A A R H U S U N I V E R S I T E T

Speciale

Teknisk Informationsteknologi / Distribuerede Realtidssystemer

” Evaluating Distributed Architectures using VDM++ Real-Time Modeling with a Proof of Concept Implementation”

af

Rasmus Ask Sørensen & Jasper Moltke Nygaard

7. december 2007

Rasmus A. Sørensen, Studerende	Jasper M. Nygaard, Studerende	Peter Gorm Larsen, Vejleder

Abstract

Developing large distributed systems poses many challenges, due to the complexity and the distributed nature of this paradigm. Empiric knowledge cannot solve all of these issues single handedly and validation today is dependent on costly late-stage assessment of a chosen architecture. However, what if early stage indications could be obtained and used to validate a potential candidate system architecture or design approach, while identifying potential bottlenecks? With the use of formal validation techniques this may be possible.

This thesis will investigate and analyze the possibility of obtaining early stage validation of potential candidate system architectures, by means of formal modelling and validation. The goal is to analyze recent research extensions of VDM++ for describing and analyzing such distributed systems (VICE) and see if the language is suitable to stress test a distributed system and validate any architectural benefits. Additionally this thesis will discus favorable approaches for realizing a case study of a transportation system in Tokyo, referred to as CyberRail. Different VDM++ models of candidate architectures for the CyberRail system are developed and validated.

The results of this master thesis demonstrates that early architectural indications are possible by the use of the new VICE extensions of VDM++ and formal modelling. However, the ability to stress test a VDM++ model is not readily possible due to the infant stage of the tool supporting the new VICE technology. VDM++ and the associated tools does however show a great deal of future potential. Regarding the CyberRail case study, several potential bottlenecks where identified and three candidate architectures where analyzed. Two critical areas of realizing the CyberRail system were identified, concerning location awareness and the communication platform interacting with the passenger.

Resum

"Brug af VDM++ Real-Tids Modellering samt en Konceptuel Implementering til Evaluering af Distribuerede Arkitekturer"

Udviklingen af store distribuerede systemer prsenterer mange udfordringer, grundet teknologiens kompleksitet og natur. Viden baseret p erfaring kan ikke lse alle problemstillinger i et distribueret system og validering er i dag afhngig af kostbare vurderinger sent i udviklingsfasen. Hvad hvis man kunne f indikationer p et tidligt stadie, der validerede en given system arkitektur eller design, mens potentielle flaskehalse blev identificeret? Dette kan muligvis lade sig gre, ved brug af formelle modeller.

Dette speciale vil undersge muligheden for at opn tidlige indikationer omkring fordelagtige system arkitekturer, ved brug af formel modellering og validering. Mlet er at undersge de seneste forskningsudvidelser til VDM++, til at beskrive og analysere distribuerede systemer (VICE). Derudover vil dette speciale diskutere fordelagtige fremgangsmder til at realisere et case study af et transport system i Tokyo ved navn CyberRail. Forskellige VDM++ modeller af kandidat arkitekturer til CyberRail, vil blive udviklet og valideret.

Resultatet af dette speciale antyder, at arkitektur indikationer p et tidligt stadie er muligt, ved brug af de nye VICE udvidelser til VDM++. Dog demonstrerer specialet at VDM++ ikke er egnet til stress test, p nuvrende tidspunkt. VDM++ og de tilhrende vrktjer viser fremtidigt potentiale. I forbindelse med CyberRail, er tre kandidat arkitekturer blevet analyseret og 2 kritiske omrder er blevet identificeret i relation til stedsans og kommunikation med passagererne.

Contents

Acknowledgements

We would like to thank Peter Gorm Larsen for his work as supervisor. The input and guidance has been invaluable and his dedication has been second to none. We would also like to thank our families and friends for being supportive.

Part I

Setting the Scene

Chapter 1

Introduction

1.1 Background

The basis for this master thesis and the involvement of VDM++ originates from VDM++ courses which we attended during our Candidate degree in Embedded and Distributed Systems at the Engineering College of Aarhus, Denmark. These courses introduced us to the world of formal modeling and lead to the evaluation and use of VDM++ in this thesis.

1.2 Motivation

The development of distributed systems poses many challenges due to the complexity and the disciplines used to realize a system of this nature. Many unknown factors exist when developing a distributed system and solving these issues often rely on empiric knowledge obtained from similar system or by testing late-stage prototypes. The areas of concern:

- Does the distributed system react correctly?
- What kind of deployment platform is needed?
- How will the system operate under high stress?
- What communication platform will be sufficient for a specific task?

Many of these issues are initial concerns for all distributed system and by using late-stage testing, it may not be feasible to alter choices made at early stages. However, what if a rapid response for questions of this nature could be achieved at a very early stage of development and thereby validate that a system path is substantiated?

A large scale distributed system is currently being developed for the transportation system of Tokyo. This system is a grand vision of improving the current transportation situation and heighten the quality of service experienced by the passengers. Many unknown

factor exist, ranging from deployment and communication platforms to functional behavior and design issues, which cannot be answered prematurely. Some of these questions could potentially be answered by means of formal modeling techniques currently offered by the VDM++ language and the associated tools. With introduction of Real-Time primitives to the VDM++ language, this is an interesting development candidate in the world of system engineering [SAE06] [SK07] [Dou06].

1.3 Thesis Goals

This goals of this thesis is to evaluate VDM++ and its engineering capabilities in terms of indicating favorable architectures, design approach and potential bottlenecks in a distributed system, at an early stage of development. By using the Real-Time primitives of VDM++ and the associated tools, it is hoped that the analysis of the CyberRail case study will indicate advantageous approaches to the design and to the deployment. As a part of this evaluation, certain beneficial system architectures for the CyberRail system are identified and pragmatic details discussed in regards to functionality and deployment. The focus area of concern are:

- Can VDM++ be used to stress test a formal model of a distributed system?
- Can VDM++, in an early stage of development, give any indication towards a favorable design or architecture or can it in any way indicate potential bottlenecks?
- What type of architecture, design and technology is beneficial to use for the CyberRail system?

To fully answer these questions, certain additional steps are required and the process of obtaining answers to this, is a partial goal for this thesis. If shortcomings, during this process, are identified, alternative solution to these problems are proposed. Focus of early stage response indications in a development phase, relies on the ability to make an approximation and achieve a reliable estimate, by the means of formal modeling.

1.4 Reading Guide

This section describes how this document is structured and provides the notational conversion for references, captions, listings and figures.

1.4.1 References

The external references in this document are structured the following way: [Acronym of the Author(s) last name(s) + Year]. If there are multiple documents by the same author(s) and year, an additional identifier will be added to avoid ambiguous references - for example [Doe06A] and [Doe06B]. When there is a reference in caption that indicates that the figure is taken from that reference. If a reference occur in the caption of a figure, then this is the origin of the image.

1.4.2 Captions

Figures containing UML diagrams such as class, sequence, flowchart and deployment diagrams, will be explicitly described as such. A class diagram caption would look like this: *"Class Diagram: The structure of the backend system"*.

1.4.3 Document content/structure

This thesis is divided into three parts and for each of these there is a short walk through providing a rationale for the structuring selected.

Part I: This part introduces the motivation for this thesis and an introduction to the aspects used. Chapter 2 describes the case study which is used as a foundation for the formal models and introduces the challenges involved. Chapter 3 introduces the formal language VDM++ and the associated tools. This chapter describe the concept of abstraction, which is used throughout this thesis.

Part II: This part shows the process of creating several VDM++ models. Chapter 4 is a detailed description of every step performed in Part II. Chapter 4 additionally contains a thorough description and discussion of the candidate architectures identified in this master thesis. Chapter 5 describes the prior work, mainly concerned with the initial functional requirements, the sequential and the concurrent VDM++ model of the CyberRail system. Chapter 6 introduces the main focus area of this master thesis and reevaluates the requirements, entities and finally introduces a preliminary Real Time VDM++ model. A proof of concept implementation of the real time VDM++ model is described in chapter 7 and timing characteristic results and relevant areas of concern are commented. Chapter 8 introduces the timing characteristics into the preliminary Real Time Model and analyze the results. The last chapter in part II discuss the alternative architectures and realize one of them as a VDM++ model, which is put through the same test scenario as the initial Real Time model.

Part III: The last part includes a discussion of the results obtained Part II and the knowledge obtained throughout this master thesis. Chapter 10 includes a series of discussions and conclusions. VDM++ and VDMTools VICE is evaluated and aspects of using the tool is discussed. Various aspects of CyberRail is also commented and certain beneficial approaches and architectures are proposed. Quality aspects, in regards to how well the VDM++ models and a Java prototype can be tested are discussed, based on tools developed for this thesis. Two appendices are included which comprises documentation for the generic logging framework for distributed systems and the Reaper utility which is used to setup deployment for VDM++ models.

1.4.4 Assumptions about the Readers

During the writing of this thesis some assumptions about the readers technical knowledge have been made.. It is assumed that the reader at least has basic knowledge of the VDM primitives and ideally knows VDM++ and VICE as well. It is also expected that the reader

has basic knowledge of issues concerning the distributed computing paradigm and some of the basic technologies typically used.

Chapter 2

The CyberRail Concept

CyberRail is a concept project initiated by the Japanese Railway Technical Research Institute in Tokyo. The project is an extensive vision with several levels of functionality and services, to provide a single frontend for both the transport companies and the passengers using public transportation.

Tokyo is a high density city with 12 million people living and working within the city limits [Wik07]. The subway and train system is the city life nerve for transporting people to and from work everyday. 7.5 million residents use the subway or train system everyday in central Tokyo and 20 millions in the greater Tokyo area. Public transportation reliability and availability, in high density areas such as Tokyo, is therefore crucial and breakdowns could potentially cost millions of dollars in lost labor.

The railway network in greater Tokyo area is composed of 136 independent railway line carriers with more than 1100 stations. Each of these carriers utilize their own payment and scheduling service and there is no apparent cross communication between these companies. This imposes a large practical challenge when having to travel with several different companies to reach a destination. This especially imposes a problem for non-locals and tourists which know little or nothing about the complex infrastructure.

CyberRail is a concept for dealing with these issues and is structured into 2 areas of concern:

- Passenger Services
- Company Services

The Passenger services relates to the usage of the transportation infrastructure. The goal is to make the public transportation in Tokyo more intuitive by providing a convenient, reliable and situation sensitive information service. This involves a number of different passenger services.

Transport Planning let the passenger conveniently plan a trip using a form of communication device - also called a token device. The transport plan function will take care of all the details in regards to different transport service providers and payment. The user simply input the departure and destination location and the transport plan service returns a complete plan.

Personal Navigation relates to a single service of helping the passenger to find a specific location. This could be vocal or visual guidance to the ticket office or the departuring platform. This service works much like car navigation in its traditional sense, however with a higher level of details and a with a highly context-aware behavior.

Transport Rescheduling enables the user to receive updated transport plans issued from CyberRail. This scenario is used when the previously calculated transport plan is no longer traversable due to e.g. breakdown and the passenger needs to be rerouted.

Passenger Assistance is a general service which incorporate many of the previous services described. The personal assistant, in the form of a token device, can aid the passenger with information such as time until departure, destination stop notification and estimated remaining traveling time. The personal assistant could also proactive initiate a dialog with the passenger to solve transport plan issues.

The company services relates to the logistics and management of information within the transport infrastructure. These service will make up the platform for developing the passenger services. The goal is to provide an efficient, flexible and competitive transport service with the following:

- Platform for distributing transportation information
- Platform for exchanging information
- Intelligent train management

In time CyberRail is supposed to extend its transportation infrastructure to cover bus, taxi and private transportation sectors and unite the means of transportation in a single user friendly organ. This will not only benefit the passengers but also increase the overall cost efficiency for the associated transport service companies involved.

2.1 Diversity of CyberRail

The different focus areas of the CyberRail concept covers a wide variety of services and functionality, and developing and realizing all of these will take a substantial amount of time. Many different aspects would be interesting to analyze of both the passenger and company services. Below the various aspect which was identified in this master thesis as potential focus area are discussed.

From the company services perspective, a very interesting area is the intelligent train management service. This service involves context-aware adaption to the railway and passenger

load status. For example during rush hour more train wagons must be put into service to accommodate for the increasing number of passenger. However, under certain circumstances, train breakdown, railway maintenance etc., the passenger load cannot be estimated and an alternative strategy must be put into effect. This strategy must specify how to move all the waiting passenger from the affected train stations to their destinations efficiently. This could be achieved by rerouting the passengers with a different train or maybe insert additional busses and taxis to relieve the load. In real life this scenario is highly plausible and most passengers often experience a very low service quality. With a system like CyberRail and the amount of information available, this situation would be easily solved with minimum degradation in service quality.

Due to the overly complex structure of the railway grid and the different transportation providers involved, a seemingly simple thing as payment has become quite an ordeal. The different companies has developed their own procedures for payment and this makes it difficult to figure out how, where and what to pay for the service. Technologies are emerging for this type of proximity payment application and many which are already deployed in various sectors. The interesting aspect in regards to the CyberRail concept is what and how this could be realized. How would the different companies be merged into the system and could this normal and seemingly simple task become more or less transparent to the passenger, while maintaining an efficient and competitive service.

Another interesting aspect is the actual passenger navigation and assistance service. The situation sensitive information service can be realized as a two-way communication paradigm with the passenger and the CyberRail backbone system. A form of communication device, a token device, will provide the interface for interaction with the system. A passenger can request a transport plan and will receive the information to successfully reach the destination. The backbone system can, upon changes in the current transport plan, notify the passenger and reroute if necessary or just provide state information periodically for the user. However the technology for such a deployment scheme is not very mature and the communication paradigm demands involved can be hard to precisely specify with any level of confidence. Will a system like this be operational with 10.000 or more concurrent users? If disaster hits and 25% of the railway grid breaks down, will the system make any difference at all? The passenger navigation and personal assistance presents many challenges and the technology of today may og may not be applicate for this type and scale of applications.

2.2 The CyberRail Vision

The full vision of CyberRail and the passenger navigation is more extensive and futuristic. The vision incorporates a personal assistant, which is represented as a token device, which the passenger carries at all times and and use to assist through all potential steps of a journey. The personal assistant can plan a journey using different means of transportation such as taxi, bus, train and subway. Figure 2.1 shows a transport plan using all the above mentioned transport methods which forms the so called the travel chain, which is a chain of transportation modes from door to door [TYKR01]. The personal assistant can order a taxi for the passenger to drive to the train station, without having to interact with the user. When existing the taxi the

passenger will be guided by the personal assistant to platform from where the destined train leaves. During the journey, the passenger will not have to figure out how to pay the different service, since the assistant will do this automatically. The assistant will keep you updated whenever you need to change train or switch to a bus. If anything effects the travel plans the assistant will change the plans accordingly to make sure that the passenger will reach the final destination as planned and on time. This implies that if sections of the railway grid breaks down or a bus route is not currently running, the assistant will reroute the passenger using other means of transportation. The personal assistant can for example also be used to guide the passenger to the food court of the ticket office at the train station. This is a especially convenient for tourists which does know their way around the city.

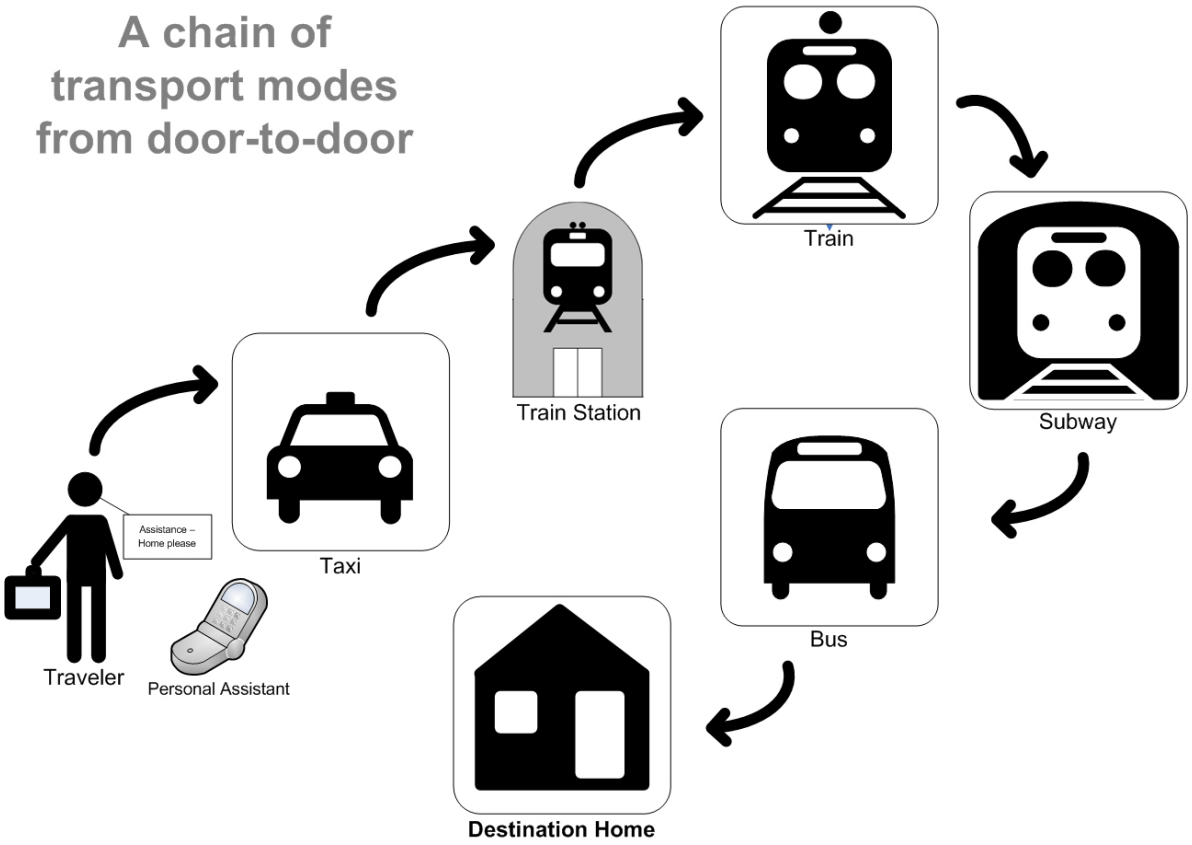


Figure 2.1: The CyberRail travel chain [TYKR01]

Chapter 3

VDM++ as a Engineering tool

The Vienna Development method or VDM is a formal modeling language that is used to model functional aspects of software systems. VDM gives the possibility to test the model by executing it and/or using a more mathematical approach and proof the functionality. At current there is three incarnations of VDM, VDM-SL, VDM++ and VDM++ VICE (VICE).

VDM started its conception in the IBM Vienna as Vienna Definition Language (VDL). VDL was made to define language and compilers, it later became replaced by VDM that had a broader scope. Through several refinement process became VDM-SL which became an ISO standard in 1996. Through the Afrodite project for European Commission VDM++ was defined [vK96]. The new VICE with distribution is from 2006.

VDM-SL VDM-SL is used to define software functionality, the models created with VDM-SL are sequential.

VDM++ VDM++ contains the core functionality of VDM-SL but extends them with bring in the object oriented design approach and concurrency [FLM+05].

VDM++ VICE VICE is a further extension of VDM++ it brings timing concerns and deployment into the models. VICE stands for VDM++ in Constrained Environment. VICE is the version worked with in this project, the following subsection will focus on explaining the special features and tool support for VICE [CSK06].

3.1 Features in Vice

Figure 3.1 shows some of the special features in VICE, the basic functionality from VDM-SL such as collections and functions are not shown. The blue boxes are features from VDM++ and the green purely VICE features. The blue has been include since they are some core functionality that is used when creating up a VICE model.

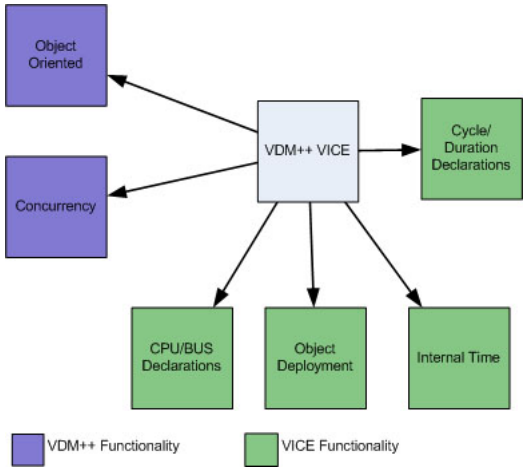


Figure 3.1: The core functionalities in VICE

CPU/BUS Declarations In VICE it is possible to set up an execution platforms topology, this is done through the primitives CPU and BUS. CPU specifies a processor, it can be declared in two modes either First Come First Served(FCFS) or Priority Based. The speed of the CPU is defined as Instructions Per Second(IPS), but can also be seen as the clock cycle of the CPU when it executes segments with in a cycle tag. BUS specifies a communication bus, it used to connect CPU's, only CPU's connected with a BUS can communicate. A BUS has three declarations, first there is the protocol used on the BUS, at the time of writing there is only FCFS. The second one is the speed of the bus, it is defined in kilo bits per second or Kb/s, the last is a set of the CPU's it connects.

Object Deployment When having declared a set of CPU's connected CPU via BUS'ses, objects can be deployed on the CPU's. Taking two objects each with a thread and deploying them on two different CPU means that they can run concurrently and not just interleaved as on one CPU.

Cycle/Duration Cycle and Duration tags are used to define how long a segment of the model will take to execute. Cycle defines the amount of clock cycles the CPU use to execute the model segment, that means the faster the CPU is, the faster the segment is executed. Duration defines a specific time, a segment within a duration tag will take the same amount of time on any speed of the processor.

Internal Time Unlike in VDM++ where the model itself has to keep track of the time, in VICE that is handled internally. The time is made to resemble milliseconds.

3.2 Tools

There is currently only one tool that can execute an VICE model, VDMTools VICE from CSK in Japan, figure 3.2 shows the features currently in VDMTools. When executing a VDM++

model in VDMTools VICE, an execution trace log is created, during use of the interpreter, with highly detailed information regarding BUS inter-communication, CPU thread processing, variable state information and most importantly timestamps. A separate tool, developed by the Overture community, called *Showtrace*, can parse the trace log and display the execution path graphically. This enables the user to readily analyze the behavior of a seemingly complex distributed system in a user friendly graphical manner. Figure 3.3 shows an example of such as visual representation.

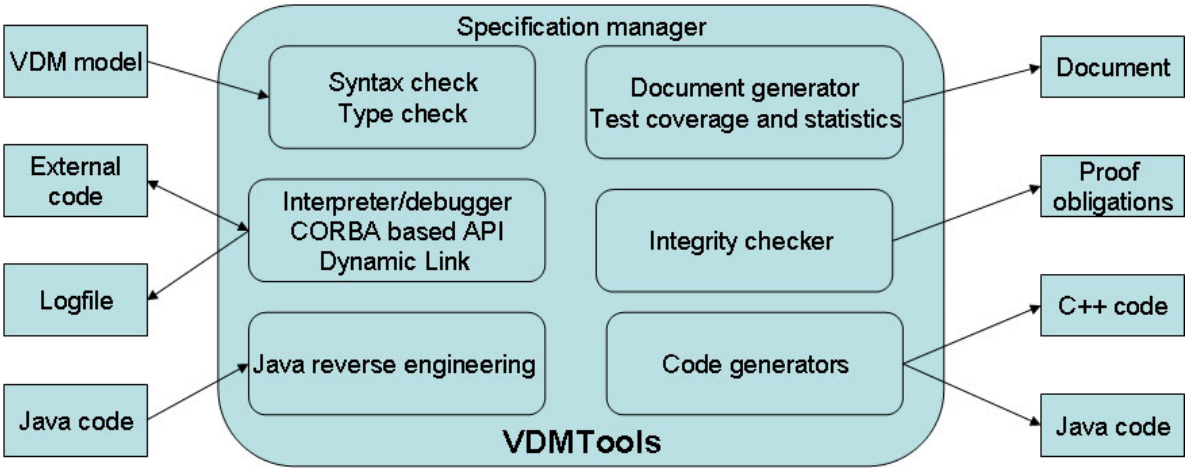


Figure 3.2: The features currently in VDMTools [FLS07]

3.3 Development with VDM

Figure 3.4 shows how VDM++ could be integrated in 2 existing development methods; waterfall and ROPES [Dou99]. In both methods the VDM++ model is positioned in the analysis and design phase with the possibility of round tripping between the phases. An application example could be the integration of 2 different system without any knowledge of cross communication.

The overall advantage of using a formal model prior to the actual developing phase, or integrate it directly in the software engineering procedures, is the ability to evaluate that certain architectural decisions make sense and that the intended behavior is feasible. When using a traditional development approach architectural and functionality changes in the later changes can be very expensive or just impossible. By using a formal model these issues can be analyzed and tested at a very early stage and increase the confidence in the design before the actual implementation progress. Trying to model the whole system in every detail is usually not the best approach as it can take as long as the actual implementation process. Instead a rational abstraction level must be specified, this decreases the complexity of the model. Making its construction faster but still keeping the ability to test the model.

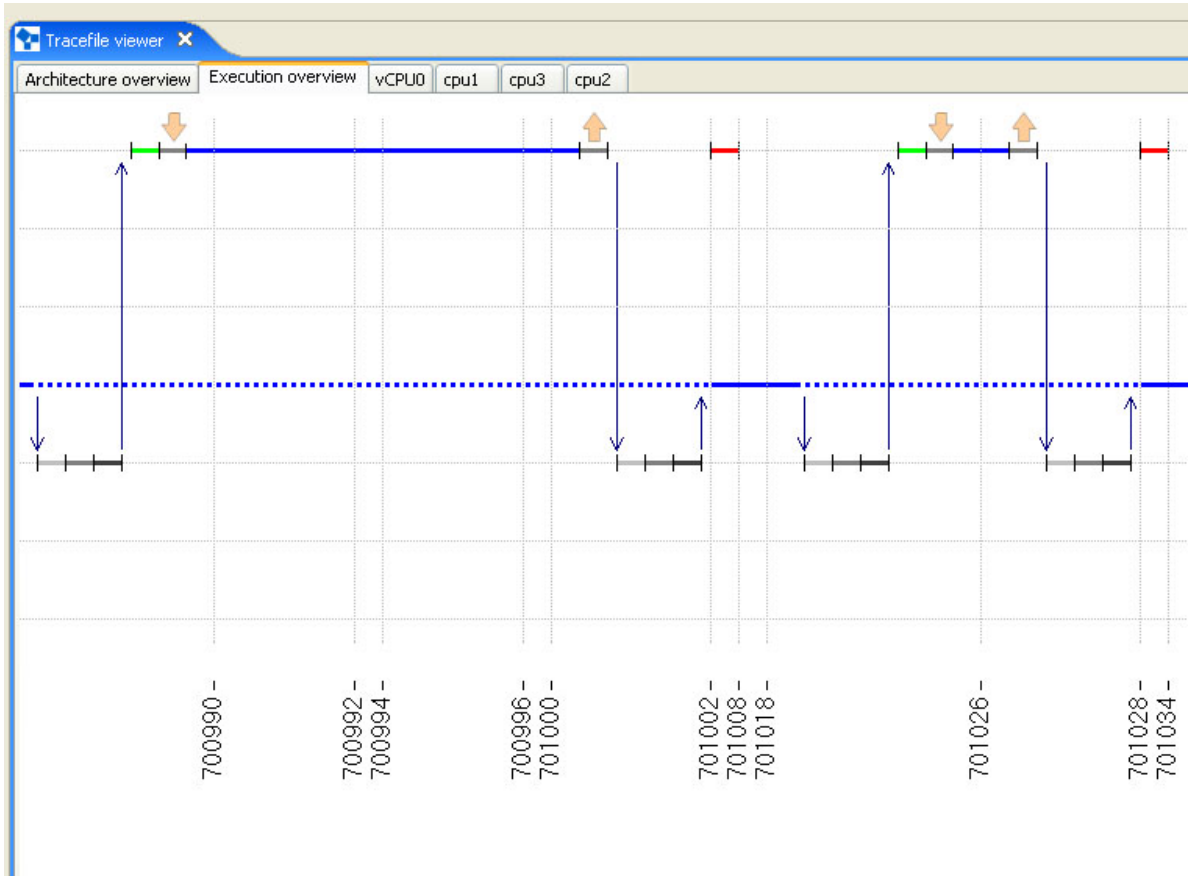


Figure 3.3: Example of showtrace

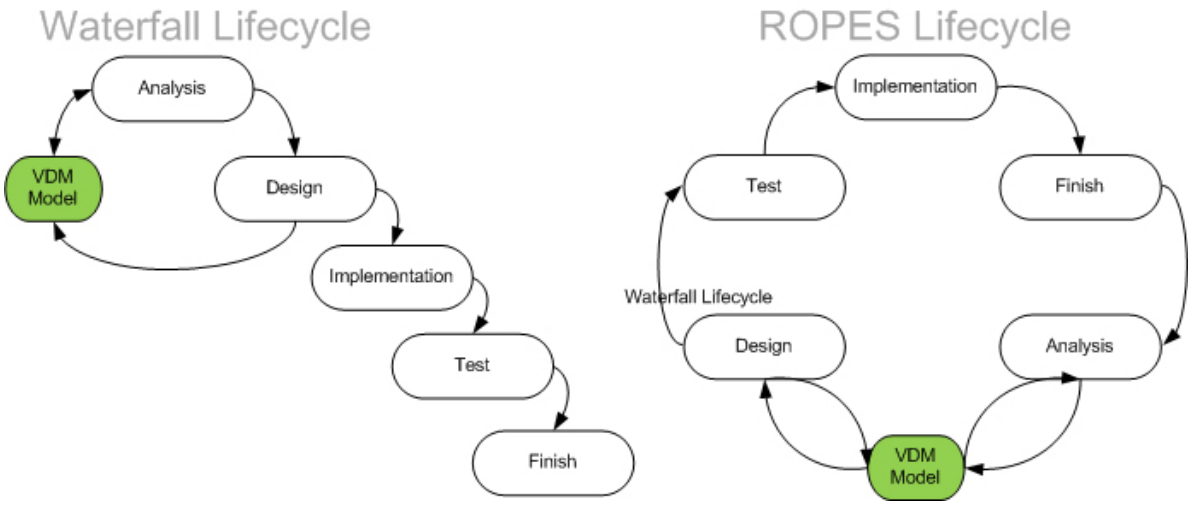


Figure 3.4: Waterfall and ROPES development lifecycle.

3.4 The Rules of Abstraction

Abstraction is a hard conceptual idea to grasp, which makes it even harder to use. However abstraction can aid the process of development by ignoring irrelevant details and focusing on the essentials. By ignoring irrelevant details, a highly complex system can be divided into iterative steps, which each identify a specific functionality or focus area of concern. However the use of different abstraction levels does not have to be a linear process. The ability to have different abstraction levels, with different focus area, enables the developer to move between the levels and hence move to a concrete focus area. This is a unique capability when solving problems in a complex system such as a distributed system.

The skill to ignore irrelevant detail can be used in various stages of development, to solve specific tasks faster and more efficiently. If a developer is working at a very low abstraction level, the task or problem at hand may seem more complex than it is and will take substantially more time to complete. If the developer on the other hand migrated to a higher level of abstraction, many unsustainable details can be ignored and the problem is potentially clarified.

Another way to argue for the importance of software abstraction, and the skill to ignore non-important details, is clarified in The Stable Abstraction Principle (SAP) [Mar96], even though it relates to software packages as building blocks. The author Robert C. Martin relates abstraction as:

> *The abstraction of a package should be in proportion to its stability.*

This can be interpreted, in terms of software abstraction, as the higher level or more accurate level of abstraction obtained, the lower the risk for unimportant details interfering with the issue at hand. Figure 3.5 illustrates the relationship between the level of abstraction and system instability. If a system development process started with a low abstraction, this would put the project in the risk zone for major instability. The system is perceived more complex and more details impact the system behavior which in the end evidently will impact the overall progress. If, however the system started out with a relative high abstraction level, the risk for instability is minimal and by decreasing the abstraction level incrementally, the risk margin is kept at a minimum.

So how do you define the abstraction level for a specific task? One approach is to incorporate the abstractions levels into the iterative development process which concerns itself with functionality and details. An iterative step will focus on a specific task or component based on prior work and is not especially concerned with any future steps. Another approach, and potentially more interesting, is to think of abstraction as conceptual ideas, rather than the level of details of a system [HK06]. This will enable the developer to define each abstraction level as a complete system (Top-down approach), with different conceptual focus areas, rather than different levels of details.

Even though the use of software abstraction for developing complex systems is a conceptual hard skill to master, and even harder to have as a unconscious cognitive tool, the reasoning is to make a seamingly complex task as simple as possible.

> *Abstraction is the essence of simple and effective software design, and logic is the essential tool for exploring and validating abstractions.* **Tony Hoare, Senior**

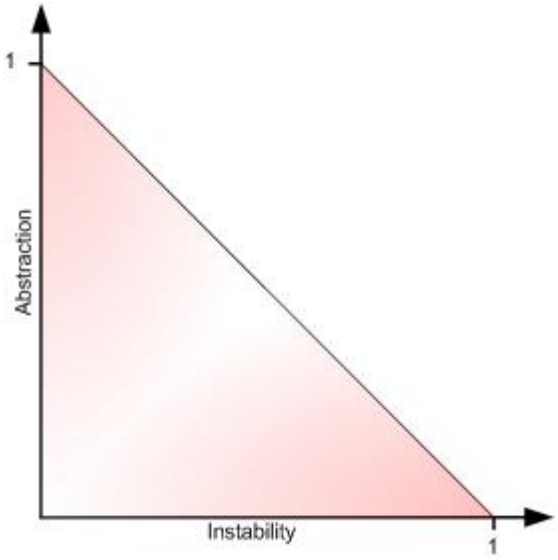


Figure 3.5: Relationship between stability and abstraction

Researcher, Microsoft [HP]

Tony Hoare identifies two very important facts in the above quote. First that abstraction is a process of making an issue more simple and hence easier to comprehend, as already discussed. The other fact is that abstraction is used for a specific purpose or with a specific focus in mind. What this means is that an abstraction level often represents a specific view of a system relating to one area of concern or purpose. For example when creating a VDM++ model, certain aspects such as time or concurrency can ignored, to first of all minimize complexity and risk of errors and secondly to be able to focus on just one view of the model without any distracting details. This view of the model could for example focus on program behavior, with the purpose of making it correct. Notion of time and concurrency behavior is not of any interest to the developer, since these issues does not impact the purpose of the abstraction level - in this case *correct behavior*. When the purpose of a giving abstraction level has been meet, the view of the model can be moved to incorporate for example the concurrency disciplines, which represent a different abstraction level.

Software abstraction is one of the key motivational factor when using VDM++ as a software development step. The formal approach lets you rethink architectural and design decisions and instantly receive rapid feedback whether or not an approach is feasible. Moving between abstraction levels is easy when working with a formal model, since external factors does not impact the system. This is an important ability and strength which should not be underestimated when evaluating VDM++ as an engineering tool.

Part II

Distributed Architecture Analysis

Chapter 4

Introduction

Many of todays projects are based on empiric knowledge when analyzing and designing a new system. This kind of knowledge is accumulated over years of experience and is a solid contributer to a project compared to the trial and error model. However certain issues differ from one project to another and it can be very challenging to accurately predict the outcome with a high level of confidence. With the means of formal modelling and validation some level of confidence can be achieved by simulating certain scenarios and thereby building a firm foundation before the actual design phase.

CyberRail presents very specific challenges when it comes to designing the system, and since there are very few systems that resembles it. There is little experience creating such systems so reliance on empiric data and experience is not really possible. To have a better understanding of what architectural factors impact systems like CyberRail, different architectures are investigated.

Three different VDM++ models with different overall system architecture are identified and analyzed. Two of these are realized in VDM++, where one of the architectures are implemented as a Java prototype, used to acquire real life timing characteristics. These characteristics are used to configure the VDM++ models and thereby increasing the overall credibility.

4.1 Thesis Methodology

When developing a model there are many ways to do. One way is to use the approach described in VICE development process [CSK06]. Figure 4.1 shows the process outlined in the VICE Development guidelines. First a requirement analysis is performed using Use Cases', VDM-SL or both. Secondly a primitive sequential model is implemented, followed by a concurrent model, individually tested to validate correct behavior. The last step involves implementing a Real-Time model with all the details and validating the model.

Another process is described in the book *Validated Designs for Object-oriented System* [FLM+05]

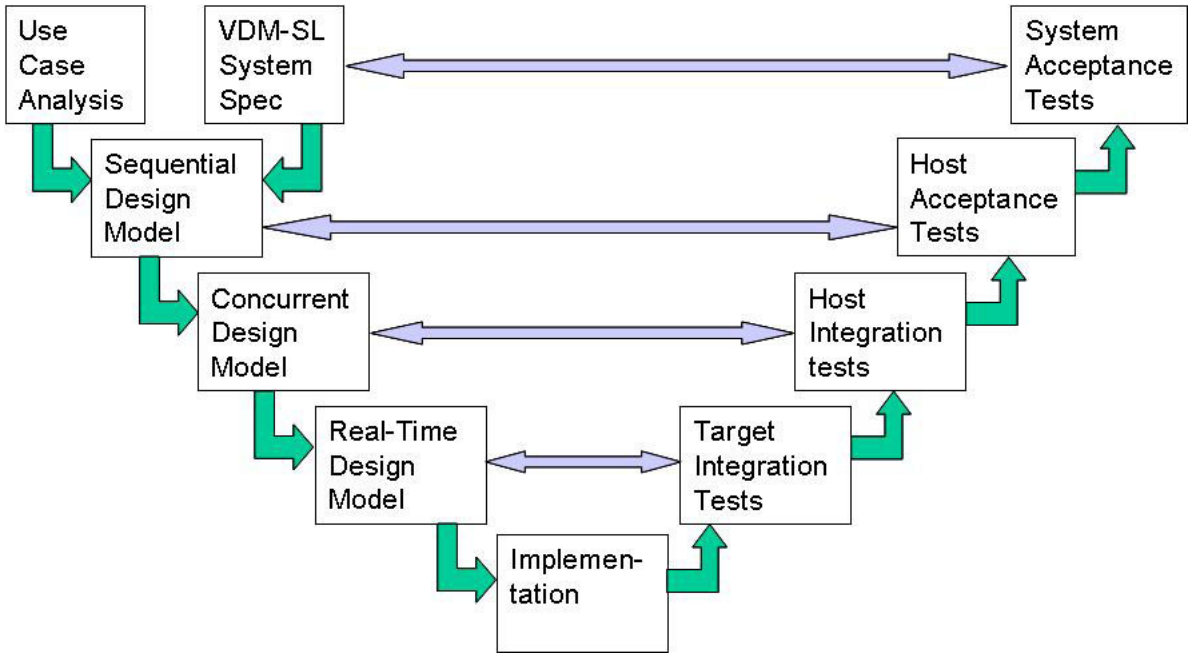


Figure 4.1: Overview of VICE development Process [CSK06]

Figure 4.2: Overview over VDM development process.

and this process only focuses on the sequential model. It does however have a different approach of defining the requirements. Figure 4.2 shows the requirement process. Instead of using Use Cases, a set of functional requirements are created. The requirements lead to a list of potential classes and types and a list of potential operations. The classes and types are usually composed of nouns found in the requirement list and the operation list is usually composed from actions that can be identified from the requirements. From the potential classes, types and operations, a class diagram is created which finally is realized in VDM++.

A combination of the two mentioned processes are used with some modification. The full list of steps in the process can be seen on figure 4.3. A more detailed description of what happens in each phase, can be seen in the following sections.

4.1.1 Initial Modeling Phase

This phase contains all the work leading up to the real time model. The initial modeling defines an abstraction level, which primary purpose is to define the basic system behavior. There is no form of performance or any consideration whether the actual designs and architecture is

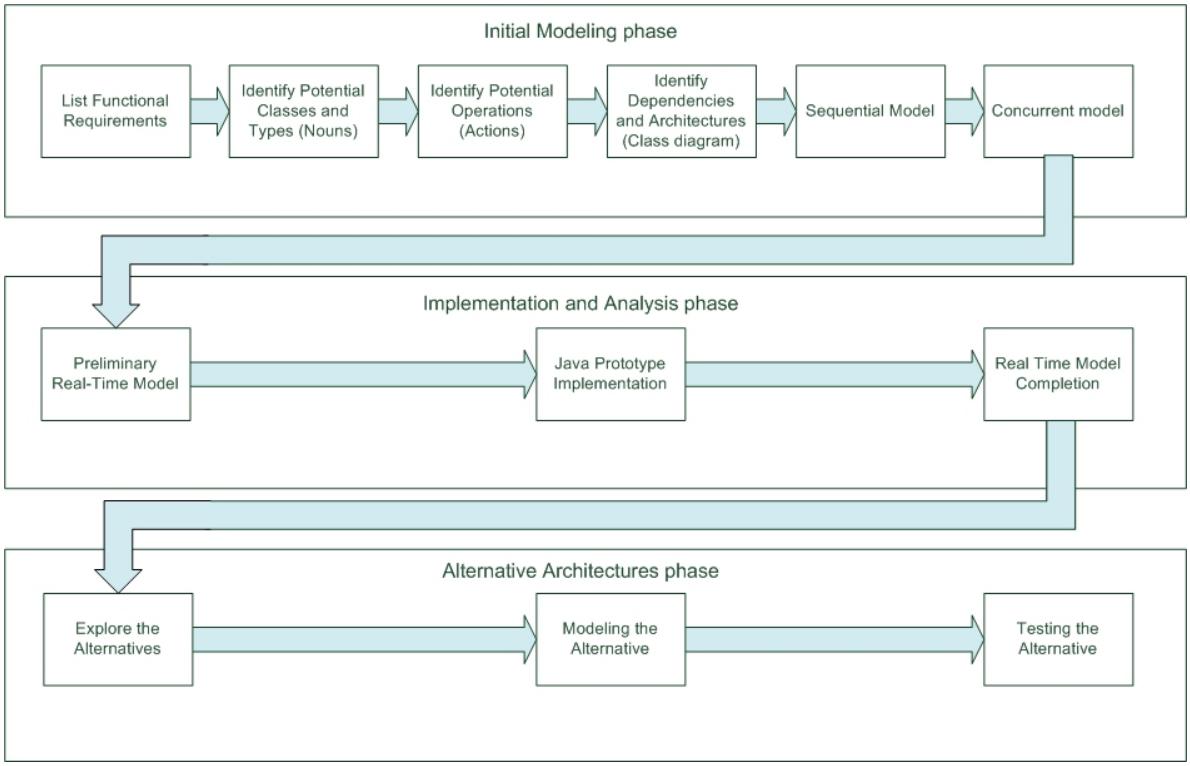


Figure 4.3: The complete development process.

a realistic approach or not. The steps of this phase are a combination of the two mentioned processes.

List Functional Requirements

In this step all the functional behavior is identified as a series of requirements. For instance a requirement for a mobile phone could be: *It should be possible to send and receive text messages.*

Identify Potential Classes and Types (Nouns)

This step involves the identification of potential classes and types. This is usually done by finding nouns in the functional requirements list. From the requirement above, a class could be *"textMessage"*. Whether it is going to be a class or a type is going to be decided later. This step is about identifying potential candidates.

Identify Potential Operations (Actions)

Based on the functional requirements, potential operations are identified. Instead of using nouns, the potential actions are identified by assigning an expressive operation name. With the

example requirements already identified, two potential operations could be *"sendTextMessage"* and *"recieveTextMessage"*.

Identify Dependencies and Architectures (Class diagram)

When the classes, types and operations have been identified, it is time to put them together. This could be realized as a UML class diagram. Not everything is necessarily found in the previous steps, so more classes, types and operations may be added during this step. Based on the previously identified items from before, then the *"textMessage"* is more a type than a class. However a *"phone"* class would be applicate with two operations, *"sendTextMessage"* and *"recieveTextMessage"*, which uses the type *"textMessage"*.

Sequential Model

This is where the first VDM model is realized by implementing the class diagram made in the previous step. The sequential VDM++ model is concerned with the basic functionality and no concurrent primitives are used in this phase.

Concurrent Model

Here the concurrency primitives is introduced into the model, threads are created and tasks can run concurrently. Concurrency problems such as race conditions and deadlocks are identified in this model, which is used as a stepping stone toward the Real-Time VDM++ model.

4.1.2 Implementation and Analysis Phase

This phase involves some additional steps, not described in any of the development guide lines, to make the Real-Time model as trustworthy as possible. The relative high abstraction level used so far, is lowered and the focus of this phase is concerned with acquiring realistic timing characteristics. As there is no past experience with the behavior of a system such as CyberRail, a Java prototype implementation is realized based on the preliminary Real-Time model. The Java prototype is used to acquire timing information in regards to execution and inter-communication between systems. The preliminary VDM model will contribute with static data, such as railway grid composition, traversable routes and time table, for the Java Prototype which in return will contribute with timing characteristics for VDM++ Real-Time Model. This feedback loop will contribute with realistic information and increases the overall trustworthiness of the model.

Preliminary Real Time VDM Model

The concurrent VDM model is made into a Real-Time model. This includes deploying active classes on CPU's and setting up the BUS's for inter-communication. The preliminary model defines all basic functionality and the basic architecture.

Java Prototype Implementation

This is the step where the preliminary real time VDM model is converted into a Java prototype. This step involves an analysis of timing characteristics in regards to system performance.

Real Time VDM Model Completion

The timing characteristics measured from the previous step are analyzed and converted to the VDM++ model, where the final tests are performed

4.1.3 Alternative Architectures Phase

This step is not part of any of the mentioned development processes, but is a common approach of exploring and experimenting with different candidate architectures. The second VDM++ model identified is based on the first, but designed with a different architecture. There are many similarities between the two models and the requirements are exactly the same.

Explore the Alternatives

First potential architectures are identified and analyzed to find the best suited alternative to the current architecture. Each alternatives have advantages and disadvantages which is carefully weighed against each other.

Modeling the Alternative

In this step the best suitable candidate architecture is realized in VDM++. Unlike the previous VDM++ models, this architecture will be realized as an Real-Time VDM model and will not go through the initial sequential and concurrent modeling phases. The Java prototype timing characteristics are reused since the core logic of both models will stay identical.

Testing the Alternative

The alternative VDM++ model is tested under the same circumstances as the first Real-Time mode. The purpose of this test is to see how this alternative architecture behaves compared to the first real-time model and if any assumptions can be made, based on indications from the test. The models are to be stress tested using the same parameters and the timing characteristics are compared.

4.2 Identifying Significant Candidate Architectures

Three different potential candidate architectures has been identified: backend, frontend- and joint responsibility, also referred to as *BR*, *FR* and *JR* respectively. Backend and frontend responsibility are archetype architectures, and joint responsibility is, as the name implies, a combination of the two. In the following subsections a short description and discussion of the three architectures is presented. These potential architectures have been identified prior to

any substantial design and implementation progress and they each present a viable and deployable approach, which could be used in the CyberRail Concept using technology available today. This aspect of deployability is very important, since it greatly decreases the number of potential architectures and technology platforms. Designing a system using 'tomorrow's' technology is a very different task and the identified architectures are only concerned with readily available solutions. Certain aspects of the original vision of CyberRail has been disregarded and the abstraction level for these candidate architectures focus mainly on the passenger service. Features such as the company services and how they interact with CyberRail, are not taken into account this thesis. Considerations in regards to passenger payment and transport modes are likewise not important and is ignored in these architectures. The purpose of these candidate architectures are to find the best suitable solution, focusing on the passenger and the the ability to travel from one point to another.

4.2.1 Architecture 1 - Backend Responsibility

All state information regarding passenger location, train and station information, transport plan assignments and section breakdowns are located in the backend system - see figure 4.4. The token device is a passive entity and is only used to present transport plan data to the user, with the exception of requesting a new transport plan. This approach assures low latency response and high performance within the backend system, in regards to servicing transport plan requests and recalculation transport plans upon section breakdowns. The token device will present data, provided by the backend system and will not initiate any actions. Keeping the passenger state information in the backend system opens up for different kinds of services, such as destination stop notification and estimated remaining traveling time. It also loosens the requirements to the token device hardware platform, since location awareness and transport plan calculation are a responsibility of the backend system and not the token device.

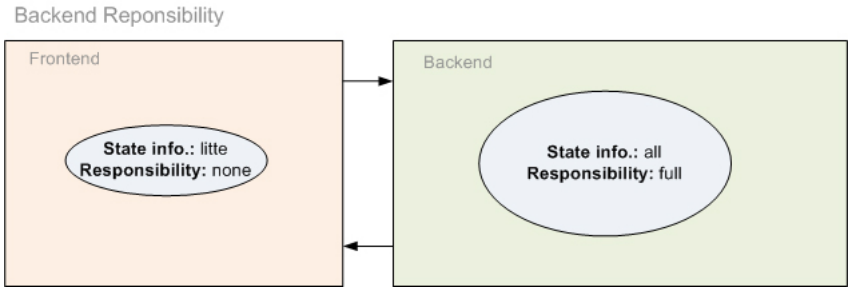


Figure 4.4: Backend responsibility characteristics.

Depending on how the system is developed and deployed, this architecture is potentially the most stable of the three solutions in regards to dependability. The functionality is centralized in an single entity, when disregarding the internal distributed design of the backend system. If the backend system operates as predefined, which can be assured by e.g. redundancy, performance scalability and failover mechanisms, the rest of the system will operate correctly.

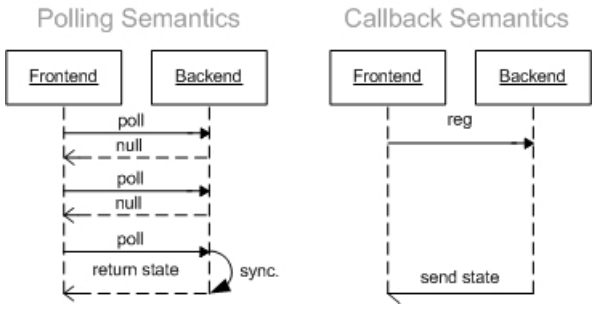


Figure 4.5: Sequence Diagram: Difference between polling and callback

This assumption is however not entirely true since the token device primarily is a passive actor in this context and the backend system represent every aspect of the CyberRail system. The essential issue is that the backend system is not dependent on any information from the outside, besides the state information it already possesses, to be able to service traveling passengers. The state information, which the token device is dependent on, is the same for all passengers since everything is located in the backend system. Some would say that this approach is a disadvantage, since it presents a single point of failure, however this does not have to be the case. The backend system may be perceived as a single entity, at least from the passengers point of view, but will most likely be deployed with different safety measurements such as load balancing, migration transparency etc. [Wag05a].

The weakest link in this approach is the the inter-communication between the passenger and the backend system, since this is highly dependent on which type of communication technology is used and the volume of concurrent users. Minimizing inter-communication with the passengers is a high priority in this design, while still maintaining a sufficient granularity of information. Additionally this design creates constraints regarding the communication platform that can be used between the frontend and the backend system. A callback paradigm is essential, since the backend system is responsible for notifying the passenger regarding changes in the transport plan. Without the callback functionality, the token device must implement a polling mechanism which would produce extensive and costly traffic between the frontend and backend system - see figure 4.5. This is therefore not considered a viable option for this architecture.

This design presents a solid and realistic architecture and is suitable for the CyberRail system. Important aspects of this system includes the centralized location awareness of passengers, highly efficient inter-communication within the backend system and consistent state information for all passengers.

4.2.2 Architecture 2 - Frontend Responsibility

This architecture distributes all responsibility and state information to the token device - see figure 4.6. This means that all functionality and the state information which was handled by the backend system, is migrated to the token device. The advantage of this architecture is the simplicity of the design and the distributed paradigm. The backend system does not

contain any type of state information regarding the passenger, but merely offers to present the information when requested. Each token device must actively poll for new information regarding the transport plan, railway grid status etc. on a need-to-use basis.

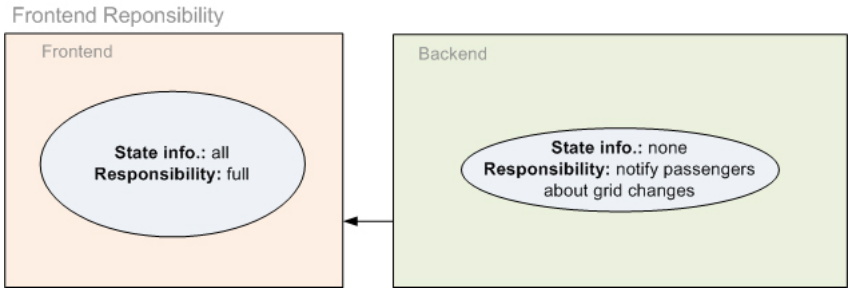


Figure 4.6: Frontend responsibility characteristics.

Since the responsibility of calculating new transport plans is distributed to the passengers, the deployment cost of the backend system would be considerable lower and this is a key factor when analyzing this type of system. If used in conjunction with a mobile phone, then the token device will preexist since most people own one already. This means lower deployment cost, faster integration and adaption period. Furthermore the system is simpler in regards to the *BR* architecture and hence more independent. A chain is no stronger than its weakest link and this especially a valid statement in a distributed system.

The challenge with this architecture, is how to maintain a synchronized view of the railway grid state information. The backend system provides the state information about inactive sections, but there is no guarantee that the distributed token devices have the updated information. They may be calculating transport plans on deprecated information and hence be receiving incorrect guidance. The challenge is to make sure that each token device uses the same railway grid information as a starting point, prior to any transport plan calculations, however limiting the update frequency and hence the strain of the backend system.

Another area of concern are the convenient services such as destination stop notification and remaining traveling time. These are not possible due to the lack of location awareness, which was handled centralized by the *BR* architecture. This problem could potentially be solved by means of technology, such as GPRS, RF location awareness such as UbiSense [Ubi07] or GSM/Bluetooth triangulation [Zha00]. However these each produce deployment difficulties which may question the usability of these technologies. RF location awareness is extremely costly and relies on location beacons which will require proprietary token devices. GSM/BlueTooth triangulation is fairly difficult to control and would require the token device to somehow communicate with the backend system about its current position. The GPS, or the more sophisticated European Galileo solution, is a popular solution when it comes to location awareness, but it also produces a number of challenges. The advantage is that mobile phones are currently emerging with GPS capabilities, however far from a standard tendency. The disadvantage is that it has a high power consumption and this decreasing the battery time on the phone. Additionally the GPS receivers in mobile phones are very weak and may not have

a sufficient signal within a train, not to mention train stations and subways. The additional cost may also be a significant deal breaker, stopping the passengers from adapting the system.

Without the extra 'nice-to-have' functionality provided by location awareness, then this architecture scales very well and by distributing the responsibility, making it very interesting cost-wise. Each passenger would have all the required information right on the token device with only little intercommunication with the backend system, which could be realized with simple SMS technology. Occasionally the passenger would need to perform a more detailed synchronization of the railway grid information with CyberRail. Many phones support the ability to communicate with a PC and uplink directly to the internet, which could be used to perform a detailed synchronization with CyberRail. This architecture does however not fully live up to the demands of the CyberRail concept, in terms of the passenger services and is not considered a favorable approach.

4.2.3 Architecture 3 - Joint Responsibility

This architecture is an attempt to combine the scalability and distribution of responsibility of the *FR* architecture with dependability and service possibilities of *BR*. The joint responsibility architecture will keep most of the responsibility on the token device, like in the *FR* architecture. However, the token device would report some state information to the backend, so the system can retain the dependability of the *BR* architecture.

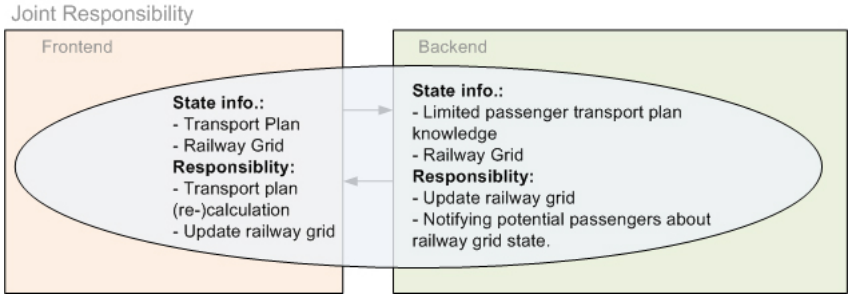


Figure 4.7: Joint responsibility characteristics.

The *JR* architecture share the responsibility between the frontend and backend system as illustrated in figure 4.7. The token device is fully responsible for calculating new transport plans and has all the railway grid state information locally. The same considerations, in terms of distributing this state information among the passengers, apply for *JR* as for the *FR* architecture. The main difference from *JR*, is that the token device send minimal state data to the backend system. If a new transport plan is calculated, the token device will send the traversable sections and the token device callback address to the backend system. This information is used by the backend system to test whether or not this token device *could* be affected by section breakdown. The backend does not know exactly where the token device is currently located, as opposed to the *BR* architecture. However, it knows the approximate location and that the passenger *may* be affected by the breakdown. This approach ensures

that all potential affected passengers are notified, whereof some might already have traveled the section and are no longer interested or affected. This is a trade off for the *JR* architecture. The backend system is responsible for 2 things; update the railway grid information locally and notify possible affected passenger. The responsibility of rerouting the individual passenger is now moved to the token device.

The advantage of this architecture is that less state information is necessary for the backend system to actively notify the passenger upon breakdowns and that this approach still retains the centralized location awareness property. Additionally the responsibility of calculation the new transport plans, still reside at the token device, as in the *FR* architecture, distributing the resource load away from the backend system. The disadvantage is that a passenger can potentially be notified about inactive sections which have already been traveled and this will decrease the overall quality of the passenger experience. This design approach unites many of the advantages of *BR* and *FR* and is a realistic architecture for the CyberRail system.

Chapter 5

Initial Modeling of CyberRail

5.1 Abstraction Considerations

As mentioned in section 3.4, setting an abstraction level can lower the complexity of the task at hand. The complexity of the CyberRail concept, makes it impossible, and impractical to incorporate every aspect and feature into a single formal model. Therefore it is important to abstract away from all the irrelevant issues with respect to the purpose of the model.

An overview of some of the areas of CyberRail, that were intially identified as potential candidate aspects to model, can be seen in figure 5.1. The aspect of communicating with passengers was identified as an appropriate area of focus, when initially identifying the abstraction level for the sequential model.

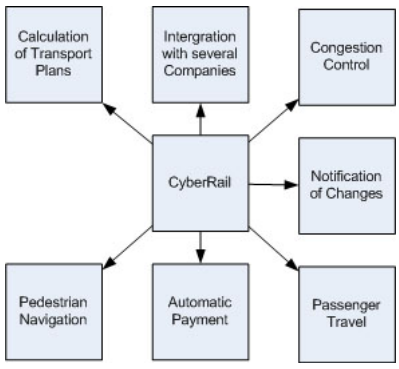


Figure 5.1: Identified functionality of CyberRail

Calculation of Transport Plans is where a passenger request a transport plan from A to B, and a plan is returned to passenger. The functionality of these aspects constitute

many of the other passenger services and how they are constructed. This area of concern will be used in the abstraction level for the sequential model.

Integration with Transport Companies covers the integration of the different service providers which contribute with transportation services to CyberRail. Companies contribute with traversable sections and the by introducing new ones and disabling existing section, this imposes a large computational resource strain on the backend system which might be interesting. However, this aspects is not taken into account due to the focus and purpose of the VDM++ model.

Congestion Control relates to congestion problems caused by for example section breakdown or a rapid increase of passenger and how CyberRail dynamically can assign more carriages to relieve the problem. This aspect is not included in the abstraction level, since it introduces several new requirements in terms of how the railway grid composed and this is not important in regards to passenger navigation and the calculation of new transport plans.

Notification of Changes is the act of notifying passengers with a new transport plan when one the sections they were planning to travel, are inactive. This aspect is taken into account in the abstraction level, since there can be several thousand affected passengers that needs an updated transport plan.

Passenger Travel defines the ability to automate the process of making the passenger travel. This was not identified as a high priority and was abstracted away.

Automatic Payment relates to how proximity payment can be introduced into CyberRail and how automatic payment can be handled by CyberRail. However this aspect is not significant for the system architecture and is abstracted away.

Pedestrian Navigation is the act of guiding people on foot, to a specific location, for instance to a platform. This issue is something that is used by the passengers and could impact the performance of CyberRail. But it is very complex problem that includes many different aspects, such as precises location awareness. It can also be seen as a separate system, from the other passenger affected functionality, as this occurs only on stations and is not on the trains. This functionality is not modeled as it will make the model too complex and change the focus.

Figure 5.2 shows which focus areas are in the sequential and the concurrent VDM++ model.

5.2 Defining the Requirements

From the abstraction definition stated above the following requirements have been formed.

R1 **A computer based system is to be developed to provide transport planning and payment.**

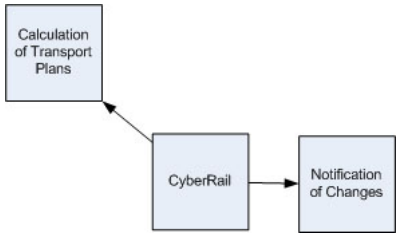


Figure 5.2: The selected functionality

This defined the basic system boundary entity which is necessary for the system to exists.

R2 **The passenger must be able to alter the transport plan at any time.**
This option ensures that the passenger is not bound to a single transport plan. The basic concept of CyberRail defined several important user services and one of these was flexibility. This requirement ensures some flexibility.

R3 **The system must offer two different transport options: quickest and cheapest.**
This requirement also ensures flexibility but also ensures efficient and competitive transport options.

R4 **The passenger must be able to request a departure or arrival time.**
This option is both a requirement and a necessity for the system to work.

R5 **The passenger must specify departure and destination location. Optionally the passenger can specify by way of locations.**
By letting the passenger create a multi point transport plan, eases the process of booking the transport plan and improves usability. The user only needs to book one transport plan and the system calculates the entire journey.

R6 **If a breakdown of a specific route occurs, the affected transport plans must be revised accordingly and reflect the original.**
The CyberRail system must identify and recalculate affected transport plans and notify the passenger. This requirement ensures a convenient, reliable and situation sensitive information service.

R7 **An alarm must be activated to notify the passenger about changes in the transport plan.**
An alarm is activated on the user input device to ensure that the passenger actually acknowledges a change in the transport plan.

R8 **The passenger must use a token device to interact with CyberRail.**
Passenger communication with the CyberRail system is performed from a mobile input device called a token device. The token device represent a number of possible devices such as a mobile phone, PDA or a proprietary CyberRail communicator device.

5.3 Identifying Entities

5.3.1 Classes and Types

- **CyberRail**
  Emulates the main system entity. This entity manages transport plans, handles the data model and is responsible for the train surveillance.
- **Token**
  Reflects the input device used by the passenger.
- **Transportplan**
  A data transfer value object containing information regarding a journey.
- **NavigationInput**
  Data transfer value object containing passenger input information.
- **TimeScheduleDB**
  Data value object containing all the data from each registered transport company. All the transport plans are composed from this database entity.
- **Alarm**
  Entity representing the alarm located at the passenger
- **PlanManager**
  Entity to handle route break downs and is responsible for notifying the user. This entity also keep track of the current passengers and their transport plans.
- **Route**
  A data value object containing information regarding specific route segment.

5.3.2 Operations

- **Calculate transport plan**
  Produces a transport plan corresponding to the passenger input. The method acquires the information needed for the transport plan from the CyberRail system entity.
- **Survey active transport plans**
  This action survey the list of transport plans currently pending in the system.
- **Notify passengers**
  When a transport change has occurred, the passenger needs to be notified. This action ensures that the user acknowledges the change in the transport plan.
- **Request transport plan**
  The system must be able to calculate a complete transport plan based on the passenger input.
- **Report inactive route**
  This method notifies the system that a specific route is down.

5.4 The Sequential VDM Model

This section shortly describes the sequential model. For a more detailed description see *CyperRail Planing and Navigation VDM++ Model* [NS06b]. As mentioned above the sequential model is about looking at functionality and whether the requirements are plausible to satisfy. Figure 5.3 shows the full class diagram for the sequential VDM++ model. The classes are:

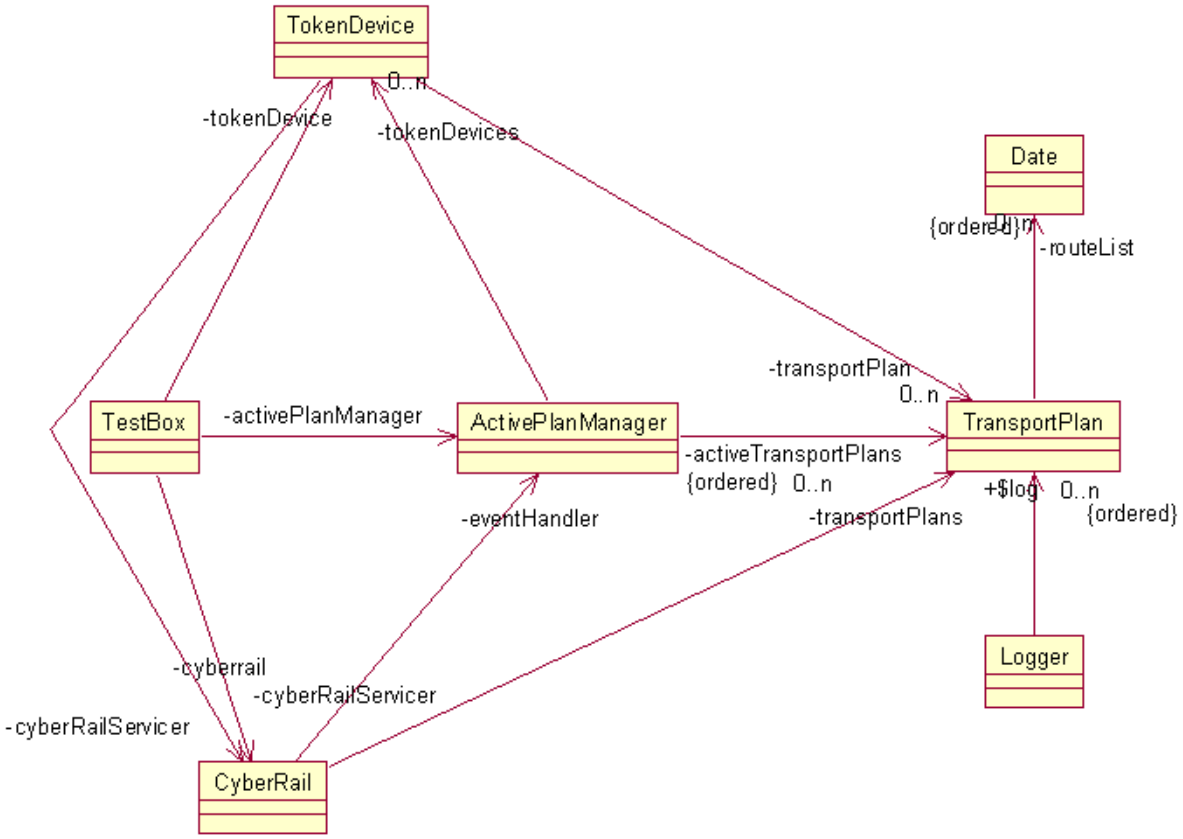


Figure 5.3: Class diagram: The sequential VDM++ model

TokenDevice This emulates the device that the passenger uses to interact with the CyberRail system and use to request a new transport plan or be notified by CyberRail.

CyberRail This is the backend of the system, where transport plans are calculated and where inactive events are received.

ActivePlanManager This is facade for CyberRail, it is here TokenDevice sends its request for transport plan. ActivePlanManager keeps track of which TokenDevice has which TransportPlan.

TransportPlan An instance of this indicates a transport plan for a single token device.

Date Contains a date but also a specific time.

Logger Logs events, it is mainly used for debugging purposes.

TextBox Contains all the different test cases which is used to validate the model.

5.5 The Concurrent VDM Model

The concurrent VDM++ model is build up like the sequential model. However, some additions are introduced into the model such as environment and system classes. Figure 5.4 and 5.5 shows the changes from the sequential to the concurrent VDM++ model. For a more detailed description of the concurrent model see *CyberRail Concurrency and Real-Time aspects VDM++ Model* [NS06a].

5.5.1 Environment classes

The reason environment classes has been added to model, is to create a greater definition of what the actual model is. The classes are:

World is the entry point to the model. It is through World that the environment and the system is initialized.

TimeStamp keeps track of the time. As there is no internal system time, it is necessary to keep track of a logical time stamp.

WaitNotify contains the logic to wake threads, its is inherited by TimeStamp.

Environment inputs stimuli into the model. Environment reads input from a file and stimulates the model.

IO This class has functions that makes it possible to read and write data to and from a file. It is used by the Environment classes to read input stimuli from a file.

5.5.2 System classes

The only new class on figure 5.5 is MessageQueue. The rest is classes were identified in the sequential VDM++ model. MessageQueue was added because it was found necessary to have some form of asynchronous messaging between the threads. This was needed because ActivePlanManager had to wait for CyberRail to finish calculating a transport plan and in the meanwhile ActivePlanManager would stay inactive. The classes are:

MessageQueue is a generic queue class used by active objects to send messages to each other. There is an instance of MessaggeQueue between each thread that communicates. The inter communication link can be defined percieved as: TokenDevice MessageQueue ActivPlanManager MessagesQueue CyberRail.

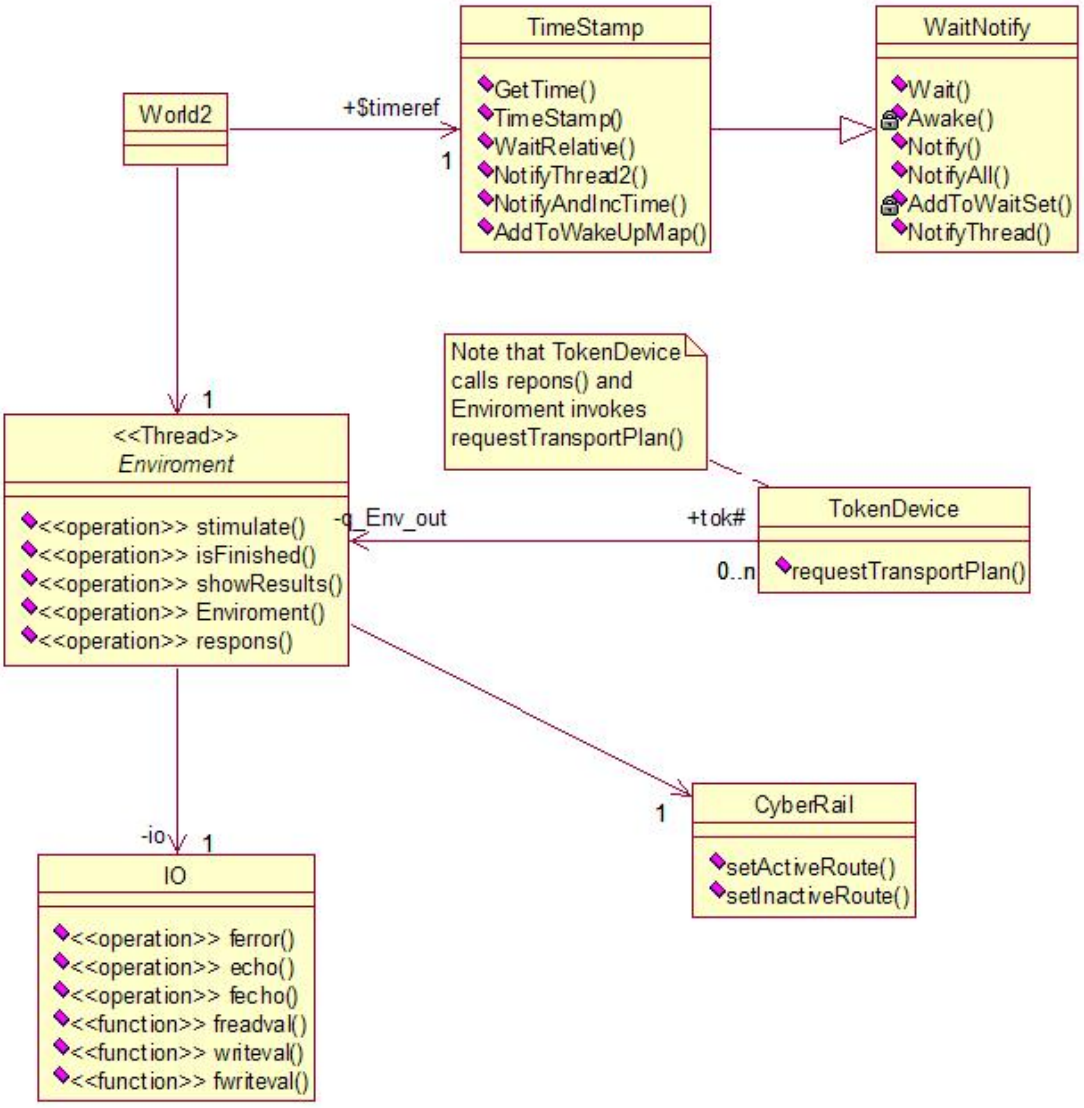


Figure 5.4: Class diagram: The environment classes in the concurrent VDM++ model

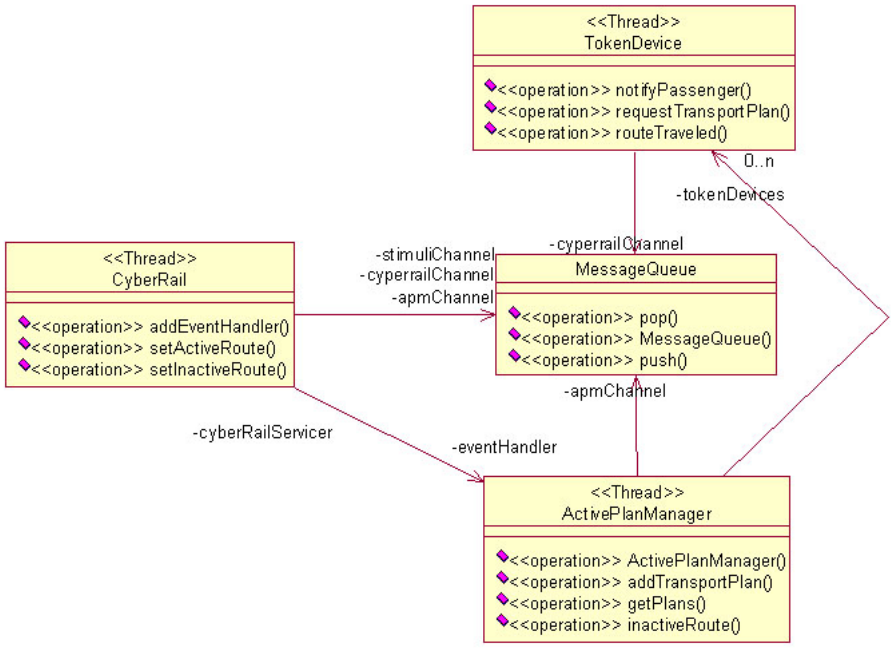


Figure 5.5: Class diagram: The system classes in the concurrent VDM++ model

TokenDevice differentiate a little from the sequential model and is now an active object and receive stimuli from the environment.

ActivePlanManager has also become an active object. It does the same job as in the sequential model, except that it now is possible to queue requests.

CyberRail is now an active object and receives input from the environment. It does the same job as in the sequential mode. Just like the ActivePlanManager it is also possible to queue request.

Chapter 6

Building the Real Time VDM Model

6.1 Abstraction Level

After developing the sequential and concurrent models it was necessary to reevaluate the current abstraction level as defined in section 5.1. It has been found that the abstraction level chosen was a bit too high and there where some cases that could not be handled. There was no notion of time in the previous models, so the passenger got the same transport plan no matter when it was requested. To make a more realistic model, time was now introduced correctly into the model. Another problem with the previous models, was the fact that the functionality of making the passengers travel, was not present. Allthough this is not a problem when only focusing on the test aspects, it did impair the overall dynamics of the models. A more realistic run through of the model was a priority in this phase. Figure 6.1 shows the aspects included in the new abstraction level.

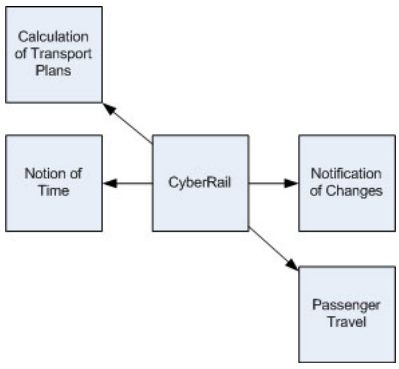


Figure 6.1: The new abstraction level aspects

6.2 Reconsidering the Requirements

Because of the changes in the abstraction levels, mentioned above, a new set of functional requirements was needed. Many of the old requirements could be reused, but some of them had to be removed and new ones was added. The new list of requirements can be seen below. Requirements marked with deprecated are to be ignore as they are no longer valid. New requirements are marked as new.

R1 **A computer based system is to be developed to provide transport planning and payment.**
The basic system boundary entity which is necessary for the system to exists.

R2 **The system must handle transport planning transparently from several companies.**
This requirement is needed to define the data model layer on which the CyberRail system will be based upon. The actual data handling is not defined, but is merely an abstraction.

R3 **The passenger must be able to alter the transport plan at anytime.**
This option ensures that the passenger is not bound to a single transport plan. The basic concept of CyberRail defined several important user services and one of the was flexibility. This requirement ensures this.

R4 **The system must offer 2 different transport options: quickest and cheapest.(Deprecated)**
This requirement also ensures flexibility but also ensures efficient and competitive transport options.

R5 **The passenger must be able to request a departure or arrival time.(Deprecated)**
This option is both a requirement and a necessity for the system to work.

R6 **The passenger must be able to travel from one station to any of the others in the model**
This requirement ensures that no requests from passengers are impossible.

R7 **Transport plans must not make round trips.(New)**
This requirement ensures that the passenger will not get a transport plan that passes the same station twice.

R8 **The passenger must receive the fastest possible transport plan.(New)**
This requirement ensures that the passenger will get the transport plan that gets to the arrival location fastest. It has been added to replace R4.

R9 **The passenger must specify departure and arrival location.(New)** By letting the passenger create a multi point transport plan, eases the process of booking the transport plan and improves usability. The user only needs to book one plan and the system calculate the entire journey.

R10 **If a breakdown of a specific route occurs, the affected transport plans must be revised accordingly and reflect the original.**
The CyberRail must identify and recalculate affected transport plans and notify the passenger. This requirement ensures a convenient, reliable and situation sensitive information service.

R11 **The passenger must be notified about any changes in the transport plan.(New)**
An alarm is activated on the user input device to ensure that the passenger actually acknowledges a change in the transport plan.

R12 **The passenger must board a train to travel.(New)**
This is added because so that the passengers just do not magically travels from one destination to another.

R13 **There must be several trains traveling the same sections at different times(New)**
This requirement ensures that a section is not just traveled once but several times.

R14 **The passenger must use a token device to interact with CyberRail.**
Passenger communication with the CyberRail system is performed from a mobile input device called a token device. The token device represent a number of possible devices such as a mobile phone, PDA or a proprietary CyberRail communicator device.

6.3 Identifying new Entities

Several different research questions comes into mind when considering a CyberRail model versus a prototype implementation. The ability to resemble real life deployments with VDM++ is very interesting. However, only if the test results of the model are precise and consistent. A very interesting aspect of all largely theoretical systems is deployment and architecture. It is easy to setup a minimalistic system in a lab and these deployment schemes are usually governed by empiric knowledge about the actual implementation methods and technologies. But how well does this paradigm scale in regards to real life performance?

This section explains the entities which is used throughout the three candidate architectures and their primary responsibilities. However some entity behavior and responsibility do change from one architecture to another.

Train contains information about passengers currently traveling on this particular train.

Stations contains information about passengers currently waiting on the train station.

RoutePlan contains a route information for a whole day (24 hours). This structure contains information regarding each route, departure/arrival locations, departure/arrival times etc.

Route contains a sequence of sections and a departure time, making up a route for a train to travel.

Section specifies a traversable path between two Stations, and parameters such as duration and price.

TokenDevice represents the passenger and/or the communication device, which interfaces with the backend system.

TPC responsible for responding to transport plan requests. This entity contains all state information regarding the all sections of the railway grid.

APM contains state information about which passengers are currently traveling which routes. APM is also responsible for notifying each passenger about changes in the transport plan assigned.

TransportPlan contains a sequence of sections, but it is not the same as a Route. The sections may cross different Routes making it necessary for the passenger to change trains.

6.4 Realizing the Real Time VDM Model

6.4.1 Architectural Description Method

In the following sections, the different candidate architectures will be described in terms of entities categorized as record types, environment- and system classes. The environment classes has no direct impact on the functional behavior of the model but input stimuli into the system. It is also the environment class that initializes the system [CSK06].

First an overall class diagram showing the model is illustrated. For each class diagram there will be a short description of each class and record type. Finally there will be a section describing the behavioral aspects of the model through scenarios. These scenarios will be the same for each architecture to make comparison easier. Two different scenarios will be used. The first is a sunshine scenario, where a transport plan is requested and the passenger travels the distance without any sections breakdowns. The other scenario, and potentially more interesting one, is where a section breakdown occurs. The passengers are affected by the breakdown and new transports plan need to be calculated and obtained by the passengers.

6.4.2 Architecture 1 - Backend Responsibility

Class descriptions

Figure 6.2 shows the full class diagram of the backend responsibility architecture. The model is split into two different parts; an environment which is used to generate stimuli and a system which react to stimuli. The environment classes includes World, Environment. The remaining classes are System classes. Below, each class and its responsibility will be described, starting with the environment classes.

Record Types:

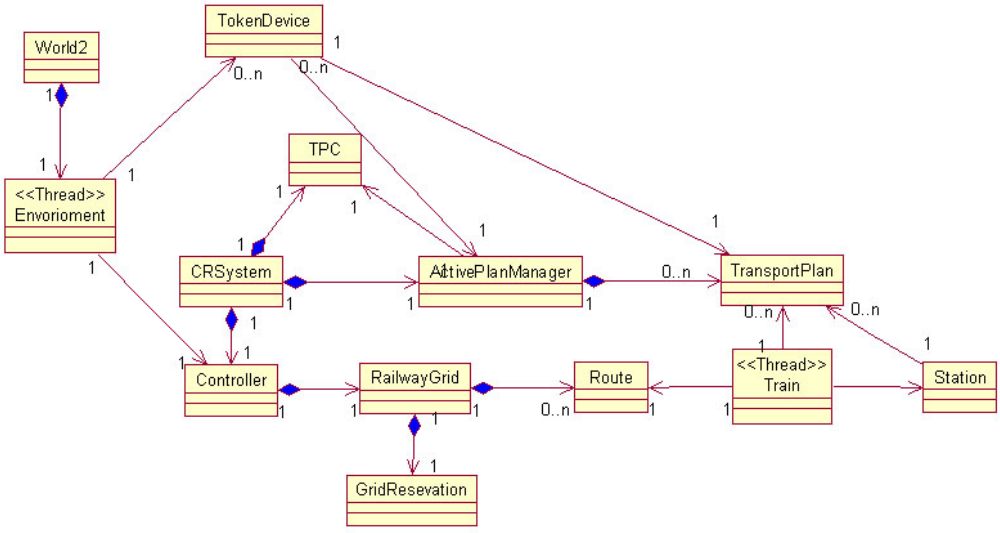


Figure 6.2: Class Diagram: The architecture for the *BR* VDM++ model

NavigationInput: This record contains the information needed for TPC to be able to create an transport plan. The information is, departure- and arrival location, departure time and the id of the token device.

Section: Each instance of a Section indicates a connection between two stations in one direction. So if it is possible to travel from station A to B and B to A there will be two Sections to indicate this. Section contains a departure- and arrival location, fee, departure- and arrival time, platform*, duration and a section ID.

TransportPlanDTO: This record type is used to transfer the TransportPlan by value from TPC to APM. VDM++ objects are always transfered by reference between CPU and to transfer by value, a record type must be used. This ensures that a copy of the object is sent and not a reference.

Environment Classes:

World: This class handles all the initial initializing for all objects in the model. It makes sure that all static system classes are initialized correctly. It also creates a point of entry to the model when is used for validating the model.

Environment: This is an active object and it is responsible for reading input from a stimuli file and input these into the system at a predefined time. Environment has access to two system classes; TokenDevice and Controller. Environment uses these objects an entry points for stimulating the system.

*This is the platform the train is at before departing from the departure station

System Classes:

CRSystem: This is the system class that contains all the static objects. It is also the CRSystems responsibility to deploy the static objects on CPU's and connect these CPU's with BUS'es.

Controller This object receives input from the environment specifying if a section has become inactive. If this happens, the Controller will notify TPC which will update the railway grid. It will also inform the APM about the inactive section. Another responsibility of the controller is to initialize the railway grid, the route plan and the trains.

RailwayGrid: This class initializes all the route plan and create the railway grid.

GridResevation: This class makes sure that no train will be traveling on the same section at the same time. In essence the object calculates a reservation time table, which the trains use, based on user defined parameters.

Route: This class contains a sequence of sections, and some departure information. A route will be assigned to a train, and a train will travel all the sections as defined in the Route.

Train: The train class is an active object and its main responsibility is to follow the route it has been assigned from controller. During the traveling of the route, passengers will board and disembark when the train arrives at a station. Depending on where the train is heading, passenger will board if it is specified in their transport plan.

Station: This is a container which has passenger waiting for a train to come and pick them up. The station class is a monitor object [MK99] between the train threads and triggered by the train objects.

TokenDevice: This recieves input from the environment and requests transport plans through the ActivePlanManager. The token device is not responsible for the traveling, since this is handled by the trains.

ActivePlanManager: This is the class that maintains all the transport plans. When a new plan is requested and activated it is stored here. If a section becomes inactive it is the ActivePlanManagers responsibility to find all the plans affected and provide a new plan to the affected token device.

TPC: TPC or Transport Plan Constructor, only has one responsibility and that is to create transport plans from the input it receives from the ActivePlanManager.

TransportPlan: A transport plan contains all the information which is needed for a passenger to travel. If a passenger is assigned a new transport plan the old one is disregarded.

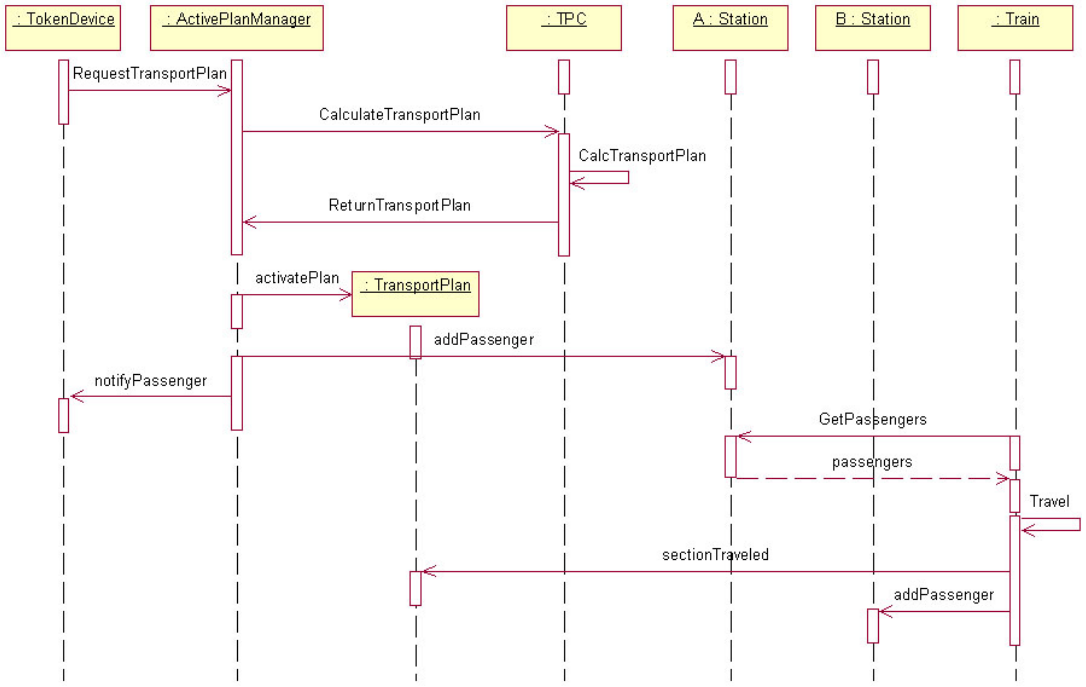


Figure 6.3: Sequence Diagram: Sunshine scenario for the *BR* architecture

Behavior

In figure 6.3, notice that the token device has no responsibility except for requesting a transport plan. Everything else is handle by the backend system. It is also the backend system that knows the location of the passenger, when the passenger has traveled.

An important aspect to notice in figure 6.4, is that when a section becomes inactive, the system will find the affected transport plans and notify the passengers with new transport plans. There is no communication with the token device, until a new plan is calculated and ready to be sent to the passenger.

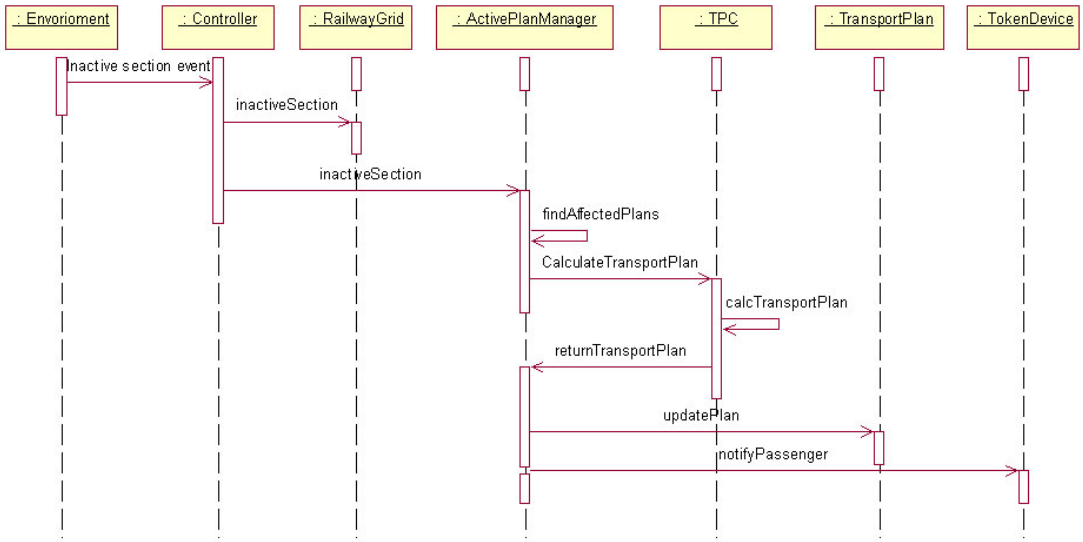


Figure 6.4: Sequence Diagram: Inactive section scenario for the *BR* architecture

Chapter 7

Proof of Concept Implementation

7.1 Introducing the Java Prototype

The purpose of the Java prototype implementation is to make the VDM++ model more accurate. The Java prototype can identify important timing characteristics, which will contribute to the accuracy of the VDM++ model. With the addition of real world timing characteristic, the formal model should behave almost identical to the Java prototype and one argue that the prototype application could just as well be used instead. Using the prototype instead is impractical, since CyberRail involves a large number of simultaneous users and a real world deployment of 1000 or 10.000 users is a very costly and impractical affair. If the VDM model can be made accurate enough, it will be more cost efficient to perform the large scale tests and alter the parameters.

7.2 Deployment

Figure 7.1 shows how the Java prototype is deployed and what technologies have been used to set up the communication. There are two new entities not in the VDM++ Model, these are *UserRegistry* and *TokenServlet*. Unlike the VDM++ Model, passengers do not enter or leaves the trains automatically, since this was abstracted from the prototype implementation. The *UserRegistry* detects when a passenger enters or leaves a train and informs the controller about the state changes. The *UserRegistry* system is a development edition of a FeliCa RFID reader, which actually represents the type of payment system currently being deployed in the Tokyo area. *Token Servlet* is an access point to the backend system, which enables a mobile phone to request transport plans. This is necessary step, due to the J2ME Java platform used on the mobile phones, which does not support RMI. The *Token Servlet* is used as a proxy object, which cleverly translates a HTTP request into RMI method invocation.

Internally in the backend system, i.e. *APM*, *TPC* and the *Controller*, Java RMI is used for inter-communication. The setup between the token device and CyberRail is a bit more

complicated. First the token device uses GPRS to send a request to the a Java servlet. The servlet then converts the request into a Java RMI request, which is sent to the *APM*. When a transport plan has been calculated by *TPC* upon request from the *APM*, it is returned to the token device as a SMS. The token device is also notified by SMS, if changes in the railway grid occurs. The communication between the *UserRegistry* and the *Controller* is a bit complex, since the software API for the FeliCa RFID reader only provided a C++ assembly. Java RMI and .NET technology is not meant to communicate with each other, however an open source project has developed an C# IIOP implementation, which enables .NET languages to inter-communicate with Java RMI [IIO]. A small C# client application was developed for the FeliCa reader, which enabled the *Controller* to communicate with the *UserRegistry* system.

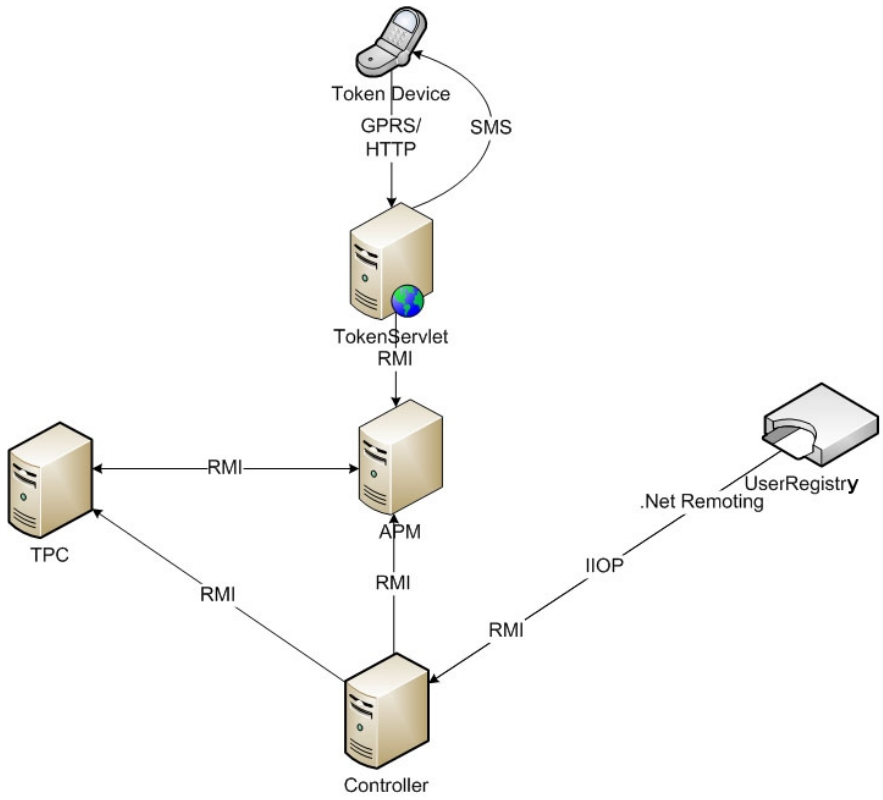


Figure 7.1: Deployment of the Java prototype.

7.3 Realizing the VDM Model in Java

Figure 7.2 shows the packages structure for the Java prototype and significant packages are described in the following sections.

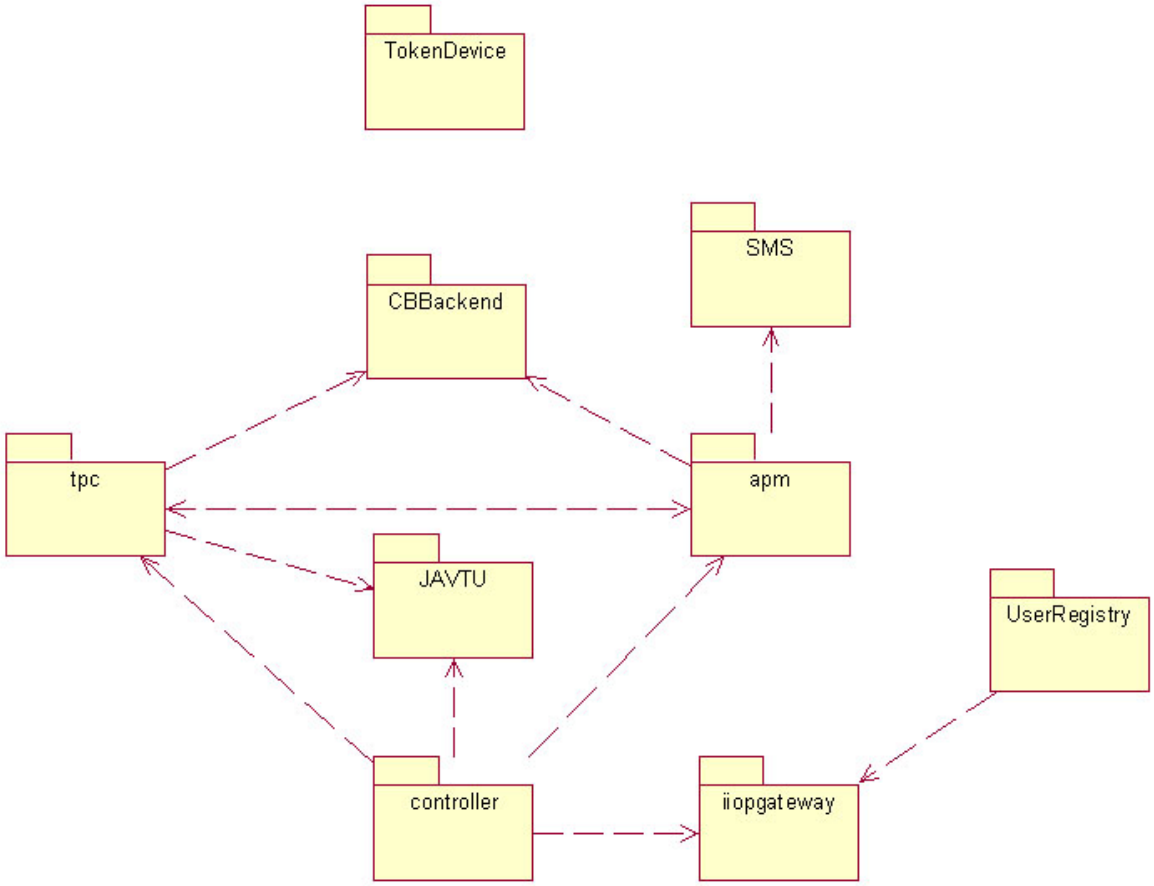


Figure 7.2: Package Diagram: Java prototype

7.3.1 Package: CBBackend

CBBackend is a shared package which means that several different systems make use of this package. It contains the *TransportPlan* class, which is used by both the *APM* and the *Controller* system. The contents of *CBBackend* can be seen on figure 7.3. *CBBackend* contains the classes that are shared between the systems

Class Section: This class is the implementation of the record type in the VDM++ Model with the same name. This class contains equivalent attributes and additionally implements ISerializable which enables an instance of this type, to be send as a stream. In Java RMI, all object implementing the ISerializable interface are sent by value instead by reference.

Class Grid: Grid is a list containing all possible combinations of sections which can be traveled. The data, which make up the section array, is read from files generated by the Real-

Time VDM++ model. The primary role of this class is to parse the data from the VDM++ generated files, to a format which can be used in the prototype application.

Class TransportPlan: This represents the transport plan class in the VDM++ model. It contains a sequence of *Sections* that makes up the plan. When a *TransportPlan* is sent over the network to e.g. a *Station*, then it is send by reference and by not value. This means that when the train instance invokes the *sectionTraveled* method on a *TransportPlan* object, it will be processed as a remote procedure call, just like in the VDM++ model.

Class TransportPlanDTO: This class is used when transport plans are needed to be send by value instead of by reference. This is necessary between the *APM* and *TPC* system, due to the fact that the transport plans are created on the *TPC* system and on the *APM*.

Class NavigationInput: This represents the record type *NavigationInput* from the model, which is used to request transport plans.

Class Util: *Util* is an utility class and it contains the functionality to perform basic I/O operations.

Class Route: This represents the *Route* class in the model and it contains sequence of *Sections* which make up a *Route* for a train. There can be several routes that has the same sequence of *Sections*, but instead differ in departure time.

Class RoutePlan: This class is responsible for loading all the routes from a file and parsing them, under the same circumstances as *Grid*.

Class RailwayGrid: This class represents all railway grid related information.

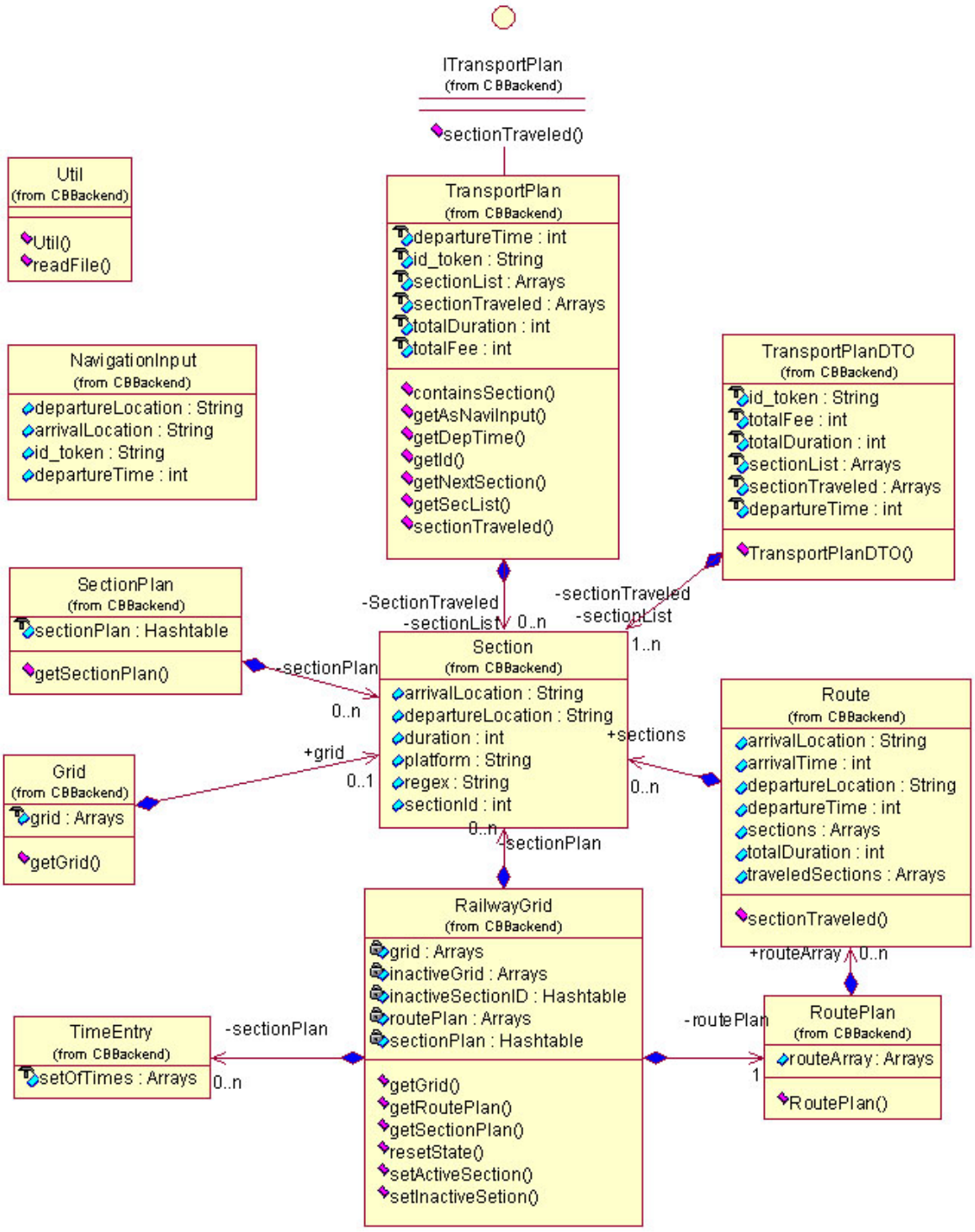


Figure 7.3: Class Diagram: The CBBackend package.

7.3.2 Package: TPC

TPC fulfills the same tasks as in the VDM++ model which is to calculate transport plans and return them to the *Active Plan Manager*. *TPC* implements two remote interfaces, used for RMI communication. These are *ITPCCtrl* and *ITPCAPM*. *ITPCCtrl* is used to inform *TPC* about a inactive route event issued by the *Controller*. *ITPCAPM* is used by the *APM* to request new transport plans. The attribute *grid* contains information about all potential section combinations and *sectionPlan* defines a time table for each section. It is also the *TPC* system that maintains the railway grid information and has the responsibility updating it upon changes.

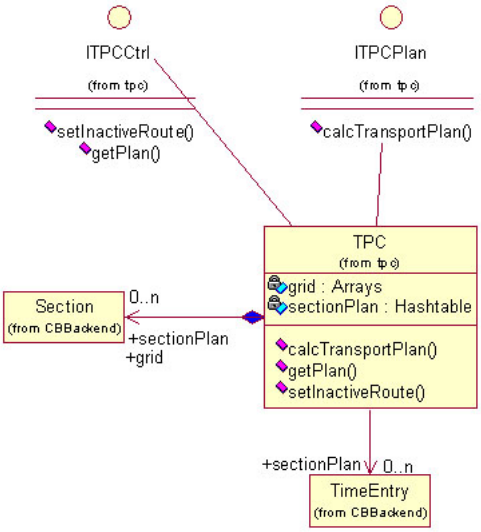


Figure 7.4: Class Diagram: The TPC package.

7.3.3 Package: APM

APM in the prototype has the same responsibility as in the VDM++ model, where it maintains a list of active transport plans and update token devices with a new transport plan if needed. *APM* implements four interfaces; *IAPMCtrl*, *IAPMToken*, *IAPMTPC* and *IAPMTrain*. *IAPMCtrl* is used by the *Controller* to report inactive routes. *IAPMToken* is used by the *Token Servlet* to send transport plan requests from the token device. *IAPMTPC* is used by *TPC* to deliver transport plans to the *APM* system upon request. *IAPMTrain* is used by the *Controller* and the *Train* objects to notify the *APM* when a passenger has traveled a section. The attribute *tokenList* list of token device ID's and the associated transport plan and *stationList* is a list of all the train stations.

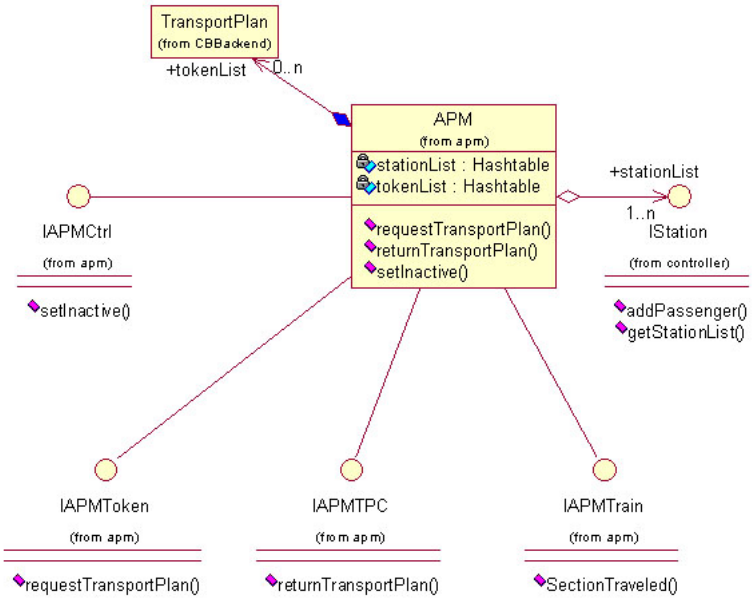


Figure 7.5: Class Diagram: The APM package.

7.3.4 Package: Controller

The *Controller* is somewhat different than in the VDM++ model. It still contains the trains and stations and is still responsible of state changes on the railway grid. Unlike the VDM++ model which contained an *Environment* class that was responsible for stimuli, section state changes and passenger entering and leaving the trains, are now handled by means of external input. *Section* state changes are handled by simple console input, where as the passenger changes are handled by input from the FeliCa system called *UserRegistry*.

7.3.5 Package: Token Device

As shown on figure 7.2, token device does not depend on any of the other packages. Figure 7.7 shows the contents of the *TokenDevice* package. There are three classes; *TokenDeviceMidlet*, *WebToMobilClient* and *Reader*. TokenDeviceMidlet is the main application used on the mobile phone and it uses the *WebToMobilClient* as proxy class to communicate with the *TokenServlet* entity, as defined in figure 7.1. The *Reader* entity is an internal* class in *TokenDeviceMidlet* and it is responsible for reading incoming SMS messages on a specific port.

7.3.6 Package: JAVTU

The *JAVTU* package contains a distributed logging framework developed during this thesis. *JAVTU* is utilized by all systems except for the *Token Device* and the *UserRegistry* system.

*An internal class is a class that exists in inside of another class and has access to private data/operation.

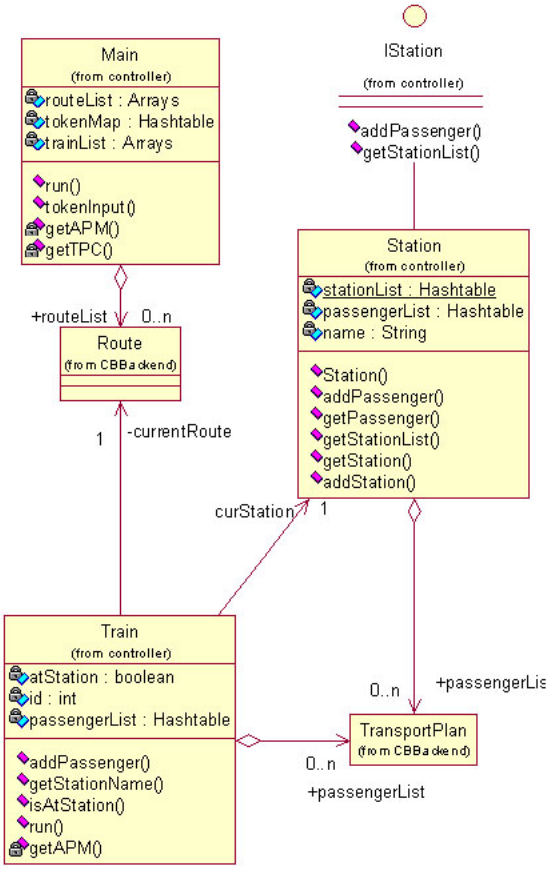


Figure 7.6: Class Diagram: The Controller package.

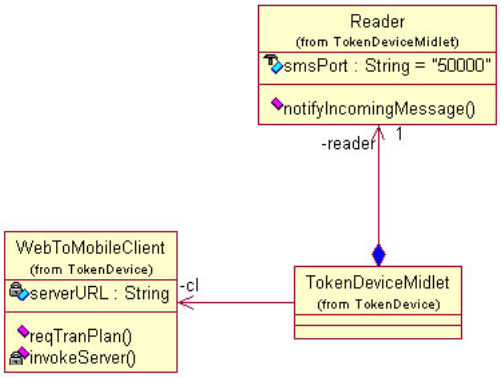


Figure 7.7: Class Diagram: The TokenDevice package.

For more information regarding the internal structure and behavior of this framework, see appendix A.1.

7.3.7 Package: SMS

This package is a utility package for the *APM*, it contains the logic needed for sending SMS's through a GSM modem. It makes use of a third party library called *smslib*. The *CService* shown on figure 7.8 is a class from *smslib* that handles that actual act of sending the SMS.

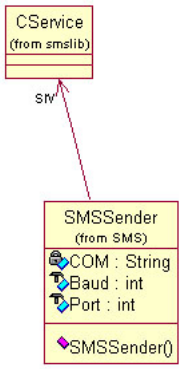


Figure 7.8: Class Diagram: The content of the SMS package.

7.3.8 Package: iiopgateway

This package contains the Java implementation for the setup that makes it possible for the C# program to make Remote Procedure Calls to the Controller.

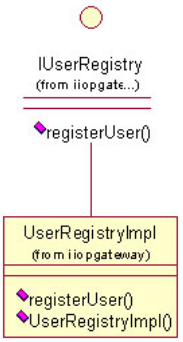


Figure 7.9: Class Diagram: The iiopgateway package.

7.3.9 Package: UserRegistry

UserRegistry is developed in C#, unlike the other packages. Figure 7.10 shows the content of the package. There are two classes supplied by the FeliCa development kit. These are FeliCa_dll_wrapper_basic and HandleContainer, the other three are custom implementations. FelicaMain is just a startup class. It initializes and then starts ContFelicaReader. FelicaReader is utility class that contains the logic needed to read the FeliCa cards. An activity diagram showing the behavior of the *UserRegistry* can be seen on figure 7.11.

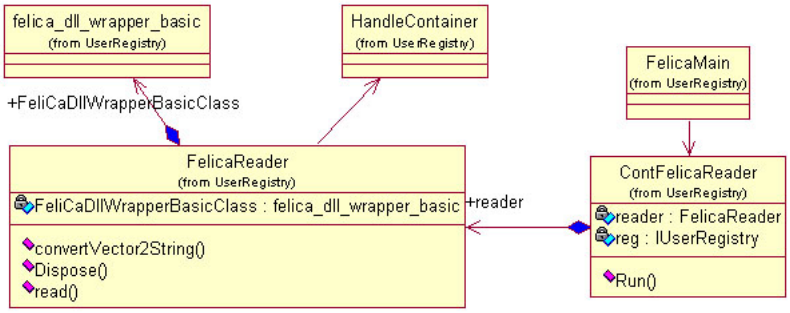


Figure 7.10: Class Diagram: The UserRegistry package.

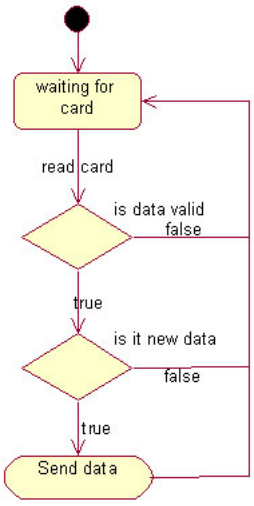


Figure 7.11: Activity Diagram: UserRegistry.

7.4 Testing the Java Prototype

Before a thorough test of the VDM++ model can be completed, we need to obtain timing characteristic from the JAVA prototype implementation. This information will be used to fine tune the VDM++ model and make it more realistic.

7.4.1 Test Tools

JAVTU Framework When working with a VDM++ model a trace log file is generated as it is interpreted. This trace log contains all relevant information regarding object activity, CPU and BUS communication and instance variable state information. This type of information is very informative in a distributed system, both for analyzing round trips and performance but also when debugging an application. Furthermore the existing tool, Showtrace from the Overture community, is a convenient way of viewing this type of data and for this reason a special JAVA trace log framework has been developed - JAVTU see appendix A.1. This enables the JAVA prototype implementation to be analyzed in a similar way as with the VDM++ model. JAVTU is also responsible for synchronizing the distributed systems, so they have a consistent sense of time. This step is crucial for the test and little abbreviation will result in inaccurate results.

JAVTU is implemented in the backend system and will register all internal characteristics, from the point of transport plan request and inactive route, to the return of a transport plan to the passenger. The JAVTU Framework will not operate on the J2ME platform due to a number of issues, but mainly because of the missing marshaling components and the lack of Java RMI support in J2ME. This means that timing characteristics and time synchronization must be performed manually prior to the test phase. Due to the expected high response times between the backend system and the token device, a relative high time synchronization inaccuracy, a maximum of +/- 1000ms, between the token device and the backend system is tolerated. The mobile phone will have to be synchronized manually with the IEEE1588's master clock. Timing characteristics will be captured and reported back to the servlet and persisted to the local file system - see figure 7.12. Note the fake *TokenDevice* object which is deployed. This is used by JAVTU in the post merging process to assign the receiver object reference.

To avoid time consuming redundant code in the Java prototype, all state information regarding railway grid, route information, time table etc. will be generated in VDM++ and read into the JAVA prototype at runtime (see figure 7.13). This will also make sure that the JAVA prototype operates on the exact same static information as the VDM++ model. The following information is read into the system.

Route information All available routes which can be traveled (filename: *routePlan.txt*).

Grid information All potential routes which may or may not be traversable. These are used to calculate alternative route compositions i case of a section breakdown. (filename: *prettyGrid.txt*).

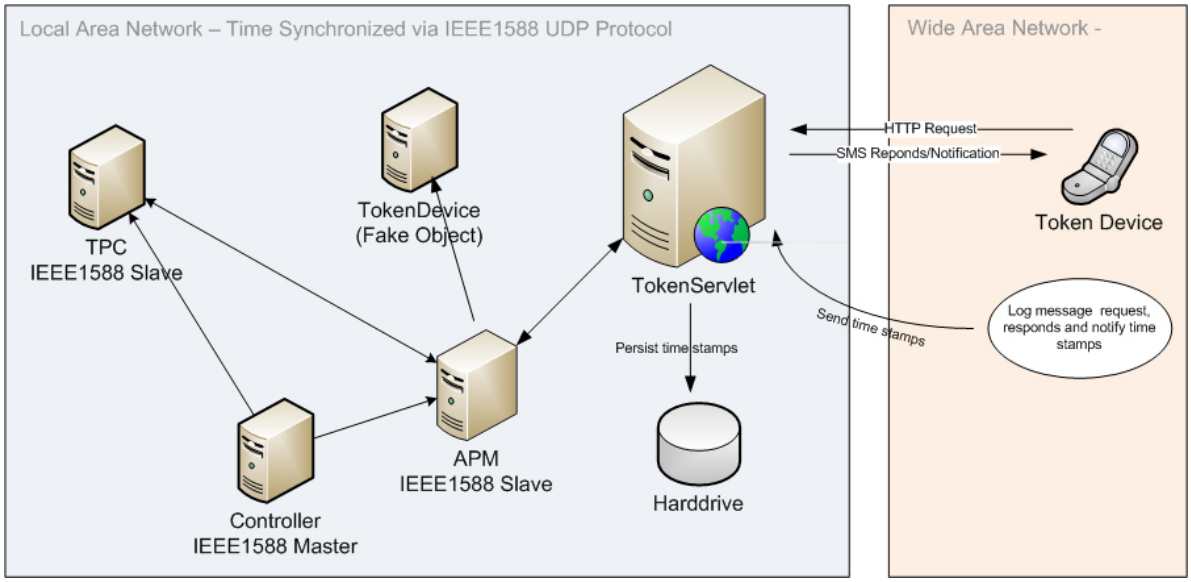


Figure 7.12: Deployment Diagram: Actual deployment for the test.

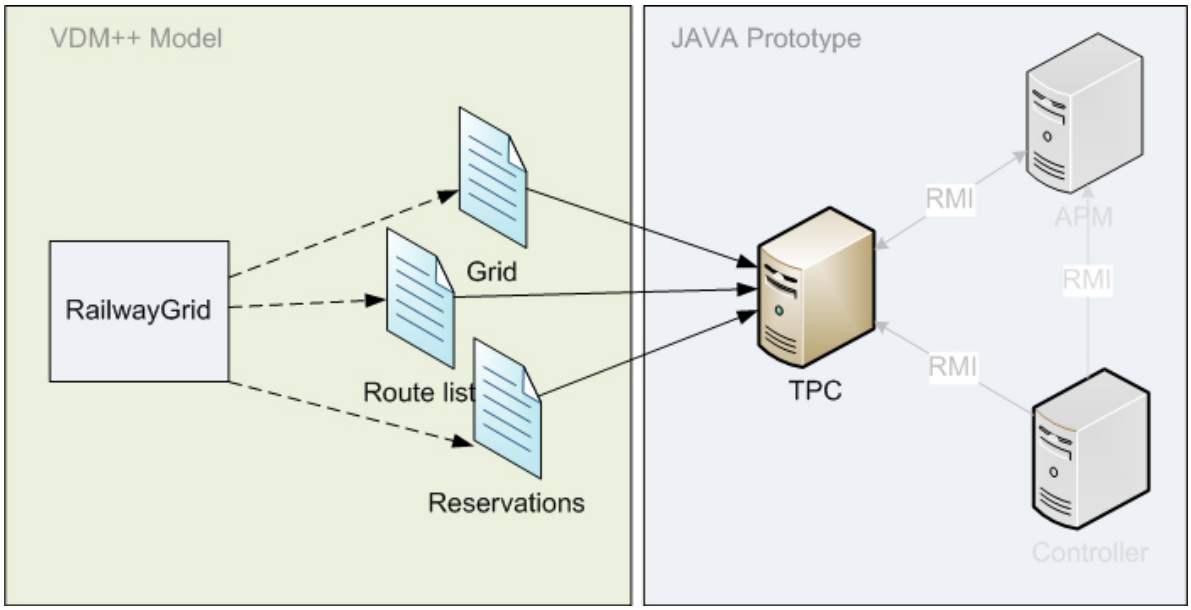


Figure 7.13: Grid information is reused in the JAVA prototype.

Reservation Information Information telling when a specific section is occupied. This data is dependent on the number of trains and number of occurrences the section will be traveled (filename: *prettyReservations.txt*).

7.4.2 Test procedure

The testing procedure is not simple, however a sufficient complex scenario will test all important timing characteristics of the system. The purpose of this test is to identify key time characteristics to optimize the formal model and certain details can be omitted to keep it simple; The railway grid composition is fairly simple with only 4 stations; all the traversable sections all take 200.000 ms to travel; only a single passenger is used in the test. How this prototype performs during a stress test with 10.000 passengers in a complex railway grid with 100 stations is not interesting since it would be very hard to deploy and execute. What is important, is how the time delays affect the performance of the CyberRail system and these results can be achieved with level of abstraction used for this test. The following scenarios are of interest:

Transport plan request Time from when the passenger requests a transport plan until the *APM* unit registers the request.

Transport plan return Time from when the *APM* unit receives the transport plan request until the passenger receives a transport plan.

Request round trip Time from when the passenger requests a transport plan until it is received.

Inactivate route Time from when the backend system receives an inactive route notification, until the passenger receives an updated transport plan.

Additional timing characteristics, such as duration for calculating a transport plan and backend intercommunication, are also interesting in this test.

An additional test will be carried out, to see whether or not it is feasible to return the transport plans synchronously from the *TokenServlet*, when requesting.

This railway grid composition and the sections which make up the grid is illustrated in figure 7.14 and figure 7.15 respectively.

Passenger Scenario

The passenger will, from a mobile phone, request a transport plan from station A to station D. The *TPC* unit will either return a route plan A-¿B-¿D or A-¿C-¿D which both fulfill the passengers request. Before the passenger starts to travel the route, a section is reported inactive. The backend system recalculate a new transport plan, which can either be A-¿B-¿C-¿D or A-¿C-¿B-¿D. This scenario is repeated 2 x 15 times to give a rough estimate of the timing delays. A flowchart of the possible scenarios are illustrated in figure 7.16

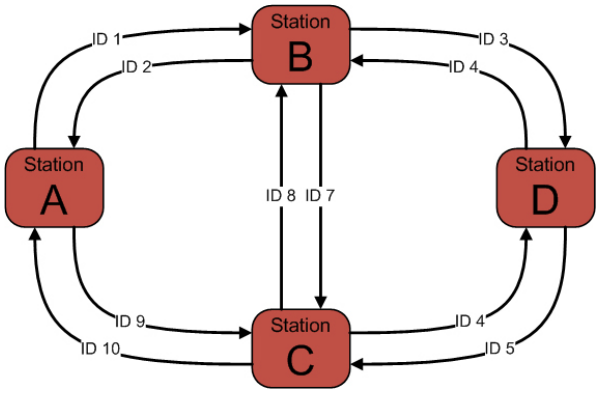


Figure 7.14: Illustrating the railway grid with stations and sections.

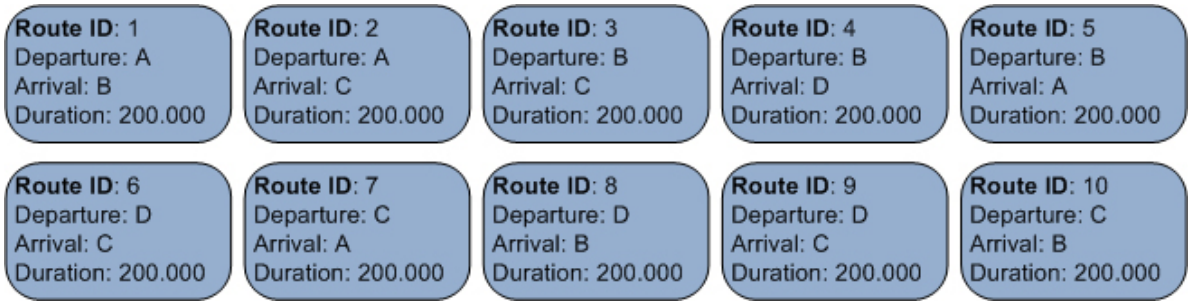


Figure 7.15: The different sections that make up the railway grid.

System	Description
APM	Intel Pentium 4 Prescot 2.8Ghz, 1Gb RAM
Controller	Intel Pentium M 770 2.13Ghz, 2gb RAM
TPC	Intel Pentium III (FC-PGA) 1Ghz, 384 Mb RAM
TokenServlet (Apache Webserser, Axis)	Intel Core Duo T2600, 2Gb RAM
TokenDevice	Sony Ericsson K750i, 105Mhz 1500Kb RAM
Network	Ovislink WMU-9000 switch 10/100Mbit.

Table 7.1: Deployment equipment used for the prototype test

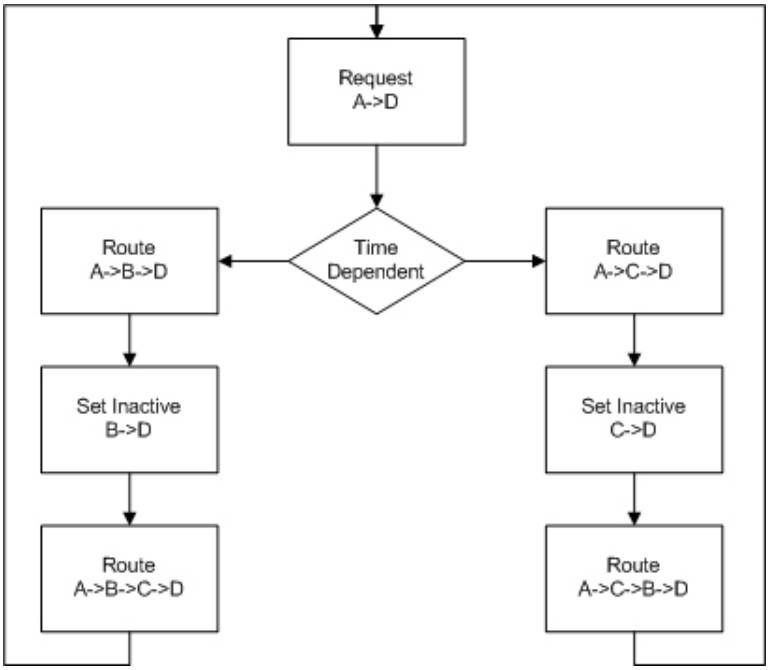


Figure 7.16: Flowchart: Illustrating the test scenarios.

7.4.3 Analyzing the Results

The Java Prototype test will contribute with timing characteristics for two different aspects; performance and operation execution times. Performance is analyzed to ensure that the technologies used are actually feasible and that the Java Prototype is performing within what can be expected. The second part, operation execution times, is done to make timing characteristics for the Real Time VDM++ Model.

Performance Analysis

Test #1 *TPC-¿TPC* is time from *TPC* receives a request until the a transport plan is calculated. This result is important when 40.000 concurrent users are using the system where this operation may be invoked many times. However, if deployed realistically, *TPC* would be a complex database structure containing the route information and probably have a different timing characteristic. However the time identified, 3.13 ms. in this test, is realistic compared to the VDM++ model.

Test #2 *APM-¿TPC-¿APM* is the time from which *APM* receives a request from a token device, request a transport plan from *TPC* and receives the new transport plan. The relative high time delay can only be a result of the Java RMI overhead, which was anticipated.

Test Id	Trace	Result
1	TPC-¿TPC	3.13 ms
2	APM-¿TPC-¿APM	24 ms
3	Ctrl-¿APM reply inactive TP.	26 ms.
4	TD-¿TD	13693 ms.
5	TD-¿TD (RAW)	373959 ms.
6	Reply deviations	7.55 %
7	Reply deviations	991902 ms.
8	Reply deviations	16.5 min.
9	Ctrl-¿TD inactive TP	9653 ms.
10	TD-¿TD GPRS Reply	4347 ms

Table 7.2: Table: Results from the prototype test.

Test #3 *Ctrl-¿APM reply inactive TP*[†] is very interesting since this trace will show how fast the backend system will react upon an inactive route stimuli. The trace describes the time delay from when the *Controller* receives the inactive route stimuli until the *APM* sends an updated transport plan to the token device. The 26 ms. will increase depending on the number of traveling passengers, however this time delay can easily be scaled in the VDM++ model.

Test #4..8 *TD-¿TD* is one of the primary traces and one of the most important. This trace tells how long it takes for a transport plan to reach the passenger using the SMS technology. The 13693 ms. is an acceptable response time for a system of this nature. However a number of deviations, **Test #6** - 7.55% in total, did occur and if accounted for, the average responds time is 373959 ms.ẇhile the average delay time of all the deviations reaches a consistent 991902 ms. (16.5 minuttes). This is however not an acceptable responds time and 7.55% is too frequent. Further reseach into this odd tendency, revealed that each service provider implements their own retry algorithm [Hug07], which specify how fast a carrier will resend the SMS upon initial failure. In this case the service provider TDC[‡] seemed to implement a retry algorithm which does not retry until 16.5 min. after the initial try. This is unfortunate on several levels. First of all the carriers does not seem to be very consequent prioritazing SMS retries, which will make the CyberRail experience quite different from passenger to passenger depending on which mobile service provider used. E.g. the US carrier NextTel uses a different approach which is far more agressive and suited for the CyberRail concept[Hug07]. The SMS retry algorithm is illustrated in table 7.4.3

The strategy NextTel implemention is far more aggressive than the danish service provider TDC. The difference might be a result of the popularity of SMS communication in the US and Europe, where European users utilizes this service far more than

[†]TD: TokenDevice, TP: TransportPlan, Ctrl: Controller

[‡]See http://www.tdc.dk.

Initial attempt	Immediate
Retry 1-15	1 min. from last delivery attempt
Retry 16-25	3 min. from last delivery attempt
Retry 26-50	5 min. from last delivery attempt
Retry 51-99	1 hour from last delivery attempt
Retry 100-124	8 hours from last delivery attempt

Table 7.3: Table: NextTel SMS retry algorithm

users in the US. The lower usage could justify the more aggressive approach, however this is not confirmed and is only speculations.

Test #9 *Ctrl-¿TD inactive TP* is the responds time from when the *Controller* initially receives an inactive route stimuli, until the token device is notified with an alternative transport plan. The 9653 ms. is a very acceptable responds time, but will most likely be dependent on the total number of passengers using the system. The time delay can be used as an indicator to what responds times a reachable with the some what optimal conditions. To reach this performance level is a question scaling the system according to the number of users.

Test #10 *TD-¿TD GPRS Reply* is the alternative method, where the transport plan is returned synchronously from the *TokenServlet* upon request and only using the SMS technology, when an inactive route stimuli has occurred, to perform the callback notification. The result of this test is very interesting due to the apparent efficiency. With an average responds time of 4347 ms. makes this approach more than 300% more efficient than using SMS. Using a GPRS solution is, in most condidions, also more reliable and during the test, there were no deviations at all. The result however is dependent on the network usage, and GPRS data request may be down prioritized in favor for speech communication. Normally at least one time slot, out of a total of 8, will be available for GRPS data transmissions [Sta02]. Returning synchronously from the servlet seems like a viable option and alternative to SMS. However callback is not feasible using this paradigm and SMS will have to be used to notify the passenger upon changes. If a complete GPRS solution was to be used, a poll strategy should be implemented. More about this issue will be dicussed later.

Source of Errors

Backend frontend time difference Due to the inability to use *JAVTU* on a J2ME platform, a maximum of +/- 1000 ms. time difference may be present between the token device and the *Controller*. This is acceptable since the communication between the backend and the frontend does not rely on any hard Real-Time requirements.

Service provider network traffic The prototype test was not performed in a 100% controlled environment and statistical information about the current traffic load on the

service providers GSM network was not available. The amount of traffic present during the test will definitely affect the results and the behavior. Since the prototype test only involved timing characteristics for a single passenger, it is possible to assume that the results are sufficient to convert and use in the VDM++ model. These results does however not tell anything about the performance during peak periods.

SMS Retry Algorithm In case of using SMS for all communication, the passengers experience is greatly dependent on the SMS retry algorithm strategy. No concrete information about TDC's strategy was to found and the deviations in the result may be caused by a third parameter.

GPRS time slots When using GRPS to synchronously return a transport plan, the results are highly dependent on the number of time slots available for GPRS communication, which is dependent on the total number of active users. The test was not performed during peak periods and it is assumed that one or more time slots where available throughout the test.

Operation Execution Times

It is not for all operations that execution times has be measured. The operations that have been measured are the ones that has an affect on the system, when the number of users increases. Operations not affected by this, is not of interest. Table 7.4.3 shows the measured operations and their average execution times.

Id	Trace	Result
1	APM.requestTransportPlan()	1.5 ms.
2	APM.returnTransportPlan()	15.7 ms.
3	APM.inactiveSection()	4.9 ms.
4	TPC.calculateTransportPlan()	3.13 ms.

Table 7.4: Execution times for selected operations

APM.requestTransportPlan() The 1.5 ms execution time of *requestTransportPlan* can seem like a long time to make a request to *TPC*. Most of the measured times for this operation were below 1 ms. going toward zero. However, it took 15 ms. to execute the operation the first time, which was the result of setting up the connection with *TPC*.

APM.returnTransportPlan() 15.7 ms average for *returnTransportPlan* seems reasonable as this operation has several different tasks. It has to create the *TransportPlan* from the *DTO*, serialize the transport plan and send it as an SMS. The 15.7 ms. does not incorporate the time the SMS takes to get to the token device. But it does incorporate the time it takes to deliver the SMS to the GSM network.

APM.inactiveSection() With an average execution time of 4.9 ms. it seems a reasonable execution time. As *inactiveSection* is responsible of finding all effected passengers, the

operation will be dependent on the total number of active token devices and this will impact the timing characteristic. This test only used a single token device and the execution time for 10 passengers would in worst case take about 10 times as long.

TPC.calculateTransportPlan() The execution time of 3.13 ms. is higher than expected, specially because the railway grid is not very complex. No search optimizations have been made to aid the calculation of transport plans and this timing characteristic may not be reliable to use in the VDM++ model.

Source of Errors

Millisecond measurements All the execution measurements has been done in millisecond resolution, which means that any time differences lower than 1 millisecond will appear as 0 ms. Higher accuracy could be obtained by changing the time resolution of JAVTU.

7.5 Results

7.5.1 Callback to mobile phones

The test of the prototype had some interesting results which indicated that different approaches, for inter-communication between the front- and backend system, could be used. This section will discuss the different calling semantics identified, which is a result of the prototype test.

Push or Poll

When deciding how communicate with the token device, there is two paradigms to consider; push or poll. The advantages of using a push model is that you only generate traffic when it is needed and it is asynchronous. Implementing a push model on the other hand, means that the sender must know all the interested receivers. The push model can be troublesome to realize, depending on which technology is used. The poll paradigm provides more control of when information can be acquired, however this approach often generate more unnecessary traffic (see figure 4.5 for more information). Some of these issues are discussed in following sections.

SMS - Push Model

During the design phase of the Java prototype, we had to decide how to return the transport plans, in a asynchronous way, to the token device. SMS was a rational choice, due to its extensive usage and its fairly reliable behavior. The reliability is assured by the *store and forward* procedure used to send a message and this enables an SMS to be stored until it can be delivered.

The test of the Java prototype, suggests that SMS technology is fast enough to deliver a transport plan most of the times. More than 90% of the times, the transport plan was delivered within 15 seconds, however 7.5% of the times, it took more than 16 minutes for the

transport plan arrive. Additional tests of the performance on SMS showed that 73.2% are delivered within 10 seconds, 17% needs more than one minute, 5% needs more than hour and a half and 1.6% is never delivered [ZMW06]. The high failure rate can be explained to the mobile phone being off for long enough for the SMS to timeout. What these numbers show, is that there will be a percentage of the SMS's that will arrive later. In regards to CyberRail, this means that transport plan requests from the token device, and notification upon changes in the transport plan, may be delayed substantially. A transport plan that is 10 mins old, may already be outdated since the train has already left the station. In addition, what happens when a section breaks down, affecting serveral thousand people? Will the local GSM be able to handle it or will the delivery time for the SMS increase even more? It is hard to say what an acceptabel delay is, but even with the numbers identified in this test, one out of every 14 passengers will wait more than 10 minuttes for a response.

GPRS - Poll Model

The advantages of GPRS over SMS, is that the transmission delay is only as big as the line delay. It does not use the *store and forward* protocol like SMS. But unlike SMS it is not possible to contact a mobile phone through GPRS unless it is already connected. For a mobile phone to stay connected for longer periods of time will result in an increase of battery usage and this is undesirable. Cost wise it has no greater impact to stay connected for longer periods of time, as most providers charge GPRS on packet volume and not time. Using a poll strategy could be used with GPRS, where the token device asks for changes every two minutes or so. This would still use more power but not as much as being connected constantly. It would however create a constant delay, but the delay is a known constant period, opposed to the SMS delays.

GPRS/SMS Combined - Push/Poll Model

In CyberRail there is two different scenarios when new information is needed on the token device. The first scenario is when a new transport plan is requested by the token device and the second one is when a section becomes inactive and the token device needs to be updated with a new transport plan. Up until now, these two cases has been solved by using the same communication platform. Another solution is to use GPRS synchronously to return the transport plans to the passenger and SMS when notifying passengers about updates. This does not solve the delay problem when notifying passengers, however it makes the process of requesting a new transport plan more efficient.

Chapter 8

Analyzing the Real Time VDM Model

8.1 Configuring the Timing Characteristics

There are three steps in the process of setting up a VDM++ model with timing characteristics. The first is getting some execution times that are close to what you would see in the finished real system. These times can for example be obtained by relying on empirical or statistical data from previous projects and systems. If such knowledge does not exist, a prototype implementations can be used to measure the execution times. In this thesis, a prototype implementation has been used to measure execution times. The second step is to convert the execution times into cycles or a durations, depending on how the VDM++ model is designed. The third step is to insert the times into the VDM++ model.

8.1.1 Calculating the timing characteristics

When calculating measured time into cycles, there are many aspects that has to be taken into account. First you need to define the speed of the processors used, but not all processors are easy to compare in speed. Comparing the speed of two CPU's is not as simple as simply comparing megahertz, due to highly advanced CPU instruction sets, multi core processor architectures etc. All these aspects makes it difficult to determine the speed of processor, as one might be faster than another in certain applications. So when taking measurements it is important to use processors that are comparable in speed.

In VDM++, processors are not declared in megahertz, instead they are declared with a parameter specifying the capacity as instructions per second (IPS). There are two methods of specifying what timing characteristics specific model section should take. Either by using the duration and cycles primitive. Duration will take the amount of time specified, no matter what computational capacity the CPU has. Cycles, on the other hand, is affected by the capacity of the CPU. Cycles indicate how many clock cycles the processor has go through before the VDM++ segment has been executed. For instance, if a CPU declared with a computational capacity of 1 million IPS and a VDM++ segment is defined with a cycles 1 million, it will

take 1000 time units to complete.

Table 8.1.1 illustrates the timing characteristics for the *calculateTransportPlan* within an acceptable range however we decided to use statistical data instead. The operation *calculateTransportPlan* is a well known challenges and deals with the *shortest path problem*. Several studies has done performance testing on different approaches to solve this problems. [HSWW06] [SS07]. Most of these studies are based on Dijkstra's shortest path algorithm and the *High-Performance Multi-Level Graphs* paper by Daniel Delling [DHM+06] suggested that a response time of 70 micro seconds was easily achievable. This operation will use the timing characteristic from Dijkstra's algorithm instead.

When the times are found and the processor speed is known, it is pretty straight forward to calculate the number of cycles for each timing characteristics - simply multiply the execution time [seconds] with computational capacity of the processor [megahertz]. The resulting calculation is illustrated in table 8.1.1:

Id	Operation	Time	Cycles
1	APM.requestTransportPlan()	1.5 ms	4200000
2	APM.returnTransportPlan()	15.7 ms	43960000
3	APM.inactiveSection()	4.9 ms.	13720000
4	TPC.calculateTransportPlan()	0.70 ms.	154000

Table 8.1: Cycles for each operation

8.1.2 Applying the characteristics to the VDM++ Model

Applying the cycles to the VDM++ model is pretty straightforward in most of the cases. In the case of the *inactiveSection* operation, it was a little different, as our Java prototype never had more than one token device connected at any time. The time measured, specifies the amount of time to find one token device. The cycle primitive declared so it triggered for each additional active token device using the system. If 15 active passengers use the system, the cycle will be executed 15 times.

Another interesting area of concern, is the timing characteristics of the SMS communication. During the test of the Java prototype, an average delay of 13.6 seconds were identified. There is no readily available functionality build into VDM++ that can mimic the behavior of a GSM network, however different approaches can be used to mimic this behavior. One way is to pad the messages with additional data with would result in a delivery time of 13.6 seconds. This however occupies the bus the entire period and would make it impossible for anything else to be send on the bus. This is not the intended behavior. Instead an additional class *SMSGateway* was created and introduced into the VDM++ model. This class is used as a proxy to send messages to the token device. It delays the message asynchronously and sends them after a predefined delay. The BUS between the SMSGateway and the receiver must have high throughput to ensure that is does not add an additional delay. This approach worked very well.

8.2 Testing the Real Time VDM Model

Before starting to test it is important to identify which tests should be run. There are three core test scenarios which is the sunshine scenario, the inactive section scenario before traveling and finally the inactive section with a notification. These three scenarios has been used during the development of the model to verify functionality. What differs from the previous test, is that additional token devices are introduced into the test.

8.2.1 The scenarios

Sunshine: A transport plan is requested, and the user travels through to the end of the transport plan.

Inactive section: An inactive section event is reported. After the event is handled, a transport plan, that normally would have used that section, is requested. The transport plan returned will not use the inactive section but still lead the user to the correct station.

Inactive section with notify: A transport plan is requested. While the user is traveling the first section of the transport plan, the next section breaks down. A new transport plan is automatically calculated and the user is notified about the changes. The user then uses the new plan outlined in the route to arrive at the requested destination.

These three test will be repeated for 1, 10, 100 and 1000 active token devices. All token devices will request the exact same thing, at the exact same time. These tests will show how the model reacts when increasing the number of token devices.

8.2.2 Setting up the Model

When deploying an object on a CPU, the object has to be created as a static attribute in the system class. Also CPU and BUS cannot be declared and initialized anywhere else than as an static attribute in the system class. These restriction makes it impossible to make the setup of CPU's and BUS's runtime. To create a test with a 1000 token devices, all setup must be performed manually. This is a tedious task of declaring a 1000 CPU's, a 1000 token devices and a 2000 BUS's, in total 4000 declarations.

To solve the problem a custom tool has been developed which can take a collection of code lines and insert them into a file at a pre-defined tag. It is possible to specify counters, for instance if the repeater where to create 1000 token devices. It would be necessary to specify a counter at the instance name to make them unique, and a counter for the ID of the token device, to make the ID unique as well. This tool is also used to generate the stimuli files, which is used by the *Environment* class to stimulate the system. For more information regarding this tool see Appendix B.

During the construction of the VDM++ model thoughts have been put into how the VDM++ model could be run more autonomously with less input from outside stimuli. In the first versions of the VDM++ model there was no way to simulate travel activity, except from outside stimuli. To solve this issue the train and station entities where introduced and added to the requirements.

The VDM++ model has been deployed as shown on figure 8.1, with every token device getting its own CPU and BUS.

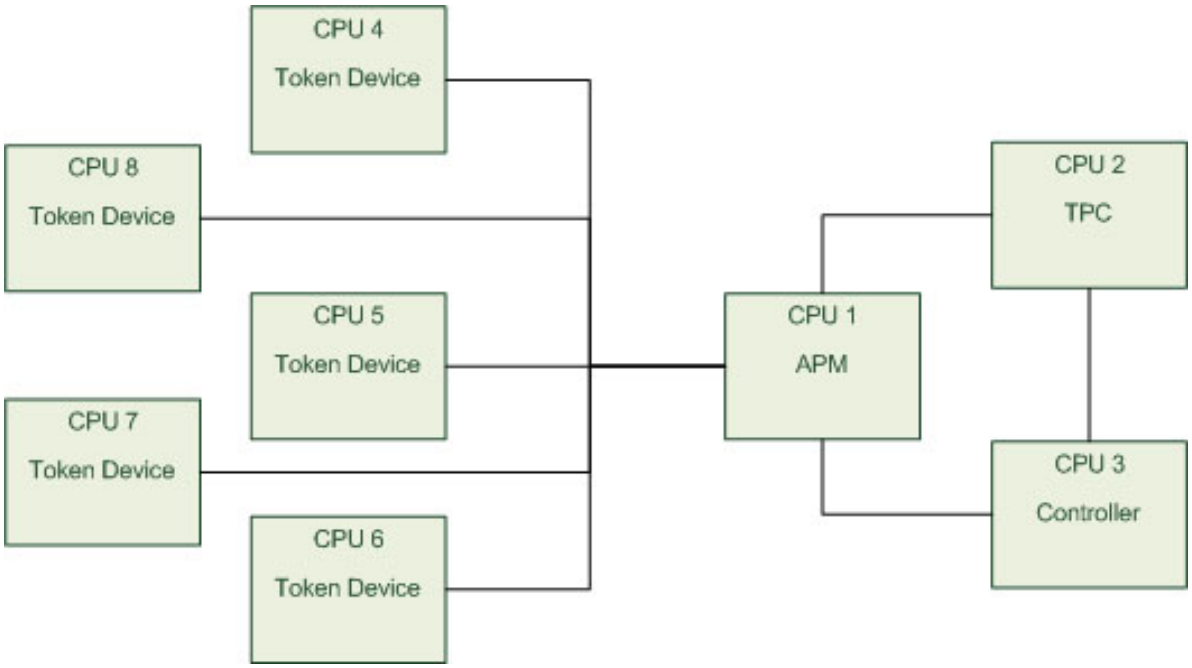


Figure 8.1: Deployment Diagram: The *BR* model deployment

8.2.3 Running the VDM++ Model

Only the tests with 1 and 10 token devices has been completed. The test with a 100 and a 1000 token devices were not able to be validated. Table 8.2.3 shows which tests were completed and which were not. The actual time it took to complete a test increased proportionally with the number of token devices (see Annex #1 for completion times with variable numbers of CPUs).

Test	**1**	**10**	**100**	**1000**
Sunshine	Completed	Completed	-	-
Inactive section	Completed	Completed	-	-
Inactive section with notify	Completed	Completed	-	-

Table 8.2: Completion status

The reason for the failure of completing the tests with more than 100 token devices, relies in the fact that asynchronous execution has not been possible with any of the version of VDMTools VICE used in this thesis. In VDM++ it is possible to make an operation asynchronous so when it is invoked, it starts a new thread. All operations between token

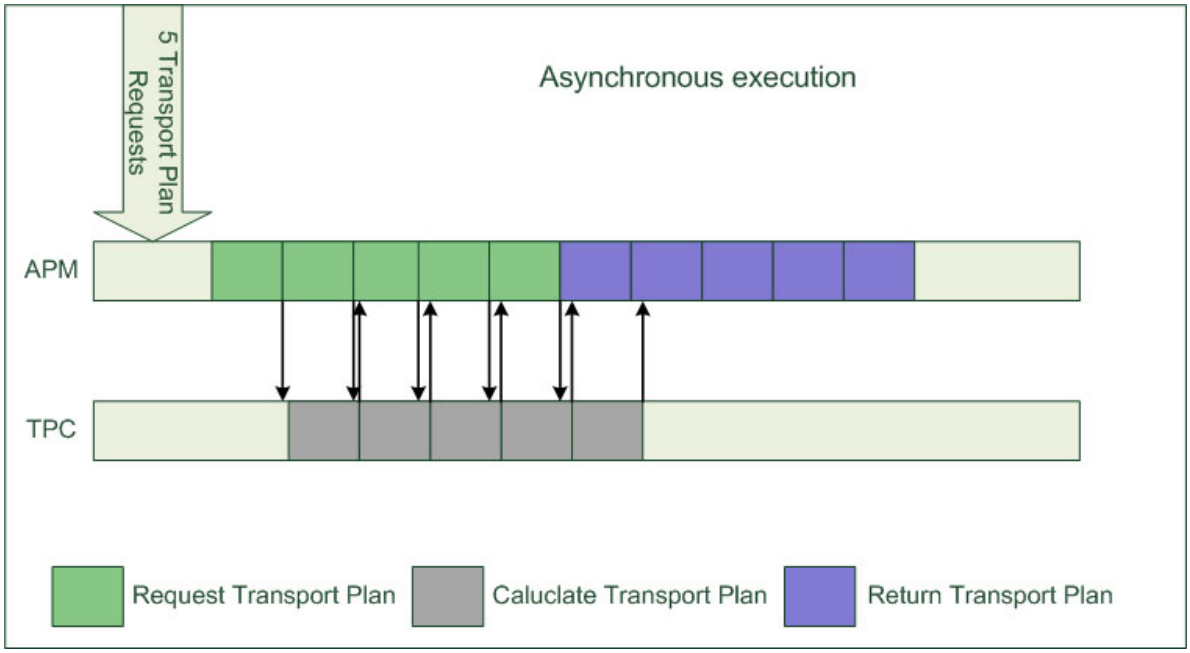


Figure 8.2: Asynchronous execution

device, APM and TPC where declared as asynchronous, this means that when there are simultaneous transport plan requests, they will be executed asynchronously. This is not working at the time of this writing. The only solution to this problem was to use synchronous execution instead. This means that a lot of CPU time is going unused. Figure 8.2 illustrates how a asynchronous system would work and likewise how a synchronous is illustrated in figure 8.3. As can be seen, the asynchronous mode utilizes the processor much more efficient than the synchronous mode does.

8.2.4 Discussion

There are definitely some problems when trying to run large scale tests in VDM++. First there is the problem with setting the deployment details, which was handled by developing the Repeater application. But even with the Repeater, it still takes time to set it up, for each different number of token devices.

Another problem is the time it takes the interpreter to execute a VDM++ model with a large number of CPU's. The tests that was run on a Intel Xeon 2.8Ghz with 4 GB of RAM. The Xeon is a high end server processor and this cannot be identified as the bottleneck. It is not possible to split the work load between several CPUs, as VDMTools VICE does not support multiple processors.

Figure 8.1 shows the deployment of the test setup. Besides the problems with the asynchronous command, the interpreter is highly affected by the numbers of CPU's deployed. A potential way to resolve this issue, and be able to perform a successful test with many token

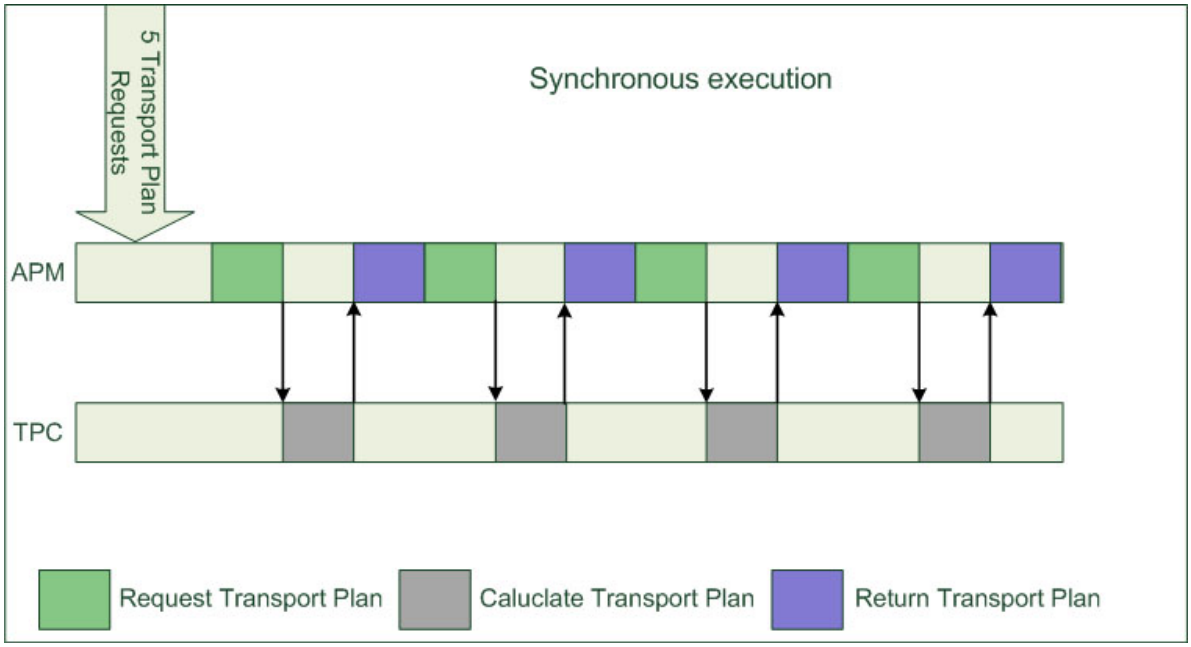


Figure 8.3: Synchronous execution

devices, is to deploy all the token devices on a single CPU defined with a very high computational capacity. This would make some of the results unusable in terms of the backend behavior. The results from the APM and TPC would be usable since the different deployment scheme would not affect their behavior. The setup is shown on figure 8.4. This approach was however not pursued due to timing constraints.

8.3 Analysis of Results

Since the only test which could be completed, were with 1 and 10 token devices respectively, only the test with 10 passengers are discussed in this section. Table 8.3 shows the average times of the performance areas of the VDM++ model. These results can only be used to compare the behavior of the VDM++ model and the prototype. The timing characteristics are very similar to the results from the prototype. The minor differences can be a side effect of environment stimulating the system synchronously.

Trace	**Result**
TD -¿ TD	16164.96667 ms
APM -¿ APM	234.3333333 ms
TPC -¿ TPC	4 ms.
Inactive -¿ TD Notify	18417 ms.

Table 8.3: Average performance times

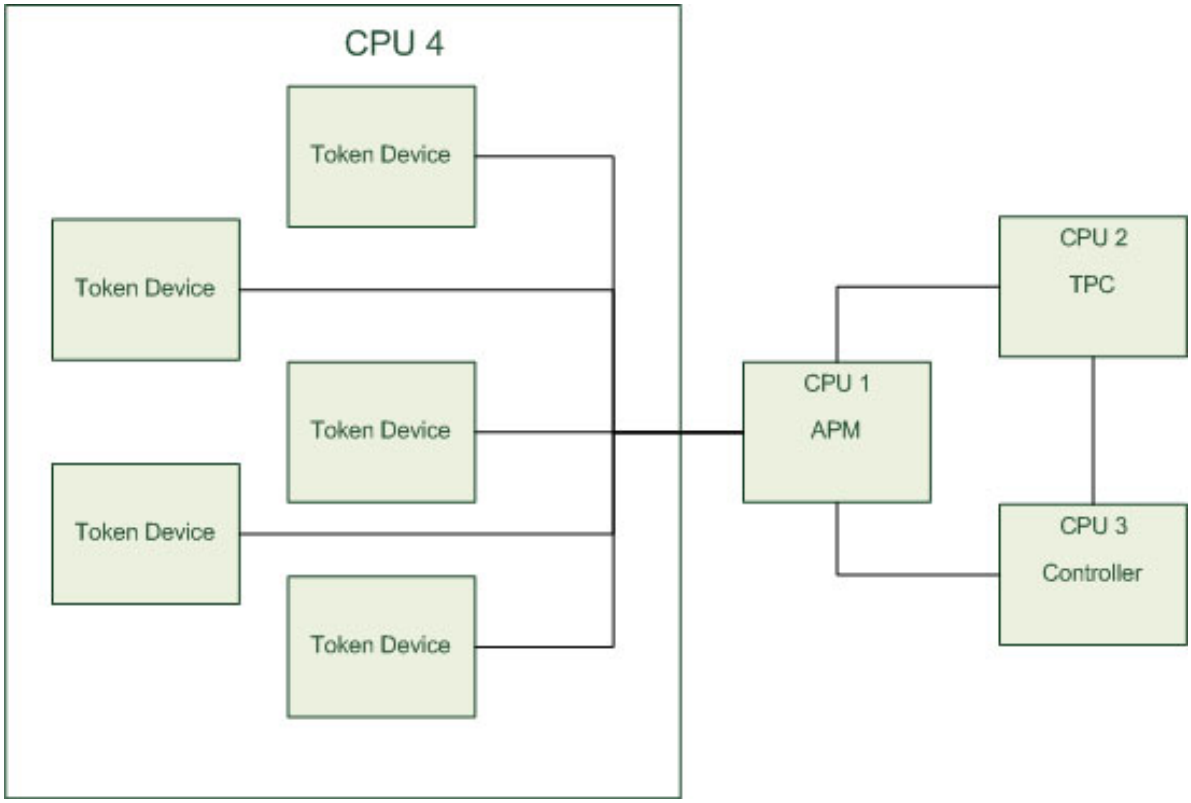


Figure 8.4: Deployment: Alternative deployment for the BR model

The performance times for test 3 is shown in table 8.3. The results described as TD X, are times for requesting transport plans. The results described as TD X IS, is the times that relates to an inactive section event. Here the effect of the asynchronous error is clearly seen, since it takes longer and longer for the token devices to receive a transport plan. This is the same sign that would occur if the system could not handle the amount of request, but then it would also show in the APM -¿ APM times.

Id	TD -¿ TD	APM -¿ APM	TPC -¿ TPC	Inactive -¿ TD Notify
TD 1	13676	213	4	
TD 2	14230	213	4	
TD 3	14784	215	4	
TD 4	15336	217	4	
TD 5	15886	219	4	
TD 6	16434	221	4	
TD 7	16980	223	4	
TD 8	17524	225	4	
TD 9	18066	227	4	
TD 10	18706	229	4	
TD 1 IS			4	13899
TD 2 IS			4	14903
TD 3 IS			4	15907
TD 4 IS			4	16911
TD 5 IS			4	17915
TD 6 IS			4	18919
TD 7 IS			4	19923
TD 8 IS			4	20927
TD 9 IS			4	21931
TD 10 IS			4	22935

Table 8.4: Test 3 performance times

Chapter 9

Alternative Architectures

9.1 Introducing the Alternatives

Not all details for each of the alternative architectures are described in this section. For details regarding identical features, the reader is referred to the *BR* architecture, section 6. The alternative architectures are meant to have the same functionality as the *BR* architecture. They have been designed from the same set of requirements as the *BR* Real-Time model. These requirements are listed in section 6.2.

The method for describing the alternative architectures is the same as the *BR* architecture, the description method is present in section 6.4.1. A lot of the classes used in the *FR* and *JR* have not been change from the form they had in the *BR* VICE and thus is not included in the description of *FR* and *JR*.

9.1.1 Frontend Responsibility

Class descriptions

The special feature of the frontend responsibility architecture, is that the calculation of transport plans are a responsibility of the token device and not the backend system. Additionally a copy of the railway grid state information is also located at the token device. Figure 9.1 illustrates the *FR* architecture.

System Classes:

TokenDevice The token device class pretty much does the same as in the *BR* VDM model. It still communicates with the Controller.

TPM This is a new entity, it has some of the responsibility the ActivePlanManager has in the *BR* architecture. When it is notified about an inactive section it will see if the token device is affected. If it is, it will make sure that a new transport plan is calculated on the token device.

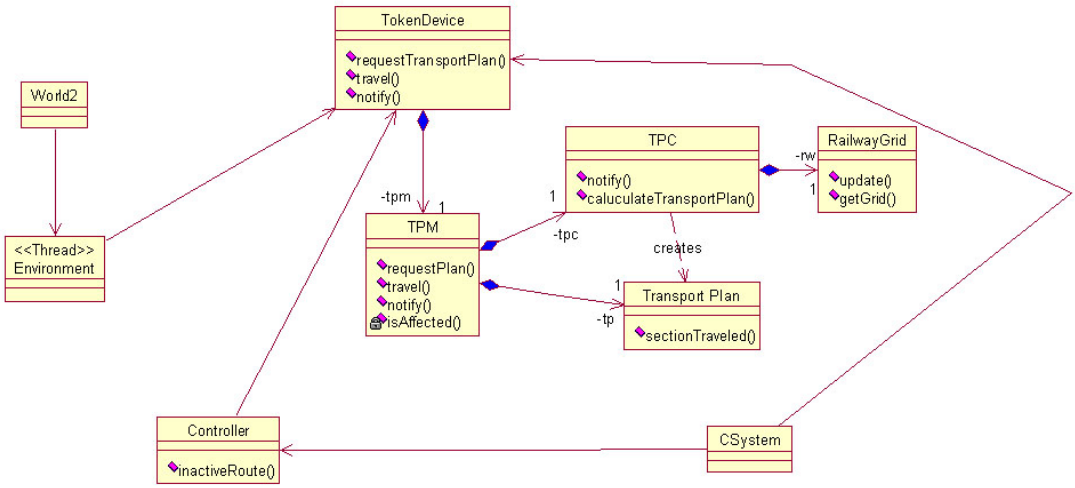


Figure 9.1: Class Diagram: Architecture of the *FR* VDM model

RailwayGrid See *BR*, only difference now is that it lives on the same CPU as the token device.

TPC See *BR*, only difference now is that it lives on the same CPU as the token device.

Controller Since there is no information kept in the backend, there is no need for the ActivePlanManager entity. The Controller is now responsible for notifying the token devices. It will notify all token devices about any changes on railway grid.

Behavior

Figure 9.2 shows the sunshine scenario for the *FR* architecture. What is special about this scenario, is that no communication with the server happens, it is all performed on the token device.

The inactive section scenario is a bit different and is shown on Figure 9.3. When the Controller receives an inactive section, it will notify all token devices even if they are not affected. All token devices will update their internal railway grid. The token devices are there to illustrate that all token devices gets notified. If a token device is affected the TPM will ask the local TPC to calculate a new transport plan

9.1.2 Joint Responsibility

Class descriptions

Joint responsibility has many similarities with the *FR* architecture, such as the calculation of the transport plans which occur on the token device. It is also the token device that keeps track of where it is. Unlike *FR*, some minor state information is kept on the backend system,

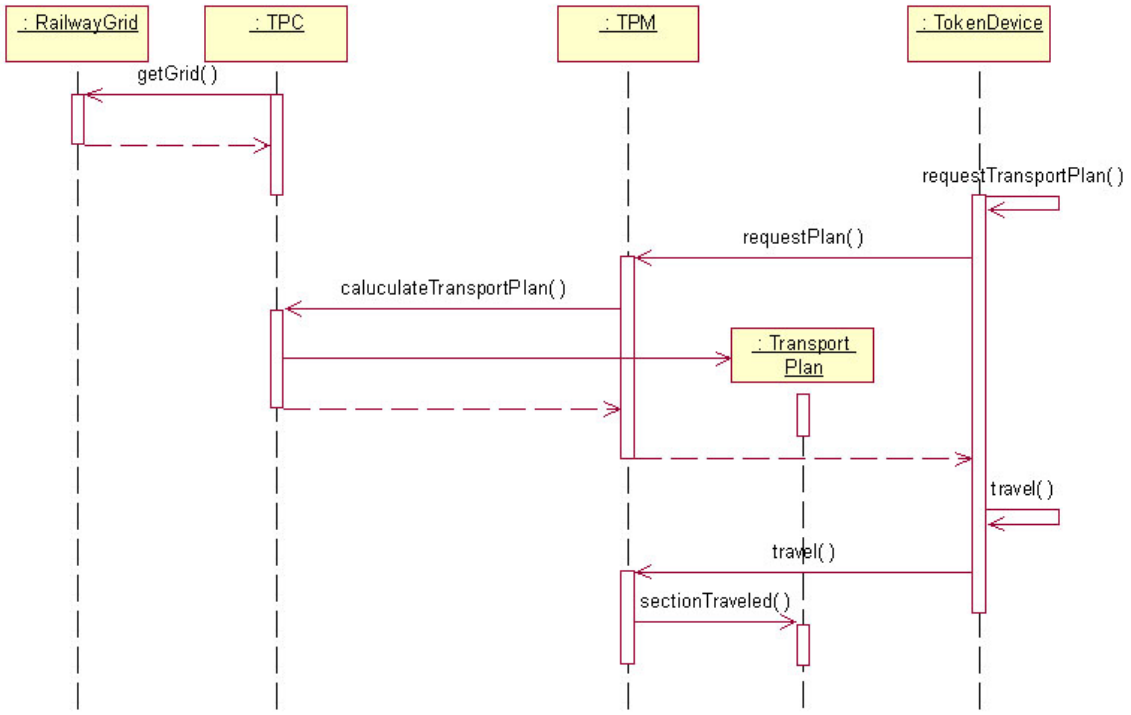


Figure 9.2: Sequence Diagram: Sunshine scenario for the *FR* architecture

which indicate which route the token device currently is traveling. However, the backend system has no information of where the token device actually is. Figure 9.4 shows the class diagram of the *JR* architecture. It is almost identical to the *FR* class diagram.

System Classes

TokenDevice Handles the communication with the *APM* and reports minor information back to the backend system by start-, stop- and update-journey invocations.

TPM See *FR*.

RailwayGrid See *FR*.

TPC See *FR*.

ActivePlanManager Keeps the transport plan information that the token devices sends, when invoking the start- or update-journey operation. If the *APM* receives an inactive section event, the *ActivePlanManager* will look through the active token devices and notify any passengers which could potentially be affected by the break down.

Controller Sends the inactive section event to the *APM* and and the *TPC*.

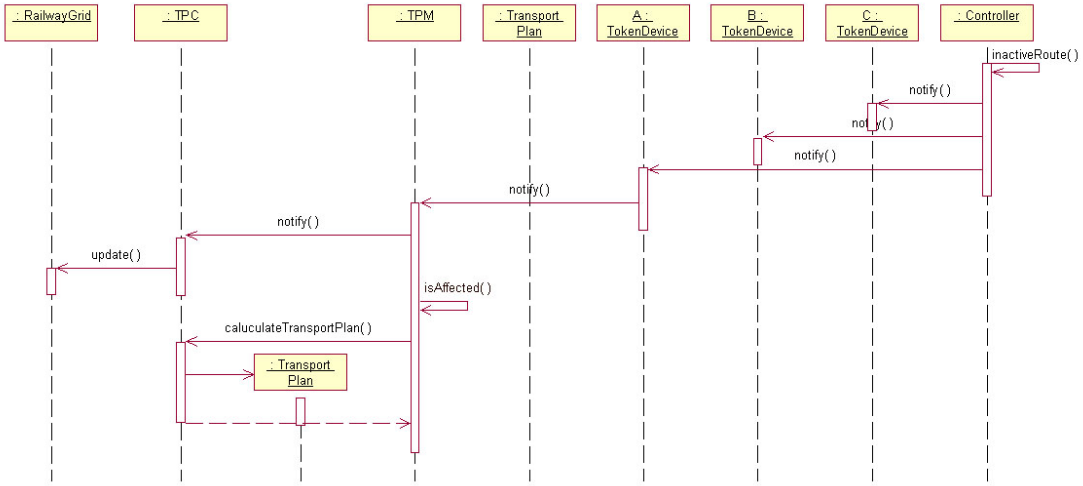


Figure 9.3: Sequence Diagram: Inactive section scenario for the *FR* architecture

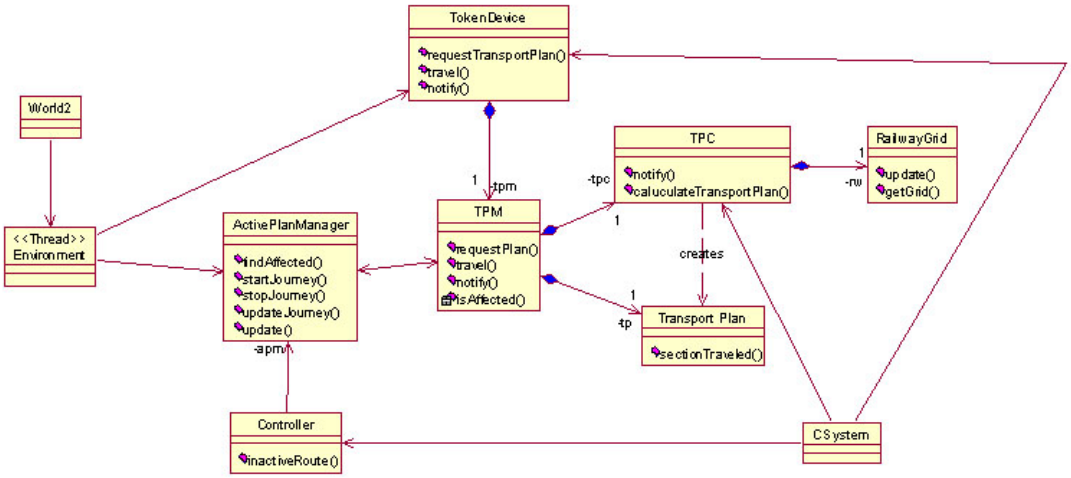


Figure 9.4: Class Diagram: The *JR* architecture

Behavior

Unlike the sunshine scenario in the *FR* model, this architecture communicates with the back-end system. Joint responsibility sunshine scenario is shown on Figure 9.5. When a transport plan has been calculated, the *startJourney* operation is invoked on the *APM*. This method invocation contains a sequence of all the section ID in the transport plan currently being traveled. If any of the sections in the sequence are inactive, a set of all inactive section will be returned to the token device, indicating the transport plan was invalid. The token device

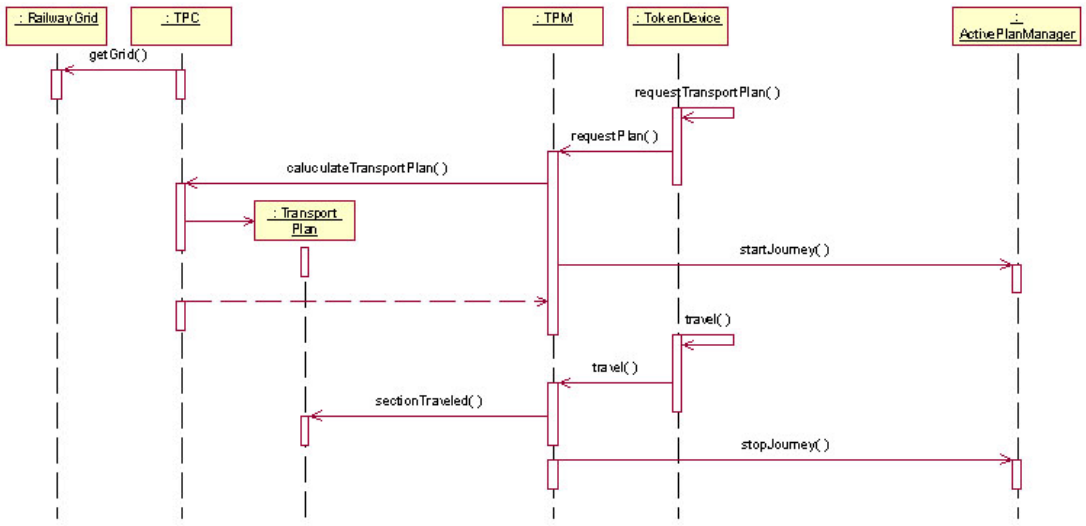


Figure 9.5: Sequence Diagram: Sunshine scenario for the *JR* architecture

then recalculates a new transport plan based on the new railway grid information. When a journey is completed the token device invokes the *stopJourney* operation and the APM will remove the token device and the corresponding transport plan from the active list.

Figure 9.6 shows the inactive section scenario for the *JR* architecture. When the *ActivePlanManager* receives a inactive section event, it will check its list of active transport plans. Whenever an affected transport plan is found, it will notify the token device. When the token device receives the notification it will update the railway grid information and check if it is affected by the breakdown. If it is affected, a new transport plan is calculated and invokes the *updateJourney* operation.

9.1.3 Choosing an Alternative

The *FR* architecture biggest advantage is that it relies very little on input from the backend, this means that each token device is very independent. But it also brings a set of challenges, for instance there is no way of checking whether the transport plan it calculates is valid. This could result in the users getting transport plans with inactive sections. Another problem is how to update the railway grid information for all the affected passengers, if there are several hundred thousand concurrent users. The performance of this scenario will rely heavily on which communication technology used.

The *JR* architecture solves some of the problems the *FR* architecture has, which comes at the expense of extra communication between the token device and the backend system. It does not have the problem with invalid transport plans as all plans are checked by the backend. This approach could turn out to be a bottleneck when large number of notifications are being sent into *APM* which needs to be verified.

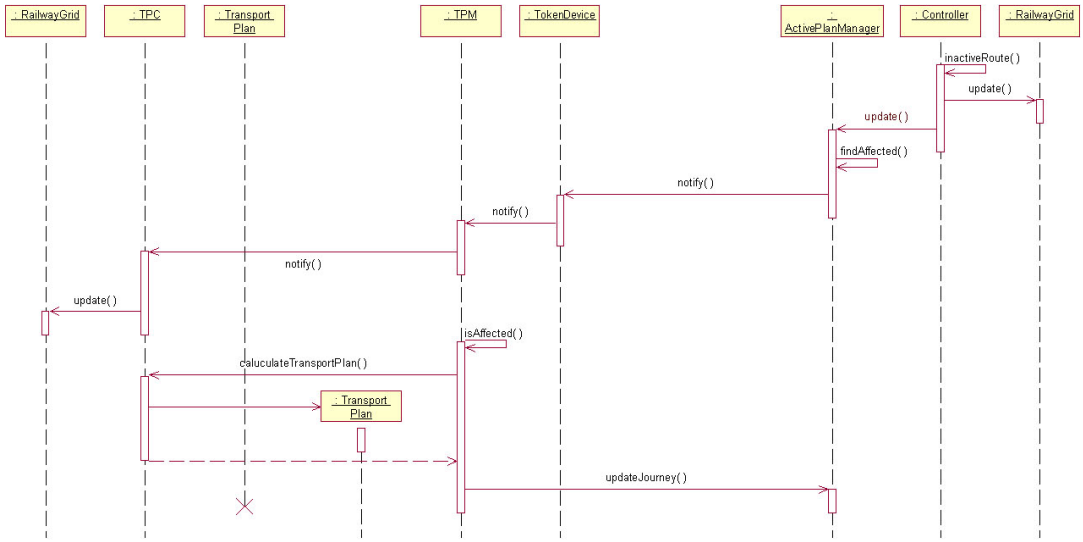


Figure 9.6: Sequence Diagram: Inactive section scenario for the *JR* architecture

The *JR* architecture was chosen to be the alternative architecture to be realized. The choice fell upon *JR*, since it can fulfill all the requirements listed in section 6.2. Unlike the *FR*, it is not possible to guarantee that a valid transport plan is calculated.

9.2 Realizing Joint Responsibility Architecture

The development phase of the *JR* did not follow the same procedure as the *BR* VDM model did, since it was developed using the same set of functional requirements and based on the *BR* VDM++ model. To realize the *JR* architecture, minor changes in the actual model was performed. Most of the changes were simple instance variable declarations and some alterations to *APM* to incorporate the verification functionality. The process of moving the railway grid information and the *TPC* entity to the token device was a fairly simple task and did not impose any major difficulties.

9.3 Testing Joint Responsibility VDM Model

9.3.1 Configuring up the Model

The timing characteristics from section 8.1 have been used in the *JR* VDM model as well.

9.3.2 Setting up the Model

The method and programs that was used in section 8.2.2, has also been used on the *JR* VDM model. See section 8.2.2 for a description. The deployment of *JR* is illustrated in figure 9.7.

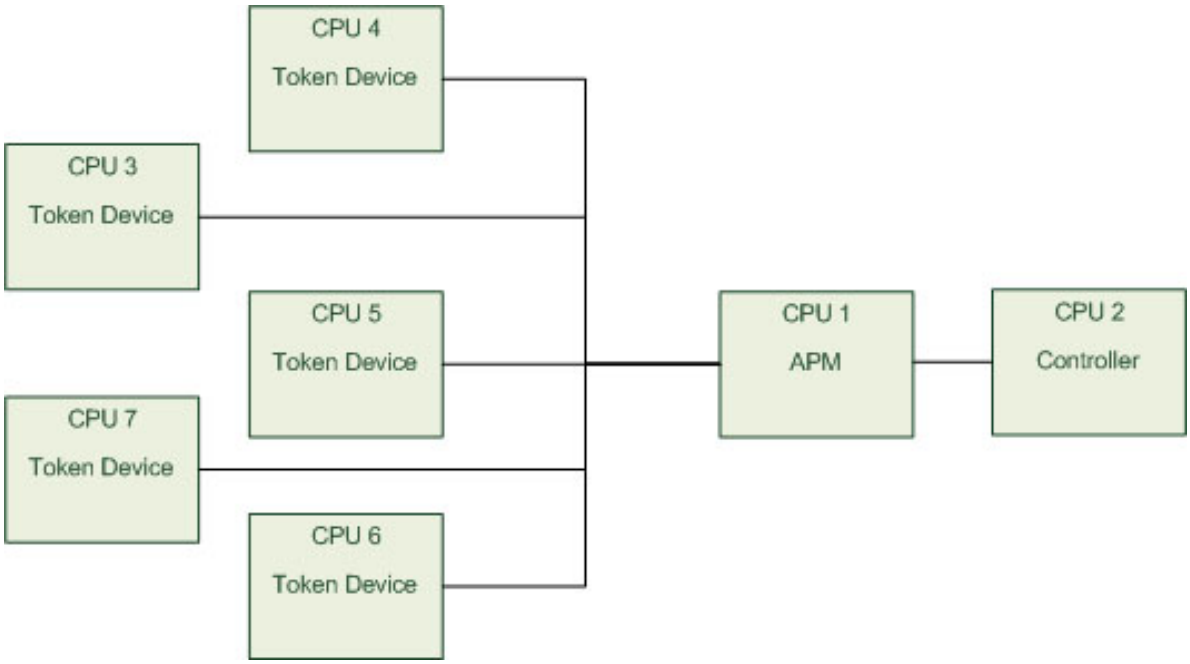


Figure 9.7: Deployment: How *JR* has been deployed

9.3.3 Running the Model

When trying to run the model several problems where encountered that made it impossible to complete the tests. During the tests, a very strange program error occurred, where the interpreter suddenly tried to cast a sequence type to a record type. This error immediately stops the execution of the model. This issue was resolved by changing to an older version of the VDMTools VICE program. However, switching back to an older version, only resulted in a reoccurance of a previously identified error, where threads are not allowed to finish their execution. This resulted in an series of thread instantiations due to use of the *period* primitive. Many hours have been spent trying to work out these issues, however without any luck. The model could not be tested and no results are obtained from the *JR* architecture.

Part III

Conclusion and Future Work

Chapter 10

Concluding Remarks

10.1 VDM++ A viable option?

In this thesis VDM++ have been used, as a design and development tool, to create several formal models of the CyberRail system as a distributed Real-Time system. This process has presented several practical challenges and elucidated many interesting aspects which are considered in this section.

10.1.1 The Development Process

The incremental approach presented as the key development strategy [FLM+05], and used in this thesis, works very well in both the traditional iterative development process but more importantly in the design and analysis phase of a project. By using this process, complexities of a distributed system such as CyberRail can be handled isolated in each incremental step. The sequential model involves basic behavioral issues, while the concurrent model deals with concurrency issues such as race conditions, dead-locking and other synchronization issues. The final step introduce the distribution and the real-time primitives which only add to the complexity of the model. Each step provides rapid feedback on the specific area of focus which in terms increases the stability for next incremental step. Increasing the stability, and hence the quality, by gradually adding more complexity, greatly reduces the risk of late-stage critical errors which can be extremely time consuming and hard to locate.

10.1.2 Using the Abstraction Principle in Practice

The design and analysis phase often poses difficult questions, in terms of program behavior, system deployment and architectural decisions, many of which the ideal solution cannot be answered using empiric knowledge. With the possibility of adjusting the abstraction layer, rapid progress can be achieved by ignoring non-interesting elements and focusing on the question at hand. The architectural questions in this thesis illustrates the advantages of abstraction,

where 3 different architectures have been identified as potential solutions for the CyberRail system and 2 of them realized as formal VDM++ models. The models abstract away from issues such as payment and location awareness. These issues are handled in theory and are not of concern for answering any of the architectural questions. However if the purpose of this thesis, and hence the abstraction level, was defined differently, this could also have been realized and validated with the help VDM++.

The level of abstraction used in this thesis has ensured that man power were spent on the distributed architecture, behavior and not least the ability to stress test the models. This thesis could easily have been based on Java solution but not without lowering the overall abstraction layer and increasing the workload of having to interface with execution platform, setting up deployment etc. Increasing the level of details will obviously extend the time period to achieve the same level of progress, not to mention increase the complexity and risk of errors, as previously discussed in section 3.4 - The Rules of Abstraction. The *BR* architecture, which was realized in Java, required substantial implementation to resemble the same behavior as in the formal model. This prototype implementation illustrates the advantages of having a formal model;

BUS Communication: The BUS communication was realized by using Java RMI as a middleware technology and required custom stubs and skeletons for each remote object such as ActivePlanManager, Controller and TransportPlanManager.

Distributed Time: The system had to be tested on different host machines which created a time synchronization problem. To be able to use any results, the distributed systems had to synchronized time wise. A custom implementation of the IEEE1588 Precise Time Synchronization protocol had to be implemented which did provide some level of time synchronization.

Token Device A mobile application, which was based on the smaller J2ME platform, had to developed. Due to the nature of the GSM network, it was not possible to synchronize the mobile phone with the rest of the hosts using the IEEE1588 protocol.

Deployment: Deployment of the system required several time consuming steps and was in general a tedious task each time a test had to be performed.

Many of these issues relates to physical deployment which consumed a substantial amount of time and prototype test only involved a single token device! If a deployment scheme with 50, 100 or maybe even 10.000 token devices should have been realized, it would be a very error prone and tedious task. When performing an equivalent test scenario in VDM++, you can ignore many of the physical deployment issues and focus on the key issues.

The process of reaching the current state, which the models represents, have required additional undocumented abstraction levels which primarily were concerned with the backend system. One of these system views, dealt with different behavioral strategies which could add some level of consistency in terms of the information sent to the passengers. However due to a focus change in the early stage of this thesis, they were abstracted away and had no further impact on the system - see figure 10.1. This view of CyberRail could easily have been revisited

later on in this thesis, but it was not pursued any further. See section 11 - Future Work, for more details regarding behavioral strategies.

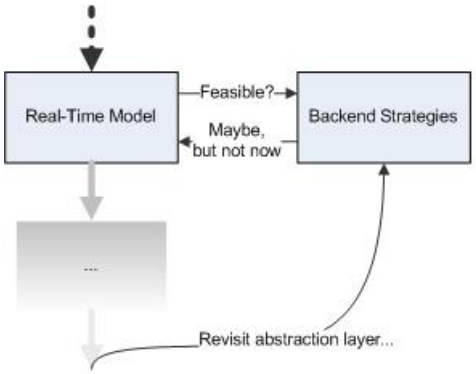


Figure 10.1: Illustrating how abstraction is used to explore possibilities

10.1.3 Feedback Loop Paradigm

Working with a formal model implies that certain real world aspects cannot be accurately estimated but merely reasoned to a plausible level. By lowering the abstraction level for a model and creating a prototype of the system, vital data can be measured and reused in the model to heighten the model accuracy or validate that a specific behavior actually performs identical in practice. A VDM++ model of copy machine pinches and their dynamic behavior has effectively used a practical prototype feedback loop to validate the accuracy of the formal behavior [MV07]. The practical test showed discrepancies between the model behavior of a belt drive and the practical behavior. This problem was credited a abnormal shaped belt cutout and actually identified a needed control feedback loop to compensate for this abnormality. The project also illustrates the added versatility of jumping forth and back between abstraction levels, to obtain an increased level of confidence in the results. Accuracy became within 1% margin between the models and the implementation.

The development phase of this thesis uses a feedback loop to accurately configure the VDM++ to resemble the real life characteristics and in term make the model more trustworthy. Figure 10.2 shows how the development process has progressed and how the prototype implementation have aided the Real-Time VDM++ model.

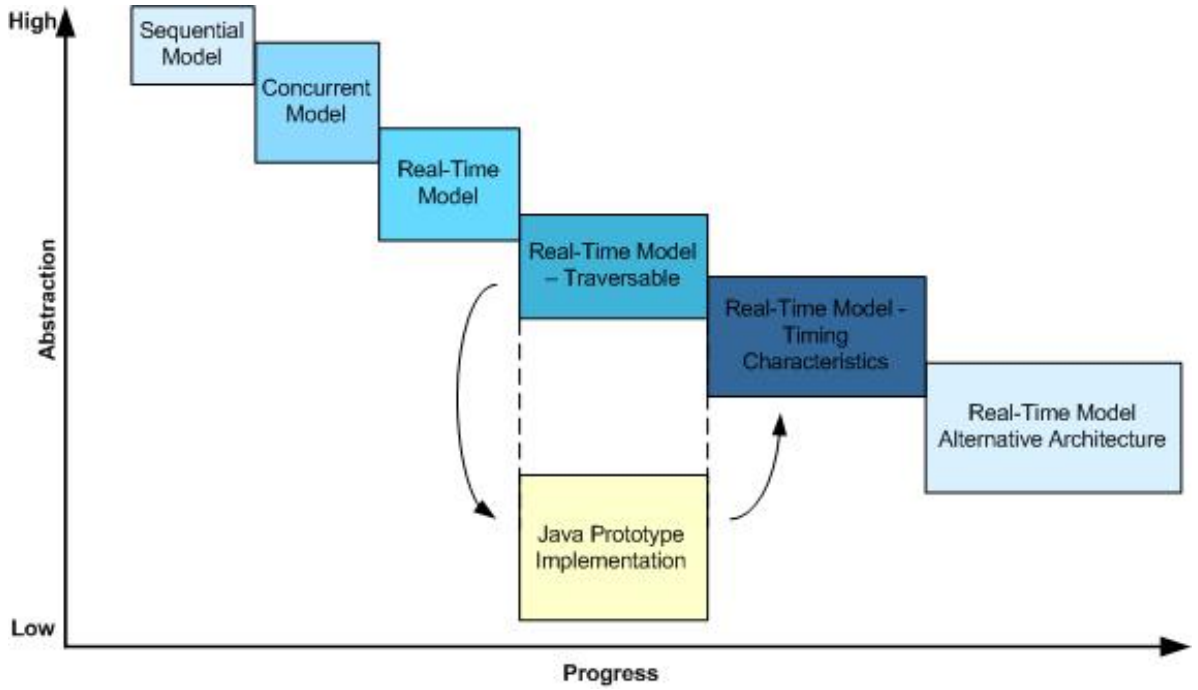


Figure 10.2: Level of abstraction during the development

The prototype implementation gave insight into how individual systems performed and especially how the inter-communication behavior between the backend system and the passenger behaved. These timing characteristics were applied to the formal model and used throughout the test phase. The actual realization of the prototype was a time consuming task which contradict the fact that this might be a profitable approach. This statement is however relative to the return of investment in regards to the added man hours used. If the resulting data neither proves or disproves any hypothesis' or does not contribute with any data to the model, then the added overhead is a waste of time. However if the prototype contribute with vital data, then the approach is very profitable. With the possibility of using code generation to create the *mechanical* code for the prototype, the potential return of investment greatly increases.

The process of implementing the VDM++ code in Java was a surprisingly simple task, since this step is usually based on a predefined design described using UML class diagrams, flow charts, sequence diagrams etc. Implementing the core functionality of the VDM++ required little insight and was in fact a direct mapped reflection of the code structure. Some issues such as mapping the ordered collections and the BUS communication paradigm did require additional effort. To optimize the process, pre-calculated railway grid information was reused in the prototype. This worked very well. However due to the lower abstraction level needed to realize the prototype and going from a virtual to a physical deployment platform, additional work had to be done which required substantial amount to work. Java RMI was used for intercommunication between the system and custom interfaces and stubs needed to be developed. The token device was represented by a mobile phone and an J2ME application had to be developed.

As the J2ME framework does not support many of the functions used in the backend system, which was based on the more comprehensive JRE6 platform, different workarounds needed to developed. A custom time logging facility, see section [REFERENCE TIL JAVA PROTOTYPE DEPLOYMENT MED SERVLET FEEDBACK DIAGRAM] needed to be developed since the mobile phone could not execute the custom developed distributed logging framework JAVTU - see appendix A.1. An alternative method for the mobile phone to intercommunication with the backend system had to be developed, since the GPRS, via the GSM network, could not communicate through Java RMI, which was solved by using a Java Servlet as a facade object into the backed System. A custom SMS class had to be developed to support callback, which utilized a mobile phone as a gateway. Many different practical deployment issues such as these emerged when developing and testing the prototype, and this was a very time consuming step. The results of this implementation, which contributed with realistic timing characteristics for the VDM++ models, did however compensate for the added work.

10.1.4 Outcome of the Test Phase

The stress tests of the 2 formal models, with the *BR* and the *JR* architecture, did not perform as was hoped, however certain usable results were still acquired. First of all the *BR* model proved to exhibit the correct behavior as anticipated, in regards to passengers traveling and using the railway grid and timetable information. Passengers were notified upon section breakdowns and new transport plans were calculated. In terms of correct program behavior

and the general approach of designing the models, it seems that the overall foundation makes sense in regards to the CyberRail case study. However no conclusion based on stress tests, to which architecture is favorable, can be made since no factual data was acquired to substantiate an argumentation of this nature. Several interesting results emerged from the Java prototype test which proved valuable in the VDM++ models and in the discussion of architectural benefits. The callback paradigm used to notify passengers about divergences in their transport plans were realized by using SMS technology. A total of 7.5% of the messages sent to the token device were delayed approximately 16 minuttes every time. This was a contant pattern and the duration was almost identical each time. Further research showed that each service provider implements a SMS retry algoritm [Hug07], which resend a SMS based on a set of rules. Apparently the service provider TDC used for the test phase, had a duration of 16.5 minuttes before SMS was sent again. Two important issues can be concluded from this:

- Each service provider configures the SMS retry algorithm and there are no guarantees that a service provider can deliver a SMS within certain period of time. This means that if 10 different service providers cover the area of which CyberRail operate, then passenger may experience different SMS delays based on the service provider they are using. This type of behavior is not acceptable in a system like CyberRail and it greatly reduces the quality of service experienced by the passenger.

- With a total of 7.5% of the messages being delayed, the importants of the SMS retry algoritm increases dramatically. If the message is retried within seconds of the initial attemp, then 7.5% is not an issue. However if several minuttes pass before the message is resend, then 7.5% is unacceptable.

SMS is a convenient technology, however less practical when dependent on many uncertain parameters. A different synchronous approach was tested in the prototype using the *Push/Poll* model described in section 7.5.1 - Callback to mobile Phones. This approach reduced the average response time to less than 5 seconds and the approach of using GPRS to synchronously return a transport plan upon request, proved very efficient. However, callback via GPRS is not possible and the only way to achieve a GPRS dependent solution, is to implement a polling paradigm. This does however present many additional concerns in terms of the added constraint of servicing these poll requests, increased battery consumption due to extensive GPRS usage and so on. It seems that a hybrid solution of using both GPRS and SMS will result in the best experience, in the question of what technology is favorable to use. SMS works reasonable well, however the setup would benefit from a custom configuration of the service providers SMS Retry algorithm, in the CyberRail coverage area

Beside the obtaining vital communication information, certain execution timing characteristics were identified and reused in the VDM++ models. The characteristics from the prototype was converted to a VDM++ relevant unit and integrated at significant repetitive operations. As soon as the conversion was done, the integration was an easy task which improved the overall credibility.

10.2 CyberRail Considerations

This thesis has identified several candidate architectures which are realistic candidates to the overall architecture of CyberRail and one of them were tested as a prototype implementation. What these architectures have in common are a realistic deployment scheme available with existing hardware technology. The fact that the architectures should be deployable with the technology available today has been an key issue throughout the master thesis. Several of the deployment issues are discussed in section 4.2, where some of the alternative are explained. This section will, based on the knowledge obtained during the master thesis, discuss the architectures and deployment issues in regards to CyberRail.

10.2.1 The Human Communication Interface Device

The CyberRail concept suggests that a passenger use a form of personal device to communicate with the system. With todays trend, and especially in Asia, the obvious choice is to use existing mobile technology readily deployed and used extensively. The discussion of whether or not to develop a proprietary device tailored only for this type of application, is not relevant anymore since pre-existing mobile phones, which typically are already owned by the users, can be utilized. Mobile technology is progressing rapidly and already incorporate many of the functions and features needed for interacting with a system like CyberRail. The list of technology includes GPS, BlueTooth, GPRS, UMTS, Wifi etc. By using existing mobile phones as token devices, the period of adapting the technology into the passengers daily routines, will potentially avoided [CA04]. Technology wise the question is how to achieve the optimal solution to use with the CyberRail system and the knowledge obtained during this thesis suggests that the mobile phone will be more that sufficient to solve this task.

10.2.2 Location Awareness

Common to all architectures identified, are the fact that most deployment and usability issues relate to location awareness or more exactly the lack of this. Technology wise this issue present many impractical problems, which potentially will impede the passengers experience. From the passengers point of view, the CyberRail system should be easy and simple to use, without any invasive hard-to-use technologies and with little or no user configuration. The CyberRail concept incorporate both railway and subway transportation (potentially all types of public traffic), which complicate the matter of location awareness and the potential technologies somewhat. To keep matters as simple and economical as possible, the use of multiple technologies are not favorable. Although vertical hand offs between communication platforms are possible, it is an error prone procedure and only adds to the complexity of the communication paradigm.

GPS & Galileo

GPS technology cannot be used in shielded locations such as the subway, but it will presumably work well above ground. GPS is a nice option since the technology is emerging in

many communication devices today, however battery issues of having GPS transmitting continuously makes this technology impractical. The granularity of detail, in terms of location awareness, is very good and more than sufficient to fulfill all passenger service proposed for the CyberRail system. However due to limited use in subway environments and the excessive power consumption, this technology is not suited for this type of application. Although still unknown how the Galileo project is going to perform in practice, many of the considerations on GPS are also valid for the European positioning system initiative [Und06].

BlueTooth

A communication paradigm using BlueTooth could also be used in a deployment scheme, where the passengers communicate with nodes positioned throughout the railway stations and trains. Figure 10.3 illustrates how the BlueTooth could work. When a passenger arrives to the train

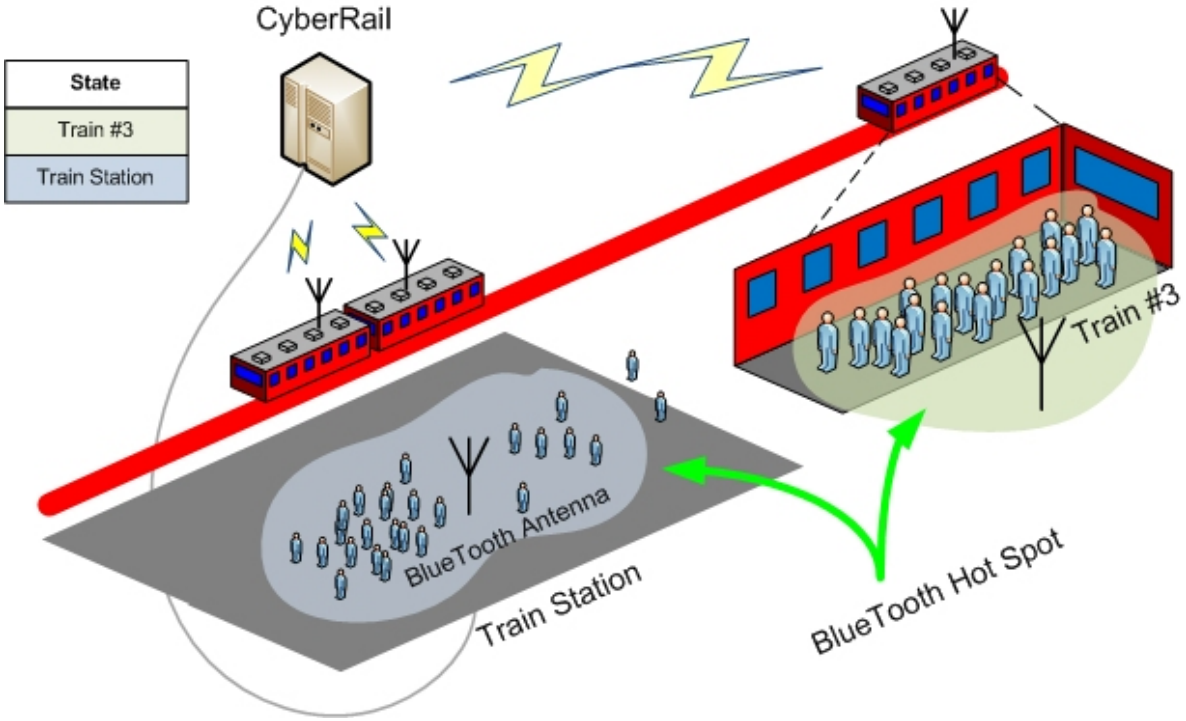


Figure 10.3: Deployment scheme using BlueTooth

station, the token device immediately connect to the BlueTooth hot spot and stays connected as long as the passenger is within the train station area. Each train must be equipped with its own BlueTooth hot spot and when a passenger boards the train, the token device connects to this one instead. The train can use a wireless technology, such as Wifi, GPRS or UMTS to inform the backend system of its current passengers. The level of location awareness using a paradigm like this, offers a lower granularity of information but sufficient enough to locate passenger upon notification. The level of granularity is however not enough to offer personal

navigation assistance such as directions to platforms, ticket office etc. The use of BlueTooth has the advantage of using existing technology available in cellular phones today and thereby lowering the overall investment needed to realize the system. This development scheme does have several disadvantages which makes this approach less favorable. Continuous BlueTooth communication will drain the the mobile phone battery very fast, which greatly decreases the passenger experience. Secondly the BlueTooth nodes which needs to be positioned on each station and train will have to be strategically installed to cover the whole area and to be able to handle the throughput. Even though a BlueTooth deployment scheme will work in theory and provide sufficient location awarenss, it is far from the optimal solution for the CyberRail system.

GSM Triangulation

A prototype application has been developed for CyberRail which use a combination of cellular signal location and the traffic timetable to assist the passenger on his or her traveling path. The signal location function was utilized in the subway, making use of cellular repeaters throughout the tunnels and thereby obtaining a sense of location awareness by registrating which repeater the passenger's mobile phone was using. This is a clever approach and proved to be quite efficient - see figure 10.4.

This type of technology could in theory be used above ground to triangulate the passengers position based on a cellular phone signal [Zha00]. A very interesting project called "Google My Location" performed by Google [For07], *geolocate* a cellphone based on nearby cellular antennas which in essence makes use of triangulation techniques - see figure 10.5a. A Java program is installed on a mobile phone which is build on top of Google Maps and the cellular phones location is plotted on the map - see figure 10.5b. If this technology is accurate enough, which depends primarily on the density of the mobile towers, this approach is very interesting for number of reasons. The system is simple from the passengers point of view. There is no need to buy new expensive technology to use the service provided and battery lifetime of the communication device is greatly increased, compared to the solutions provided above. The deployment scheme is inexpensive from CyberRails' point of view, since it will use an existing system. This approach is very suitable if the technology is sufficiently matured.

Registration by Proximity Payment

Another area of concern in CyberRail, which could solve the location awareness issue, is how to handle payment in a non-invasive manner. Different type of proximity payment technologies have been investigated during the process of this thesis and they can be used to handle the location awareness issue (SonyEricsson & Eurocard (Bluetooth) [Eur07], Nokia & MasterCard(RFID) [Mas07], ZOOP (Irda, IrFM) [Inf02], FeliCa (RFID) [OMGR05], Philips & VISA (NFC) [CA04], EMVCo: Eurocard, MasterCard, VISA [EMV06a] [EMV06b]), however they are not to be pursued in this thesis. The different proximity payment technologies all share the same feature of being in direct contact with both CyberRail and the passengers. The typical usage scenario is that passengers registrate upon entering the train and again when leaving the train. The total cost can then be calculated by CyberRail and credited the passengers

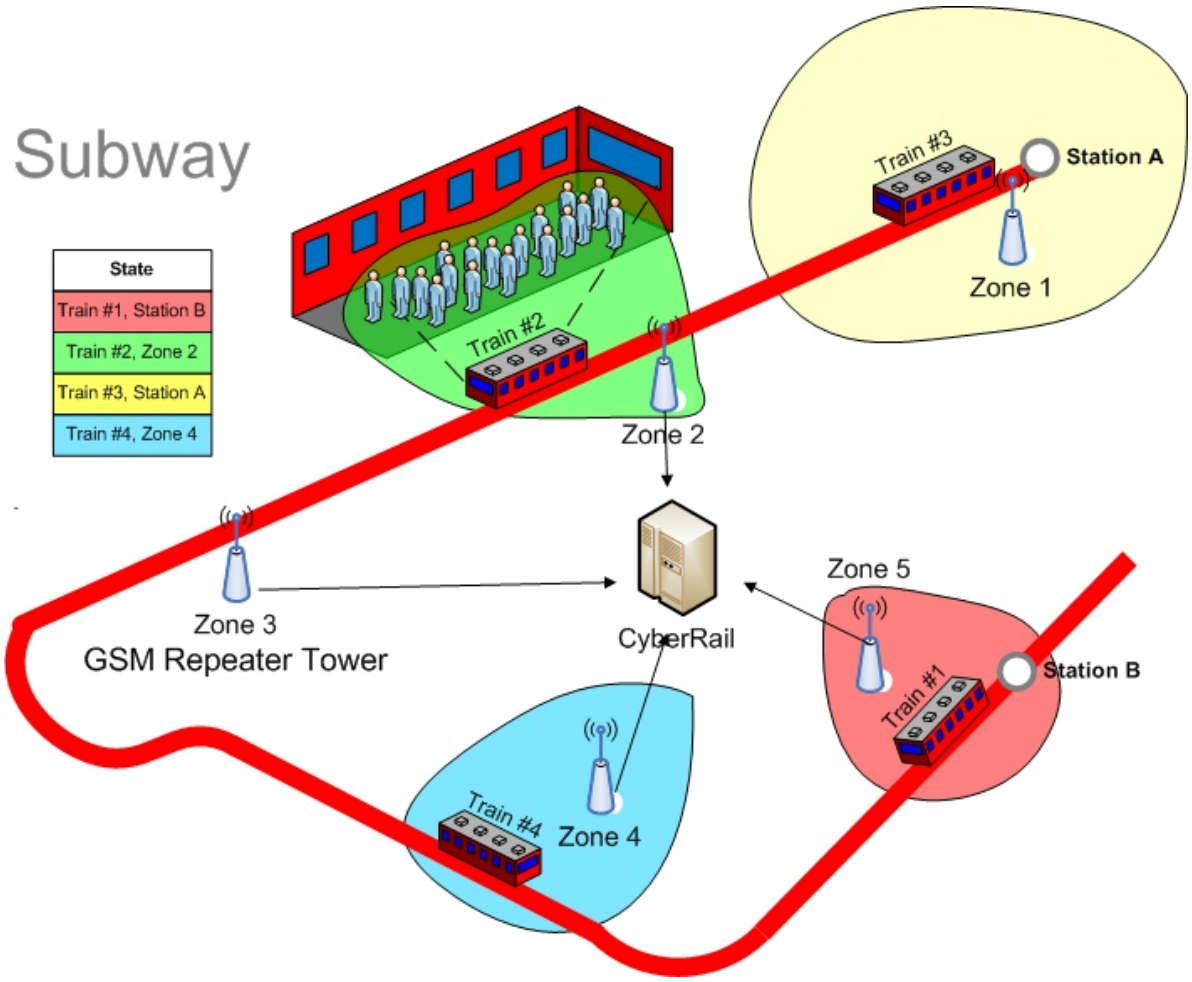


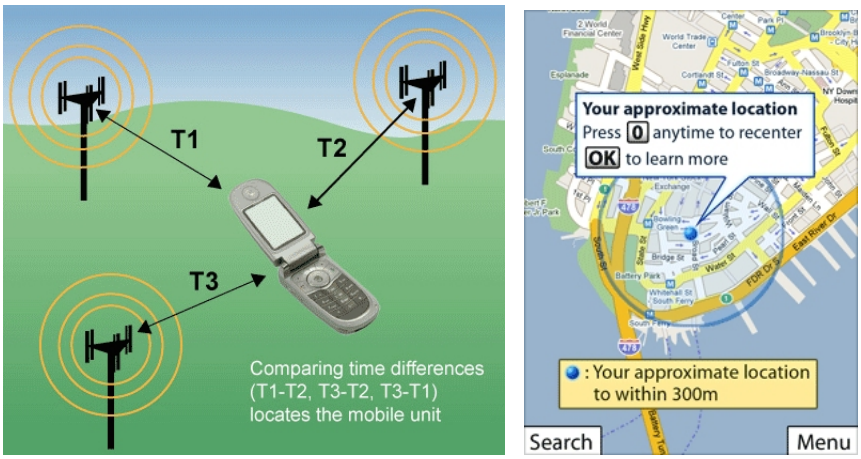
Figure 10.4: Using underground GSM repeater towers to registrate mobile signals

account or credit card. CyberRail might as well registrate the passengers position based on which payment node was used and use this information to locate the passenger. Figure 10.6 illustrates how proximity payment can be used to locate passengers.

A very good granularity of location awareness can be achieved by registrating proximity payments. However a couple scenarios exist where the passengers location is either unknown or incorrect;

Pre-Payment: The passenger must register a payment at the payment node before CyberRail knows the location of the passenger - see figure 10.6 state *Unknown*. This really is not an issue since most CyberRail services rely on the fact that passengers are actively using the system. However personal assistance can be an issue if the passenger has not registered at a payment node.

Train departure: The period from when passenger leaves the train until a re-registration is



(a) GSM triangulation

(b) Google My Location Application

Figure 10.5: GSM Triangulation used in the new Google application

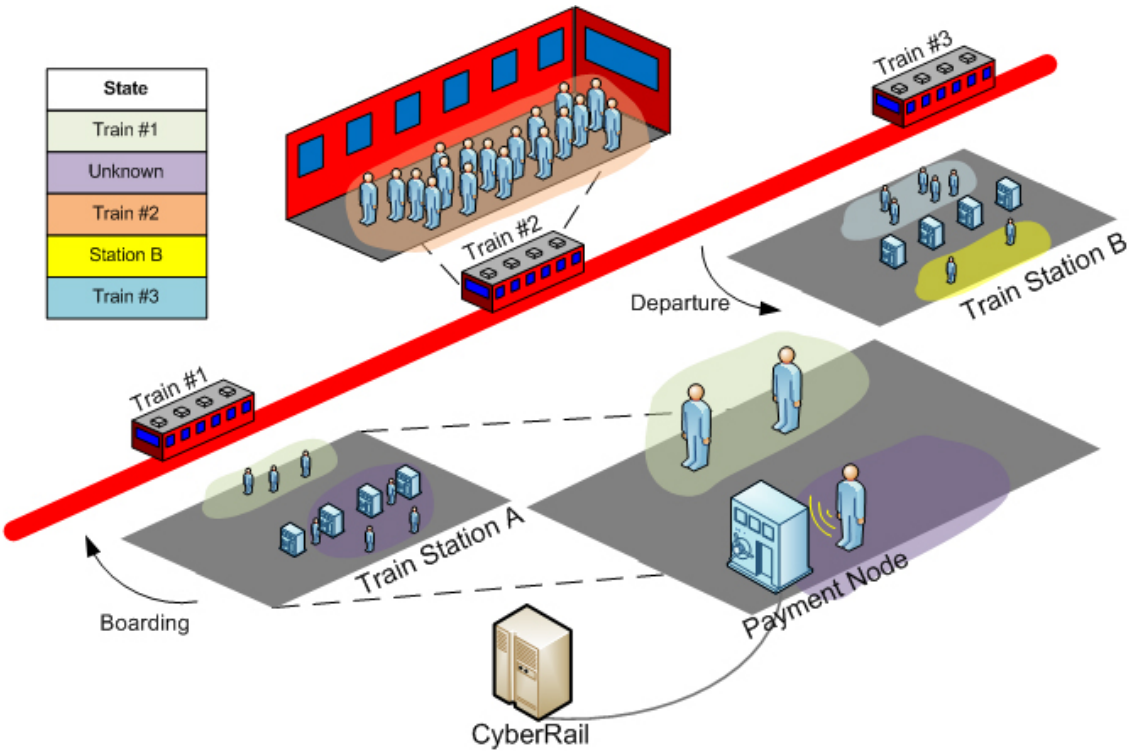


Figure 10.6: Location awareness by proximity payment

made at a payment node, CyberRail will think that the passenger is still on the train - see figure 10.6 state *Train #3*. This is a temporary state issue which can result in inactive route notification from CyberRail, which does not concern the passenger. However this is a minor issue! From a deployment point of view, the location awareness obtained from registrating proximity payment transactions is very advantageous since the overall cost will be greatly reduced compared to GPS or the BlueTooth deployment scheme*. Complexity wise the solution is also favorable solution and in the end might be a very suitable approach, if trade offs can be accepted.

Alternative Solutions

Other more exotic solution can be used to acquire the state information needed to successfully aid the passenger, however most not readily available or less practical.

A peer to peer approach could be used where each passenger was tagged with a communication device which communicated with the peer passengers. Eventually a peer passenger will communicate with a base terminal reader located on either the train station platform or on the trains, which will report all the state information to CyberRail. The system will resemble a group communication paradigm where state information is shared across several distributed systems [Bir05]. This deployment scheme would give CyberRail near Real-Time state information about who was located on a specific station or train. With a system like this, the granularity of information would be sufficient enough to provide personal location assistance on e.g. the train station. This type of technology is emerging in the industry and currently the company Chess† has developed a peer to peer protocol based on gossip principles, which would be very suitable for this type of application. They have also developed a hardware platform which was used in a proof of concept project called WIreless Gossip With Asset Management (WIGWAM). The project demonstrated a form of track and trace service using these active gossip devices. With the cost per unit and battery lifetime estimated, this product and the general approach could be an interesting technology in the near future [Che07].

Other Real-Time location awareness technologies are readily available [Ubi07] but are still far too expensive for a deployment scheme of more than 1100 stations and millions of passengers per day [Wika].

10.2.3 Architectural Analysis

The identified architectures in this master thesis each possesses certain qualities which make them lucrative in different areas. The functional advantages have already been discussed, cf. section 4.2, and deployment considerations have briefly been discussed. In regards to deployment, the main difference in the architectures are a question of cost and pragmatic deployment issues, not functionality as initially thought. Each architecture, may it be the backend responsibility or the joint responsibility architecture, have the ability to fulfill the requirement to passenger and company service (cf. 2). Frontend responsibility is not considered since the

*Cost in regards to both the passengers and CyberRail.

†for more information see http://www.chess.nl.

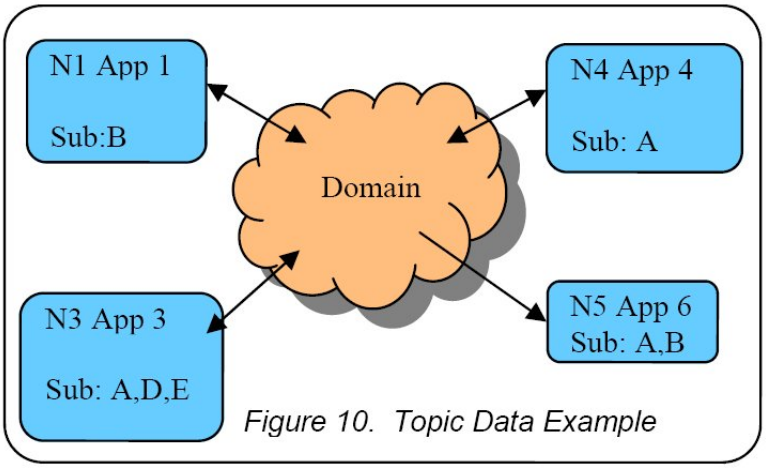


Figure 10.7: Data Centric Publish-Subscribe paradigm (DCPS)

approach cannot fulfill the requirements of the CyberRail system, in terms of the passenger services.

The real question is what architecture makes most sense in terms of deployment and usability. Even though the test of each architecture could not supply information about howf they performed with a high number of concurrent passengers, certain assumptions can be made. The main difference between backend- and joint responsibility is the distribution liability. The backend architecture, which is responsible for both calculating transport plans and notifying the passengers about transport plan changes, will require substantial more resources to service the passengers traveling. By distributing the responsibility of calculating the transport plans to the passengers, it greatly reduces the resources needed. Additionally the quality of service, which joint responsibility represent, will be at a constant level, since the railway grid is located directly on the token device. The disadvantage of the joint responsibility architecture, is that the railway grid information must always be up-to-date. The paradigm used to update the grid information in the VDM++ model, proved quite efficient. The paradigm would obviously have to be extended to include some type of versioning support, so only the changes since the last update are retrieved. However the fundamental approach works very well. This issue could make use of some of the principles introduced for Data Distribution Service(DDS) [PCFW05] for Real-Time Systems. It introduces a so called Data Centric Publish-Subscribe (DCPS) paradigm which relies on topic based notification. Applications subscribe to a domain, with one or more specific topics of interest. Upon changes in the domain, subscribed application are notified based on the topic - see figure 10.7.

The granularity of information in this type of publish-subscriber system if far more detailed, than the traditional publish-subscriber approach [GHJV95], and this would scale very well in the JR architecture. The domain would translate to the railway grid information and topics would be different geological transportation zones of interest. A frequent traveler would subscribe to zones used most often, and the backend system would only be notified upon *relevant* changes. This approach would greatly decrease the number of updates performed

daily and with a total of 7.5 million passengers per day, this area is of great importance.

With the information gathered in this thesis, it seems that the joint responsibility architecture is a favorable approach in regards to functionality, deployment considerations and cost. The architecture is a reasonable solution which should fulfill all present and future requirements for the CyberRail system. The architecture is especially advantageous for the high volume of users which use CyberRail everyday.

10.3 Quality & Testability

This thesis has demonstrated several disciplines which have improved the qualitative properties in regards to the formal models and the Java prototype. However, what is the conceptual meaning of quality and how is it defined? Different quality aspects are used depending on the area of focus and the abstraction level. In the world of software engineering, Robert S. Pressman defines software quality, as how well it is designed (quality of design) and how will the software conforms to a design or specifications (quality of conformance) [Pre05]. The ISO9000:2000 standard defines quality as *"Degree to which a set of inherent (existing) characteristics fulfills requirements."* [Hua07]. These definitions all relate to a specific area of concern such as program correctness, incorporation of best practices or fulfillment of a demand specification. This thesis defines a different quality factor referred to as *Testability* and is defined as *"The totality of measurements taken to achieve a satisfying and meaningful assessment"*. For example a software component is supposed to have an intended behavior and to assure this, module tests and integration tests are often performed. To be able to perform the module test, a *satisfiable* scenario must be composed, which test all the behavioral aspects for this specific component. Additionally a program shell needs to be developed, which is used to stimulate the component as predefined in the module test scenario. These steps all contribute to the component's stability [Mar96] and increases the components quality in regards to testability.

This section will describe the additional measurements taken to increase the testability of VDM++ models and the Java prototype developed during the process of this master thesis.

10.3.1 JAVTU: A Common Denominator

In order to incorporate feedback from a prototype implementation it was necessary to obtain realistic timing characteristics. These characteristics could be used to optimize the behavior of the VDM++ models and increase the trustworthiness of the results. As described in section 3, the VDMTools VICE application creates a very informative execution trace which can be readily analyzed using the Showtrace tool. This is a very convenient way to analyzing, and not to mention debug, a distributed system and it would be favorable to also have this level of detail in the Java prototype. The added quality of having the same analysis tool both in VDM++ and Java outweighed the cost. This resulted in the development of a custom distributed logging framework, JAVTU - see appendix A.1, which conformed to the execution trace log protocol used in VDMTools VICE. The JAVTU framework significantly increased the testability of the Java prototype, since existing tools could be used to analyze the traces and acquire the *relevant* timing characteristics. The sudden increased overview of the Java prototype

gave valuable insight on how the system performed and exposed a potential bottleneck of using SMS technology. Additionally, faulty behavior could be readily seen in the graphical traces, which in terms decreased the total hours used for debugging the prototype. The JAVTU framework is not a proprietary, but a generic component and can easily be used in other distributed applications. The added testability, generated from developing and using the JAVTU framework, improved the actual task of testing and improved the aftermath of analyzing the results.

10.3.2 Solving Pragmatic Deployment Issues

When defining the deployment platform for a VDM++ model in VDMTools VICE, primitives such as CPU's, BUS's and active objects must be declared statically in the System class object. This means that new CPU's and active objects cannot be instantiated at runtime which impair the ability to use scale a deployment platform. One of the goals of this thesis, was see how the a giving architecture would behave during a stress test. The initial idea was to acquire statistical data from the Japanese Railway Company and used this to stimulate the model. By using the correct statistical data, assessments gain an increased level of trustworthiness and any indications and speculations would be better founded. However, with the current procedure of setting up for example 30.000 token device object, declaring 30.000 independent CPU's and connecting these with 30.000 individual BUS's manually, we anticipated that a custom tool needed to be developed. The tool is Called Repeater, see appendix B, and is used to generate repetitive entries into a file, based on identification tags and input strings. The application is used to generate 3 files for each VDM++ test phase.

System: Used to Generate CPU and BUS declarations and deploy the appropriate CPU's and connect the CPU's to the corresponding BUS's.

World: Used to generate method invocation statements in regards to token device initialization.

Environment: Used to generate stimuli, as in transport plan requests, for each traveling passenger.

With the added testability of the Repeater tool, the deployment platform for 100 or even 10.000 passengers, could be generated in less than 10 minutes. This task would be a *very* tedious and error prone one, if performed manually. The notion of being able to stress test the formal models proposed in the thesis, relied heavily on the Repeater tool and the increased testability saved us a lot of time.

10.3.3 Populating the Models

As mentioned above, we would have hoped to stress test the VDM++ models with the use of statistical data. However, to efficiently analyze the behavior of a candidate architecture, the passengers needed to have a purpose for traveling and some environment constraints which affected the journey. A form of automatic behavior was favorable, where trains were traveling according to time tables and on a specific route. The abstraction level was therefore altered

and train and station entities were introduced into the initial real-time model. Additionally a timetable and a set of routes needed to be generated for the trains. The generation of these would have to be configurable, so each test could be scaled to fit the number of passengers. The result was a set of classes, namely *RailwayGrid* and *GridReservation*. In figure 10.8 a flowchart illustrates how the railway grid is composed. The result is 3 different data structures, containing viable information used by either the train entities, token devices or by the backend system. The grid data contains all combination of section, which is potentially traversable. Time table is a matrix specifying when a section is occupied. This matrix depends on the reservation parameters specified - see listings 10.1 for details.

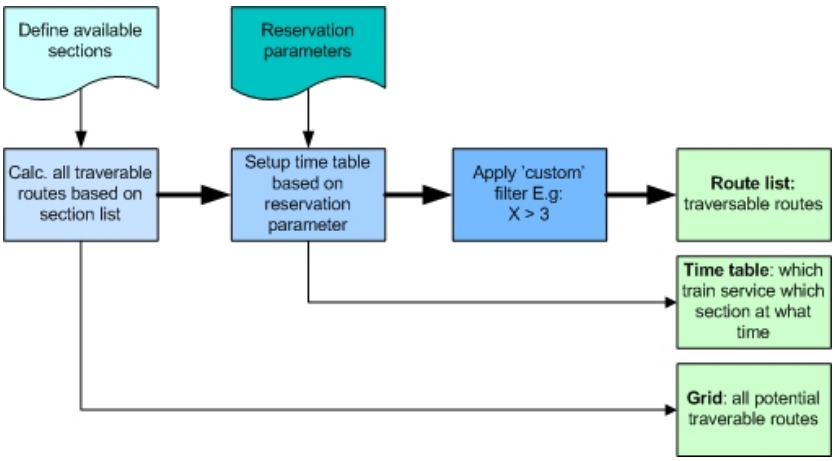


Figure 10.8: Flowchart: Showing how the railway grid information is calculated

Listing 10.1: Time table configuration parameters

```
13 --Configuration Parameters--
14 --The minimal time resolution used in the time table
15 private minDelta : nat := 50000;
16 --The number of trains which can occupy the grid at the same time
17 private noTrain : nat :=1;
18 --Length of period which the reservations be calc.
19 private periode : nat :=1000000;
20 --Max delay between train station stop and start
21 private maxDelay : nat :=minDelta * 2;
22 --Number of instances for each train should run
23 private noInstances : nat := 2;
```

The ability to dynamically generate new railway grid structure and the time table information, this greatly improves the versatility and hence the testability of the formal VDM++ models.

10.4 Related Work

The nature of this master thesis and the focus on early indication of favorable architectures in distributed systems, does not relate to many similar projects and finding related work in regards to either stress testing by use of formal models has not been possible. Many projects are emerging using VDM++ and VICE, however many of these are only concerned with program correctness and not performance. A couple of related studies a briefly described in this section.

Hugo Daniel dos Santos Macedo has done a master thesis on a study of the requirements for a pacemaker using VDM++ and VICE. However, the focus was not concerned with timing characteristics or stress testing. This thesis main focus was to find and validating the requirements for the pacemaker system [dSM07].

Formal modeling analysis of CyberRail has been performed earlier, in the article *Towards a formal model of CyberRail* [BCJ+04]. This work focuses more on formalizing the behavior of CyberRail than requirements and timing concerns.

10.5 Thesis Conclusion

The objective of this thesis was partially to evaluate the recent extensions of VDM++ called VICE and the associated tools as an engineering approach for developing and stress testing real-time distributed systems and analyze the CyberRail concept using the VICE technology mentioned above. Three different candidate architectures have been identified and discussed as viable options for the CyberRail concept. Two of these have been modeled in VDM++, of which one have been realized in Java and contributed with valuable timing characteristics. Deployment considerations for each architecture have been discussed in regards to existing technology and realistic deployment schemes. The process of developing and testing the CyberRail case study with the current VDM++ tool set indicates that a formal model approach can contribute with vital information at a very early stage of development.

During the process of this thesis, several shortcomings have been identified, which impair the ability to use the VICE extensions of VDM++ to stress test a distributed system as intended. Beside the application errors discovered in VDMTools VICE during the work of this thesis, some fundamental shortcomings still exist. To be able to efficiently stress test a system, certain optimizations to the interpreter must be implemented to obtain the volume of stimuli needed for a distributed system like CyberRail. Initial tests of the models shows that a test with only 100 traveling passenger will be a very time consuming affair. If a realistic stress test was to be completed, with a potential peak value of 50.000 concurrent travelers and throughput of 3.5 million passengers per day for a single train station, it would take 6.7 years to complete a simple scenario test with the current level of performance. Other shortcomings have also been identified, but these have been solved by developing custom tools to achieve the intended functionality. A deployment generation tool have been developed to aid the creation of both the execution platform file and environment stimuli file (see appendix B). A generic distributed logging mechanism for Java has been realized, using the real-time primitives defined in VICE, which makes a VDM++ and a Java application comparable using the same

set of tools. The models were not tested to their full potential, due to issues with VDMTools VICE. However, an optional approach to test certain aspects of the models, has been described in section 8.2.2, but have not been pursued any further due to lack of time. The process of this thesis suggests that future versions of VDMTools VICE and VDM++ potentially could be used to stress test a formal distributed system, however focus on performance issues must be attended. Even though VDM++ and VDMTools VICE does not perform as we would have hoped, one must take into account the technology's infant stage when fully evaluating VDM++ as an engineering tool. The language is by no means a leading edge technology, but rather a bleeding edge contester.

This thesis has identified several potential bottlenecks in regards the proposed candidate architectures and deployment schemes, which must be considered carefully when realizing the CyberRail system. Especially in regards to location awareness and to inter-communication with the passengers. A combined SMS and GPRS solution was found to be the most beneficial approach in regards to both performance and feature wise. Furthermore special attention to the service providers in the CyberRail coverage area is needed to realize the solution proposed. Location awareness has been identified as a key issue for realizing the CyberRail system, however the technologies of today, such as the Google My Location technology, seems sufficiently mature to resolve this issue.

Chapter 11

Future Work

The deployment of the BR VICE model needs to be changed to the identified approach of all token devices on one CPU. So that a more valid test can be performed. The *JR* VICE model has to be refactored so it will avoid the VICE Tool errors that it encounters at the moment, and run through the same tests as *BR* VICE model.

Right now there is limited focus on model quality and areas that needs pre and post conditions and invariances has to be identified and created to ensure that the model performs correctly but also give a higher degree of confidence in the results accumulated.

A problem that has not been handled concerning notification of token devices, is the scenario where several sections that go inactive. Currently the token devices would be notified of each brake down and this is not a desirable conduct. Behavioral strategies can be introduced into the backend system, which defines different notification schemes based on different levels of stress. This is an interesting aspect and especially how the strategies would impact the system under different stress levels is a relevant question.

Terminology

.Net Remoting .Net Remoting(Remoting), is a middleware developed by Microsoft, it can be used by all programming languages supported by the .Net Framework. Remoting can only be found in the full .Net Framework and not in .Net Compact Framework that is used for mobile devices. Remoting utilizes remote method invocation like Java RMI, but unlike Java RMI there is the possibility to choose between SOAP (Simple Object Access Protocol) or Binary communication. Remoting contains no build-in ability to communicate through IIOP, there has been made extensions by 3rd party developers so Remoting can communicate with IIOP [IIO].

API Application Programmers Interface

APM Active Plan Manager

backend Comprised by everything except the actual passenger.

CORBA Common Object Request Broker Architecture. A platform, language and architecture independent middleware technology by OMG.

CyberRail Concept The overall vision for the CyberRail project with both passenger and company services.

Frontend Represented by the passenger(s) or token device(s). The fronend is one of two subsystems of CyberRail. The other subsystem is the backend system, which provides the core functionlity of the CyberRail system.

GPRS General Packet Radio Service. GPRS can be used for services such as Wireless Application Protocol (WAP) access, Short Message Service (SMS), Multimedia Messaging Service (MMS), and for Internet communication services such as email and World Wide Web access. GPRS technology is often referred to as 2.5G and offers a theoretical bandwidth of 230 kbit/s

HTTP Hypertext Transfer Protocol, mainly used for transfering webpages from servers to browers

IIOP Internet Inter-ORP Protocol (IIOP), is based on GIOP (General Inter-ORB Protocol) the protocol specification is maintained by OMG [Bol02]. It was developed to enhance cross ORB communication and in 2002 it became part of the CORBA v3.0 core specification. Since then other middleware such as Java RMI has included this protocol into their specification.

IR Infra Red. Short range wireless protocol typically used in mobile phones, remotes and computeres to transfer small amount of data

J2ME JAVA 2 Micro Edition (SUN) for small embedded devices, e.g. mobile phones.

Java RMI Java Remote Method Invocation (RMI), is a middleware developed by Sun for Java, it is included in most of the JVMs (Java virtual machine). RMI is however not in J2ME (Java Micro Edition) which is used for small mobile devices such as mobile phones and PDA's. As the name implies Java RMI uses remote method invocation, that is where methods on objects are being called from remote machines. Java RMI also has in build functionality to make calls through IIOP, which makes it possible for Java RMI to communicate with a CORBA ORB. When using IIOP there are though some functionalities that are not supported, such as remote garbage collection [Wag05a] [Wag05b].

JAVTU JAVA VDM Trace Unit. JAVTU is a logging framework that enables a distributed systems, where applications run on seperate systems and seperate networks, to log interaction. The framework provides tools to sucessfully merge each individual log into a single file, which support the format of Overture Showtrace graphical trace viewer.

JRMP Java Remote Method Protocol is the wire protocol implemented by Sun to handle the traffic in Java RMI. JRMP is the standard protocol used by RMI objects, however there JRMP is not the only protocol for Java RMI. Several companies has made their own implementation instead of JRMP and not all of them recognise RMP. [Wag05a] [Wikb]

JVM Java Virtual Machine

Midlet This is what an J2ME application is called

NFC Near Frequency Communication

RF Radio Frequency

RFID Radio Frequency Identification

Route A sequence of section with a departure time

RPC Remote Procedure calls is a way to hide network traffic from the programmer, when calling a remote method it will look like a local method call on the callers side [Bol02].

SE System Engineering

Section A connection between two station and the travel between them. A -¿ B and B -¿ A are two different sections

Sentient World Simulation Sentient World Simulation (SWS) will be a continuously running, continually updated mirror model of the real world that can be used to predict and evaluate future events and courses of action.

Service Provider A company providing some type of service. Celluare companies and internet providers are usually referred to as service providers

Servlet This is java based web application, but unlike a normal HTML it has Java code running in the background

SMS Short Message Service

SWS See Sentient World Simulation

Token Device Gender less device which could be represented by a proprietary device or a mobile phone. A token device is used be a passenger to communicate with the CyberRail system

TPC Transport Plan Constructor

transport plan A complete passenger plan with date, time, fee, departure location, arrival location etc

UMTS Universal Mobile Telecommunications System. This is the third generation of celluare technology and is often reffered to as 3G. This technology offers a much higher data transmission bandwidth which makes services such as music and video streaming possible.

VDM Vienna Development Method. Formal development method similar to the Z language.

VICE VDM++ In a Constrained Environment. VICE is a recent research extension of VDM++ for describing and analyzing distributed systems

VM Virtual Machine

Wifi Is a brand created by Wi-Fi Alliance, to improve interoperability between wireless technologies based on the IEEE802.11 standard. In essense the term Wifi is used when referring to a wireless local area network.

List of Figures

List of Tables

Bibliography

[BCJ+04] Dines Bjrner, Peter Ciang, Morten S. T. Jacopsen, Jens Kielsgaard Hansen, and Michael P. Madsen. *Towards a formal model of CyberRail*, pages 657–664. IFIP 18th World Computer Congress. Kluwer Academic Publishers, Toulouse, France, 2004. [cited at p. 101]

[Bir05] Kenneth P. Birman. *Reliable Distributed Systems - Technologies, Web Services and Applications.*, volume 1, pages 661–254–260. Springer, USA, 2005. [cited at p. 96, 122]

[Bol02] Fintan Bolton. *Pure Corba.* Sams, [Indianapolis, IN], 2002. ISBN 0672318121. 921 pp. Available: `http://isbndb.com/d/book/pure_corba`. [cited at p. 106]

[CA04] Jiajun Jim Chen and Carl Adams. Short-range wireless technologies with mobile payments systems. page 649, 2004. [cited at p. 91, 93]

[Che07] Chess. Wireless gossip with asset management. wigwam - proof of concept project. pages 11–30–2007, 2007 2007. Available: `http://www.chess.nl`. [cited at p. 96]

[CSK06] CSK. Development guidelines for real-time systems using vdmtools. 2006. [cited at p. 13, 21, 22, 42, 108]

[DHM+06] Daniel Delling, Martin Holzer, Kirill Muller, Frank Schulz, and Dorothea Wagner. High-performance multi-level graphs. 2006. [cited at p. 68]

[Dou99] Bruce Powel Douglass. *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns.* Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 1999. ISBN 0-201-49837-5. [cited at p. 15]

[Dou06] Bruce Powel Douglass. Embedded.com - sysml - the systems modeling language, 2006. Available: `http://www.embedded.com/shared/printableArticle.jhtml;jsessionid=4THZGKJZUGIH4QSNDLPSKHSCJUNN2JVN?articleID=192700665`. [cited at p. 6]

[dSM07] Hugo Daniel dos Santos Macedo. Validating and understanding pacemaker requirements. 2007. [cited at p. 101]

[Ecl07] Eclipse. Eclipse.org home, 2007 2007. Available: `http://www.eclipse.org/`. [cited at p. 121]

[EMV06a] EMVCo. Emv - a brief explanation, 2006. Available: `http://www.accesskeyboards.co.uk/emvoverview.htm`. [cited at p. 93]

[EMV06b] EMVCo. Emvco, 2006. Available: `http://www.emvco.com/contactless.asp?show=58`. [cited at p. 93]

[Eur07] Eurocard. Ericsson, eurocard team on bluetooth wireless payments - product development newsbytes news network - find articles, 2007. Available: `http://findarticles.com/p/articles/mi_m0NEW/is_2001_April_5/ai_72803493`. [cited at p. 93]

[FLM+05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer, 2005. ISBN 1-85233-881-4. 402 pp. [cited at p. 13, 21, 85]

[FLS07] John Fitzgerald, Peter Gorm Larsen, and Shin Sahara. Vdmtools: advances in support for formal modeling in vdm. 2007. [cited at p. 15, 108]

[For07] Brady Forrest. Google's my location: A peek at location on android, 2007. Available: `http://radar.oreilly.com/archives/2007/11/googles_my_loca.html`. [cited at p. 93]

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. 1995. ISBN 0-201-63361-2. [cited at p. 97, 138]

[Ham99] K. Hamzeh. Rfc 2637 point-to-point tunneling protocol (pptp), 1999. [cited at p. 122]

[HK06] Orit Hazzan and Jeff Kramer. Abstraction in computer science & software. *System Design Frontier Journal*, December 2006. [cited at p. 17]

[HP] Tony Hoare and The MIT Press. Software abstractions - the mit press. Available: `http://mitpress.mit.edu/catalog/item/default.asp?ttype=2\&tid=10928`% fi. [cited at p. 18]

[HSWW06] Matin Holze, Frank Schul, Dorothea Wagne, and Thomas Willhalm. Combining speed-up techniques for shortest-path computations. March 9, 2006 2006. [cited at p. 68]

[Hua07] Ding Huamei. Quality management systems fundamentals and vocabulary. page 21, 2007. Available: `www.uml.org.cn/rjzl/doc/Ejichushuyu.doc`. [cited at p. 98]

[Hug07] Tom Hughson. Messaging services on the iden network developers' guide, 2007. Available: `http://developer.sprint.com/site/global/develop/technologies/wireless_markup_msg/developer_guide/p_coding_messaging.jsp\#SMS_Retry`. [cited at p. 62, 90]

[IIO] IIOP.Net. Iiop.net - overview. Available: `http://iiop-net.sourceforge.net/`. [cited at p. 48, 105]

[Inf02] HAREX InfoTech. Zoop - welcome to harex infotech, 2002. Available: `http://www.mzoop.com/en/zoop/demo.html`. [cited at p. 93]

[Jac07] JacORB. Jacorb, 2007. Available: `http://www.jacorb.org/`. [cited at p. 138]

[Mar96] R. C. Martin. The dependency inversion principle. 1:1, 1996. Available: `http://www.objectmentor.com/resources/articles/stability.pdf`. [cited at p. 17, 98]

[Mas07] MasterCard. Mastercard product & services - documentation, 2007. Available: `http://www.paypass.com/documentation.html`. [cited at p. 93]

[MK99] Jeff Magee and Jeff Kramer. *Concurrency: state models & Java programs*. Wiley, Chichester, [England], 1999. ISBN 0471987107. 355 pp. Available: `http://isbndb.com/d/book/concurrency`. [cited at p. 44]

[Moh03] Dirk S. Mohl. Short description of ieee 1588. Technical report, http://www.industrialnetworking.com, 2003. 1 pp. Available: `http://www.industrialnetworking.com/support/general_faqs_info/Precise_Time_Sync.pdf`. [cited at p. 122]

[MV07] Jozef Hooman and Marcel Verhoef, Peter Visser. Co-simulation of real-time embedded control systems. In *IFM 2007: Integrated Formal Methods*. LNCS, July 2007. [cited at p. 87]

[NS06a] Jasper Moltke Nygaard and Rasmus Ask Sorensen. Cyperrail concurrency and real-time aspects vdm++ model, 2006. [cited at p. 36]

[NS06b] Jasper Moltke Nygaard and Rasmus Ask Sorensen. Cyperrail planing and navigation vdm++ model, 2006. [cited at p. 35]

[OMGR05] Pat O'Brien, Laith Murad, Simon Gibbs, and Richter Rafey. Smart card product development in an internet-based crm environment. In *DUX '05: Proceedings of the 2005 conference on Designing for User eXperience*, page 56. AIGA: American Institute of Graphic Arts, New York, NY, USA, 2005. ISBN 1-59593-250-X. [cited at p. 93]

[Ove07] Overturetool.org. Overturetool.org twiki, 18 Jul 2007 2007. Available: `http://www.overturetool.org/twiki/bin/view`. [cited at p. 121]

[PCFW05] Gerardo Pardo-Castellote, Bert Farabaugh, and Rick Warren. An introduction to dds and data-centric communications. 2005. Available: `www.omg.org/news/whitepapers/Intro_To_DDS.pdf`. [cited at p. 97]

[Pre05] Robert S. Pressman. *Quality Management*, volume 6 of *Software Engineering: A Practitioner's Approach.*, pages 746–474–749. McGraw-Hill Education, 2005. [cited at p. 98]

[SAE06] SAE. The sae architecture analysis & design language (aadl). 2006. Available: `www.aadl.info`. [cited at p. 6]

[SK07] Lars Sarbk and Martin Kjeldsen. Examination of scade in system engineering. 2007. Available: `http://ihase.wikispaces.com/space/showimage/scade-shortpaper.pdf`. [cited at p. 6]

[SS07] Peter Sanders and Dominik Schultes. Engineering fast route planning algorithms. 2007. [cited at p. 68]

[Sta02] William Stallings. *Cellular Wireless Networks*, pages 287–287–297. Wireless Communications & Networks. Prentice Hall, 2002. [cited at p. 63]

[Sys07] CSK Systems. Csk holdings : Vdm, 2007 2007. Available: `http://www.csk.com/support_e/vdm/index.html`. [cited at p. 121]

[Tow99] W. Townsley. Rfc 2661 layer two tunneling protocol "l2tp", 1999. [cited at p. 122]

[TYKR01] OGINO Takahiko, SATO Yasuo, SEKI Kiyotaka, and TSUCHIYA Ryuji. Cyberrail - the fabric of its enhanced by railway transport. 2001. [cited at p. 11, 12, 108]

[Ubi07] Ubisense. http://www.ubisense.net/, 2007. Available: `http://www.ubisense.net/`. [cited at p. 28, 96]

[Und06] Galileo Joint Undertaking. Galileo - the european programme for global navigation services. pages 11–12–2007, 2006 2006. Available: `http://www.galileoju.com/page.cfm?voce=s2&idvoce=38&plugIn=1`. [cited at p. 92]

[vK96] van Katwijk. Real time formal specification using vdm/sup ++/. page 17, 1996. [cited at p. 13]

[Wag05a] Stefan Wagner. Engineering distributed objects - principles & applications - volume 1, 2005. [cited at p. 27, 106]

[Wag05b] Stefan Wagner. Engineering distributed objects - principles & applications - volume 2, 2005. [cited at p. 106, 127]

[Wika] Wikipedia. Greater tokyo area. [cited at p. 96]

[Wikb] Wikipedia. Java remote method protocol. Available: `http://en.wikipedia.org/wiki/java_remote_method_protocol?oldid=167720298`. [cited at p. 106]

[Wik07] Wikipedia. Tokyo population, 2007. [cited at p. 9]

[Zha00] Yilin Zhao. Mobile phone location determination and its impact on intelligent transportation systems. *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS*, 1, 2000. [cited at p. 28, 93]

[ZMW06] Petros Zerfos, Xiaoqiao Meng, and Starsky H. Y. Wong. *A Study of the Short Message Service of a Nationwide Cellular Network*, pages 263–264–268. Proceedings of the 6th ACM SIGCOMM conference on Internet measurement. ACM, New York, NY, USA, 2006. ISBN 1-59593-561-4. [cited at p. 66]

Appendices

Appendix A

Java VDM Trace Unit Framework

A.1 Introduction

The JAVTU framework is a generic logging framework which mirrors the behavior of the VDMTools [Sys07] trace log feature. However since this is a real distributed system, it has extra complications that must be taken into account. The main reason for developing JAVTU, is to be able be able to see inter communication timing characteristics between distributed systems and display the information in the Overture Showtrace application plugin [Ove07] for Eclipse SDK [Ecl07]. This framework will give future developers the possibility of implementing a prototype of an existing VDM++ model and verify certain practical and non-formal factors, with the same set of tools and procedures.

This appendix describes the design of the JAVTU framework and how it is envisioned to be used, in a simple example application. Certain aspects about the present state of the framework and the functionality will be discussed at the end of this appendix.

A.2 Defining the Functionality

The initial idea for this framework was to develop a component which could be integrated into an application in a transparent and non-invasive manner, reflecting the current behavior of the VDMTools VICE application. However certain obstacles emerge when trying to acquire the same level of runtime information in a distributed system, while maintaining acceptable real-time performance. Having a central entity to control message ID's and global object reference state across systems, would greatly increase latencies and make the execution traces inaccurate. The solution is to maintain local trace files at each CPU and use a merging application to weave each distributed trace file into a single entity. The process of combining the trace files includes additional manipulation to assign the unknown runtime parameters, based on either automatic analysis or user interaction.

Figure A.1 illustrates how the merging paradigm works - Each distributed application

creates a trace file with the information available at runtime.

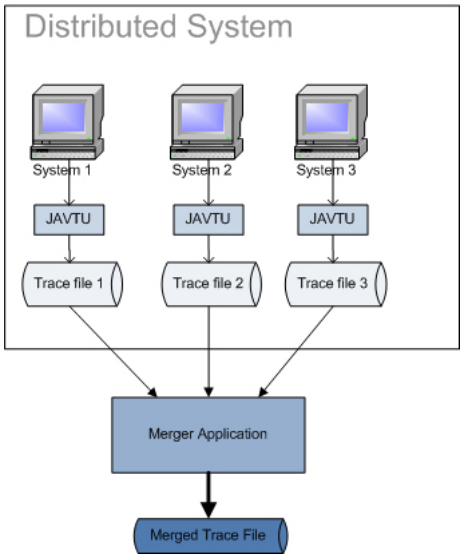


Figure A.1: The process of merging three trace files.

The trace files generated are used as input in the merger application which combines the files into a single trace file readable in the Showtrace application. The advantage of this approach is that no central entity is needed to control the logging process and this enables a more simple usage of the framework.

The concept of consistent time, is another well known issue in a distributed paradigm and is a fundamental parameter in the logging mechanism. Without a synchronization of the time attributes across the systems, the log entries are useless, since replies may be received before a request is issued. This is not an issue in a VDM++ model, since the CPU's are not actually processing asynchronous and VDMTools VICE executes in a controller and sequential environment. Since a prototype implementation measures and relies on practical timestamps, opposed to logical time, and a *consistent snapshot* approach used in group communication context [Bir05], is not sufficient in a performance wise context. The interesting aspect in the prototype performance, is not primarily program correctness but rather performance based on a specific architecture and communication paradigm. To manage the time synchronization issues, an implementation of the IEEE1588 Precise Time Synchronization protocol [Moh03] is included in the JAVTU framework. IEEE1588 relies on a master/slave paradigm and is used to synchronize time continuously and at the same time take connection latencies into account. The protocol does however rely on UDP broadcast and this does impact the degree of distribution when deploying a system. VPN communication technologies such as PPTP [Ham99] and L2TP [Tow99] can be used to create LAN environments where UDP transmissions are possible. However, note that this increases the overall communication latency between the systems due to the added protocol overhead. Using this protocol ensures that the distributed systems have a sufficient synchronized time notion at all times.

A.3 The JAVTU Framework Composition

The JAVTU framework is composed of a single package - JAVTU. This contains both the runtime logging modules and the merger application and associated modules. Figure A.2 illustrates the package content.

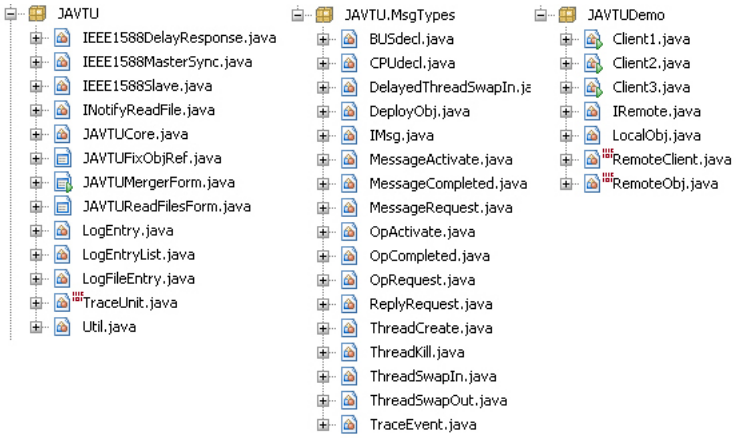


Figure A.2: JAVTU Package overview.

The *JAVTU* package contains the core modules, including the Merger application and the sub package *JAVTU.MsgTypes* contains the individual log file message types. This paper will describe the packages in two different usage scenarios; logging and post processing. The *JAVTUDemo* package is an example application utilizing the JAVTU framework.

A.4 Logging

The logging scenario involves the usage of the framework to log essential information in a distributed system. Figure A.3 shows the classes and packages, when using the framework to log system activity. This section will describe JAVTU by using an example application.

The central entity in each distributed system is the singleton class *TraceUnit*. This class holds all the log messages acquired during execution and makes sure that the correct information is persisted. *TraceUnit* uses a utility class to acquire runtime information and perform I/O activity. The *IEEE1588MasterSync* and *IEEE1588Slave* are the Precise Time Synchronization classes to synchronize each distributed system. Package *JAVTU.MsgTypes* contains all messages types (see Figure A.4).

A.5 Deployment

In listing A.1 a simple non-distributed application illustrates how to deploy an object. First a CPU is instantiated with ID 1 and the *this* pointer, and a communication bus with ID 1 and the "bus1". An object is instantiated and deployed on the CPU 1. When instantiating

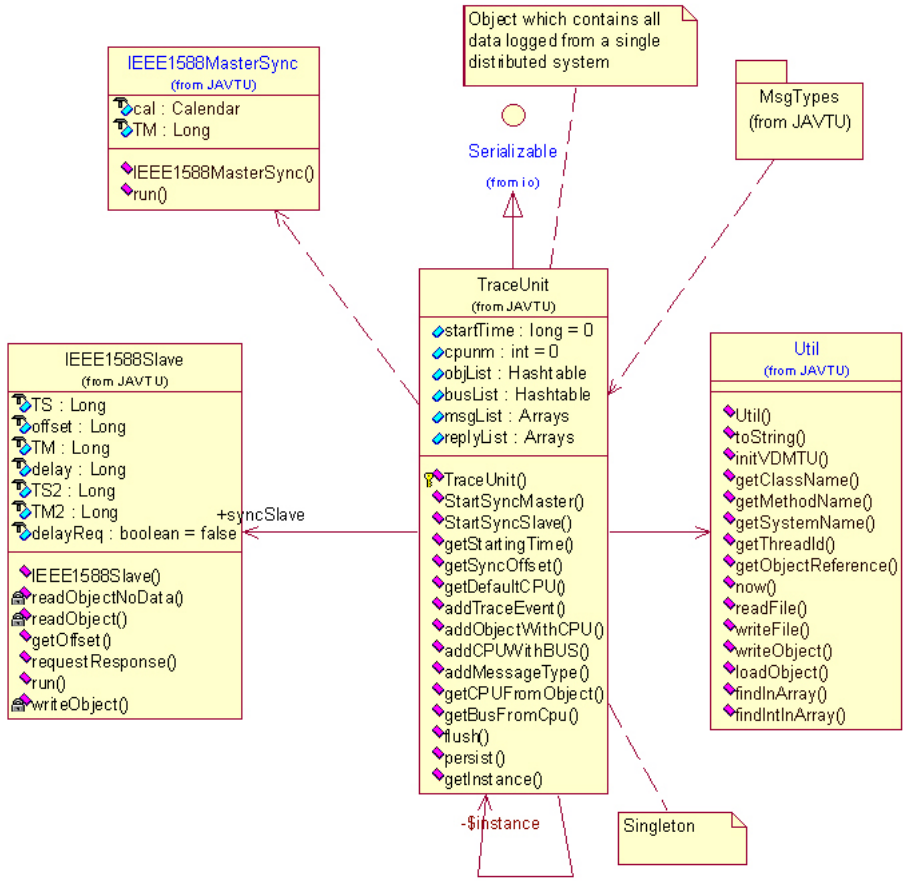


Figure A.3: Class Diagram: Integrating the logging mechanism into an existing application.

JAVTU message types, a *this* pointer is always added as a parameter. This pointer ensures that the *CPUdecl*, *BUSdecl* and the object deployment* log entry can acquire the information necessary to populate the object. Listing A.2 shows how an object utilizes the pointer to obtain the relevant information for the log entry. Finally, the object also adds a reference to itself. Each object entry makes use of the *TraceUnit.addTraceEvent* method to register the object. Additional information may be required to be registered, as is the case with the *DeployObj* log entry. This log entry register the reference to the object and which CPU it is deployed on. This information is very important when merging several trace files into one.

Listing A.1: Deployment of a system using JAVTU.

```
0
```

*Performed indirectly in the *CPUdecl.deployObj method.*

```
1 CPUdecl cpu1 = new CPUdecl(1, this);
2
3 /*Specify communication bus*/
4 BUSdecl bus1 = new BUSdecl("bus1",1);
5 bus1.addCPU(cpu1);
6
7 /*Create object instances*/
8 local1 = new LocalObj();
9
10 //Deploy objects on CPU's'
11 cpu1.deployObject(local1);
```

Listing A.2: DeployObj log entry population.

```
0
1 public DeployObj(Object obj, int cpunm) {
2   objref = Util.getObjectReference(obj);
3   clnm = Util.getClassName(obj);
4   this.cpunm = cpunm;
5   time = Util.now();
6   TraceUnit.getInstance().addTraceEvent(this);
7   TraceUnit.getInstance().addObjectWithCPU(cpunm, objref);
8   TraceUnit.getInstance().deployedObjects.add(this);
9  }
```

Each log entry object in *JAVTU.MsgTypes* share the same basic structure, but differs in the amount of information registered with the *TraceUnit* object. Figure A.4 shows a class diagram of all JAVTU message types in the *JAVTU.MsgTypes* package.

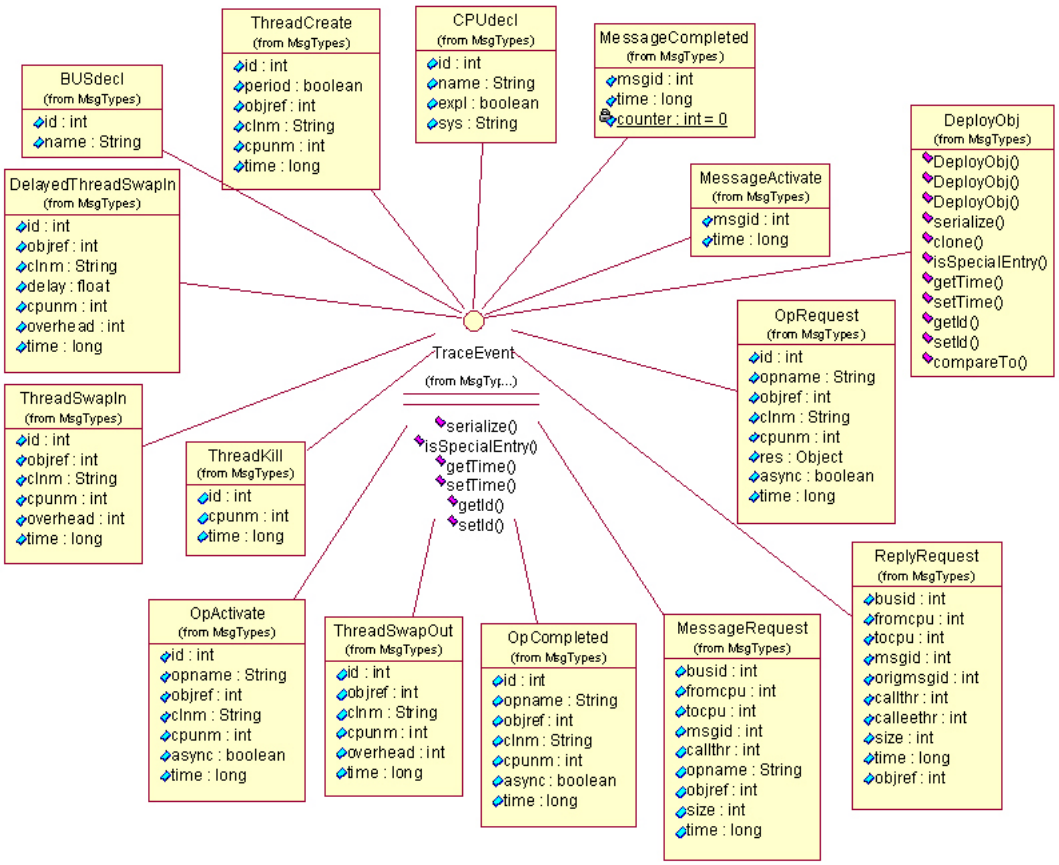


Figure A.4: Class Diagram: Message types in the JAVTU framework.

Each message type implements a *TraceEvent* interface which force the object to behave in a certain way. The abstract method *isSpecialEntry* specifies whether or not the entry is a object which must be processed differently in the post processing stage (e.g. CPU, BUS declarations). Additionally the interface specifies some basic get/set methods and a serialize method to actually create the text based log entry.

A.6 Using the JAVTU framework

The deployment scheme, where local and distributed method invocation occur, can be seen on figure A.5

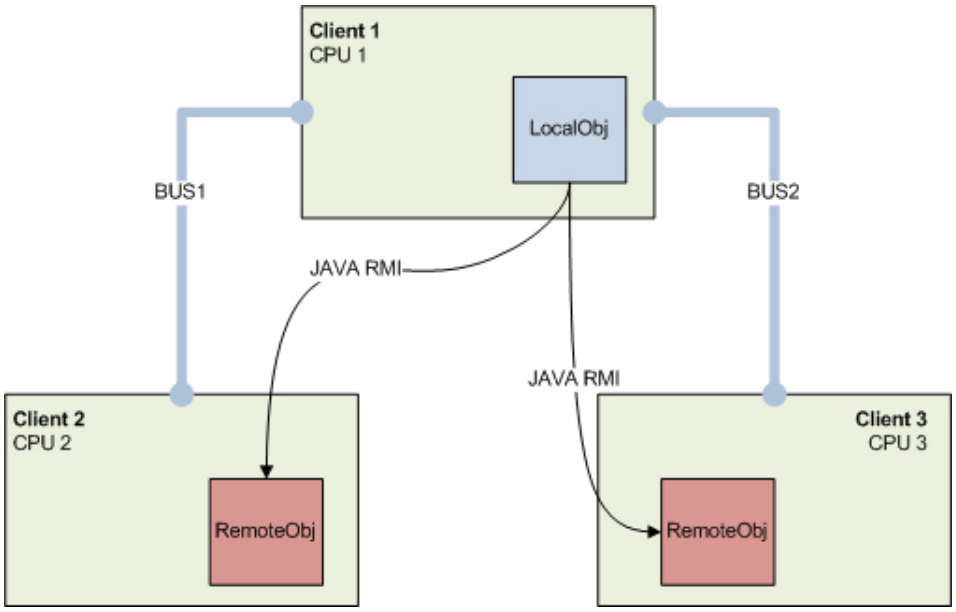


Figure A.5: Simple deployment scheme with three CPU's

The deployment make use of JAVA RMI as middleware technology [Wag05b] to distribute the two objects in *Client 2* and *Client 3*. *Client* communicates with the 2 remotes objects via RMI and invokes the touch method declared in the custom interface *IRemote*. Figure A.6 shows the class diagram for the demo project. For further reference see the source code.

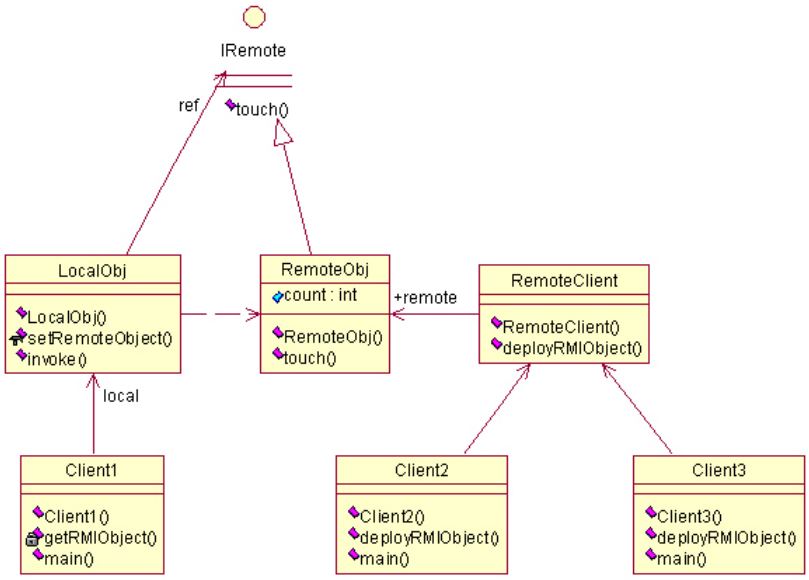


Figure A.6: Class Diagram: The demo application of JAVTU

Listing A.3 and listing A.4 shows how the two objects *LocalObj* and *RemoteObj* are constructed respectively.

Listing A.3: *LocalObj* object structure

```
20 public class LocalObj {
21     IRemote ref;
22     // Creates a new instance of LocalObj
23     public LocalObj() {
24     }
25
26     void setRemoteObject(IRemote o){
27         new JAVTU.MsgTypes.OpRequest(this);      //JAVTU
28         new JAVTU.MsgTypes.OpActivate(this);     //JAVTU
29         ref = o;
30         new OpCompleted(this);                   //JAVTU
31     }
32
33     public void invoke(){
34         new JAVTU.MsgTypes.OpRequest(this);      //JAVTU
35         new JAVTU.MsgTypes.OpActivate(this);     //JAVTU
36         try {
37             new MessageRequest(this);            //JAVTU
38             new MessageActivate(this);           //JAVTU
39             ref.touch();
40             new MessageCompleted(this);          //JAVTU
41         } catch (RemoteException ex) {
42             ex.printStackTrace();
43         }
44         new OpCompleted(this);                   //JAVTU
45     }
46 }
```

Listing A.4: *RemoteObj* object structure

```
 6 public interface IRemote extends java.rmi.Remote{
 7     public void touch() throws RemoteException;
 8 }
 9
10 public class RemoteObj
11 extends UnicastRemoteObject implements IRemote, Serializable {
12     public int count;
13     public IRemote ref;
14     public RemoteObj() throws java.rmi.RemoteException {
15             new OpRequest(this);                 //JAVTU
16             new OpActivate(this);                //JAVTU
17             ref = null;
18             new OpCompleted(this);               //JAVTU
19     }
20
```

```
21      public int touch() throws RemoteException {
22          new OpRequest(this);                          //JAVTU
23          new OpActivate(this);                         //JAVTU
24          new ReplyRequest(this);                       //JAVTU
25          new MessageActivate(this);                    //JAVTU
26          count = count +1;
27          System.out.println("Touch called: " + count);
28          new MessageCompleted(this);                   //JAVTU
29          new JAVTU.MsgTypes.OpCompleted(this);         //JAVTU
30          return count;
31      }
32  }
```

The important thing to note in listing A.3 and A.4 is the usage of operation and message entry cycles[†]. Listing A.3, line 27, 28 and 29 shows how a method must declare a *OpRequest* and a *OpActivate* upon entry and upon existing a *OpCompleted*. This tells the JAVTU framework that a method invocation has occurred and it will be logged accordingly. When executing a remote method invocation over a BUS connection, a message request cycle must be declared. This occurs in listing A.4, line 37, 38, 40. Note that prior to any message entry declaration, a operation request cycle must be completed. The response to the message request can be seen in listing A.4, line 24, 25 and 28. Note that the *MessageCompleted* and *OpCompleted* must be declared before any data can be returned.

The deployment of the JAVTU framework for this is illustrated in listing A.5

Listing A.5: Deployment of JAVTU for *Client 1* (*JAVTUDemo.Client1.java*)

```
29 public class Client1 {
30  LocalObj local;
31  public Client1() {
32  try {
33     //TraceUnit.getInstance().StartSyncMaster();
34     ...
35     //Create CPU*/
36     CPUdecl cpu1 = new JAVTU.MsgTypes.CPUdecl(1, this);
37     //Create object instances*/
38     local = new LocalObj();
39     //Specify communication bus & deploy CPU's*/
40     BUSdecl bus1 = new BUSdecl("bus1",1);
41     BUSdecl bus2 = new BUSdecl("bus2",2);
42     bus1.addCPU(cpu1);
43     bus2.addCPU(cpu1);
44     //Deploy objects on CPU's'
45     cpu1.deployObject(local);
46     //Acquire RMI remote object references*/
47     Object remoteObject1 = getRMIObject(...);
```

[†]A message cycle implies that *OpRequest*, *OpActivate* and *OpCompleted* has been declared. An operation cycle implies that a *MessageRequst* (or *ReplyRequest*), *MessageActivate* and *MessageCompleted* has been declared.

```
48      Object remoteObject2 = getRMIObject(...);
49      local.setRemoteObject((IRemote)remoteObject1);
50      Thread.sleep(1000);
51      local.invoke();
52      local.setRemoteObject((IRemote)remoteObject2);
53      Thread.sleep(1000);
54      local.invoke();
55      ...
56  }
```

The deployment is fairly straightforward and self explanatory. Note however that cpu1 is deployed on both bus1 and bus2 as depicted in figure A.5. Note also the command on line 33 that is commented out. If it is taken in, this command would initiate the IEEE1588 Master and initiate synchronization with any active slaves on the local network. The slaves would be activated by invoking the *StartSyncSlave* method on the *TraceUnit* object. This is however not used in the demo application since the systems are deployed on the same CPU. The source code for bootstrapping the code and deploying the RMI objects will not be covered here. Deployment of the JAVTU framework is similar to the entries in listing A.1. For further reference see the source code.

A.7 JAVTU Results

The outcome of running the demo application is three JAVTU resource files in the runtime folder with the extension **.javtu.res*. These three files represent the log entries for *Client 1*, *Client 2* and *Client 3* respectively. Which is which plays no role, however the filenames are composed of the system hostname and a process ID. The JAVTU framework automatically save these objects every 10 seconds during runtime. Note that these objects are not actual human readable log files, but binary files which the JAVTU Merger application can read. Next step is to merge the units into a single log file.

A.8 Post Processing

The three JAVTU resource files must now be merged into 1 readable log file. This is done with the classes illustrated in figure A.7. The key component is the *JAVTUCore* class. This class is both the placeholder for the distributed resource files and the toolbox for manipulating the files into a single execution trace. Several unknown factors exist during runtime - especially in regards to bus communication.

- What is the remote reference of the receiver object?
- What is the next global synchronous message ID number?
- Which original message request is a reply request responding to?

These issues must be dealt with in the post processing phase and the *JAVTUCore* class contains the methods to handle this.

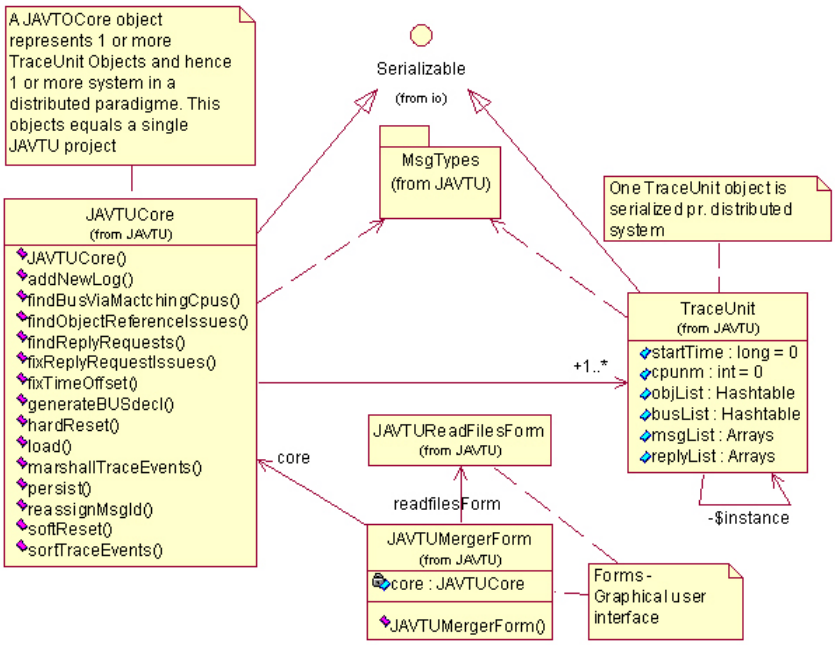


Figure A.7: Class diagram: Modules used for post processing in the JAVTU Merger application.

The *JAVTUCore* object is controlled by the user, utilizing a graphical user interface represented by *JAVTUMergerForm* and *JAVTUReadFilesForm*. Each resource file, created from the distributed applications, are actually serialized *TraceUnit* objects, which are read into the *JAVTUCore* object and manipulated accordingly. Figure A.8 shows the *JAVTUMergerForm* primary interface.

The GUI has a number of buttons:

Open: Open resource file and add to current project

Reload: Reload current resource files.

Reinitialize: Unload all log files and meta data - alternative to restarting the application.

Save Merged file: save merged file to a custom destination.

Step 1: Merging phase 1 with a number log file manipulations. This step will require user interaction.

Step 2: Merging phase 2 with the final log file manipulations. Merged file is finished.

File: Basic load/save project option.

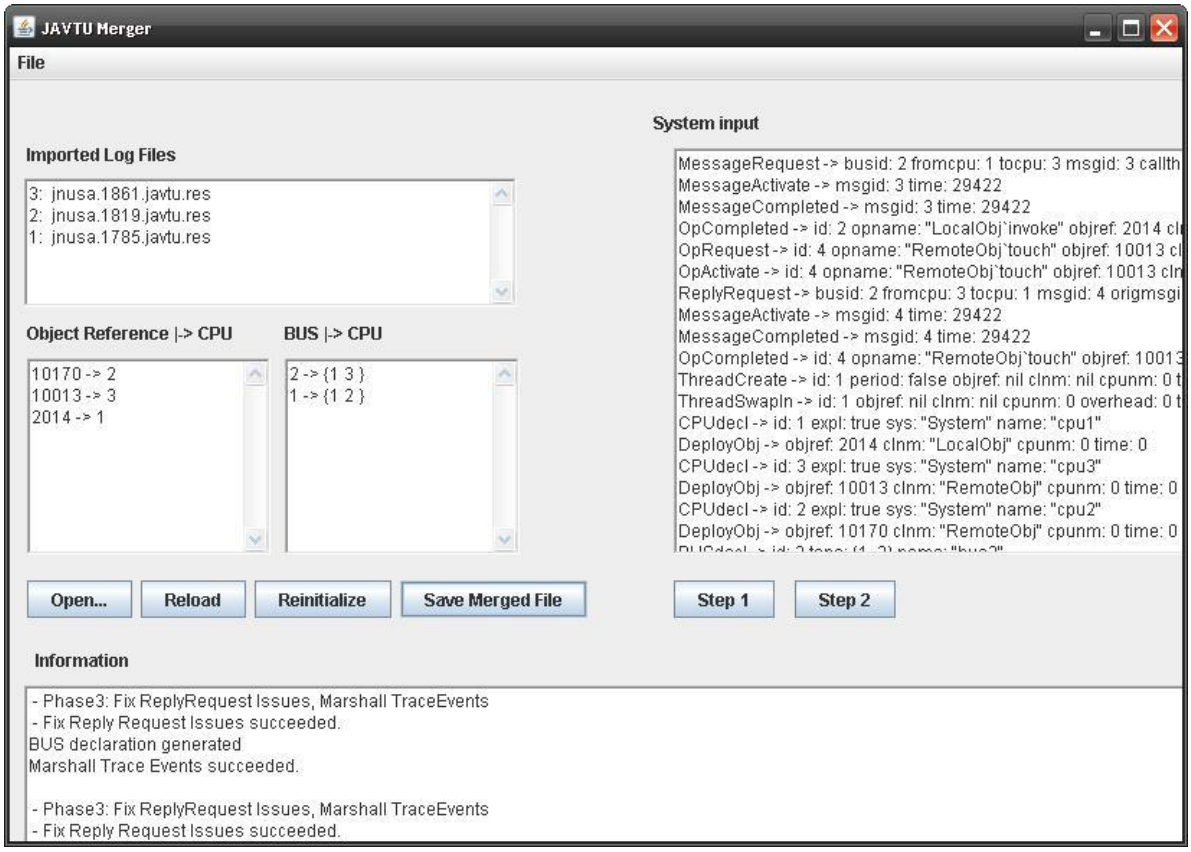


Figure A.8: Screenshot of the JAVTU Merger.

Additionally, the user interface displays deployment information, such as which objects are deployed on which CPU'es, which resource files a loaded and which CPU's are deployed on which BUS'es. This information can be used to confirm the actual deployment of the JAVTU framework is correct. If any errors occurs, it will be displayed in the information area.

When the resource files are loaded, step 1 is performed. This step is responsible for the following manipulations:

1 Sort trace events based on time (*JAVTUCore.sortTraceEvents*)

2 Fix time offset on every trace event. Each distributed system logs the time as the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC. This is a very large number and to counteract this, the smallest time registered in the entries, is subtracted from all the other trace events. The result is a more manageable unit which is readable in the Overture plugin (*JAVTUCore.fixTimeOffset*).

3 Reassign message ID's on all bus communication so they are consecutive numbered. This only applies to message request cycles and not the reply request cycles (*JAVTU-*

Core.reassignMsgId).

4 Fix object references issues in regards to remote methods invocation over a BUS. This manipulation requires user interaction to precisely define which message requests are connected to which remote object reference (*JAVTUCore.fixObjectReferenceIssues*).

It is not possible to fix the object references in the fourth step (at the present moment) automatically even at post processing, with the amount of information currently in the *MessageRequest* log type. This is a tedious step, if the execution traces are very large - the process is depicted in figure A.9.

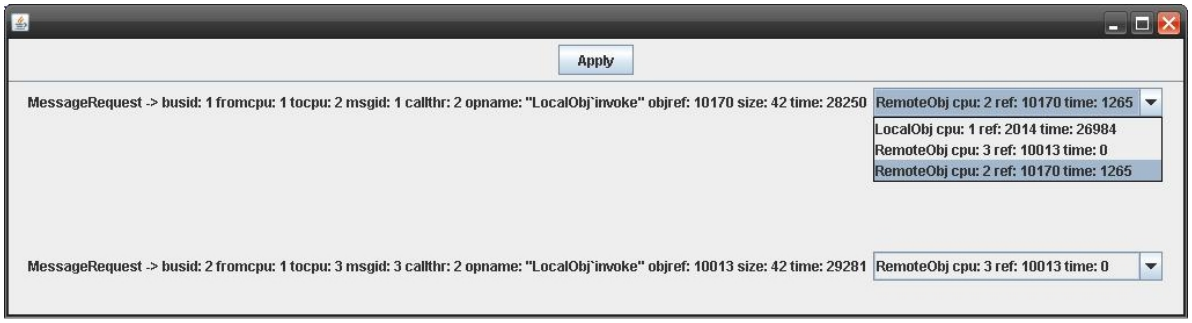


Figure A.9: Screenshot of JAVTU Merger, fixing object references.

After the user has assigned each message request to an object reference, step 2 can be initiated. This step involves the following manipulations:

1 Fix reply request cycles in regards to message ID's and object references. This information is now available after step 1 has been executed (*JAVTUCore.fixReplyRequestIssues*)

2 Generate BUS, CPU, thread declarations and marshal all trace events. The merged log file is now generated and ready to be saved.

When the 2 steps above have been executed successfully, the log file can be saved and loaded into the Overture Showtrace tool. The result of the merged log file is illustrated in listing A.6.

Listing A.6: Merged log file from the demo application (*merged.logfile.juvta*)

```
0 ThreadCreate -> id: 1 period: false objref: nil clnm: nil cpunm: 0 time: 0
1 ThreadSwapIn -> id: 1 objref: nil clnm: nil cpunm: 0 overhead: 0 time: 0
2 CPUdecl -> id: 1 expl: true sys: "System" name: "cpu1"
3 DeployObj -> objref: 2014 clnm: "LocalObj" cpunm: 0 time: 0
4 CPUdecl -> id: 3 expl: true sys: "System" name: "cpu3"
5 DeployObj -> objref: 10013 clnm: "RemoteObj" cpunm: 0 time: 0
6 CPUdecl -> id: 2 expl: true sys: "System" name: "cpu2"
7 DeployObj -> objref: 10170 clnm: "RemoteObj" cpunm: 0 time: 0
8 BUSdecl -> id: 2 topo: {1, 3} name: "bus2"
9 BUSdecl -> id: 1 topo: {1, 2} name: "bus1"
10 ThreadCreate -> id: 4 period: false objref: nil clnm: nil cpunm: 3 time: 0
11 ThreadSwapIn -> id: 4 objref: nil clnm: nil cpunm: 3 overhead: 0 time: 0
12 OpRequest -> id: 1 opname: "RemoteObj`RemoteObj" objref: 10013 clnm: "
    RemoteObj" cpunm: 0 async: false time: 141
13 OpActivate -> id: 1 opname: "RemoteObj`RemoteObj" objref: 10013 clnm: "
    RemoteObj" cpunm: 0 async: false time: 141
14 OpCompleted -> id: 1 opname: "RemoteObj`RemoteObj" objref: 10013 clnm: "
    RemoteObj" cpunm: 0 async: false time: 141
15 DeployObj -> objref: 10013 clnm: "RemoteObj" cpunm: 3 time: 141
16 MessageRequest -> busid: 1 fromcpu: 1 tocpu: 2 msgid: 1 callthr: 2 opname:
    "LocalObj`invoke" objref: 10170 size: 42 time: 28391
17 MessageActivate -> msgid: 1 time: 28406
18 OpRequest -> id: 3 opname: "RemoteObj`touch" objref: 10170 clnm: "
    RemoteObj" cpunm: 2 async: false time: 28406
19 OpActivate -> id: 3 opname: "RemoteObj`touch" objref: 10170 clnm: "
    RemoteObj" cpunm: 2 async: false time: 28406
20 ReplyRequest -> busid: 1 fromcpu: 2 tocpu: 1 msgid: 2 origmsgid: 1 callthr
    : 2 calleethr: 3 size: 42 time: 28406 objref: 10170
21 MessageActivate -> msgid: 2 time: 28406
22 MessageCompleted -> msgid: 2 time: 28422
23 OpCompleted -> id: 2 opname: "LocalObj`invoke" objref: 2014 clnm: "
    LocalObj" cpunm: 1 async: false time: 28422
24 OpRequest -> id: 2 opname: "LocalObj`setRemoteObject" objref: 2014 clnm: "
    LocalObj" cpunm: 1 async: false time: 28422
25 OpActivate -> id: 2 opname: "LocalObj`setRemoteObject" objref: 2014 clnm:
    "LocalObj" cpunm: 1 async: false time: 28422
26 OpCompleted -> id: 2 opname: "LocalObj`setRemoteObject" objref: 2014 clnm:
    "LocalObj" cpunm: 1 async: false time: 28422
27 MessageCompleted -> msgid: 1 time: 28422
```

The architectural overview, execution overview and CPU1 execution output from Overture Showtrace, can be seen in A.10, A.11 and A.12 respectively.

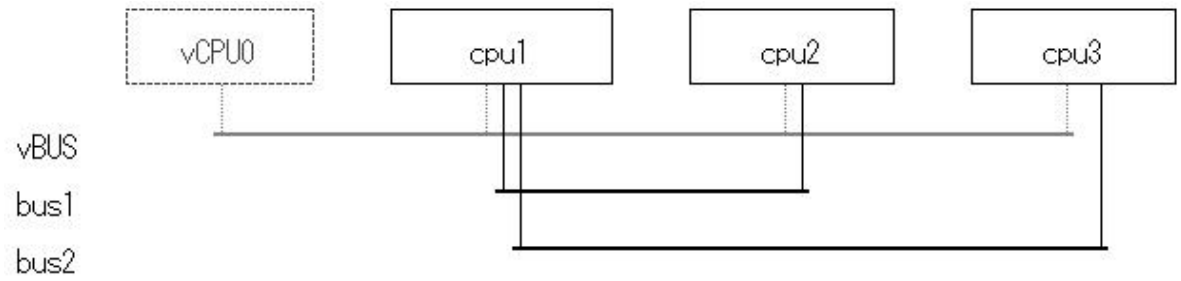


Figure A.10: Screenshot of Overture Showtrace - Architectural overview.

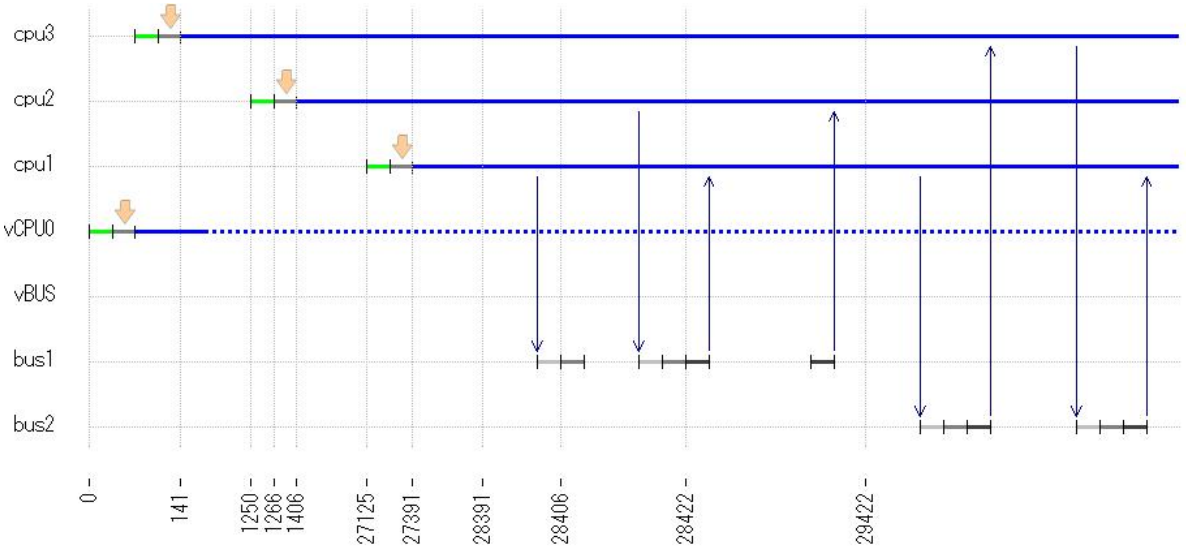


Figure A.11: Screenshot of Overture Showtrace - Execution overview.

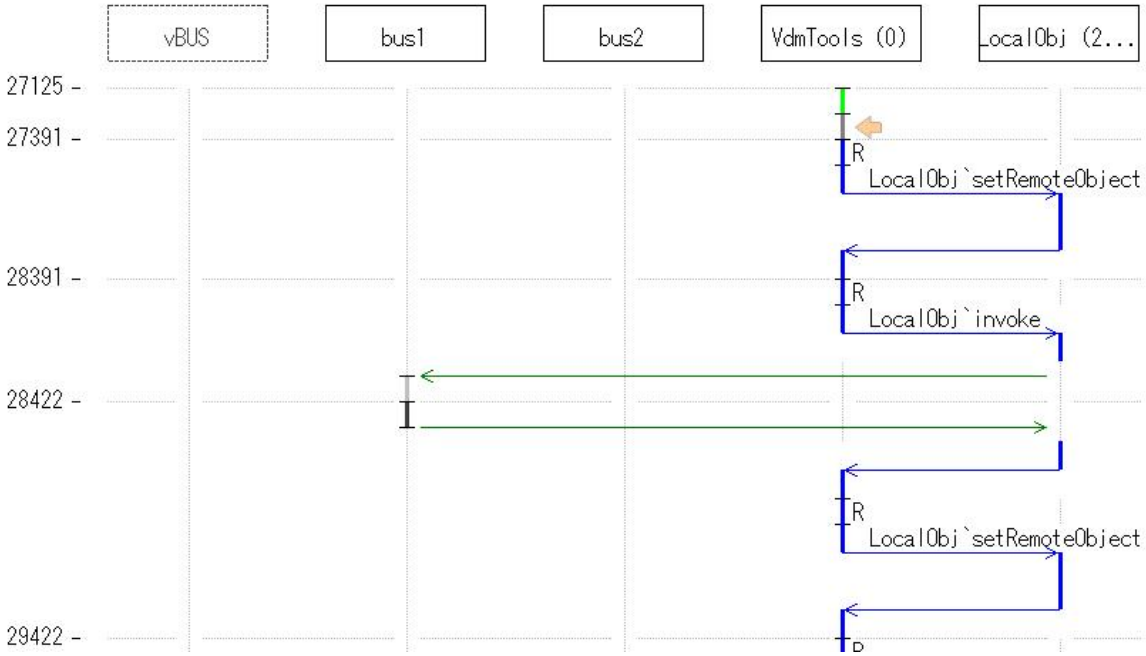


Figure A.12: Screenshot of Overture Showtrace - CPU1 execution.

A.9 JAVTU Framework Status

At the current stage of implementation, the JAVTU framework has a number of limitations, due to the fact that all state information is acquired at the application layer, as opposed to OS layer (e.g. virtual OS layer in VDMTools). For this reason certain log entries does not have any impact on the trace. E.g. *OpRequest/OpAvtivate* does not have the intended behavior where an *OpRequest* entry identify that a request for execution has occurred and *OpActivate* identify that the execution has commenced. The same is the case with *MessageRequest* and *MessageActivate*. When a *MessageRequest* occurs when executing model in VDMTools, the MessageActivate will not appear until the BUS is available. An additional restriction is that only one thread per CPU can exist, due to the early stage of development. No apparent reason have been identified for this not to work with multiple threads, however this is again widely dependent on the amount of information available from the JAVA virtual machine. These issues may be solved by using some kind of Java Virtual Machine hook methods or by using the Java VM Debug interface to obtain relevant information. The accuracy of the log entries, in regards to time, cannot be improved, without being able to obtain detailed information about the underlying execution behavior of the JVM.

With the current design, the JAVTU framework is very invasive and a user must insert many identical method invocations. This not only makes the code harder to read, but also make the program subject to a more error prone behavior, in regards to the trace files. Entries such as operation cycles could be hidden by utilizing a kind of execution pointer or execution object based on a Decorator Pattern [GHJV95]. Entries such as message request cycles are a little more complicated since they rely on which type of communication technology is used. In this demo application, JAVA RMI was used. However an application could just as well be using a Corba ORB such as PrismTech's JacORB [Jac07] or even a third middleware technology. So in regards to BUS communication there's no easy way to solve this. Either the developer use JAVTU as demonstrated or to some extend wrap the middleware technology of choice which declare message request cycles implicit.

Another issue briefly discussed in the Post Processing section, is the unknown runtime state information in regards to *MessageRequest* entries. No apparent automated methods has been identified to solve this and the user has to manually insert the correct object reference (see figure A.9). If a JAVTU project contains hundreds of *MessageRequests*, this step could be very tedious and subject to many errors which can be very hard to find. The *MessageRequest* object instantiation could easily be altered to include an identifier or a textual description. This would allow the JAVTU Merger application to group identical *MessageRequest* entries and more importantly group the object reference assignments. This will however only work when a *MessageRequest* communicates with the same object reference at all times and this may not always be the case. By leaving an identifier or clue for the user, it will be considerably easier to match the object references to a message request.

Due to the use of JAVA's own serializable service, *IO.Serializable*, to marshall generic collections used in *JAVTU.TraceUnit* and *JAVTU.JAVTUCore*, the JAVTU framework cannot be used in conjunction with the J2ME platform. To deal with this issue, explicit implementations of the serializing routines must be specified for all collections and other non atomic types used in the JAVTU framework. Currently there is no JAVA road map showing seri-

alizing support in J2ME due to the vast resource requirements used to perform this type of operation.

A.10 Concluding Remarks

Even though certain aspects of JAVTU is impractical, the framework does work for the intended use. For this thesis a distributed logging utility was needed, which mirrored the behavior of the VDM++ execution traces and was compatible with the Overture Showtrace tool. This was achieved and a formal VDM++ model and an prototype implementation can now be compared side by side, using the same set of tools and procedures.

Appendix B

Repeater Application

The Repeater is a program developed to ease the deployment of VDM++ VICE models, that deploys hundred to several thousand objects on many different CPU's. In general it can repeat code that only have minor changes in each repetitions. Figure B.1 shows the GUI of the Repeater with numbers corresponding to a description.

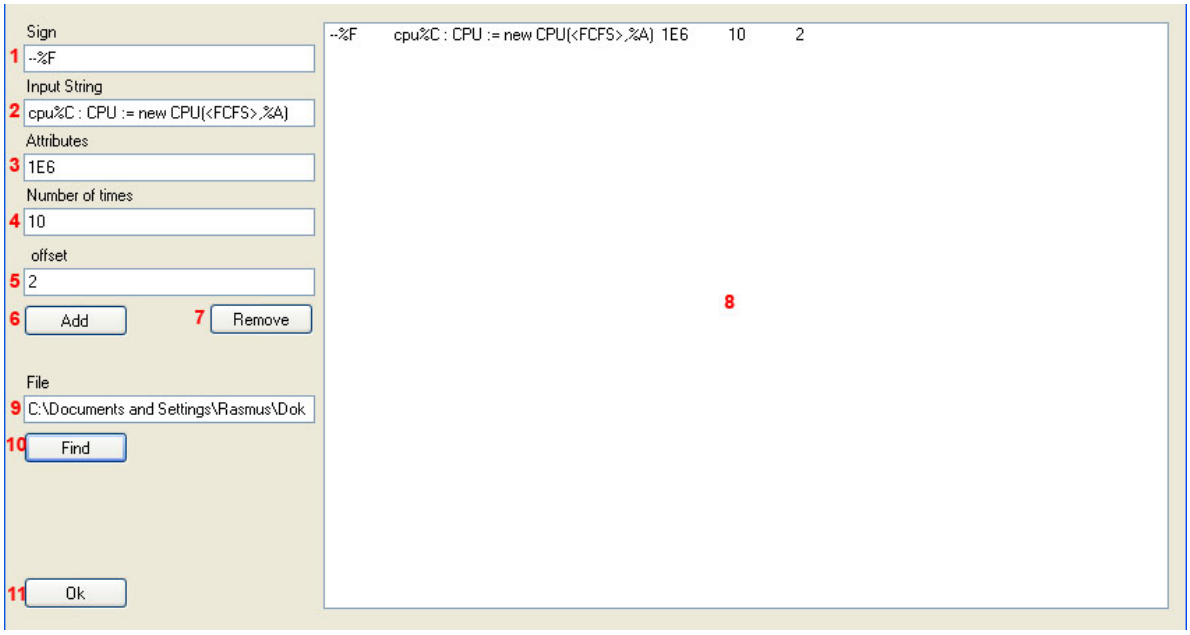


Figure B.1: The GUI of the Repeater

1 The sign is a string in the file, the Repeater looks for. At the signs location, it will insert the repetitions. The sign can be any string and the only requirement is that is has to

be unique in the file. Listing B.1 shows how a sign can be written into a file. The sign is on line 14 and it will be removed when the repetitions are inserted.

Listing B.1: Snapshot of a file with a sign

```
0 public static ctrl : Controller := new Controller();
1 public static sms : SMSGateway := new SMSGateway();
2 operations
3 CRSystem : () ==> CRSystem
4 CRSystem()==
5 (
6   cpu1.deploy(cb);
7   cpu1.deploy(grid2);
8   cpu3.deploy(grid);
9   cpu3.deploy(ctrl);
10  cpu2.deploy(apm);
11  cpu4.deploy(sms);
12
13  --Deploy
14 )
15 end CRSystem
```

2 This is where the string, that shall be repeated, is written. It is a possible to insert a counter (%C), to create differences in each repetition. The counter can only count in numbers, it is not possible to define an alphabetic range. There can be more than one counter in an input string, they will however be counted up synchronously. There can be inserted a attribute sign (%A) which inserts what is written in field 3. There can also be more than one attribute in a input string.

3 This is where the attributes are written. There has to be the exact same number as there are attribute signs in the input string. The attributes has to be separated with a %.

4 This is the number of times the input string has to be repeated. It is also the amount of times that the counters will be incremented.

5 This is the offset of the counters, in this case there are two. This means that the first time the string will be inserted, the number inserted into the counters place will be 2, then 3 etc.

6 Adds the above fields as an repetition command to the input list.

7 Removes the selected repetition command from the input field.

8 This is the input list and it contains all the added repetition commands. Each repetition command needs to have a unique sign.

9 Path to the file.

10 Opens a file dialog and the selected path will be added to the file field (9).

11 Applies all the repetitions commands in the input list to the selected path.