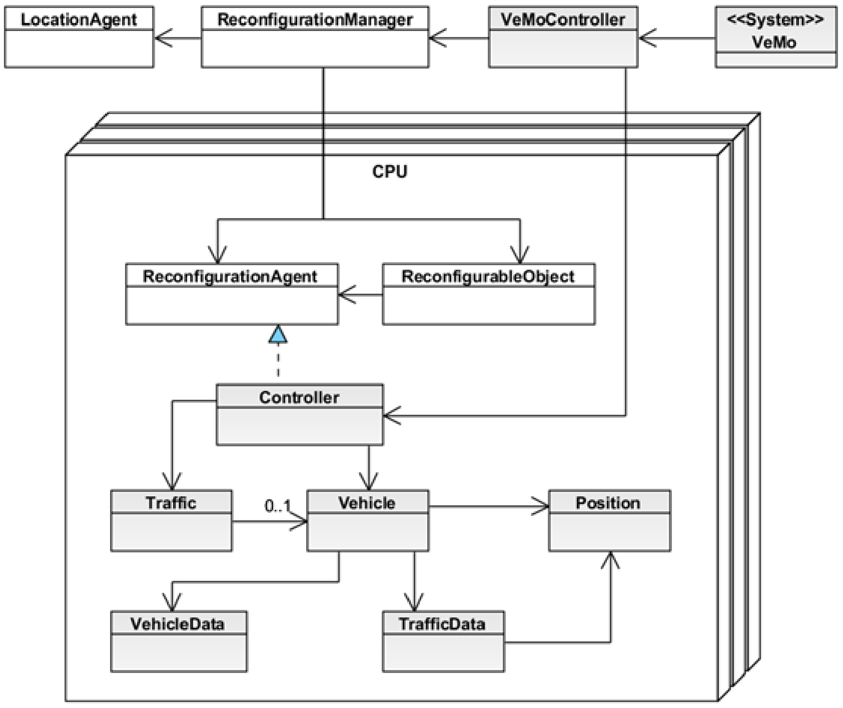


Figure 5.8: Deployment diagram of VeMo objects in relation to CPUs

The idea of having `Reconfigurable-Object Factories` has been left out because while it makes sense in a middleware context where the distributed objects in general originate from some type of factory, it does not really serve a point in VDM-RT. Instead objects that implement the `ReconfigurableObject` interface will be registered with the `ReconfigurationManager` automatically by the interpreter when the object is instantiated.
A relation between the `VeMoController` and the `ReconfigurationManager` has been created to allow the `VeMoController` trigger a reconfiguration.

However as to reconfiguration the approach is really not that applicable for the case study, as the middleware on which it is build only require object references to communicate, as explained in Section 5.3.2. With this approach the existing VeMo model would be sufficient as object references are only kept during communication.
Nonetheless it is still interesting to consider the overall approach in the context of VDM-RT because it contains some very interesting solutions as the root in middleware technology utilizes some very different mechanisms from what exists in the current version of VDM-RT.

5.4. Summary and Considerations for Adaption Into VDM-RT

The purpose of the chapter were to get knowledge and inspiration from existing literature in order to establish the best approach for incorporating dynamic reconfiguration into VDM-RT.

The objective were to investigate two central aspects, namely the *language construct aspect* and the *technical design aspect*, as these are the two main characteristics needed for a dynamic reconfiguration extension.

The different approaches was put in the context of VDM-RT, by applying them to the VeMo case study, in order to investigate how the dynamic reconfiguration language primitives could be incorporated into VDM-RT , while still keeping the language structure and basic methodology of VDM-RT intact.

5.4.1 Language construct aspect

The choice on a language construct is essentially a subjective choice as it depends on a practical and reasonable way of describing and applying the dynamic reconfiguration change to a model. However an important objective is to avoid language construct that differs to greatly from the existing VDM-RT dialect.

5.4.1.1 Enabling reconfiguration

In order to enable reconfiguration all three approaches introduce some kind of encapsulation for the reconfigurable entity. [Johnsen&08] encapsulates multiple concurrent objects into single network components between which communication can occur. This construct resembles the `CPU` class in VDM-RT.

[Jaghoori08] also uses components but in a much more elaborate approach containing elements such as `component connectors` and `connector ends`. In a VDM-RT context this seems overly complex and would lead to an unnecessary complication of the system.

In [Wegdam&03] objects have to inherit from the `Reconfigurable Object` to become reconfigurable. The inheritance adds operations for getting state-access to the object during reconfiguration. The benefit of having these operations is that the System Developer gets to decide what data belongs to the object state, the disadvantage is that it forces operations into objects which must be implemented. This is a functionality which can be abstracted away in a modeling language, since the interpreter has direct access to state date. It does nonetheless have the cost of preventing the System Developer from specifying what comprises the object state. Overall [Wegdam&03] has the most elaborate set of elements needed to enable the reconfiguration, this however has origin in the approach being based on middleware which does not enable cutting corners, like the formal languages.

5.4.1.2 Language keywords

Both of the approaches which have a base in formal methods introduce new language keywords in order to handle the reconfiguration.

In [Johnsen&08] keywords were introduced to handle the **`link`** /**`unlinking`** of network relations and to select the network type, while [Jaghoori08] introduced a keyword to raise the reconfiguration event. Introducing a new keyword has the cost of removing an identifier from the model and thereby risking naming collisions with existing models, as a result a solution without any new keywords would be beneficial. [Wegdam&03] were building on top of existing middleware technology and therefore did not introduce any language keywords, instead all reconfiguration was performed through operation calls to a `reconfiguration manager`.

5.4.1.3 Managing reconfiguration

All three approaches use some notion of centralized management of the reconfiguration.

[Wegdam&03] has a reconfiguration manager which coordinates reconfiguration by having a reference to all the other components in the system. The dynamic reconfiguration is initiated through operation calls to the reconfiguration manager.
[Johnsen&08] define the system architecture through a **`system`** class, which at the same time function as a `reconfiguration manager`. The dynamic reconfiguration of the network is initiated by predicates define within the **`system`** class. In a VDM-RT context this would require the language to be extended with a new reconfiguration declaration group in the **`system`** class. This has the benefit of centralizing the initiation of reconfiguration, where the approach used by [Wegdam&03] has the weakness of reconfiguration becoming more unmanageable because it can be issued from the entire system.
[Jaghoori08] use a combination of these approaches where reconfiguration can be initiated from the entire model, but the exact reconfiguration is controlled by a centralized policy in a Network manager. Only Jaghoori enables the reconfiguration to be managed through a replaceable policy.

A policy ensures that the dynamic reconfiguration takes place in a predictable manner [Payne08]. While reconfiguration policies are an important subject, it is considered out of scope for this thesis by subjective choice.

5.4.1.4 Error Reporting

While [Jaghoori08] does not consider errors, [Johnsen&08] introduce error reporting through an `error message` type which will be returned from an operation call if network errors occur. This means that the error must be handled by examining the return value, it will not propagate up the system like an exception. [Wegdam&03] on the other hand make use of the existing error handling found in middleware, such as CORBA, which is based on exceptions (see Section 3.5.3).

VDM-RT has exception handling natively, which makes this type of error reporting a strong candidate.

5.4.2 Technical design aspect

The technical part concerns how dynamic reconfiguration is actually performed in the interpreter and what impact a reconfiguration change will have on the model.
The approach proposed by [Wegdam&03] is intended to work with middleware technology, which entail some boundaries not found in the formal models. Consequently a large part of [Wegdam&03] concerns implementation mechanisms for managing system state and ensuring system consistency. A large part of these challenges is avoided in the formal models. Both [Johnsen&08] and [Jaghoori08] use the power of the system being a model. They assume that reconfiguration occur instantaneously and takes up no time, meaning that the reconfiguration can occur without interruptions. Here by the necessity of doing transaction management or elaborate implementations to ensure the persistency is avoided.

Chapter 6

Engineering a Dynamic Reconfiguration Extension to VDM-RT

In Chapter 5 two central aspects were investigated in order to derive a design for the reconfiguration extension. The first aspect was the language construct aspect which is focused on a language extension for describing reconfiguration. In this chapter this aspect is covered in Sections 6.1 and 6.2. Section 6.1 defines the language extension itself, while Section 6.2 defines the possible semantics of the operations defined. The second aspect is the Technical design aspect which concerns the implementation for the language constructs and semantics defined via the language construct aspect. For the extension and its semantics to be adopted into the Overture Project, a number of adjustments to the VDMJ interpreter are necessary. The Technical design aspect is the focus of Sections 6.3 and 6.4. Section 6.3 concerns how the langauge extension can integrated into VDM-RT, while Section 6.4 contain implementation specifics in relation to VDMJ. Lastly a summary of the implementation changes is supplied in Section 6.5

6.1. Creating a language extension for dynamic reconfiguration

With a basis in the existing language structure and basic philosophy of VDM-RT and the related work presented in Section 5, the language constructs for the dynamic reconfiguration can be defined.

Given that the distributed system architecture is defined in the special **`system`** class and all `BUS` and `CPU` 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 network manager, but a specific network manager class is not created. Adding a new a manager class would just add an unnecessary complication of the language.

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 arise. 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 dynamic reconfiguration operations are presented in Table 6.1.

Operation	Parameters	Description
Sys.ConnectToBus(obj , bus)	Object $*$ BUS	Connect object with a BUS.
Sys.DisconnectFromBus(obj , bus)	Object $*$ BUS	Disconnect object from BUS.
Sys.Migrate(obj, cpu)	Object $*$ CPU	Migrate object to a specific cpu.

Table 6.1: Dynamic reconfiguration operations

The extension does not introduce any new classes or 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 either extended or inherited. Likewise despite it being defined by the System 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.

6.2. Possible semantics of the dynamic reconfiguration extension

Section 6.1 defined the operations needed for describing dynamic reconfiguration in VDM-RT. For each of these operations the possible semantics needs to be defined in order to determine their exact behaviour. The dynamic reconfiguration extension could have numerous possible behaviours which is to be identified, so that an assessment can be made as to which should be the preferred semantics.

In general all reconfiguration changes will occur as atomic operations, meaning that they are uninterruptible by any other processing. This is necessary to ensure the consistency and integrity of the system as continued processing on the affect objects, `CPUs` or `BUS` connections will make it very difficult to ensure the behavior of the reconfiguration and of the system following the reconfiguration. For instance an active object might start placing messages on the BUS in the middle of a reconfiguration change, because the object has not yet been affect by the reconfiguration, but the intended receiver of the messages has. This will result in an inconsistency, because the objects do not share the same understanding of the system topology. Likewise the atomicity ensures that multiple reconfigurations cannot occur at the same point in time.
From a semantic point of view this means that a reconfiguration will have no *duration*. *Duration* is the only representation of time progress in the existing semantics defined for VDM-RT [Verhoef08]. In the semantics a model is executed by stepping from one *execution state* (denoted as a *configuration* in the semantics) to another. This stepwise progress entails that as instructions are executed the *execution state* is affected, but the global time of the model is not changed until one global time is issued for all configurations. The global step can only occur when all current threads are ready to execute a *duration* instruction or have terminated. This enables the semantics to distinguish between moving time and processing instructions.

6.2.1 Connecting a CPU to a BUS

The `ConnectToBus` operation is used to establish communication between an object and other objects which are already connected to the given `BUS`. Connecting objects over a `BUS`, in reality

entails the connection of `CPU`s, to which the objects are deployed, through a `BUS`.
This operation has the simplest functionality of the dynamic reconfiguration operations and accordingly the semantics are also the easiest to define:

- The `ConnectToBus` operation (defined in Section 6.1) requires the reference to an object as well as to a `BUS` in order to connect the object to the specified `BUS`.
- As the `ConnectToBus` operation merely establishes a connection and does not alter existing connections there is no need to take care of either reference changes or current communication of the concerned objects.
- When the `ConnectToBus` operation is completed no event or other type of notification will be supplied implicitly to the concerned objects to give awareness of the reconfiguration change. Instead it is left for the System Developer to ensure that the distributed objects are aware of new connections if this information is needed by the remote objects.

  This semantic behavior is chosen in order to keep the VDM-RT model as abstract as possible. If a specific method of notification were to be used it may bind VDM-RT to a specific technology. Instead the extension merely offers the possibility of actually performing reconfiguration, the exact details and events occurring before, during and after the reconfiguration is left for the System Developer to implement in the model.
- Once the `ConnectToBus` operation is called the reconfiguration is to be considered successful. If the connection already exists nothing will be changed, otherwise the connection will be established

6.2.2 Disconnecting a CPU from a BUS

The `DisconnectFromBus` operation is used to tear down an existing connection between an object and a `BUS`. Again this in reality entail changing the `BUS` connection of the `CPU` to which the object is deployed. This operation can affect the existing system and ongoing processing as it alters existing connections which may influence reference changes and current communication. Accordingly the semantics are more comprehensive.

A disconnect reconfiguration scenario can be divided into two parts; where the first concerns messages currently processing on a `CPU` or on a `BUS` during the dynamic reconfiguration and the second relates to processing taking place after a dynamic reconfiguration.

Disconnecting an active entity from its network is a disruptive action which can have great impact on the system. An approach could be followed where disconnection occurs gracefully by allowing the system to halted and be brought to a safe state where there is no processing on the BUS allowing no new messages to be transferred until after the reconfiguration. This would follow the approach by [Wegdam&03] as described in Section 5.3. Conversely an approach could be used where there is no graceful disconnection and any message currently processing will result in data loss and a kind of error notification.
The former has the benefit of ensuring transfers and data in the system by entering a safe-state, but the time taking to reach this state is unpredictable and the safe-state might never be reached.
The latter has the benefit of occurring instantly and always; it does however have the risk of affecting current processing and involves more complex error handling.

To enable VDM-RT to model as many types of dynamic reconfiguration systems as possible, the non-graceful approach is chosen. Unlike the graceful approach it allows the System Developer to get an insight into errors occurring during dynamic reconfiguration, which will result in a greater understanding of the system, than if this processing was handled automatically.
In relation to modelling in the most abstract way, the non-graceful approach is the least intrusive as it does not superimpose a specific way of doing reconfiguration. It does define a specific way of handling errors which may not be available in all types of systems. However in the context of formal modelling is important that errors are either reported or that the system is intentionality built so it will handle would-be errors. Therefore implementing error reporting is not considered limiting in relation to modelling different types of distributed systems.

Essentially the non-graceful approach is chosen for two reasons. Firstly because it allows for a better insight into the model as it requires the System Developer to take more decisions. Secondly it provides a more abstract notion of dynamic reconfiguration and therefore allows a wider range of systems to be modelled. For example a graceful behaviour can be implemented in a model by using VDM-RT with non-graceful disconnection behaviour, while it is more difficult to implement non-graceful behaviour in a model using VDM-RT that has a graceful behaviour from the start.
In order to handle the complexity of errors occurring in current processing and messages in transit, a more comprehensive set of semantics needs to be defined. These are described in Section 6.2.3.

Regarding the processing taking place after a dynamic reconfiguration has finished, there are also two possible behaviors. Here the processing considered is specifically objects trying to use a now disconnected `BUS`, because it is unaware of the reconfiguration change.
Either the existing behavior of VDM-RT can be used, where attempts to perform communication between unconnected `CPU`s results in a run-time error or the disconnect operation can issue a notification to the affected objects to inform them of the changes.
Again the objective of keeping the dynamic reconfiguration extension as open and abstract as possible is the deciding factor. If the latter approach were to be used, specific mechanics for notifying objects is superimposed on all models and thereby all types of distributed systems. Instead it is the task of the System Developer to ensure that any object which may be using the connection is notified prior to the reconfiguration, so that precautious can be made to prevent attempts to communicate following the reconfiguration. Otherwise such a call will result in a run-time error.

As with the `ConnectToBus` operation the `DisconnectFromBus` operation will occur as an atomic reconfiguration to maintain the consistency of the system.

6.2.3 Message Loss

The goal of moving VDM-RT from a static system to a system with dynamic reconfiguration, is to enable the modelling of new types of systems and that systems that were difficult to model before can be modelled in a more natural manner. However the changes can also add to the complexity of the system, as increased functionality also creates more error scenarios to consider.

An apparent error source is the loss of data on the `BUS` connections. Because network relationship can be changed during execution, messages currently on the `BUS` may never reach its receiver. As message loss cannot occur in the existing VDM-RT interpreter, there is no error reporting or default behaviour defined for communication errors. Consequently the behaviour and error reporting occurring when a message cannot be delivered has to be established.

Section 5.4 determined that the best approach for reporting and handling network errors is to use exception handling, a mechanism which exists natively in VDM-RT. However before it can be defining when an exception should be thrown, the criteria for a message being lost must be established.

When a BUS is disconnected from a CPU it can be considered as having one of two different message loss behaviours, which are denoted as *connectionless* and *connection-oriented* behaviour.

The semantics of these behaviours can be illustrated with the initial system in Figure 6.1.

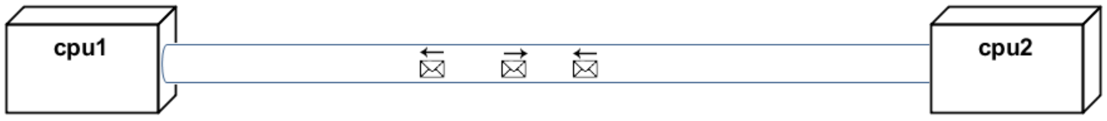


Figure 6.1: BUS connection between two CPUs

Given a dynamic reconfiguration scenario where cpu2 gets disconnected from the BUS, the two message loss behaviours are defined as follows.

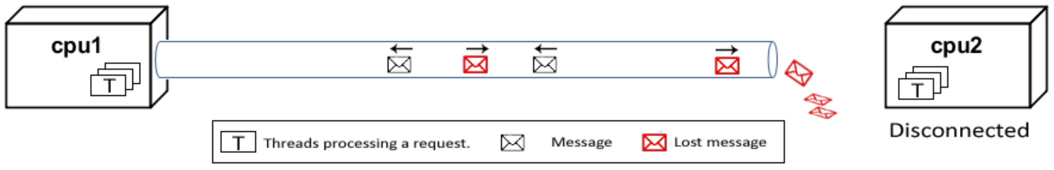


Figure 6.2: Connectionless behaviour

Figure 6.2 illustrates the *connectionless* behaviour. In Figure 6.2 the BUS can be thought of as only being detached at the end of the disconnected CPU, when seen from the perspective of the message travelling on the BUS.
Only considering the BUS as interrupted at one end results in a behaviour where all messages en-route towards the disconnected CPU will be dropped, but all messages being sent from the disconnected that made it on to BUS prior to reconfiguration will still be transmitted.

Connectionless behaviour can be considered as a network type without any reliability technique. Once a message is on the BUS it is considered transmitted.

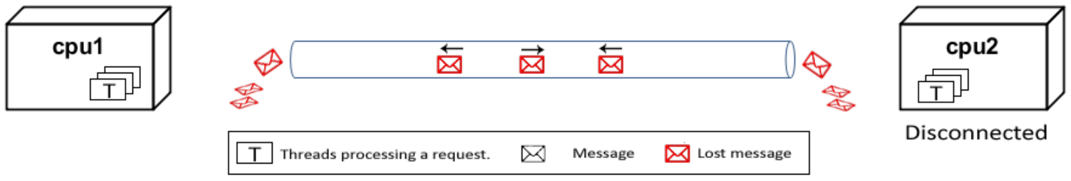


Figure 6.3: Connection-oriented behaviour

Figure 6.3 illustrates the *connection-oriented* behaviour. In Figure 6.3 the `BUS` is detached at both ends meaning that all messages which are on the `BUS` and going *to* or *from* the disconnected `CPU` will be dropped.
When considering the *connection-oriented* behaviour it is important to realize that the disconnection only applies in relation to `cpu2`, meaning that only messages moving *to* or *from* `cpu2` will be lost. So although the `BUS` seems disconnected from `cpu1`, this is only for illustrative purposes to indicate that all message relating to `cpu2` will be dropped no matter their direction. `cpu1` is however still connected to the `BUS` and can continue communication with other `CPUs` as shown in Figure 6.4.

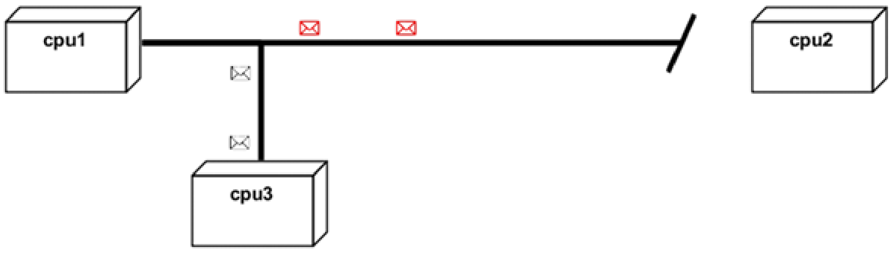


Figure 6.4: Relationship between BUS and CPUs following a disconnect from cpu2

The *connection-oriented* behaviour resembles a connection type in which a technique such as positive acknowledgment is applied to ensure reliability. When the sender or receiver gets disconnected the lack of acknowledgment will cause the message to be dropped. Normally the positive acknowledgment technique includes attempts to retransmit messages, this however is not relevant in relation to the type of message loss discussed in this thesis. Messages cannot be lost due to noise, congestion or signal degradation but only occur when there is no network connection at all.

The two behaviours can roughly be related to the behaviours found in the UDP $^{\mathcal{T}}$ [rfc768] and the TCP $^{\mathcal{T}}$ [rfc793] protocol, which are *connectionless* and a *connection-oriented* respectively.

Figures 6.2 and 6.3 shows each `CPU` containing threads processing requests, this illustrates remote operation invocation request processing at the time of the dynamic reconfiguration change (see Figure 2.10 for details of remote invocation in VDMJ). These are not affected by the two behaviours defined above, because they have no messages on the `BUS` during the reconfiguration. But they will at a later point, following the reconfiguration, attempt to used the BUS for returning response messages with the result of the synchronous requests that were being processed during the reconfiguration. However this BUS connection may no longer be valid because of the reconfiguration.
It was determined in Section 6.2.2 that any attempts to do communication between unconnected `CPUs` should result in a run-time error. However because these requests were initiated prior to the reconfiguration, they will be included as part of the error handling.

With the criteria for message loss determined, the possible scenarios for error reporting can be considered.

The fundamentals of the error reporting are defined as follows:

- With a basis in the communication error handling found in CORBA (see Section 3.5) and

Java RMI (see Section 3.6) a communication exception will always occur on what can be considered as the client side, meaning that the exception will be thrown on the initiator of the request.

- Any message that is lost on the `BUS` will result in an exception being thrown.
- A response resulting from a request that has finished processing after reconfiguration and is going *to* or *from* a disconnected `CPU` will cause an exception.
- Asynchronous calls are not considered in the error handling because; as there is no one waiting for the response, there is nowhere to throw the exception in order to report the error. Consequently asynchronous messages that would cause an exception if they were part of a synchronous call will be dropped silently.

  Consequently if the System Developer wants to have certainty and overview of the communication, synchronous messages must be used, so insight can be assured through either a successful return value or an exception.
- Any messages which are attempted sent over the disconnected `BUS` to or from the disconnected `CPU` following the reconfiguration, will result in a runtime error as there is no `BUS` connection. This is the chosen default behaviour of VDM-RT.

The error reporting for the *connectionless* and *connection-oriented* behaviour is listed in Tables 6.2 and 6.3 respectively.

Type	**From**	**To**	**Result**
Request on Bus	CPU (cpu1)	Disconnected(cpu2)	Exception
Request on Bus	Disconnected (cpu2)	CPU (cpu1)	Ok / Exception
Response on Bus	CPU (cpu1)	Disconnected (cpu2)	Ok
Response on Bus	Disconnected (cpu2)	CPU (cpu1)	Exception
New Request	CPU (cpu1)	Disconnected (cpu2)	Error
New Request	Disconnected (cpu2)	CPU (cpu1)	Error
Processing on Disconnected CPU	CPU (cpu1)	Disconnected (cpu2)	Exception
Processing on other CPU on behalf of Disconnected CPU	Disconnected (cpu2)	CPU (cpu1)	Exception

Table 6.2: Connectionless behaviour during dynamic reconfiguration

Table 6.2 shows the result of combining the *connectionless* behaviour with the rules determined for error reporting. Messages moving away from the disconnected `CPU` will be delivered, while any messages moving towards it will be lost on the `BUS`, and will cause an exception. In keeping with existing VDM-RT, any attempt to make a request to or from a `CPU` over a `BUS` to which it has no connection will result in a run-time error, denoted as a New Request in the table.
This approach is the most complicated to implement, as it requires more knowledge of the current `BUS` state and involves more complex decision making. For instance some requests are allowed to remain on the `BUS`, but their response will cause an exception. An example of this is row two in the table containing the OK/Exception value. This means that the request will be allowed to be transfered on the `BUS`, but there can never be a response.

Type	From	To	Result
Request on Bus	CPU (cpu1)	Disconnected(cpu2)	Exception
Request on Bus	Disconnected (cpu2)	CPU (cpu1)	Exception
Response on Bus	CPU (cpu1)	Disconnected (cpu2)	Exception
Response on Bus	Disconnected (cpu2)	CPU (cpu1)	Exception
New Request	CPU (cpu1)	Disconnected (cpu2)	Error
New Request	Disconnected (cpu2)	CPU (cpu1)	Error
Processing on Disconnected CPU	CPU (cpu1)	Disconnected (cpu2)	Exception
Processing on other CPU on behalf of Disconnected CPU	Disconnected (cpu2)	CPU (cpu1)	Exception

Table 6.3: Connection-oriented behaviour during dynamic reconfiguration

Table 6.3 shows the *connection-oriented* behaviour which is the easiest approach to implement, as all messages on the `BUS` which are related to the *Disconnected CPU* are dropped and an exception is issued.

One more aspect has to be considered for the error handling to be complete. Simply throwing the same exception type on every error carries a risk of leaving the system in an inconsistent state. Because although the request initiator is informed of an error it will still be uncertain if the lost message was a request or response, meaning that the initiator does not know if the receiver actually received and processed the request and thereby potentially changed its state. To enable the System Developer to handle this type of inconsistency, two types of exceptions are defined. One type is to be thrown when a request is lost and another type if a response is lost.
Here the advantage of VDM-RT being a modelling language is utilized, as this behaviour would be difficult to guarantee in a real-life system. For instance although CORBA (see Section 3.5) exceptions do contain a completion status which can indicate the processing completed, it is always permitted to return a maybe status. Nonetheless it is valuable in a model context, where the possibility of indentifying error scenarios is fundamental for obtaining the wanted overview of the modelled system.

Both behaviours have been implemented and tested during the development of the extension to ensure the validity of each behaviour.

As the main focus of this thesis, in relation to the VeMo case study, is to apply the dynamic reconfiguration for modelling wireless communication, the *connection-oriented* behaviour is chosen for the standard implementation. It has a behavior that resembles those found in wireless networks where the transfer medium is unreliable and a reliability technique such as positive acknowledgment would be employed. Please note this reliability technique does not relate to the *connection-oriented* TCP protocol mentioned above, as wireless networks have these techniques implement at the lowest level for efficiency. For instance WLAN 802.11 [IEEE-Std802.11-1999] implements CSMA/CA $^{\mathcal{T}}$ and RTS/CTS $^{\mathcal{T}}$ mechanisms at the MAC $^{\mathcal{T}}$ layer.

In the current VDM-RT any notion of a network protocol has been abstracted away but given the two different behaviours described in this section, it has to be considered if it should be possible for the System Developer to choose which type of network behaviour that the model uses.

6.2.4 Migrating an object from one CPU to another

The `Migrate` operation moves an object from one `CPU` to another `CPU` while preserving its identity and state. When migrating objects the behaviours for the objects references and the objects messages on the `BUS` during migration, should be considered.

6.2.4.1 Object References

When moving an object to a new `CPU`, referential integrity is a concern because all existing object references *to* and *from* the object must be managed. Existing references *from* the *migrated object* to other objects is a dilemma because what is considered to be part of the object and how much should be move? Similarly the existing object references *to* the migrated object should be considered.

If the references *from* the *migrated object* to other objects is considered to begin with, the question is what is considered as part of an object and what should be moved during the migration?

One approach is to only move exactly the *migrating object*, an approach used by [Wegdam&03] as described in Section 5.3. Here migration is performed by moving the object state, and updating the location-independent object reference with the new location. [Wegdam&03] make use of CORBA, in which each CORBA object instance has its own universal object reference IOR (see Section 3.5.1). Now considering a scenario where the *migrating object* has a compositional relationship to, (i.e. created), another CORBA object. Then all operation calls to the compositional object from the *migrating object* will be remote calls. This behavior is not that apparent, as it requires detailed insight into CORBA, and it may cause remote invocation inadvertently. However CORBA has the option of defining objects as valuetypes, meaning that they are not remote enabled and therefore will be moved along with the object state.

To foresee the issue described above another approach can be used which seeks to determine the expected scope of the objects being moved. In such an approach the scope can be defined as all compositional objects of the *migrating object* and all of their transitive references. Meaning that the scope includes all objects created by the *migrating object* (composition), as well as all objects created by the compositional objects, and so on. In VDM-RT all objects can be referenced remotely, and therefore they are all candidates for migration.

An object is defined as all its instance variables as well as its threads. This definition is based on the goal of keeping the changes to VDM-RT language as limited as possible. As a result no state accessors or state translators (see Section 5.3.1) have been forced into the VDM-RT classes. Instead the power of having a model is to be used to access and move the object.

The existing object references to the *migrated object* is not considered an issue, as it is more natural that they become remote references when an object is migrated. This means that existing object references to the *migrated object* will still be valid after the reconfiguration.

6.2.4.2 Messages on the BUS during migration

The target `CPU`, to which the object is migrated, must be connected to the same `BUS` as the source `CPU` from which the object is migrated. This is done to ensure a connection over which the object can be transferred and at the same time it has the benefit that the objects current requests and reponses to other objects will still be able to process over the `BUS`.

Messages *to* or *from* the *migrating object* might be on the BUS at time of the migration. To handle this a message forwarding approach, which is used in most types of migrating systems[Milojicic&00], will be utilized. All messages send prior to reconfiguration will be forwarded to the *migrating object*'s new location in order for the invocation to complete.

6.3. Handling semantics in VDM-RT

This section describes how the semantics defined in Section 6.2 can be conceptually integrated into VDM-RT and VDMJ. The integration of the semantics will not be described in every detail, however details which are considered essential in relation to the VDM-RT deployment model and VDMJ are described.
The general semantics of the dynamic reconfiguration defined that all reconfiguration operations must occur as atomic operations (see Section 6.2). This is actually achieved automatically by the combination of VDMJs threading model (see Section 6.4.2) and the fact that all reconfiguration operations are placed in the **system** class. Because the system cannot be deployed to a CPU, it will automatically be deployed to the virtual CPU (Refer to Section 2.2 for VDM-RT Deployment Model) on which processing appears to take no time. Therefore no other processing will be able to run during reconfiguration.

6.3.1 Changing system topology

The ConnectToBUS operation takes an object as it argument; however as connections in the VDM-RT semantics are made between CPUs the interpreter will have to resolve an object reference to a specific CPU. Internally this means that ConnectToBus will look up the CPU on which the object is deployed and then establish the connection with the BUS supplied in the operation's argument list.
It should be emphasized that the operation only creates a connection from a CPU to a BUS, it is up to the System Developer to establish the reference between objects. This is illustrated in Figure 6.5 where an object reference between *Obj1* and *Obj3* exists, as indicated by the dotted line, but the BUS connection is needed to establish communication.

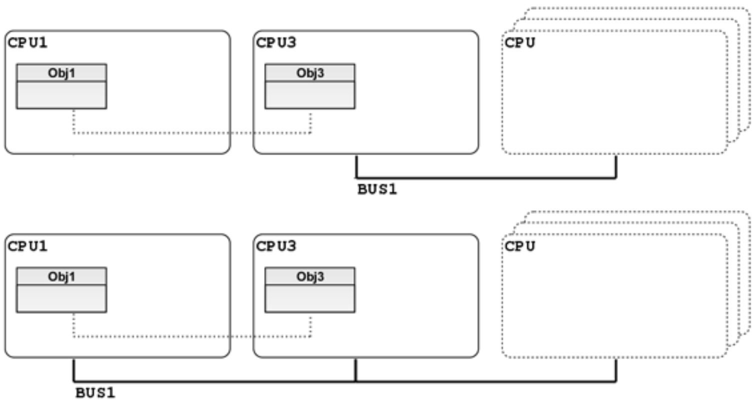


Figure 6.5: Missing BUS connection

Listing 6.1 demonstrates how this connection can be created by using the object reference:

```
--connect obj 1 to BUS 1
Sys'ConnectToBus(obj1, Sys'bus1);
```

Listing 6.1: Create a new connection

Similarly, although the `DisconnectFromBus` operation is called with an object reference, the entire `CPU` the given object is deployed on will be disconnected from the `BUS`. This means that the System Developer must be aware that because connection and disconnection occurs on a `CPU` to `BUS` basis, a system reconfiguration will affect the connectivity of all objects deployed on the `CPU` in question.

The possibility of changing the system topology presents an issue with the current implementation of the system and the use of the *cpumap* to determine connections between CPUs (see Section 2.3.3 System Topology).

Considering an initial system configuration with three `CPU`s and two `BUS` connections having a topology as illustrated in Figure 6.6. The dotted connection is ignored in the initial setup.

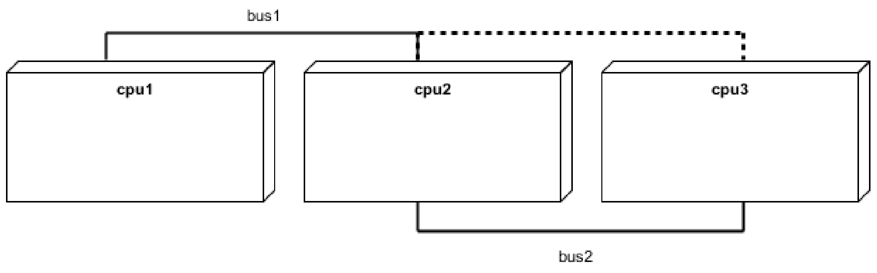


Figure 6.6: System topology with multiple BUS connections.

This configuration will have a cpumap as presented in Table 6.4.

CPU	CPU	BUS
cpu1	cpu2	bus1
cpu1	cpu3	null
cpu2	cpu1	bus1
cpu2	cpu3	bus2
cpu3	cpu1	null
cpu3	cpu2	bus2

Table 6.4: Initial CPU map based on the toplogy in Figure 6.6

Now consider a scenario where `cpu3` needs to communicate with `cpu1` and as a result a dynamic reconfiguration is made which connects `cpu3` to `bus1`, as indicated by the dotted line. This creates a situation where `cpu2` and `cpu3` have two `BUS` connections between them, a situation that cannot be handled with the current implementation of the *cpumap*, as it only allows for one connection.

Since only one connection is needed, an implementation could be made which concludes that since `bus1` connects all of the `CPU`s, the connection to `bus2` is no longer needed. However the logic to determine this in more advanced topologies might be extremely complex and additionally the solution presents a problem as it essentially discard parts of the topology the System Developer has designed. For instance if dynamic reconfiguration once again was used to disconnect `cpu3` from `bus1`, it will result in `cpu3` not being connected to any `BUS` as `bus2` was discarded during the previous reconfiguration.
Instead a solution has been implemented where the *cpumap* contains a list of `BUS` connections, to enable multiple connections between `CPU`s. The list is sorted so that the `BUS` with the highest capacity is used for transmissions when multiple connections exist. This is a subjective decision; other methods could be used such as the load on the BUS.
Accordingly the reconfiguration scenario in Figure 6.6 will have the *cpumap* shown in Table 6.5.

CPU	**CPU**	**BUS**
cpu1	cpu2	bus1
cpu1	cpu3	bus1
cpu2	cpu1	bus1
cpu2	cpu3	bus1, bus2
cpu3	cpu1	bus1
cpu3	cpu2	bus1, bus2

Table 6.5: New CPU map, allowing multiple connections, reflecting the toplogy in Figure 6.6

Both the connect and disconnect reconfiguration is performed by manipulating the *cpumap*, when the respective operations are called.

6.3.2 Error Reporting

In VDMJ a `BUS` is represented by a `BUSValue` and a `BUSResource` where the former is the representation of the VDM-RT `BUS` class and the latter is the active processing part of a `BUS`.
The implementation of `BUSResource` is changed so it can be used for determining message loss and for storing exception occurring during a dynamic reconfiguration.
Because a message lost exception is always thrown at the initiator of the request, the error reporting is implemented such that an exception is thrown instead of the value result in a `MessageResponse`.
A new field has been added to the `MessageResponse` in order to let it carry an `ExitException` which is the Java equivalent of the exception thrown by the VDM-RT **`exit`** statement.

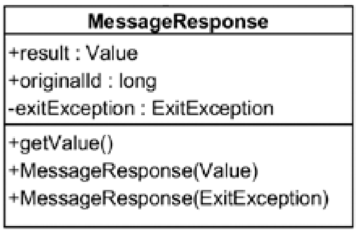


Figure 6.7: Class diagram of Message Response following implemention of the extension

For the error reporting the semantics defines that one type of exception is to be thrown when a response is lost and another type if a request is lost. In the implementation a <*ResponseLost*> exception is thrown if a reponse is lost, otherwise a <*MessageLost*> exception is thrown.

The error reporting of message loss has been implemented by using the existing mechanism for returning the value of a remote invocation, namely the `BUS` and `MessageResponse`.
All `MessageResponses` carrying an exception are placed on the `BUS` queue and thus will be processed as any other message on the BUS. This is done to prevent exceptions from being thrown during reconfiguration. The messages which cause the message lost exceptions are removed from the queue, as they obviously were determined lost.

The implementation requires that although there might not be a valid connection between a `BUS` and a `CPU`, the `BUS` will still be able to deliver messages containing exception. Here the `BUS` should be considered more as a complete representation of a network layer and not just the physical connection.

To handle the semantic rules for request processing during the reconfiguration some extra management is needed. The current version of VDMJ does not keep track of requests currently processing on `CPU`s. So if a dynamic reconfiguration occurs, when a request is being processed on `CPU`, it cannot be seen on the `BUS` and therefore it is not covered by the two message loss behaviours, as explained in Section 6.2.3. This means that any response to a request made prior to the dynamic reconfiguration will be placed on a disconnected `BUS` and it will erroneously be transferred, because the connection checks in VDMJ are only performed on the request.

Aborting the processing threads is not possible as they might be in the middle of processing VDM statements and a termination will leave an unmanageable state. Therefore the threads are left to finish their execution and return their response to the `BUS` as normally, even though there may no longer be a connection. As mention above, this is still possible because of the way the connection check is implemented.

Instead the handling is moved into the processing of the `BUS` message queue. When the `BUS` pulls a `MessageResponse` from the queue it will check if the response should be discarded because its `BUS` has been reconfigured. If a match is found the reponse will be replaced with an exception.

The implementation of the error reporting into VDMJ results in the message loss behaviour shown in Table 6.6.

Type	**To**	**From**	**Result**
Request on Bus	Disconnected(cpu2)	CPU (cpu1)	<MessageLost>
Request on Bus	CPU (cpu1)	Disconnected (cpu2)	<MessageLost>
Response on Bus	Disconnected (cpu2)	CPU (cpu1)	<ResponseLost>
Response on Bus	CPU (cpu1)	Disconnected (cpu2)	<ResponseLost>
New Request	Disconnected (cpu2)	CPU (cpu1)	Error
New Request	CPU (cpu1)	Disconnected (cpu2)	Error
Processing on Disconnected CPU	Disconnected (cpu2)	CPU (cpu1)	<ResponseLost>
Processing on other CPU on behalf of Disconnected CPU	CPU (cpu1)	Disconnected (cpu2)	<ResponseLost>

Table 6.6: Error reporting with final implementation of dynamic reconfiguration

6.3.3 Migration

The `migrate` operation takes an object as the first argument, and the `CPU` to which the object should be migrated as the second argument.
Listing 6.2 shows the migration of *obj1* to *cpu2*, which is illustrated in Figure 6.8.

```
Sys.Migrate(obj1, cpu2);   -- migrate obj 1 to cpu 2
```

Listing 6.2: Migrating an object to another cpu

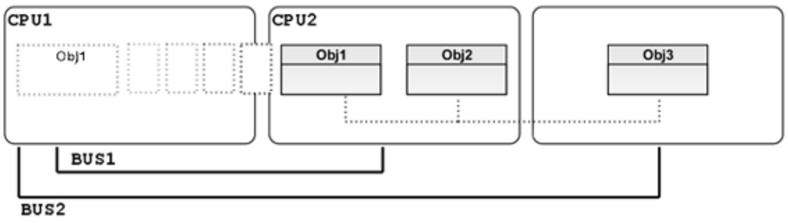


Figure 6.8: Object migration example.

6.3.3.1 Object References

The approach of moving the object, including all of its compositional objects and all of their transitive references, will be used to handle references. To a large extent this is a subjective decision in relation to VDM-RT, in which it is considered to be the best approach to move the compositional objects with their creator.

The approach is implemented by letting every object knowing its parent (creator) and its children (objects it has created). When an object is constructed it will implicitly be attached to its parent set of children and it will keep a reference to the parent. This creates a family tree structure from which the scope of the migration can be derived.
Consider a scenario in which object *A* creates two objects *B* and *C*, and *C* creates object *D*. This is shown as a sequence diagram in the left column of Figure 6.9, while the center column shows the family tree of that scenario. If object *A* is handed to the migrate operation, itself along with all its children will be moved.

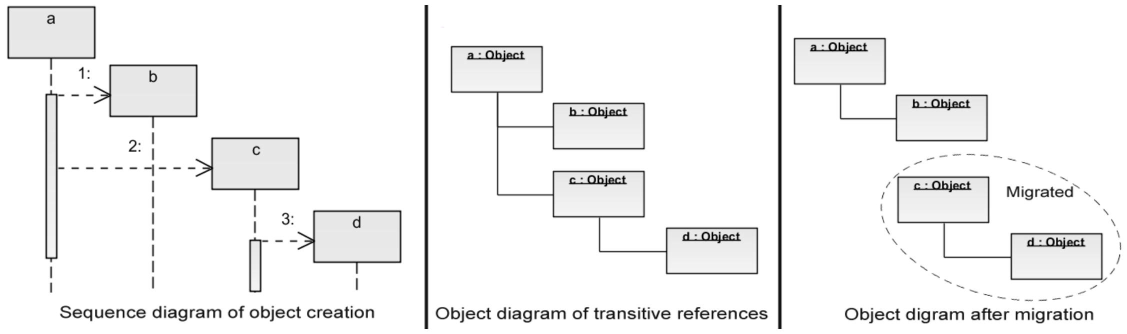

a
1:
b
2:
c
3:
d
Sequence diagram of object creation
a : Object
b : Object
c : Object
d : Object
Object diagram of transitive references
a : Object
b : Object
c : Object
Migrated
d : Object
Object digram after migration

Figure 6.9: Object references family tree.

The reason an object has to know its parent is that once a migration is performed the top most object of the tree (the one given to the `migrate` operation) must be detached from its parents children set, otherwise it will get moved to another `CPU` if its parent is migrated later on.
Consider the same scenario as before, but with object *C* passed to the `migrate` operation instead. Here object *C* and *D* will be moved, and at the same time object *C* is detached from its parent object *A*,as shown in the right column in Figure 6.9.

6.3.3.2 Issue in VDMJ

During the development of the extension an issue was found in the existing VDMJ regarding the deployment of objects on to `CPU`s.
The issue occurs during the initialization of a model and the thread context the creator of the object is currently in. The thread context is important because objects are deployed to the `CPU` on which their constructor is called. During the initialization, and prior to deployment in the **`system`** class, VDMJ initially deploys all objects to the `vCPU` (see Section 2.1). When the constructor of **`system`** class is called the objects are deployed to a `CPU` by changing its `CPU` reference from the `vCPU` to the specific `CPU`. However only the object handed to the `deploy` operation will be set to this `CPU`, any compositional object within the deployed object, as instance variables, is still on the `vCPU`/`vBUS`. Meaning that their execution will take no time and they will be able to do remote operation calls even though there is no `BUS` connection.

To solve this issue the family tree of object references was used, so that when a `deploy` operation is called on a specific `CPU` the tree will be traversed and the `CPU` will be set in every object.
This solution has been adopted into the main branch of VDMJ.

6.3.3.3 Messages

To handle messages moving `to` or `from` the migrating object a message forwarding mechanism has been implemented.

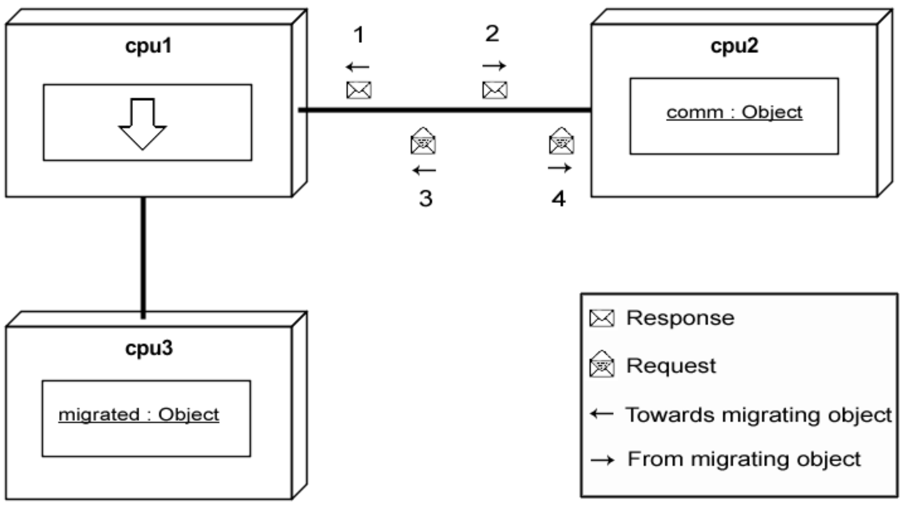


Figure 6.10: Messages on BUS during migration.

Figure 6.10 shows a scenario where an object is migrated from *cpu1* to *cpu3*, while simultaneously having communication with another object on *cpu2*.

The forwarding mechnaism is implemented as follows:

1. A reponse going towards the migrated object is forwarded,
2. A response from the migrating object is left untouched,
3. A request going towards the migrated object is forwarded and so will the response, and
4. A request from the migrated object is left untouched, but the reponse is forwarded.

6.4. Building on the existing VDMJ implementation

When changing the implementation of VDMJ in order to adopt the dynamic reconfiguration extension the existing implementation has obviously been examined and the existing solutions have been built upon. This includes the approach used for adding predefined operations to existing classes and the use of VMDJ's threading model to ensure concurrency.

6.4.1 Extending the language

For the language constructs, identified in Section 6.1, to be adopted into the Overture Project (see Section 2.3.1), a number of adjustments to the VDMJ interpreter are necessary.
VDM-RT already has some predefined operations which are added to existing classes directly in VDMJ, for example the `deploy` operation in the `CPU` class. The operation added in the extension follows the same approach as used for the `deploy` operation.
The approach involves changes to the type checker and the interpreter. Adding a new operation to a VDM-RT defined class involves three steps.

- The type checker needs to be changed to allow for the VDM-RT operation to be accepted as valid in VDM-RT,
- calls to the added VDM-RT operation needs to be coupled to the equivalent Java operation in the interpreter, and
- the implementation of the new operations which performs the required changes needs to be created in an Java implementation in VDMJ.

As the steps above imply the only relation to the VDM-RT dialect is the added operations, whereas the actual implementation and functionality of the new VDM-RT operation is placed at the interpreter level and is written purely in Java.
Because there are no syntactical changes to the VDM-RT dialect, no changes has to be made to the lexical analyzer or syntax analysis mechanisms, as the new operations added will be read like all other operations.
The predefined VDM-RT types and classes are defined in VDMJ by an equivalent Java class, for example a VDM-RT class is defined in the Java `ClassDefinition` class.
The `ClassDefinition` is the basis for all VDM-RT classes and it contains the management of all the universal properties of a class such as accessibility, inheritance, member value and operation definitions, instance invariants as well as type check and object instantiation methods. The operation definitions and the type check are particularly interesting when adding new operations to a predefined class.
As all the new operations introduced by the extension are made available to the System Developer$^{\mathcal{T}}$ through the `System` class, the changes are made in the `SystemDefinition` class which inherits from `ClassDefinition`.
A new operation can be added to the **`system`** class by inputting the operation as a string literal into an VDMJ type object which can then be added to the Abstract Syntax Tree (see Section 2.3.3).

An example of a new operation string literal is shown in Listing 6.3.

```
operations
public static connectToBus: ? * BUS ==> ()
connectToBus(obj, bus) == is not yet specified;
```

Listing 6.3: Definition of CPUs

The "?" in the parameter list is a placeholder for any VDM-RT value and it is used in VDMJ to allow for any object to passed as an argument. This is necessary because VDM-RT does not have a base type for object such as Object in Java.[1] Any argument that has a "?" as its type will be considered as valid by the VDMJ type checker. Therefore all operations which have a "?" in the parameter list must have an additional type check added in order to determine if the supplied arguments are of the required type.
As the **system** class cannot be instantiated, the operations added must be declared static which mean that additional type checks are implemented to allow for static operations in the **system** class.

The process of adding and calling an operation as explained above is shown in the sequence diagram in Figure 6.11.

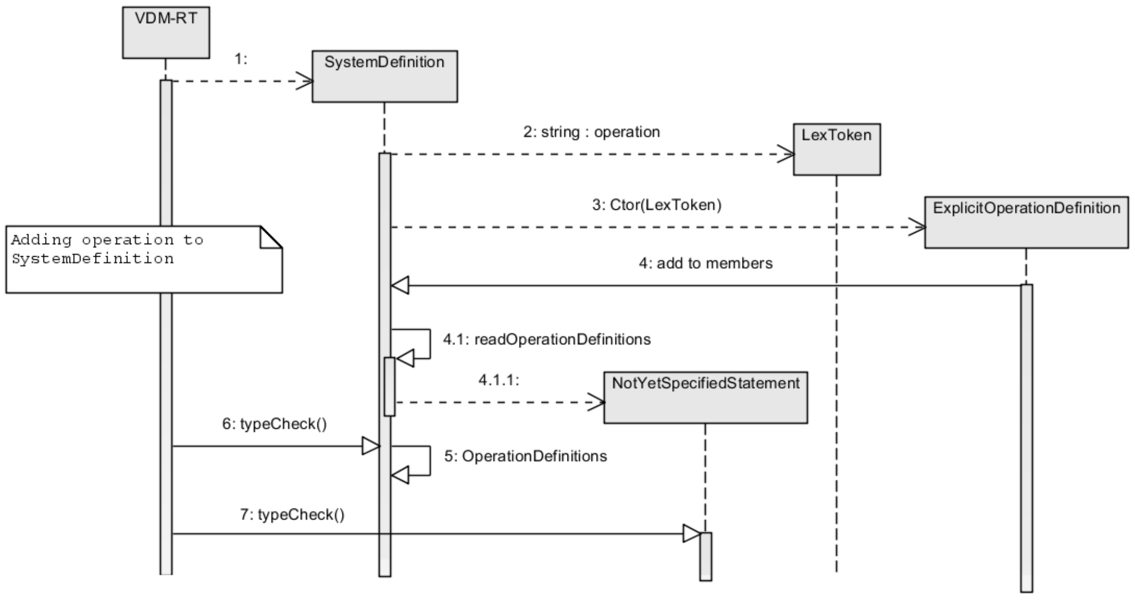


Figure 6.11: Sequence diagram of adding an operation in the System class

[1]Java Technical Documentation : Object http://download.oracle.com/javase/6/docs/api/java/lang/Object.html

6.4.2 Ensuring thread safety

To ensure a deterministic execution and to coordinate time between `CPU`s and the global time, VDMJs threading model has to be built so that the all processing in a model can be controlled precisely with respect to the global time. The threads in VDM-RT are implemented using real Java threads, which entails that the processing of threads, including the duration, is managed by the Java VM$^{\mathcal{T}}$.
This is makes it difficult to enforce VDM-RT concurrency constructs, such as permission predicates and mutexes, and to ensure that threads on various `CPU`s actually receive the same processing time in relation to the VDM-RT global time.
To take control of the Java threads, VDMJ's threading model is built around a VDMJ concept referred to as a resource. Everything requires scheduling is defined as a *resource*, e.g. a `CPU` resource or a `BUS` resource. These resources have a restriction that confine them to only single processing, so a `BUS` can only transmit one message at a time, and each `CPU` can only have one thread running. The request of the messages and threads are placed in queues and can then be swapped as time progresses.
All resources in the model are controlled by a resource rescheduler which functions as a centralized master that determines which resource is allowed to run.
Essential this threading model entails that only one Java thread, that can affect the thread safety of the model, is running at a time.

Thread safety for the dynamic reconfiguration extension is ensured firstly because the reconfiguration operations are executed as any other statement in VDMJ, meaning it will be executed by a resource. Secondly because the reconfiguration only changes values that are defined in the interpreter and not in the model. This means that all the values which may be changed during a reconfiguration are only accessible by the interpreter and cannot be reached by threads defined in the VDM-RT model itself. Combined with VDMJ's threading model this makes it a lot easier to identify and control concurrency concerns.

6.5. Overview of implementation in VDMJ

The implementation of the dynamic reconfiguration extension was made in a branch of the VDMJ interpreter from the Overture Project repository at SourceForge[2].

The implementation of the extension affected 16 existing classes and added one new class (In order handle multiple `BUS` connections between `CPUs`).
Approximately 780 lines of code were added or altered in the 17 classes which contained approximately 4050 lines of code. This means that the percentage of change in the affect classes is about 20 %.

Table 6.7 provides an overview of the changes made to VDMJ.

Class	**Description**
SystemDefinition	Added definition for reconfiguration operations. Changed type check to allow reconfiguration operations in system class. Added implementation of reconfiguration operations.
NotYetSpecifiedStatement	Added reconfiguration methods with loopback to the system class.
BUSResource	Added management of currently processing request. Added message loss and error reporting functionality. Added different exception types. Made it possible to forward messages following a migration.
BusValue	Added functionality to connect and disconnect CPUs from the BUS dynamically. Made it possible to forward messages following a migration. Changed *cpumap* to handle multiple CPUs.
BusConnection	Class added to handle multiple BUS connection between CPUs and sorting of BUS connections based on BUS speed.
ClassDefinition	Management of transitive references added to fix existing deploy issue and to enable migration.
CallStatement	Changed type check to ensure the types of the arguments handed to the added operations.

Table 6.7: Overview of implementation changes made to VDMJ.

[2]Dynamic Reconfiguration branch of VDMJ in the Overture Project: `https://overture.svn.sourceforge.net/svnroot/overture/branches/vdmj_dyn_reconfig`

Class	Description
ObjectValue	Management of transitive references added to fix existing deploy issue and to enable migration.
MessagePacket	The `replyTo` field is moved to the superclass MessagePacket as it existed in both MessageResponse and MessageRequest.
MessageResponse	Changed to allow MessageResponse to carry an ExitException, as part of the error reporting mechanism.
MessageRequest	The `replyTo` field is moved to the superclass MessagePacket.
CPUClassDefinition	Changed the deploy operation to function as redeploy meaning that the object is undeployed on the old CPU and deployed on the new CPU and ensures that transitive references are moved as well.
CPUValue	Added method for removing a deployed object again.
FPPolicy	Made it possible to migrate a thread from one CPU to another.
FCFSPolicy	Made it possible to migrate a thread from one CPU to another.
AsyncThread	Changed implementation of remote operations returning newly initialized objects.
SchedulingPolicy	Made it possible to migrate a thread from one CPU to another.

Table 6.7: Overview of implementation changes made to VDMJ.

Chapter 7

Revisiting the Case Studies

This chapter describes how the extension is integrated into the existing case study models and to what extend the models had to be modified to adopt the dynamic reconfiguration functionality. Secondly the impact and value of having the extension in relation to the case studies is evaluated.

In order to evaluate the effect of the dynamic reconfiguration in VDM-RT, the extension will be applied to the case studies presented in Chapter 4.

7.1. VeMo case study

The VeMo case study (see section 4.1) models a system in which the autonomous movement of vehicles sets the basis for a system that is strongly based on distributed computing and has a rapid change of topology with constant connections and disconnections.
This is a challenge in the existing VDM-RT dialect because of the static network topology, which makes it difficult to construct a model that represents the nature of the system correctly. An executable model of the case study was constructed in existing VDM-RT, however the model was not able to simulate a check of `BUS` and `CPU` connectivity, because of the static network layout used.

With a dynamic reconfigurable model the topology can be changed, meaning that the vehicles can be connected and disconnected as they move around. This allows for the construction of a model that is a closer representation of the real life scenario. Furthermore by using the dynamic reconfiguration functionality the VDM-RT interpreter validation of connectivity between `BUS` and `CPU`s/objects is ensured. This way the correctness of the communication will be evaluated and any communication disregarding the controller will be identified.

7.1.1 Applying the extension

Dynamic Reconfiguration has been applied to the case study by altering two existing classes.

- the `VeMo` class, which is the system class and
- the `VeMoController` class, which keeps track of all vehicles in the model and determines when they are close enough to establish communication.

In the VeMo system class the existing instantiation of the `BUS` connection, shown in Listing 7.1, has been altered so it no longer statically links all of the `CPU`s together during initialization.

```
static bus : BUS := new BUS (<FCFS>,1E6,{cpu0, cpu1, cpu2, cpu3,
                                        cpu4, cpu5,cpu6});
```

Listing 7.1: Original Bus instantiation

Instead the `BUS` is initialized with an empty set of `CPU`s, shown in Section 7.2, which means that the model has an initially unconnected `BUS`, to which the vehicles can be attached as needed. This change is not necessary for the dynamic reconfiguration extension to function, though it is a result of the static topology no longer being required.

```
static bus : BUS := new BUS (<FCFS>,1E6,{});
```

Listing 7.2: New Bus instantiation

Note that the model uses a single BUS to simulate the wireless network, by connecting all vehicles in range to the same BUS. This is an imprecise representation of the wireless network in the VeMo system, because it is not on large joint network but a large number of smaller networks. Consider two groups of vehicles that have established a wireless network within each group, now if the distance between these groups is too great to establish a combined wireless network between all the vehicles, the models use of a single BUS is incorrect. However because of the autonomic movement of the vehicles the VeMo model has a nearly unpredictable number of wireless networks. This is problematic when the number of BUS connections available is static. However, ignoring the possibility of brute force, it is unfeasible to solve this issue in the existing capabilities of VDM-RT.
The `VeMoController` is naturally the subject for the majority of the changes, as this is the class which is responsible for determining when communication is possible and it is thereby the representation of the wireless network.
In the original implementation the `VeMoController` class had a calculation loop that merely needed to determine which vehicles were close enough to each other for it to allow communication. The `VeMoController` did not need to worry about connections or disconnections as communication was always allowed, if they were in range.

This approach is too simple when incorporating the dynamic reconfiguration extension into the model. If the existing calculation loop were to be used as it is, the operations that performs the connect or disconnect reconfiguration would be called each time a set of vehicles are determined to be in or out of range, even though they already are connected or disconnected. Multiple calls to the reconfiguration operations with the same arguments will have no effect, as the interpreter ignores attempts to establish a connection that already exists or connections that do not exist if

the disconnect operation is called. However given that the calculation loop is called each time a vehicle moves it will involve some unnecessary overhead.
Instead the `VeMoController` has been expanded to perform some bookkeeping which keeps track of connections from one vehicle to another. The mapping `controllerConnections` is introduced, as shown in Listing 7.3, which maps a vehicle ID to a set of Controllers. Recall that the `Controller` class is the representation of a vehicle in the model and is responsible for exchanging information with passing vehicles (Controllers). The `controllerConnections` mapping is used to store a history of which vehicles that were in range during the previous run-through of the calculation loop.

```
private controllerConnections : inmap nat to set of
                                Controller := {|->};
```

Listing 7.3: Map added for bookkeeping of connections

The calculation loop has been altered, so that for each vehicle the difference between the set of other vehicles which is currently in range and the set of vehicles which were in range in the previous loop can be used to determine both new connections as well as lost connections. The result of the `difference` evaluations is stored in the `newConnection` set and `lostConnection` set respectively.
An implementation of the altered calculation loop is shown in Listing 7.4

```
for all unit in set units do
(
 let inrange = FindInRangeWithOppositeDirection(unit, units)
  in
  (
 -- find new controllers that have come in range since last check
    let newConnection =
       inrange \controllerConnections(unit.GetVehicleID()) in

 -- find controllers that have gone out of range since last check
    let lostConnection =
        controllerConnections(unit.GetVehicleID()) \ inrange  in

 -- remember inrange for a specific unit, to enable
 --history/bookkeeping of new and lost connections
    controllerConnections(unit.GetVehicleID()) := inrange;
  )
);
```

Listing 7.4: In range calculation loop which keeps track of new and lost connections

The knowledge of new and lost connections is used to determine when the dynamic reconfiguration should take place. Whenever new connections are identified the reconfiguration operation `connectToBus` will be called for each `Controller` in the `newConnection` set, and oppositely the `disconnectFromBus` operation is called for the `lostConnection` set.
The calculation loop with the full implementation of the dynamic reconfiguration is shown in Listing 7.5.

```
for all unit in set units do
(
 let inrange = FindInRangeWithOppositeDirection(unit, units)
  in
  (
  -- find new controllers that have come in range since last
  -- check and create a bus connection.
    let newConnection =
        inrange \controllerConnections(unit.GetVehicleID()) in

    for all conn in set newConnection do
    (
      VeMo`connectToBus(unit, VeMo`bus);
    );

  -- find controllers that have gone out of range since last
  -- check and tear down the bus connection.
    let lostConnection =
        controllerConnections(unit.GetVehicleID()) \ inrange  in

    for all lost in set lostConnection do
    (
      VeMo`disconnectFromBus(unit, VeMo`bus);
    );

  -- remember inrange for a specific unit, to enable
  -- history/bookkeeping of new and lost connections
    controllerConnections(unit.GetVehicleID()) := inrange;

    [...Initiate data communication ...]
    )
);
```

Listing 7.5: Calculation loop with dynamic reconfiguration implementation

7.1.2 Impact on the case study

The effect of the extension can be observed visually by a graphical representation made of the VeMo case study. The representation is an animated reflection of the scenario which is executed in the model and the animation is completely controlled by the VDM-RT model.
The difference between the VeMo model evaluated by the original VDM-RT interpreter and the VDM-RT interpreter with the dynamic reconfiguration extension is shown in Figures 7.1 and 7.2, respectively. The exact same scenario of vehicle movements is executed in the two models and the only difference between the models is the changes made to enable the dynamic reconfiguration, as described in Section 7.1.1.

Both figures show a storyboard which illustrates a scenario where two vehicles pass each other

with a distance that is small enough for them to exchange traffic information. In the scenario the blue car passes a traffic warning to the red vehicle.
The circles indicate the range of the radio communication and the circles has to cover the cars, as in radio communication it is not enough that the fields cover each other if the receiver is not reached. The current connections between the vehicles are indicated by the orange lines.

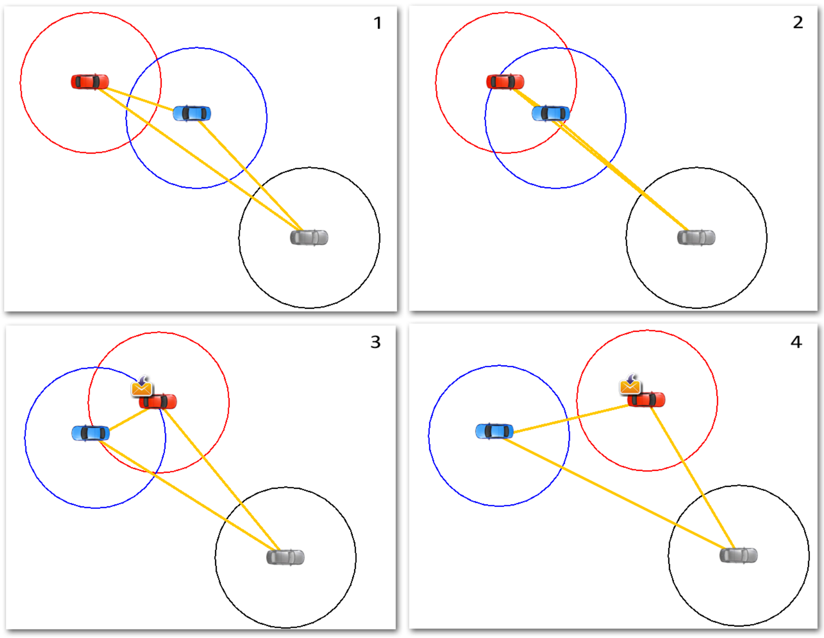


Figure 7.1: Model using the original VDM-RT interpreter with static topology.

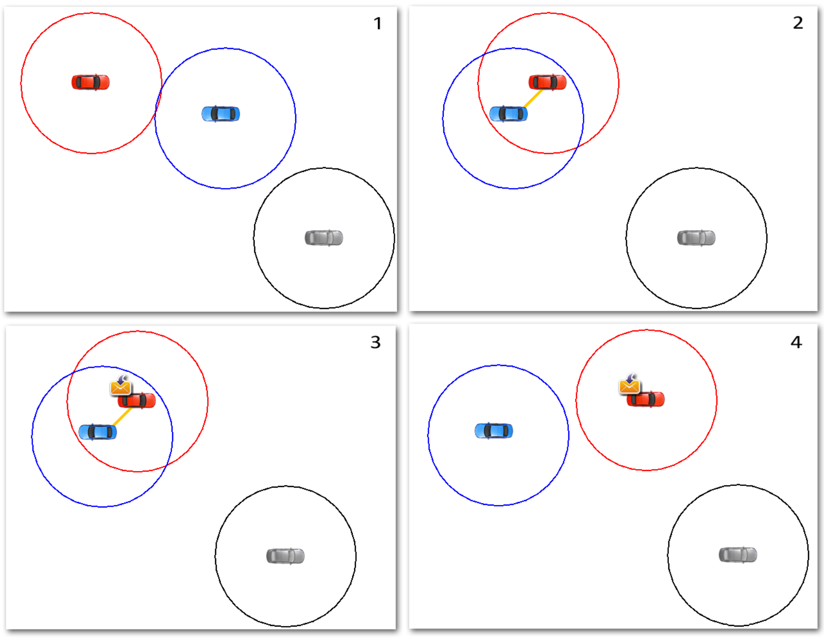


Figure 7.2: Model using the interpreter with the dynamic reconfiguration extension and a dynamic topology.

In the model using the original VDM-RT interpreter (Figure 7.1) it can be seen that the connections between the vehicles remain constant and that movement or proximity of the vehicles does not affect the connections.

When using the interpreter with the dynamic reconfiguration extension (Figure 7.2) the connection only exists between the two vehicles which are in range of each other and only for this short period of time. This reflects the real life scenario the case study seeks to model and it ensures that communication only occur when a connection is possible.
Because the connection is only established when the vehicles are in range of each other the model becomes more precise and a greater confidence in the system is gained.

7.1.3 An error discovered in the VeMo case study

During the analysis of the extensions effect on the VeMo model an error was discovered in the existing realization of the model.

When traffic data is sent from one vehicle to another vehicle, a deep copy is made in order to replicate the data. This is done to ensure that each vehicle has full ownership of the data it receives and that the original object cannot be affected, as would be the case if merely an object reference is sent. However in the existing model the deep copy was performed on the wrong `CPU`, meaning that although the data were replicated and only one reference existed to this data, any data access was not performed locally but actually as a remote call. Recall from Section 2.1 that when an object is created it will be deployed to the `CPU` on which the instantiation was executed. In the existing model the instantiation was made on the `CPU` that was holding the data and not on the `CPU` from which the request for the data occurred.

To create a more sustainable solution to this issue, an alternation to the semantics of making a remote call has been made as well as changes to the VDMJ interpreter. The change in semantics entails that when a remote call invokes an operation that purely returns a new object and does nothing else, the returned object will be deployed on the invoking `CPU`, that made the remote call, and not on the invoked `CPU` where the object is actually created. Because the new object is returned over the `BUS` in the same statement as it is initialized, the object containing the invoked operation can never hold a reference to the new object and therefore it indicates that there should be no relationship to the created object. For that reason it makes good sense to deploy the new object on the invoking `CPU`.

To implement the semantics an extra check has been added to the interpreters implementation of remote operation invocations. When a remote invocation returns an object or a sequence of objects, the AST (see Section 2.3.3) is inspected to see if the returned value is a direct result of a new expression or a set comprehension of new expressions. If this is the case the object or objects will be deployed to the invoking CPU immediately, before processing of the model continues.

7.2. ConPlay

The dynamic reconfiguration extension is applied to the ConPlay case study to examine the impact on the existing model and to evaluate the effect of the having migration.

7.2.1 Applying the extension

Having the migration capability of the extension entails that central parts of the existing model can be removed. Because the entire object is moved automatically, the Memento Design pattern is no longer necessary. Similarly there is no longer a need for a Player object to be deployed to each CPU. With the migrate capability the same object can be moved from one CPU to another CPU directly. The new model is illustrated on Figures 7.3 and 7.4 where the changes in the class diagram and the deployment diagram can be compare to the original model illustrated on Figures 4.6 and 4.7.

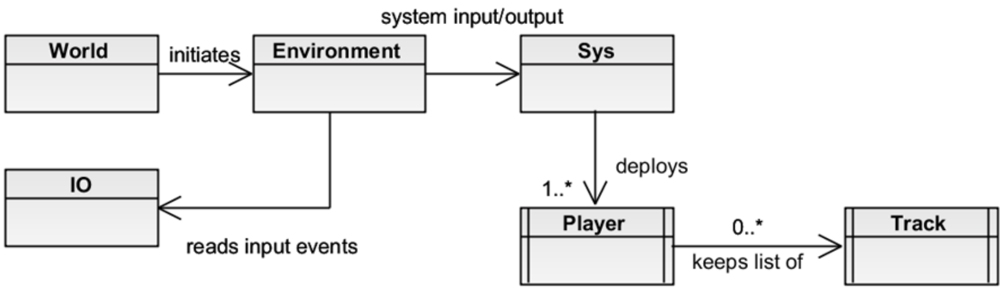


Figure 7.3: Class diagram of the ConPlay model with the dynamic reconfigration extension.

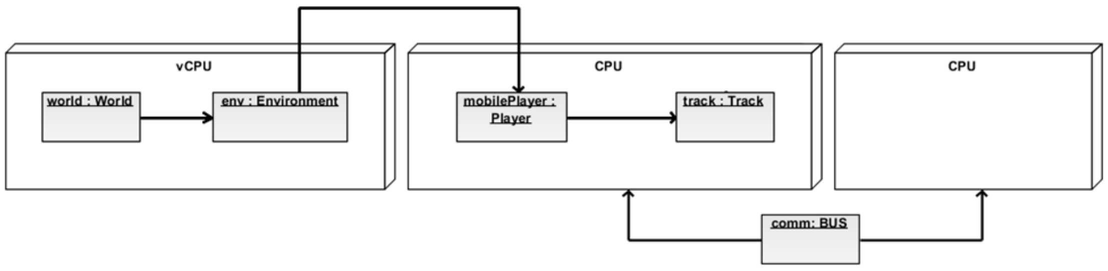
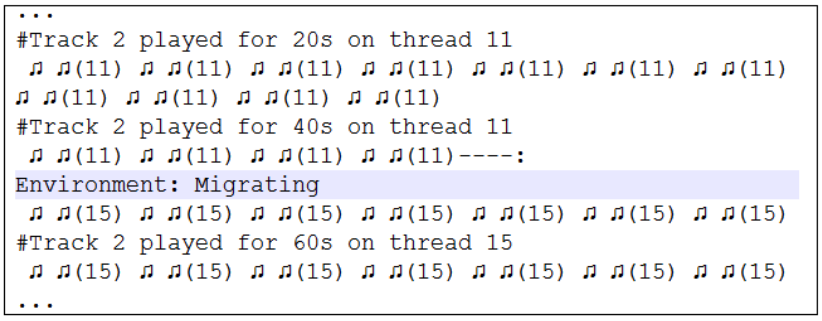


Figure 7.4: Deployment diagram of the ConPlay model with the dynamic reconfigration extension.

When looking at the changes the extension has entailed, it appears as if the model has been simplified. There are fewer classes, less logic and the movement of music seems clear-cut. Conversely the practical experience of using the migration capability was that it was not that simply. Despite knowing the semantics, it was difficult to figure out exactly what was moved and some effort was needed to understand the migrate process in relation to the concrete model. Having the feeling of losing the overview of what exactly happens is unsatisfactory in a modelling language, where the purpose is to gain a better understanding of the system.

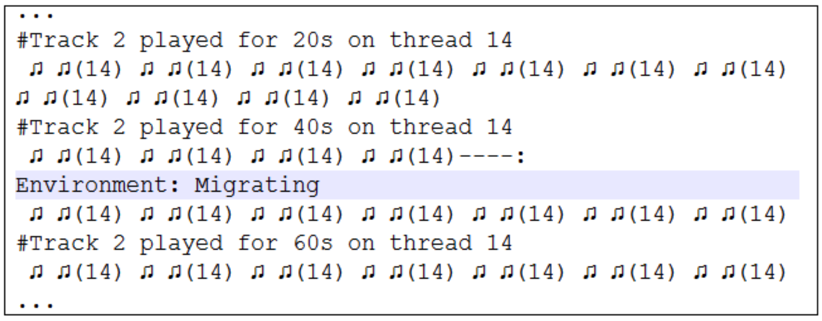
7.2.2 Impact on the case study

The effect of the extension can be observed through the console output of the Conplay model *before* and *after* applying dynamic reconfiguration. The parenthetical numbers indicates the **`threadid`** of the thread processing the music.
The difference between the two models can be seen in Figures 7.5 and 7.6.



```
...
#Track 2 played for 20s on thread 11
 ♫ ♫(11) ♫ ♫(11) ♫ ♫(11) ♫ ♫(11) ♫ ♫(11) ♫ ♫(11) ♫ ♫(11)
♫ ♫(11) ♫ ♫(11) ♫ ♫(11) ♫ ♫(11)
#Track 2 played for 40s on thread 11
 ♫ ♫(11) ♫ ♫(11) ♫ ♫(11) ♫ ♫(11)----:
Environment: Migrating
 ♫ ♫(15) ♫ ♫(15) ♫ ♫(15) ♫ ♫(15) ♫ ♫(15) ♫ ♫(15) ♫ ♫(15)
#Track 2 played for 60s on thread 15
 ♫ ♫(15) ♫ ♫(15) ♫ ♫(15) ♫ ♫(15) ♫ ♫(15) ♫ ♫(15) ♫ ♫(15)
...
```

Figure 7.5: Output of the original ConPlay model.



```
...
#Track 2 played for 20s on thread 14
 ♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14)
♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14)
#Track 2 played for 40s on thread 14
 ♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14)----:
Environment: Migrating
 ♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14)
#Track 2 played for 60s on thread 14
 ♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14) ♫ ♫(14)
...
```

Figure 7.6: Output of the ConPlay model with the dynamic reconfigration extension.

In Figure 7.5 the **`threadid`** changes after the migration. While the **`threadid`** remains the same in Figure 7.6, because the entire object, including the thread, has been migrated to a new CPU. This is the behavior defined in the semantics. However when assessing the results achieved with the case study in relation to the real implementation of the modeled system, this behavior might be misguiding. Since the movement of an object and thread cannot occur in a real implementation. In reality an approach where the state is serialized and moved, like originally used in the case study, would be employed in a real system.

It should be noted that this particular case study does not enable an evaluation of the message forwarding defined in the semantics for the migrate capability (see Section 6.3.3.3).

Chapter 8

Concluding Remarks and Future Work

This chapter concludes the thesis.
Firstly an overview over the achievements of the performed work is presented in Section 8.1. The extension and the value of having dynamic reconfiguration in VDM-RT is assessed in Section 8.2. Suggestions for future development is supplied in Section 8.3, before the chapter and thesis is concluded with the final remarks in Section 8.4.

8.1. Achieved results

In this thesis the possibilities of introducing dynamic reconfiguration into VDM-RT has been examined.
The achievements of the performed work is:

- Existing literature of dynamic reconfigurable distributed systems has been evaluated in relation to VDM-RT;
- A dynamic reconfiguration language extension and its semantics has been designed with minimal changes to the language;
- The extension has been incorporated into VDM-RT in a prototype implementation of VDMJ; and
- The value of having dynamic reconfiguration has been evaluated on two case studies.

The achievements are described in more detail in the following paragraphs:

Existing literature of dynamic reconfigurable distributed systems has been evaluated in relation to VDM-RT
In Chapter 5 existing approaches in dynamic reconfiguration has been examined prior to the VDM-RT extenions and their semantics.
The approaches were evaluated with regard to the two reconfiguration capabilities considered in this thesis, namely *changing network topology* and *migration*. The evaluation included the language constructs and semantics used to perform the reconfiguration, as well as the mechanisms used to ensure the integrity and consistency of the system following a reconfiguration.

By applying the language constructs and methodologies of the different approaches to the VeMo case study (presented in Section 4.1) an insight into which elements were applicable for VDM-RT was obtained. This established the foundation for the dynamic reconfiguration extension to VDM-RT.

A dynamic reconfiguration langauge extension and its semantics has been designed with minimal changes to the language

Based on the evaluation of existing approaches in Chapter 5 and the theory on dynamic reconfiguration in Chapter 3 a language extension has been created for dynamic reconfiguration in VDM-RT.

A conservative language extension has been created which does not introduce any new classes or language keywords into VDM-RT, as the new reconfiguration operations are added to the existing predefined `system` class. In combination with the existing VDM-RT semantics this means that any risk of ambiguity and keyword conflicts with existing VDM models is prevented.

The power of abstraction was used to establish the semantic rule that; reconfigurations will take no time in reference to the global time of the model. This utilize the strength of having a modelling language, as the action of performing reconfiguration changes is simplified. Not having to relate to time makes it possible to perform a reconfiguration change without the necessity of doing transaction management or elaborate implementations to ensure the persistency of the system during reconfiguration. When validating a model in relation to a real implementation the System Developer needs to be aware of the reconfiguration time being abstracted away, as this cannot be achieved in a real implementation.

Two types of reconfiguration operations are supported in the extension, explicitly *Changing Topology* and *Migration*. Different types of semantics for these operations has been evaluated, and a choice was made by considering the effect the semantics would have on the case studies.

To support the semantics of these operations the risk of message loss has been introduced into VDM-RT. To assist the error handling of message loss the language semantics and interpreter has been altered to support remote exceptions.

The extension has been incoporated into VDM-RT in a prototype implementation of VDMJ

The language extension and its semantics have been successfully implemented into the Overture interpreter VDMJ. The implementation builds on top of the existing approach used in VDMJ to implement predefined classes and operations. The implementation makes it possible to create running VDM-RT models that utilize the dynamic reconfiguration extension.

During the implementation of the *migration* operation an issue was discovered with the existing deployment mechanism in VDMJ, meaning that objects did not get deployed as expected (see 6.3.3.2). A proposal for fixing the error was created, and the solution has now been incorporated into the main branch of VDMJ.

The value of having dynamic reconfiguration has been evaluated on two case studies

Two case studies have been modelled in the existing VDM-RT dialect, to show how the case study systems could be designed without dynamic reconfiguration. This has established the possibility of getting a “before” and an “after” picture in relation to the use of the dynamic reconfiguration extension.

For the primary case study, the VeMo system, a graphical animation was created in Java which visualizes the modelled scenarios. The animation is a reflection of the actions occurring in the model and it is completely controlled by the model via VDM-RT operations.

The case studies were altered to utilize the possibilities of the new extension and the functioning prototype of VDMJ was used to execute the new models.

The results from this exercise can be divided into the two reconfiguration operations:

Changing Topology: The extension was fairly easy applied to the VeMo case study and it supplied the desired result; a closer representation of the real life scenario. The extension provided the solution to a problem which could not be solved satisfactorily in existing VDM-RT.

The introduction of dynamic reconfiguration actually revealed an error in the existing model. Some objects turned out to be deployed differently than expected, resulting in some remote invocations occurring inadvertently. This resulted in a change to the VDM-RT semantics and implementation of remote operations returning newly initialized objects.

Migration: The changes to the ConPlay case study was more extensive because the extension made it possible to simulate the same system with a simpler implementation.

Having the *migrate* capability makes it easier to model systems that needs this functionality, however it does not provide a functionality which cannot already be modeled in existing VDM-RT.

8.2. Assessment of dynamic reconfiguration in VDM-RT

With a basis in the achieved results, this thesis has demonstrated that it is possible to integrate a dynamic reconfiguration into VDM-RT which enable existing models to achieve closer representation of the real life scenario.

The dynamic reconfiguration is controlled through a centralized entity; but a dedicated network or reconfiguration manager as in used the related work of [Jaghoori08] or [Wegdam&03] has not been introduced. Instead all reconfiguration has been implemented in the **`system`** class, which resembles the approach used in [Johnsen&08]. This has the advanced of no new concepts being introduced into VDM-RT, that a System Developer^{T} has to recognize and comprehend, instead the dynamic reconfiguration is controlled through a well known predefined class.

Contrasting both of the formal methods presented in the related work (see Chapter 5) the language extension has deliberately been kept conservative by not introducing new language keywords or concepts for describing the dynamic reconfiguration. This has the benefit of existing models being unaffected and it enables the adoption of the dynamic reconfiguration capabilities into to the existing model with minor changes.

Because the reconfiguration capabilities are accessed through static operations in the **`system`** class, the reconfiguration can be initiated from the entire model. This differs from the approach of [Johnsen&08] where reconfiguration can only be defined within the **`system`** class, but it also has the weakness of reconfiguration becoming more unmanageable. A combination of these approaches is used by [Jaghoori08] where reconfiguration can be initiated from the entire model, but the result of an initiation is controlled by a centralized policy. This line of attack has not been introduced into VDM-RT as it would require substantial changes to the methodology of the VDM-RT **`system`** class.

Because reconfiguration is an intrusive process, the possibility of *message loss* has been introduced as well as *error reporting* for handling message loss. The *error reporting* is based on remote exceptions which is the same approach used in middleware technologies, such as presented in Sections 3.5.3 and 3.6.3. A similar approach is used in [Johnsen&08] where the return messages type can be an *Error*, however this has to be handled where the operation returns and it can not propagate like exceptions.
Using exceptions for error reporting of errors seem rational as this is a commonly used approach and it already exists as a primitive in the existing VDM-RT notation. The interpreter had to be changed to allow for remote exceptions, but remote exception can only be raised by the interpreter in connection with reconfiguration and not from within model defined objects. This is done to keep the impact and changes to the language to a minimum.

The advantage of the entire system being modelled is utilized to stop the time during a reconfiguration, thereby allowing the reconfiguration to occur in no time. This has the advantage of simplifying the reconfiguration mechanism, both in relation to the implementation but also from the perspective of the System Developer. Considerations regarding transactions, interleaving and mutual consistency is circumvented. This cannot be achieved in a real implementation, but in a modelling context it is considered as an acceptable abstraction, which can be tolerated as long as the System Developer is aware. An approach used in both of the formal methods explored in Sections 5.1 and 5.2.
The capability of dynamically *changing the topology* of the system architecture has been added to VDM-RT language while still respecting the methodology of existing VDM-RT. Adding more capabilities to a modeling language, with the intent of getting a better representation of the real implementation, has the risk of making the language too complicated and too focused on concrete implementations. However given that the system architecture, e.g. BUS and CPU relations, is already specified in the **`system`** class by the System Developer, the additional capability of changing the network topology is not considered to dilute the abstraction level already choosen for VDM-RT.
As shown through the VeMo case, in Section 7.1.2, the possibilities supplied by making the topology dynamic has enabled a more precise representation of dynamic systems and the modelling of certain network types such as wireless networks.

In the *migrate* capability the power of the entire system being modelled is utilized, as migration moves the exact same object, including running threads, from one CPU to another while preserving the execution state of the thread. This makes it easy to *migrate* objects, but this behaviour may be too abstract to be valuable, as this cannot occur in real implementations. In most implementation of migration the object is suspended and serialized, and manually moved across the network before being reinitialized. This approach is used in [Wegdam&03], as described in Section 5.3.
Another issue with having the *migrate* capability is that it is very difficult to generalize exactly what should be migrated. Is it only the object itself; is it all compositional objects or all referenced objects? And what about threads? This is handled through the semantics defined for the capability, but from a System Developer perspective, and own experience, it can be established that it is very difficult to determine exactly what will occur when the *migrate* functionality is used. Through the

results obtained by revisiting the ConPlay case study in Section 7.2, it can be determined that the utilization of the migrate operation did not benefit the model.
With these issues in mind it must be concluded that the *migrate* capability, as proposed in this thesis, does not have justification as it in the best case provided no additional functionality and in the worse case created more confusion.

A most interesting result made during the thesis process was the discovery of a distinction between the types of dynamic reconfiguration that exist in VDM-RT. There is a difference in the *reconfiguration of the system architecture* and the *reconfiguration of objects*. *Reconfiguration of objects* is capabilities such as *migration*, while *reconfiguration of the system architecture* is changes made to `CPU`s and `BUS` connections, such as *changing topology*.
The distinction is that *reconfiguration of objects* involves capabilities, that to a great extend can be modelled using in existing VDM-RT. Essentially the capability does not require a reconfiguration extension, only a little modelling effort. Oppositely *reconfiguration of the system architecture* involves changes to the internals of the **`system`** class; changes that cannot be performed in existing VDM-RT regardless of the effort. This not only includes topology changes but also the creation and removal of `CPU` and `BUS` objects on the fly.
Consequently the biggest effects of the dynamic reconfiguration extension is actually the possibilities of doing reconfiguration of (simulated) hardware elements.

8.3. Future Work

Based on the impressions and insight gained through the thesis work, some possibilities for further development of are suggested in the follow subsections.

8.3.1 Dynamic creation of CPU and BUS instances

The VeMo case study revealed a challenge when modeling systems that have a large number of units with autonomous behavior, that use wireless connections to communicate (see Section 7.1.1). They need an unpredictable number of network connections in order to have a correct representation of the simulated wireless networks which may be established. This is problematic when the number of `BUS` connections available is static. There are two possible solutions, both assuming a dynamic reconfiguration capability:
The *first* is creating a pool of `BUS` connections from which a `BUS` can be borrowed to simulate a wireless network and then returned when there is no longer any in range of the network. However this involves the risk of exhausting the pool.
The *second* solution would be to add the dynamic creation of busses in VDM-RT. as this would enable a more realistic representation of how wireless networks operate.

If `BUS` connections could be dynamically added and removed via the **`system`** class it would be possible to model types of adaptive systems that currently are difficult to model in VDM-RT. The dynamic creation/removal could be expanded to include `CPU`s as well, which would make it possible to model the effect of adaptively adding more processing power or simulated failover scenarios where the processing power of a `CPU` is lost.

8.3.2 Network/Bus protocol type

As described in Section 6.2.3 the type of network protocol has an impact on the behaviour of the system during reconfiguration. The choice between a *connectionless* and a *connection-oriented* network protocol is based on the functionality and requirements of the modelled system. To allow the modelling of a larger variety of systems it could be beneficial to enable the possibility of choosing which kind of protocol a `BUS` should use.

In making this option available in VDM-RT the level of abstraction supplied by the language should be considered. Choosing the protocol can be considered as a technology bound decision and the choice may complicate the use of the modeling language to an extend where it becomes to concrete and too difficult to use while still maintaining the abstract representation.

8.3.3 Loss on BUS

To enable a closer representation of a real world, the `BUS` class could be expanded to allow a property of disturbance which can cause message loss. The message loss could be handled by the message loss exception handling introduced in this thesis. Thereby enabling the modeling of the system reaction to an apparent random message loss, which however remains deterministic from an interpreter perspective. This would make it possible to model the imperfections of reality; while still maintaining the level of abstraction supplied by the model. This is a behavior which is available in [Johnsen&08], as described in Section 5.1.

Again the abstraction level is important to take into account, as message loss is a property which may be closer bound to the implementation of a system, than to the functionality. Therefore message loss is a property that is abstracted away in current VDM-RT. Still for modeling certain types of systems this option

8.3.4 BUS Status

The introduction of dynamic reconfiguration increases the need for knowledge of the current activity an object has on a `BUS`. Because a change in network topology or an object migration can result in the loss of message data, currently in transmission, on a `BUS`, it could be considered enabling the System Developer$^{\mathcal{T}}$ to specify a certain `BUS` status before the reconfiguration occurs. This would make it possible to model systems in which seamless data transfer is the most important. It would however have the price of losing the predictability over the reconfiguration time. In some cases the reconfiguration may never occur. This utilizes the approach of detecting safe-state as described in Section 3.3.2.

It could be implemented by overloading the existing reconfiguration operations with another parameter. An enumerated type representing the current activity the given object has on the `BUS`, can be used as a guard that must be fulfilled before the reconfiguration takes place.
The enumeration is defined as follows:

Idle:
The object has no communication on the *BUS*.

ProcessingRequest:
The object is currently processing an operation invocation by a remote object.

WaitingForReply:
The object is waiting on a reply from an operation invocation on a remote object.

RequestAndReply:
The object is both processing a request and waiting for a reply.

The different indications supplied by the enumeration will allow the System Developer to use different strategies for reconfiguration, based on the current state of the object.

8.4. Final Remarks

In this thesis a dynamic reconfiguration extension to VDM-RT has been proposed, implemented and evaluated through two case studies. This has enabled the possibility of modeling dynamic systems and to create models which has a closer representation of the real system.

An evaluation on the accomplishment of the thesis subgoals follows:

1. **To evaluated the feasibility and value of extending VDM-RT with dynamic reconfiguration in order to facilitate the modeling of dynamic distributed systems.**

A dynamic reconfiguration extension for VDM-RT has been created and implemented in a prototype of the VDMJ interpreter. The extension has been applied to the case studies and the models have been executed successfully using the VDMJ prototype.
The effect on the case studies has shown that it is indeed possible to model dynamic distributed systems in VDM-RT with the added dynamic reconfiguration capabilities. Particularly the possibility of changing the network topology has been demonstrated to be advantageous in improving the modeling of system with volatile networks. Thus it must be concluded that this subgoal has been achived.

2. **To expand the language in such a way that the changes to the existing VDM-RT language and methodology is kept to a minimum.**

A language extension has been made that enable dynamic reconfiguration in VDM-RT, with minimal changes to the existing language. No new language keywords or elements have been added and the extension is based on existing classes and is in keeping with the existing methodology. Thus it must be concluded that this subgoal has been achived.

3. **To ensure that the dynamic reconfiguration extension maintains an appropriate relationship between the modeling level and the implementation level, such that dynamic reconfiguration models can be used for validation of subsequent implementations**

The case studies used to evaluate the extension has not been implemented in a real system, therefore is difficult to assess the fulfillment of this goal. However the extension has been based on existing approaches found in both middleware technology as well as in other formal modeling languages. The provides for a sound foundation for the dynamic reconfiguration modeling level in relation to the implementation level. Here it must be concluded that this subgoal has partially been achieved since it has not been possible to make a real implementation of the VeMo case study for practical reasons.

Based on the findings above, the goals of the thesis are considered accomplished. Especially for the VeMo case study, the effect of having dynamic reconfiguration was excellent, as it not only improved the quality of the model but also reveal an issue that would have been difficult to identify in the old model.

In retrospect it would have been more beneficial for the extension to included the dynamic creating of CPUs and BUS connections, instead of the migration capability. Nonetheless the inclusion of the migration functionality revealed that its value in a VDM-RT context is limited.

This thesis has successfully approached and opened a new capability of VDM-RT which has not previously been explored. Hopefully it can inspire to an increased research and development effort in the area and in VDM in general.

References

[Almeida&01] João Paulo A. Almeida and Maarten Wegdam and Luìs Ferreira Pires and Marten Van Sinderen. An approach to dynamic reconfiguration of distributed systems based on object-middleware. In *Proceedings of the 19th Brazilian Symposium on Computer Networks (SBRC 2001)*, pages 246–274, Springer-Verlag LNCS 417, 2001. [cited at p. x, 25]

[Arbab06] F. Arbab. Coordination for Component Composition. *Electronic Notes in Theoretical Computer Science*, 160(1):15–40, August 2006. . [cited at p. 6, 46]

[Avvenuti&01] Avvenuti, M. and Vecchio, A. Embedding Remote Object Mobility in Java RMI. In *Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 98–104, IEEE Computer Society, September 1997. ISBN 0-7695-1384-0. [cited at p. 29]

[Bates98] John Bates. *The State of the Art in Distributed and Dependable Computing*. Technical Report, Laboratory for Communications Engineering, University of Cambridge, 1998. 71 pages. [cited at p. 21]

[Battle09] Nick Battle. *VDMJ User Guide*. Technical Report, Fujitsu Services Ltd., UK, 2009. [cited at p. 14, 17]

[Benini&02] L. Benini and G. De Micheli. Networks on chips: A new SoC paradigm. *Computer*, 35(1):70–78, January 2002. [cited at p. 1]

[Bjørner00] Dines Bjørner. Pinnacles of software engineering: 25 years of formal methods. *Annals of Software Engineering*, 10:11–66, 2000. [cited at p. 9]

[Clarke&96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996. [cited at p. 9]

[Dijkstra75] Edger W. Dijkstra. Guarded Commands, Nondeterminancy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, August 1975. 5 pages. [cited at p. 42]Our major interest in guarded commands and CSP in this connection is restricted to the fact that the semantics was developed /Bafter the proof rules were found. This inspired us to think that the same idea could be applied to the BSI/VDM.

[Elmstrøm&94] René Elmstrøm and Peter Gorm Larsen and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994. 4 pages. [cited at p. 13]

[Fich&04] Faith Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 121–163, February 2004. 43 pages. [cited at p. 1]

[Fitzgerald&05] John Fitzgerald and Peter Gorm Larsen and Paul Mukherjee and Nico Plat and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005. [cited at p. 10]

[Fitzgerald&07d] J. S. Fitzgerald and P. G. Larsen and S. Tjell and M. Verhoef. *Validation Support for Real-Time Embedded Systems in VDM++*. Technical Report CS-TR-1017, School of Computing Science, Newcastle University, April 2007. 18 pages. Revised version in Proc. 10th IEEE High Assurance Systems Engineering Symposium, November, 2007, Dallas, Texas, IEEE. [cited at p. 10]

[Fitzgerald&08a] J. S. Fitzgerald and P. G. Larsen and M. Verhoef. Vienna Development Method. *Wiley Encyclopedia of Computer Science and Engineering*, 2008. 11 pages. edited by Benjamin Wah, John Wiley & Sons, Inc. [cited at p. 9]

[Fitzgerald&08b] John Fitzgerald and Peter Gorm Larsen and Shin Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2):3–11, February 2008. 8 pages. [cited at p. 9, 13]

[Fitzgerald&09] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edition, 2009. ISBN 0-521-62348-0. [cited at p. 10]

[Floyd67] R.W. Floyd. Assigning Meanings to Programs. In *the Symposium on Applied Mathematics*, pages 19–32, American Mathematical Society, 1967. [cited at p. 9]

[Fowler&03] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2003. 185 pages. [cited at p. 50]

[Gamma&95] E.Gamma, R.Helm, R.Johnson and J.Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Volume of *Addison-Wesley Professional Computing Series*, Addison-Wesley Publishing Company, edition, 1995. 395 pages. . [cited at p. 39, 56]

[Ganek&03] Alan G. Ganek and Thomas A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003. [cited at p. 2]

[Goudarzi&96] Goudarzi, K.M. and Kramer, J. Maintaining node consistency in the face of dynamic change. In *Configurable Distributed Systems*, pages 62 –69, 1996. 8 pages. [cited at p. 22]

[Goudarzi99] K. Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed systems*. PhD thesis, Imperial College London, 1999. . [cited at p. 23]

[Hooman&10] J. Hooman and M. Verhoef. Formal Semantics of a VDM Extension for Distributed Embedded Systems. In *de Roever Festschrift*, pages 142–161, LNCS 5930, Springer-Verlag, 2010. [cited at p. 10]

[IEEE-Std802.11-1999] The Institute of Electrical and Electronics Engingeers, Inc. *IEEE Standards for Information Technology Specifications. Telecommunications and Information Exchange between Systems – Local and Metropolitan Area Network – Specific Requirements. Part 11*. Technical Report, IEEE, NY, USA, 1999. 90 pages. . [cited at p. 70]

[ISOVDM96short] P. G. Larsen and B. S. Hansen and others. 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. [cited at p. 9]

[Jaghoori08] M. M. Jaghoori. Coordinating object oriented components using data-flow networks. In *6th International Symposium on Formal Methods for Components and Objects FMCO'2007*, pages 280–311, Springer-Verlag, Springer, Berlin Heidelberg, October 2008. [cited at p. 5, 46, 48, 60, 61, 95, 96]

[JavaRMI] Java RemoteMethodInvocation Specification 1.5.0. http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf, 2004. [cited at p. 29]

[Johnsen&07] Einar Broch Johnsen and Olaf Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and Systems Modeling*, 6(1):39–58, March 2007. [cited at p. 5, 42, 46]

[Johnsen&08] E. B. Johnsen and O. Owe and J. Bjørk and M. Kyas. An object-oriented component model for heterogeneous nets. In *6th International Symposium on Formal Methods for Components and Objects FMCO'2007*, pages 257–279, Springer-Verlag, Springer, Berlin Heidelberg, October 2008. [cited at p. 5, 6, 25, 42, 44, 49, 60, 61, 95, 96, 98]

[Jones99] Cliff B. Jones. Scientific Decisions which Characterize VDM. In J.M. Wing and J.C.P. Woodcock and J. Davies, editors, *FM'99 - Formal Methods*, pages 28–47, Springer-Verlag, 1999. Lecture Notes in Computer Science 1708. [cited at p. 9]

[Kramer07] Jeff Kramer. Is Abstraction the Key to Computing? *Communications of the ACM*, 50(4):37–42, 2007. [cited at p. 2]

[Kramer&85] Jeff Kramer and Jeff Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, April 1985. 13 pages. [cited at p. 2, 21]

[Larsen&10a] Peter Gorm Larsen, Sune Wolff, Nick Battle, John Fitzgerald and Ken Pierce. *Development Process of Distributed Embedded Systems using VDM*. Technical Report TR-2010-02, The Overture Open Source Initiative, April 2010. [cited at p. 13]

[Larsen&10b] Peter Gorm Larsen and Nick Battle and Miguel Ferreira and John Fitzgerald and Kenneth Lausdahl and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *ACM Software Engineering Notes*, 35(1):, January 2010. 6 pages. [cited at p. 13, 14]

[Milojicic&00] Milojíčić, Dejan S. and Douglis, Fred and Paindaveine, Yves and Wheeler, Richard and Zhou, Songnian. Process migration. *ACM Comput. Surv.*, 32:241–299, September 2000. 59 pages. [cited at p. 22, 72]

[OMG&04] *Common Object Request Broker: Naming Service Specification*. OMG, October 2004. [cited at p. 28, 56]

[OMG&96] *The Common Object Request Broker: Architecture and Specification*. OMG, July 1996. [cited at p. 28, 55]

[Payne08] Payne, Richard J. RPL: a policy language for dynamic reconfiguration. In *Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*, pages 73–78, ACM, New York, NY, USA, 2008. [cited at p. 61]

[Pellegrini99] Pellegrini, N.-C. Dynamic reconfiguration of Corba-based applications. In *Technology of Object-Oriented Languages and Systems, 1999. Proceedings of*, pages 329–340, July 1999. [cited at p. 28]

[Plat&92] Nico Plat and Peter Gorm Larsen. An Overview of the ISO/VDM-SL Standard. *Sigplan Notices*, 27(8):76–82, August 1992. 7 pages. [cited at p. 9]

[rfc768] J. Postel. *User Datagram Protocol*. RFC 768, Internet Engineering Task Force, August 1980. [cited at p. 68]

[rfc793] J. Postel. *Transmission Control Protocol*. RFC 793, Internet Engineering Task Force, September 1981. 85 pages. [cited at p. 68]

[Satyanarayanan01] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001. 7 pages. [cited at p. 1]

[Sørensen&07] Rasmus Ask Sørensen and Jasper Moltke Nygaard. *Evaluating Distributed Architectures using VDM++ Real-Time Modelling with a Proof of Concept Implementation*. Master's thesis, Enginering College of Aarhus, December 2007. [cited at p. 10]

[Varaiya93] Pravin Varaiya. Smart cars on smart roads. Problems of control. *IEEE Transactions on Automatic Control*, 38(2):195–207, February 1993. 12 pages. [cited at p. 32]

[VDMTools] CSK. VDMTools homepage. *http://www.vdmtools.jp/en/*, 2007. pages. . [cited at p. 13]

[Verhoef05] Marcel Verhoef. On the Use of VDM++ for Specifying Real-Time Systems. *Proc. First Overture workshop*, November 2005. [cited at p. 10]

[Verhoef&06] Marcel Verhoef and Peter Gorm Larsen and Jozef Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Jayadev Misra and Tobias Nipkow and Emil Sekerinski, editors, *FM 2006: Formal Methods*, pages 147–162, Lecture Notes in Computer Science 4085, 2006. [cited at p. 10, 50]

[Verhoef&07] Marcel Verhoef and Peter Gorm Larsen. Interpreting Distributed System Architectures Using VDM++ – A Case Study. In Brian Sauser and Gerrit Muller, editors, *5th Annual Conference on Systems Engineering Research*, March 2007. [cited at p. 10]

[Verhoef08] Marcel Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2008. ISBN 978-90-9023705-3. [cited at p. 2, 10, 64]

[Wegdam&03] Maarten Wegdam and João Paulo A. Almeida and Marten J. Sinderen van and Lambert J.M. Nieuwenhuis. *Dynamic Reconfiguration for Middleware-Based Applications*. Technical Report, 2003. 30 pages.

[cited at p. 6, 26, 55, 60, 61, 65, 71, 95, 96]

[Wegdam03b] M. Wegdam. *Dynamic Reconfiguration and Load Distribution in Component Middleware*. PhD thesis, University of Twente, 2003. ISBN 90-75176-36-8. [cited at p. 23, 28]

[Weiser91] Mark Weiser. The Computer for the Twenty-First Century. *Scientific American*, 265(3):10, September 1991. 94-104 pages. [cited at p. 2]

[Willig&05] Andreas Willig and Kirsten Matheus and Adam Wolisz. Wireless Technology in Industrial Networks. In *Proceedings of the IEEE*, pages 1130–1151, IEEE, 2005. . [cited at p. 26]

[Wolff08] Sune Wolff. *Universal Multiprotocol Home Automation Framework*. Master's thesis, Aarhus University/Engineering College of Aarhus, December 2008. 127 pages. [cited at p. 10]

[Woodcock&09] Woodcock, Jim and Larsen, Peter Gorm and Bicarregui, Juan and Fitzgerald, John. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, October 2009. [cited at p. 9]

Appendix A

Terminology

^{T}**CSMA/CA**

Carrier Sense Multiple Access/Collision Avoidance, is a protocol for carrier transmission where wireless transmission of a node is prevented if another node is transmitting.

^{T}**CSMACD**

Carrier Sense Multiple Access Collision Detection, is a protocol for carrier transmission where the idle state of the communication line is checked before the line is used. If a collision is still detected, the transmitter backs off for a random periode of time before retransmitting.

^{T}**DBGp**

ugger Protocol, a simple protocol for use with language tools for the purpose of debugging applications.

^{T}**DTO**

Data Transfer Object, is value object which is used to transfer data between objects or subsystems.

^{T}**FCFS**

First Come First Serv, is a service policy where requests are handled in the order that they arrive.

^{T}**FP**

Fixed Priority, a scheduling policy where the scheduling is determined by a priority given to specific operations.

^{T}**IDE**

Integrated development environment denotes an application suite for software development.

^{T}**JVM**

Java Virtual Machine, Java run-time environment.

^{T}**MAC**

Media Access Control layer, 802.11 layer implementing Carrier Sense Multiple Access/-Collision Avoidance

^{T}**QoS**
: Quality of Service, assurance as to the value a service. Normally QoS identifies potential issues with system processing and may handle the issue by adapting or reconfiguring the system.

^{T}**RTS/CTS**
: Request to Send / Clear to Send is a mechanism in 802.11 reducing frame collisions by coordinating communication between nodes.

^{T}**TCP**
: Transmission Control Protocol, is connection-oriented reliable network protocol, which is reliable and guarantees message order.

^{T}**SystemsDeveloper**
: Denotes a developer that uses VDM-RT to model a distributed system.

^{T}**UDP**
: User Datagram Protocol is an connectionless unreliable network protocol which does not guarantee ordering, reliability or data integrity.

^{T}**URI**
: Uniform Resource Identifier, is string of characters that can identify a name or a resource on the Internet.

Appendix B

Original VeMo Model

1 Config Class

```
-----------------------------------------------
-- Class:   Config
-- Description:  Config contains configuration values
-----------------------------------------------

--
-- class definition
--
class Config

--
-- instance variables
--
instance variables
--
-- Types definition section
--
types

--
-- Operations definition section
--
operations

--
-- Functions definition section
--
functions

--
-- Values definition section
```

```
--
values
--indicates the range in which units in the system can see each
--other
public static Range : nat = 12;
--indicates the periode for which a TrafficData Message is valid
public static TrafficDataLifeTime : nat = 5000;
--indicates the number of TrafficData Message held by the vdm
--units
public static TrafficDataKeeptNumber : nat = 5;
--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
```

Function or operation	Coverage	Calls
Config.vdmrt	100.0%	0

2 Controller Class

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


--
-- class definition
--
class Controller

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

--keep track of whom we have communicated with.
private communicatedWith : seq of nat := [];
inv len communicatedWith <= Config`TrafficDataKeeptNumber;

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 cant 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];

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

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

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


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())
      | i in set inds internalTrafficData ];
```

```


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

Function or operation	Coverage	Calls
AddInternalTrafficData	100.0%	32
AddOncomingVehicle	100.0%	18
AddTrafficData	79.0%	18
Controller	100.0%	14
EnvironmentReady	100.0%	153
GetDirection	100.0%	363
GetPosition	100.0%	800
GetTrafficData	100.0%	18
GetVehicleID	100.0%	347
Step	39.0%	154
getVehicle	100.0%	5
getVehicleDTO	100.0%	18
run	100.0%	3
Controller.vdmrt	69.0%	1943

3 Environment Class

```
-----------------------------------------------
-- 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`VechicleRun(-,-) ->
   (
    if event.t <= curtime
    then
    (
         Printer`OutWithTS("Environment: Start Vehicle event "
               ^ Printer`natToString(event.ID));
      let ctrl = vemoCtrl.getController(event.ID) in
      (
      ctrl.run();
      );
         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
  --duration(10)
  (
   Events();
--   Printer`OutWithTS("Environment done");
   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
```

Function or operation	Coverage	Calls
Environment	100.0%	2
Events	52.0%	52
goEnvironment	100.0%	51
handleEvent	100.0%	1
isFinished	100.0%	1
report	100.0%	1
run	100.0%	1
setVeMoCtrl	100.0%	1
Environment.vdmrt	64.0%	110

4 Position Class

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

Function or operation	Coverage	Calls
Compare	0.0%	0
Position	100.0%	67
X	100.0%	530
Y	100.0%	548
deepCopy	100.0%	52
inRange	100.0%	306
setX	100.0%	68
setY	100.0%	86
toString	100.0%	155
Position.vdmrt	87.0%	1812

5 Printer Class

```
-----------------------------------------------
-- 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) ==
  (
   let last =
   cases n mod 10:
   0-> "0",
   1-> "1",
   2-> "2",
   3-> "3",
   4-> "4",
   5-> "5",
   6-> "6",
   7-> "7",
   8-> "8",
   9-> "9"
   end

    in
    if n < 10
    then return last
    else return (natToString(n div 10) ^ last)

  );

  public static intToString : nat ==> seq of char
  intToString(i) ==
  (
   dcl s : seq of char := "";

   if i < 0
   then
   s := "-";

   return s ^ natToString(i);
  );

sync
 mutex(OutWithTS)

end Printer
```

Function or operation	Coverage	Calls
Echo	0.0%	0
OutAlways	100.0%	10
OutWithTS	100.0%	167
intToString	0.0%	0
natToString	100.0%	1107
Printer.vdmrt	76.0%	1284

6 Traffic Class

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

Function or operation	Coverage	Calls
AddVehicle	70.0%	2
AverageSpeed	0.0%	0
Congestion	72.0%	153
ExistVehicle	0.0%	0
ExistVehicleData	100.0%	18
FindInRangeWithSameDirection	47.0%	72
GetVehicles	0.0%	0
InRange	0.0%	0
Traffic.vdmrt	49.0%	245

7 TrafficData Class

```
-----------------------------------------------
-- 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;

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

--
-- Operations definition section
--
operations
public TrafficData: MessageType * Position * Types`Direction ==>
  TrafficData
  TrafficData(m,p,d) ==
  (
  pos := p ;
  message := m;
  dir := d;
  timeToLive := time + Config`TrafficDataLifeTime;
  );

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

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

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

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`natToString(timeToLive - time);

--
-- 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
```

Function or operation	Coverage	Calls
Expired	100.0%	150
GetDirection	100.0%	10
GetMessage	100.0%	10
GetPosition	100.0%	10
MessageTypeToString	50.0%	1
ToString	100.0%	1
TrafficData	100.0%	42
TrafficData.vdmrt	94.0%	224

8 TrafficLight Class

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

--
-- class definition
```

```
--
class TrafficLight

--
-- instance variables
--
instance variables

private pos: Position;
private greenLightTime : nat1;
private greenDir: Types`Direction;
private id : nat;
--
-- Types definition section
--
types

--
-- Operations definition section
--
operations

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

 greenDir := <NORTH>
);

public AddTrafficData: TrafficData ==> ()
AddTrafficData(data) ==
is not yet specified;

public GetTrafficData: () ==> set of TrafficData
GetTrafficData() ==
is not yet specified;

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

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

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

```
GetID() ==
 return id;

private Step: () ==> ()
Step() ==
(
    if (time mod greenLightTime) = 0
    then
     (
  greenDir := CrossDirection(greenDir);
    )
);

--
-- 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
 periodic (1000,10,900,0) (Step)


--
-- sync definition section
--
sync
mutex(GreenLightPath);
mutex(Step,GreenLightPath);

end TrafficLight
```

Function or operation	Coverage	Calls
AddTrafficData	100.0%	2
CrossDirection	0.0%	0
GetID	0.0%	0
GetPosition	0.0%	0
GetTrafficData	100.0%	2
GreenLightPath	0.0%	0
Step	0.0%	0
TrafficLight	100.0%	1
TrafficLight.vdmrt	47.0%	5

9 Types

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

--
-- class definition
--
class Types

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

public Event = VechicleRun | TrafficLightRun | VehicleUpdateSpeed
      | VehicleUpdatePosition | VehicleUpdateDirection
      | VehicleLowGrip | VehicleTurnIndication | WasteTime;

public VechicleRun ::
        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
```

Function or operation	Coverage	Calls
DirectionToGraphics	100.0%	156
DirectionToString	100.0%	159
Types.vdmrt	100.0%	315

10 VeMo System

```

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


--
-- class definition
--
system VeMo

--
-- instance variables
--
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  cpu4 : CPU := new CPU (<FCFS>,1E6);
public  cpu5 : CPU := new CPU (<FCFS>,1E6);
public  cpu6 : CPU := new CPU (<FCFS>,1E6);
public  cpu7 : CPU := new CPU (<FCFS>,1E6);
public  cpu8 : CPU := new CPU (<FCFS>,1E6);
public  cpu9 : CPU := new CPU (<FCFS>,1E6);
public  cpu10 : CPU := new CPU (<FCFS>,1E6);
public  cpu11 : CPU := new CPU (<FCFS>,1E6);
public  cpu12 : CPU := new CPU (<FCFS>,1E6);
public  cpu13 : CPU := new CPU (<FCFS>,1E6);
public  cpu14 : CPU := new CPU (<FCFS>,1E6);

--public static bus : BUS := new BUS (<FCFS>,1E6,{cpu1});
public bus1 : BUS := new BUS (<FCFS>,1E6,{cpu0,cpu1, cpu2, cpu3,
                    cpu4, cpu5, cpu6,cpu7, cpu8, cpu9, cpu10,
                    cpu11, cpu12, cpu13, cpu14});

static e : Environment := World`env;

```

```
-- Vehicles
public static ctrl1 : Controller := new Controller(
         new Vehicle(1,
         new Position(1, -20), 1, <NORTH>));

public static ctrl2 : Controller := new Controller(
         new Vehicle(2,
         new Position(-20, 25), 3, <SOUTH>));

public static ctrl3 : Controller := new Controller(
         new Vehicle(3,
         new Position(1, 1), 1, <NORTH>));

public static ctrl4 : Controller := new Controller(
         new Vehicle(4,
         new Position(1, 1), 1, <NORTH>));

public static ctrl5 : Controller := new Controller(
         new Vehicle(5,
         new Position(1, 1), 1, <NORTH>));

public static ctrl6 : Controller := new Controller(
         new Vehicle(6,
         new Position(1, 1), 1, <NORTH>));

public static ctrl7 : Controller := new Controller(
         new Vehicle(7,
         new Position(1, 1), 1, <NORTH>));

public static ctrl8 : Controller := new Controller(
         new Vehicle(8,
         new Position(1, 1), 1, <NORTH>));

public static ctrl9 : Controller := new Controller(
         new Vehicle(9,
         new Position(6, 20), 1, <SOUTH>));

public static ctrl10 : Controller := new Controller(
         new Vehicle(10,
         new Position(1, 10), 1, <SOUTH>));

public static ctrl11 : Controller := new Controller(
         new Vehicle(11,
         new Position(1, 1), 1, <NORTH>));

public static ctrl12 : Controller := new Controller(
         new Vehicle(12,
         new Position(7, 5), 1, <WEST>));

```

```
public static ctrl13 : Controller := new Controller(
         new Vehicle(13,
         new Position(12, 5), 1, <WEST>));

public static ctrl14 : Controller := new Controller(
         new Vehicle(14,
         new Position(14, 5), 1, <WEST>));


--traffic lights
public static tl1 : TrafficLight := new TrafficLight(20
             ,new Position(1, 1)
             , 100);

public static vemoCtrl : VeMoController := new VeMoController();

--
-- Operations definition section
--
operations

public VeMo: () ==> VeMo
 VeMo() ==
 (
 cpu1.deploy(ctrl1);
 cpu2.deploy(ctrl2);
 --cpu3.deploy(ctrl3);
 --cpu4.deploy(ctrl4);
 --cpu5.deploy(ctrl5);
 --cpu6.deploy(ctrl6);
 --cpu7.deploy(ctrl7);
 --cpu8.deploy(ctrl8);
 cpu9.deploy(ctrl9);
 --cpu10.deploy(ctrl10);
 --cpu11.deploy(ctrl11);
 --cpu12.deploy(ctrl12);
 --cpu13.deploy(ctrl13);
 --cpu14.deploy(ctrl14);

 --cpu0.deploy(vemoCtrl);
 --cpu0.setPriority(VeMoController`getController,500);
 --cpu0.setPriority(VeMoController`CalculateInRange,100);
 --cpu0.deploy(e);


 );


end VeMo
```

Function or operation	Coverage	Calls
VeMo	100.0%	1
VeMo.vdmrt	100.0%	1

11 VeMoController Class

```
-----------------------------------------------
-- 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).GetVehicleID() = id;
inv forall id in set dom lights
    & lights(id).GetID() = id;

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

static env : Environment := World`env;

--
-- Types definition section
--
types

--
-- Operations definition section
--
operations
```

```

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

public addController: Controller ==> ()
addController(ctrl) ==
(
 ctrlUnits := ctrlUnits munion {ctrl.GetVehicleID() |->  ctrl} ;

    let vecID = ctrl.GetVehicleID() 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)
                                );

       for all unit in set rng ctrlUnits do
       graphics.connectVehicles(unit.GetVehicleID() ,vecID);
    )
)
pre ctrl.GetVehicleID() not in
    set (dom ctrlUnits union dom lights);

public addTrafficLight: TrafficLight ==> ()
addTrafficLight(light) ==
(
    lights := lights munion {light.GetID() |-> light};
)
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 = rng ctrlUnits 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 pos = unit.GetPosition() in
           graphics.updatePosition(unit.GetVehicleID()
                                    ,pos.X(), pos.Y());

     let dir = unit.GetDirection() in graphics.updateDirection(
                                 unit.GetVehicleID(),
                                 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.GetVehicleID()
              ,oncomingVehicle.GetTrafficData());

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

    graphics.sleep();

 -- synchronization, when we have calculated inrange allow
 -- vehicles to move again
    for all u in set rng ctrlUnits 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 : Controller * set of Controller ->
                                   set of Controller
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 : Controller * Controller -> 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 : Controller *
       set of Controller -> set of Controller
FindInRangeWithOppositeDirection(u ,us) ==
 let dir = OppositeDirection(u.GetDirection()) in
  let inrange = { ir | ir in set FindInRange(u, us)
```

```
                 & dir = ir.GetDirection()}
    in inrange;

--
-- Values definition section
--
values


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

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


-- has heavy performance loss
mutex (CalculateInRange, addController, getController);
mutex (addController);
mutex (getController);
mutex (CalculateInRange)
end VeMoController
```

Function or operation	Coverage	Calls
CalculateInRange	100.0%	51
EnvironmentReady	100.0%	52
FindInRange	100.0%	153
FindInRangeWithOppositeDirection	100.0%	153
InRange	100.0%	306
OppositeDirection	100.0%	153
VeMoController	100.0%	1
addController	100.0%	3
addTrafficLight	0.0%	0

getController	100.0%	8
getTrafficLight	0.0%	0
VeMoController.vdmrt	84.0%	880

12 Vehicle Class

```
-----------------------------------------------
-- 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.deepCopy();
return pos;

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

public GetID: () ==> nat
GetID() ==
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() ==
(
let vd = new VehicleData(id, pos.deepCopy(), speed, dir, lowgrip)
in return vd;
)

--
-- Functions definition section
--
functions

public static IndicatorToString : Indicator -> seq of char
IndicatorToString(i) ==
(
cases i:
<LEFT>-> "LEFT",
<RIGHT>-> "RIGHT",
<NONE>-> "NONE"
end
)



--
-- Values definition section
--
values

--
-- 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
```

Function or operation	Coverage	Calls
GetDirection	100.0%	435
GetID	100.0%	363
GetPosition	100.0%	800
GetSpeed	100.0%	32
IndicatorToString	0.0%	0
Move	100.0%	154
SetDirection	100.0%	4
SetPosition	0.0%	0
SetSpeed	0.0%	0
TurnIndicator	100.0%	153
Vehicle	100.0%	2
getDTO	100.0%	18

getLowGrip	100.0%	153
setLowGrip	100.0%	1
setTurnIndicator	0.0%	0
Vehicle.vdmrt	88.0%	2115

13 VehicleData Class

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

Function or operation	Coverage	Calls
GetDirection	100.0%	2
GetID	100.0%	18
GetPosition	100.0%	2

GetSpeed	100.0%	2
TurnIndicator	0.0%	0
VehicleData	100.0%	18
getLowGrip	100.0%	2
VehicleData.vdmrt	92.0%	44

14 World Class

```
-----------------------------------------------
-- 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");

 --vehicle
 VeMo`vemoCtrl.addController(VeMo`ctrl1);
 VeMo`vemoCtrl.addController(VeMo`ctrl2);
 --VeMo`vemoCtrl.addController(VeMo`ctrl3);
 --VeMo`vemoCtrl.addController(VeMo`ctrl4);
 --VeMo`vemoCtrl.addController(VeMo`ctrl5);
 --VeMo`vemoCtrl.addController(VeMo`ctrl6);
 --VeMo`vemoCtrl.addController(VeMo`ctrl7);
 --VeMo`vemoCtrl.addController(VeMo`ctrl8);
 VeMo`vemoCtrl.addController(VeMo`ctrl9);
 --VeMo`vemoCtrl.addController(VeMo`ctrl10);
 --VeMo`vemoCtrl.addController(VeMo`ctrl11);
 --VeMo`vemoCtrl.addController(VeMo`ctrl12);
 --VeMo`vemoCtrl.addController(VeMo`ctrl13);
 --VeMo`vemoCtrl.addController(VeMo`ctrl14);


-- 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
```

Function or operation	Coverage	Calls

Run	100.0%	1
Verbose	0.0%	0
World	100.0%	1
World.vdmrt	92.0%	2

15 guiGraphics

```
class gui_Graphics

 instance variables
-- TODO Define instance variables here
 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 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
```

Function or operation	Coverage	Calls
addVehicle	100.0%	5
connectVehicles	100.0%	8
disconnectVehicles	100.0%	2
init	100.0%	3

receivedMessage	100.0%	3
sleep	100.0%	53
updateDirection	100.0%	158
updatePosition	100.0%	158
gui_Graphics.vdmrt	100.0%	390

Appendix C

Dynamic VeMo Model

Only the classes which have been altered when applying the dynamic reconfiguration extension is included. The dynamic reconfiguration operations are marked with blue.

1 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).GetVehicleID() = 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 : inmap nat to
                                set of Controller := {|->};


static env : Environment := World`env;

--
-- Types definition section
--
types

--
-- Operations definition section
--
operations

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

public addController: Controller ==> ()
addController(ctrl) ==
(
 ctrlUnits := ctrlUnits munion {ctrl.GetVehicleID() |->  ctrl} ;
 controllerConnections(ctrl.GetVehicleID()) := {};

    let vecID = ctrl.GetVehicleID() 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.GetVehicleID() not in set
                       (dom ctrlUnits union dom lights);

public addTrafficLight: TrafficLight ==> ()
addTrafficLight(light) ==
(
  lights := lights munion {light.GetID() |-> light};
)
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 = rng ctrlUnits in
    for all unit in set units do
    (
     let pos = unit.GetPosition() in
       graphics.updatePosition(unit.GetVehicleID()
                              ,pos.X(), pos.Y());

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

     let inrange = FindInRangeWithOppositeDirection(unit, units)
     in
     (
       -- find new controllers that has come in range since last
       -- check and create a bus connection.
       let newConnection =
       inrange \controllerConnections(unit.GetVehicleID()) in
       for all conn in set newConnection do
       (
         VeMo`connectToBus(unit, VeMo`bus);
         graphics.connectVehicles(unit.GetVehicleID()
                                 ,conn.GetVehicleID());
         Printer`OutWithTS
         (
          "+" ^ Printer`natToString(conn.GetVehicleID())
         );
       );

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

      -- 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.GetVehicleID()
                            ,oncomingVehicle);

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

    graphics.sleep();

 -- synchronization, when we have calculated inrange allow
 -- vehicles to move again
 for all u in set rng ctrlUnits 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 : Controller * set of Controller
                                 -> set of Controller
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 : Controller * Controller -> 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 : Controller
                                        * set of Controller
                                        -> set of Controller
FindInRangeWithOppositeDirection(u ,us) ==
 let dir = OppositeDirection(u.GetDirection()) in
  let inrange = { ir | ir in set FindInRange(u, us)
             & dir = ir.GetDirection()}
   in inrange;

--
-- Values definition section
--
values


--
```

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

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


-- has heavy performance loss
mutex (CalculateInRange, addController, getController);
mutex (addController);
mutex (getController);
mutex (CalculateInRange)
end VeMoController
```

Appendix D

ConPlay Model

1 Environment

```
-----------------------------------------------
-- Class:   Environment
-- Description:
-----------------------------------------------

--
-- class definition
--
class Environment

--
-- instance variables
--
instance variables

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

--
-- Types definition section
--
types
InputTP   = seq of inline;
inline  = Types`Event;
--
-- 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 doEvents: () ==> ()
doEvents() ==
(
   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`play(-) ->
       (
         if event.t <= curtime
         then
         (
            Printer`OutWithTS("Environment: Playing");
            start(Sys`mobPlayer);
           eventOccurred := true;
         )
       ),
       mk_Types`migrate(-) ->
       (
         if event.t <= curtime
         then
         (
           let memento = Sys`mobPlayer.Migrate() in
            Sys`homePlayer.Reinitialize(memento);

            Printer`OutWithTS("\nEnvironment: Migrating");
            eventOccurred := true;
```

```
          )
        ),
        mk_Types`WasteTime(-) ->
        (
         if event.t <= curtime
         then
          (
            Printer`OutWithTS("\nEnvironment: Wasting time");
            eventOccurred := true;
          )
        ),
        others ->
        Printer`OutWithTS("\nEnvironment:  No match found")
       end;

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

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

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

--
-- Values definition section
--
values


--
```

```
-- Threads definition section
--
thread
(
 while busy do
  (
    duration(1000)
    doEvents();
  );

)
--
-- sync definition section
--
sync
 per isFinished => not busy;

end Environment
```

Function or operation	Coverage	Calls
Environment	100.0%	1
doEvents	96.0%	102
isFinished	100.0%	1
run	100.0%	1
Environment.vdmrt	97.0%	105

2 Player

```
class Player
 instance variables
    protected name : seq of char;
    protected playlist : seq of Track;
    protected suspended : bool := false;

    public  tr1 : Track := new Track("Track 1", 20);
    public  tr2 : Track := new Track("Track 2", 100);
    public  tr3 : Track := new Track("Track 3", 100);

    operations

    public AddTrack : Track ==> ()
    AddTrack(track) == playlist := playlist ^ [track];

    public Player : seq of char ==> Player
    Player(playername) ==
    (
        playlist := [];
        name := playername;

        start(self);
    );

    public Player : PlayerMemento ==> Player
    Player(pm) ==
    (
        let tms = pm.playlist in
        playlist := [new Track(tms(i)) | i in set inds tms];
        name := pm.name;
    );

    public Reinitialize : PlayerMemento  ==> ()
    Reinitialize(pm) ==
    (
        let tms = pm.playlist in
        playlist := [new Track(tms(i)) | i in set inds tms];
        name := pm.name;
        start(self);
    );

    public Run : () ==> ()
    Run() ==
    (
      while(len playlist > 0 and not suspended) do
      (
        let track = hd playlist in
        (
          start(track);
          track.Finished();
        );
        playlist := tl playlist;
      )
    );

    public Migrate : () ==> PlayerMemento
    Migrate() ==
    (
      dcl pm : PlayerMemento := new PlayerMemento();

      suspended := true;
      pm.playlist := [ playlist(i).Migrate() |
                           i in set inds playlist ] ;
      pm.name := name;

      return pm;
    );

    thread
    Run()

end Player
```

Function or operation	Coverage	Calls
AddTrack	100.0%	3
Migrate	100.0%	1
Player	0.0%	0
Reinitialize	100.0%	1
Run	100.0%	2
Player.vdmrt	85.0%	7

3 PlayerMemento

```
class PlayerMemento
 instance variables
        public name : seq of char;
        public playlist : seq of TrackMemento;
end PlayerMemento
```

Function or operation	Coverage	Calls
PlayerMemento.vdmrt	100.0%	0

4 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 Out: seq of char ==> ()
  Out(pstr) ==
    def - = new IO().echo(pstr) 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);
  );


sync
 mutex(OutWithTS)

end Printer
```

Function or operation	Coverage	Calls
Echo	100.0%	1
Out	100.0%	231
OutAlways	100.0%	1
OutWithTS	100.0%	5
natToString	100.0%	5
Printer.vdmrt	100.0%	243

5 Sys

```
system Sys
instance variables

  public static cpu1 : CPU := new CPU(<FCFS>, 22E6);
  public static cpu2 : CPU := new CPU(<FP>, 22E6);

  public static mobPlayer : Player := new Player("mobilePlayer");
  public static homePlayer : Player := new Player("homePlayer");

  public static bus : BUS := new BUS(<CSMACD>, 1E1,{cpu1, cpu2});

operations

public Sys : () ==> Sys
Sys () ==
(
    mobPlayer.AddTrack(mobPlayer.tr1);
    mobPlayer.AddTrack(mobPlayer.tr2);
    mobPlayer.AddTrack(mobPlayer.tr3);

    cpu1.deploy(mobPlayer.tr1);
    cpu1.deploy(mobPlayer.tr2);
    cpu1.deploy(mobPlayer.tr3);
    cpu1.deploy(mobPlayer);
    cpu2.deploy(homePlayer);

);

end Sys
```

Function or operation	Coverage	Calls
Sys	100.0%	1
Sys.vdmrt	100.0%	1

6 Track

```
class Track
 instance variables
    protected name : seq of char;
    protected playtime : nat;
    protected length : nat;
    protected finished : bool;
    protected suspended : bool := false;

    operations

    public Track : seq of char * nat ==> Track
    Track(trackname, tracklength) ==
    (
        name := trackname;
        playtime := 0;
        length := tracklength;
        finished := false;
    );

   public Track : TrackMemento==> Track
    Track(tm) ==
    (
        name := tm.name;
        playtime := tm.playtime;
        length := tm.length;
        finished := tm.finished;
    );


    public Play : () ==> ()
    Play() ==
    (
        while(not suspended) do
        (
            playtime := playtime + 1;

            if(playtime mod 20 = 0) then
            (
                Printer`Out("\n");
                self.Status();
            )
            else
                Printer`Out(" â™« â™«("
                ^ VDMUtil`val2seq_of_char[nat](threadid) ^ ")");

            if(playtime >= length) then
            (
                finished := true;
                return;
            );
        )
    );

    public Migrate : () ==> TrackMemento
    Migrate() ==
    (
        dcl tm : TrackMemento := new TrackMemento();
        suspended := true;

        tm.name := name;
        tm.playtime := playtime;
        tm.length := length;
        tm.finished := finished;
        tm.playing := playtime > 0 and not finished;

        --release any thread blocking on finished
        finished := true;
        return tm;
    );

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

    public Status : () ==> ()
    Status() ==
    (
        Printer`Out("#" ^ name ^ " played for "
                ^ VDMUtil`val2seq_of_char[nat](playtime)
                ^  "s on thread "
                ^ VDMUtil`val2seq_of_char[nat](threadid) ^ "\n");
    );

thread
    Play();

sync
per Finished => finished;

end Track
```

Function or operation	Coverage	Calls
Finished	100.0%	4
Migrate	100.0%	2
Play	96.0%	4
Status	100.0%	11
Track	100.0%	2
Track.vdmrt	98.0%	23

7 TrackMemento

```
class TrackMemento
 instance variables
        public name : seq of char := [];
        public playtime : nat;
        public length : nat;
        public finished : bool;
        public playing : bool;

end TrackMemento
```

Function or operation	Coverage	Calls
TrackMemento.vdmrt	100.0%	0

8 Types

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


--
-- class definition
--
class Types

types
public Time = nat;

public Event =  WasteTime | play | migrate;

public WasteTime ::
        t : Time;

public play ::
        t : Time;

public migrate ::
        t : Time;


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

```

```
end Types
```

Function or operation	Coverage	Calls
Verbose	100.0%	1
Types.vdmrt	100.0%	1

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

--
-- class definition
--
class World

--
-- instance variables
--
instance variables

public static env : [Environment] := nil;

--
-- Types definition section
--
types

--
-- Operations definition section
--
operations

-- create a := new World().Run()
public World: () ==> World
World() ==
(
 Printer`OutAlways("Start");
 env := new Environment("inputvalues.txt");

);

public Run: () ==> ()
Run() ==
(
    Types`Verbose(false);

    env.run();
    env.isFinished();

    Printer`OutWithTS("Stop");
);

end World
```

Function or operation	Coverage	Calls
Run	100.0%	1
World	100.0%	1
World.vdmrt	100.0%	2

Appendix E

Dynamic ConPlay Model

Only the classes which have been altered when applying the dynamic reconfiguration extension is included. The dynamic reconfiguration operations are marked with blue.

The `PlayerMemento` and `TrackMemento` has been removed from the model.

1 Environment

```
-----------------------------------------------
-- Class:   Environment
-- Description:
-----------------------------------------------

--
-- class definition
--
class Environment

--
-- instance variables
--
instance variables

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

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

```
-- 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 doEvents: () ==> ()
doEvents() ==
(
   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`play(-) ->
       (
         if event.t <= curtime
         then
         (
           Printer`OutWithTS("Environment: Playing");
           start(Sys`mobPlayer);
           eventOccurred := true;
         )
       ),
       mk_Types`migrate(-) ->
       (
           if event.t <= curtime
           then
           (
              Printer`OutWithTS("\nEnvironment: Migrating");
```

```

                 Sys`migrate(Sys`mobPlayer, Sys`cpu2);

                 eventOccurred := true;
              )
         ),
         mk_Types`WasteTime(-) ->
         (
          if event.t <= curtime
           then
           (
            Printer`OutWithTS("\nEnvironment: Wasting time");
            eventOccurred := true;
           )
         ),
         others ->
         Printer`OutWithTS("\nEnvironment: No match found")
        end;

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

        eventOccurred := false;
       );
     )
     else busy := false;
 );

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

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

--
-- Values definition section
```

```
--
values


--
-- Threads definition section
--
thread
(
 while busy do
 (
   duration(1000)
   doEvents();
 );

)
--
-- sync definition section
--
sync
 per isFinished => not busy;

end Environment
```

2 Player

```
class Player
 instance variables
    protected name : seq of char;
    protected playlist : seq of Track;
    protected suspended : bool := false;

    public  tr1 : Track := new Track("Track 1", 20);
    public  tr2 : Track := new Track("Track 2", 100);
    public  tr3 : Track := new Track("Track 3", 100);

    operations

    public AddTrack : Track ==> ()
    AddTrack(track) == playlist := playlist ^ [track];

    public Player : seq of char ==> Player
    Player(playername) ==
    (
        playlist := [];
        name := playername;

        start(self);
    );


    public Run : () ==> ()
    Run() ==
    (
        while(len playlist > 0 and not suspended) do
        (
            let track = hd playlist in
            (
                start(track);
                track.Finished();
            );
            playlist := tl playlist;
        )
    );

    thread
    Run()

end Player
```

3 Sys

```
system Sys
instance variables

  public static cpu1 : CPU := new CPU(<FCFS>, 22E6);
  public static cpu2 : CPU := new CPU(<FP>, 22E6);

  public static mobPlayer : Player := new Player("player");

  public static bus : BUS := new BUS(<CSMACD>, 1E1,{cpu1, cpu2});

operations

public Sys : () ==> Sys
Sys () ==
(
    mobPlayer.AddTrack(mobPlayer.tr1);
    mobPlayer.AddTrack(mobPlayer.tr2);
    mobPlayer.AddTrack(mobPlayer.tr3);

    cpu1.deploy(mobPlayer);
);

end Sys
```

4 Track

```
class Track
 instance variables
    protected name : seq of char;
    protected playtime : nat;
    protected length : nat;
    protected finished : bool;
    protected suspended : bool := false;

    operations

    public Track : seq of char * nat ==> Track
    Track(trackname, tracklength) ==
    (
        name := trackname;
        playtime := 0;
        length := tracklength;
        finished := false;
    );

    public Play : () ==> ()
    Play() ==
    (
        while(not suspended) do
        (
            playtime := playtime + 1;

            if(playtime mod 20 = 0) then
            (
                Printer`Out("\n");
                self.Status();
            )
            else
                Printer`Out("_/Â¯\\("
                ^ VDMUtil`val2seq_of_char[nat](threadid) ^ ")");

            if(playtime >= length) then
            (
                finished := true;
                return;
            );
        )
    );

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

    public Status : () ==> ()
    Status() ==
    (
        Printer`Out("#" ^ name ^ " played for "
          ^ VDMUtil`val2seq_of_char[nat](playtime)
          ^  "s on thread "
          ^ VDMUtil`val2seq_of_char[nat](threadid) ^ "\n");
    );

thread
    Play();

sync
per Finished => finished;

end Track
```