Cairo University

Faculty of Engineering

Credit Hours System

Communication and Computer Engineering

CCEN481: Graduation Project – 2

Spring 2017

Mentor®
A Siemens Business

VRM Performance Analyzer

Prepared by:

HUSSEIN FADL OSMAN

MAHEED HATEM MOSTAFA

MOHAMMED AMR ABDELFATTAH

OMAR MOHAMED IBRAHIM

Supervised by:

DR. MAYADA HADHOUD

Sponsored by:

MENTOR GRAPHICS

ACKNOWLEDGEMENTS

We would like to thank **Dr. Mayada Hadhoud** for her unparalleled mentorship, guidance, and encouragement throughout our project's inception, design, and final implementation. We would like to thank everyone at Mentor Graphics for their commitment, patience, and for the opportunity they have given us to work on this project. Special thanks go to **Ayman Sheirah** for his ongoing support and for helping us get the tools that we needed. To **Mennatallah Amer** for her help with research, for introducing us to her domain, and for helping us develop a model for the complex system that is VRM. And last but definitely not least to **Amr Hany** for his unfaltering guidance in making sure the project makes it into the final product and for constantly providing feedback on our written thesis.

ABSTRACT

Mentor Graphics Questa Verification Run Manager (VRM) provides users with an adaptable solution for running verification regression. Completing verification regression iterations quickly and efficiently is crucial to the success of any project. However, performance issues may arise with more complex regression runs; this stalls the development process. Our project focuses on using process mining techniques to analyze the performance of VRM. The result of which is a tool for use by Mentor Graphics affiliates to debug verification regression runs and resolve misconfigurations before they become a hazard for project development.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

ACK	Acknowledgement
AI	Artificial Intelligence
API	Application Program Interface
BPMN	Business Process Model and Notation
DUT	Design Under Test
ERP	Enterprise Resource Planning
ESL	Electronic System Level
ETL	Extract Transform and Load
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
IDE	Integrated Development Environment
IT	Information Technology
OS	Operating System
RMDB	Run Manager Database
RTL	Register Transfer Level
SAP	Systems, Applications and Product (German Company)
SoC	System on Chip
SW	Software
TCL	Tool Command Language
UCDB	Unified Coverage Database
UI	User Interface
VRM	Verification Run Manager
XML	eXtensible Markup Language
Yasper	Yet Another Smart Process EditoR

Chapter 1 :

Introduction

The analysis of large systems has always been a difficult engineering task. Due to the ever-growing nature of system requirements, this might not change for the near future. This is especially true of flexible systems. Such systems allow numerous customizable parameters to meet the needs of the user. With great flexibility, conventional means of analyzing performance and the resultant system model fall short. This project focuses on using techniques from a new fast-growing field focused on system analysis, Process Mining, to analyze the performance of a large complex and immensely flexible system, Mentor's Questa Verification Run Manager. We begin by providing the reader with some basic background information before presenting our project idea.

1.1 Background Information

1.1.1 Design Verification

Design verification is the process of ensuring that a specified design adheres to the requirements set forth at beginning of the product's inception. If the final product does not meet the customer's requirements, then the product would have to be redesigned and the costly tools needed to turn the design into a tangible physical product would have to be drastically changed. This would lead to an increase in the time taken by and the cost of product development resulting in the dissatisfaction of customers and potential loss of business. For this reason, verification has become a corner stone of the product development life cycle and can single handedly lead to the project's failure or success.

Verification can come in many flavors according to the extent of verification required and frequency. Each type of verification is suited for a specific need according to what point the product is in its life cycle. Tools have been created to suit these needs. One such tool, Mentor Graphics' Questa Verification Run Manager (VRM), aims at providing a platform for verification of any type and in any environment.

1.1.2 Verification Run Manager (VRM)

Verification Run Manager is a tool developed by Mentor Graphics within Questa Verification Solutions. The tool is designed to automate most of the verification process in any

system. It is an automated flexible tool designed to increase the efficiency of the verification processes.

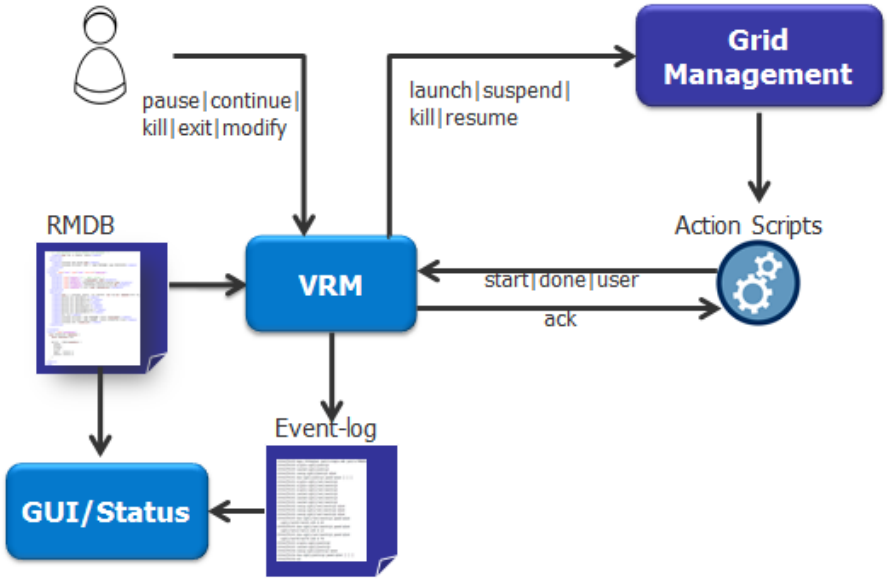

Figure 1-1: VRM Block Diagram

As illustrated in Figure 1-1the execution of VRM goes as follows:

- A VRM process is set up using a configuration file written in RMDB format (Run Manager Database) which is a mixture between XML and TCL.
- VRM then communicates with a Grid Management System or in some cases creates worker instances on a local machine to execute action scripts.
- The action scripts send event signals to VRM.
- The user can manipulate the process through VRM during execution.
- The event-log of the process is generated with all the action scripts events.
- The results of the regression are stored in multiple files including multiple log files and a UCDB file for coverage information.

1.2 Motivation

As it stands, the user or VRM engineer does not have a substantial tool that explains the performance of a VRM regression instance. This presents a few performance challenges as summarized below.

- ***User Misconfiguration and Resource Utilization***

Not all users are fully versed in the functionality VRM provides; because of this, the user might not expect VRM to behave the way they intend for it to behave. A user misconfigured RMDB file might lead to avoidable performance slowdowns.

- ***Replaying processes requires running all the action scripts***

As it stands, VRM does not provide a utility to replay event log data post execution. In order to debug execution of an RMDB file and conduct performance analysis, one would need to rerun an entire regression.

- ***Complexity Due to Generality***

VRM is a generic tool that provides many powerful features. Endless regression scenarios can be configured. This complexity is required to match user needs in different regression environments. As such, the resultant regression process can be overwhelming for even the most experienced user. This process needs analysis to convey core performance metrics.

- ***Enhance VRM***

VRM can use regression history to make better decisions.

1.3 Project Idea

Our project focuses on developing a tool to aid in analyzing regression performance in VRM, the VRM Performance Analyzer. The tool offers three main features.

- ***Process Modeling***

During operation of the tool, the VRM process is modeled by processing both an RMDB file and an event log. The output of this stage is an abstracted process model of the VRM/RMDB combination. This aids the user in understanding what VRM would have worked on during execution.

- ***Replay***

The tool allows an event log to be replayed on the abstract model it created. This helps in debugging event data to trace where misconfiguration might have taken place. In addition, it also

helps in analyzing event data performance during execution and in checking how VRM responds to user input and configuration.

- ***Performance Visualization***

The tool provides a user-friendly utility to visualize and monitor the performance of the generated process model. The tool renders visualizations of bottlenecks in the regression run if they exist along with performance statistics extracted in the performance analysis stage. This visualization provides an easier way to trace problems and a more effective way to display overall system performance.

1.4 Document Overview

In this report, we present our tool, VRM Performance Analyzer. We first introduce the reader to the necessary background needed to comprehend our solution. Since the number of different areas involved in this project are numerous, the chapter is split up into three distinct sections. One section introduces Design Verification in detail. The second section explains the structure of VRM execution and the terminology used when referring to VRM regression runs. The third and final section explores the rapidly evolving field of Process Mining, of which our approach is based on.

In the third chapter, we look at available process mining commercial software, namely Fluxicon Disco and Celonis Pi. We also explore an instance in the literature where a custom solution is formulated to tackle a real life problem using Process Mining.

After the basis is laid, we dive into the design of our tool. We explain the inner connections of our tool using a block diagram. We then describe each module in our system in detail from parsing and initial modeling all the way to final performance analysis and visualization. We then dedicate two chapters to the tools and libraries used in designing and creating our tool and to the testing plan used in ensuring our tool meets the requirements we set for it.

In chapter 7, we conclude our findings and outline future work that we see can be beneficiary to the operation of our tool in the real world. We provide an appendix containing a brief but thorough User Manual describing how to operate our tool.

Chapter 2 :

Necessary

Background

In this chapter, we will provide necessary background information crucial to understanding our solution. Since the design verification process is of great importance in the project life cycle as it controls the quality of the final product and reduces the risk of re-spins, we will start by exploring it. With its significance in mind, we will move on to discuss Mentor Graphics' tool, Questa Verification Run Manager (VRM), and discuss its terminology, its main features, and its execution.

Developing a performance analysis tool to detect bottlenecks and performance issues due to misconfiguration of RMDB files, bad resource utilization, etc. requires full understanding of VRM's process model. The last section of this chapter is devoted to process mining, a technique for process management that allows for the analysis of business processes based on event logs.

2.1 Design Verification

2.1.1 Introduction

Design verification is a process of demonstrating functional correctness of a design under test and assuring that the design conforms to its specification. The process of verification is usually coupled with questions such as "Does the proposed design behave as intended?" and "In what ways does the design deviate from what is expected?" As complex as it may seem, many development projects specify design verification testing as a major contract requirement and many customers may even invest a substantial amount of resources on ensuring that such testing is completed to satisfaction.

The importance of the verification process is unarguable. For many reasons, it is almost impossible to guarantee a design without verification due to incorrect or insufficient specifications, misinterpretations and misunderstandings of specifications, incorrect interactions between hardware, or more commonly due to the unexpected behavior of the system. Moreover, bug escapes to silicon can be costly not to mention the time it would take to "re-spin" the design. These reasons were enough for the verification process to emerge as an elite solution. Although potentially costly, the expense of not testing can be significantly greater; and hence development projects should devote a considerable budget for design verification as a part of the development program.

Many aspects of a design are subject to verification, mainly functional verification, timing verification, and performance verification. Such design aspects can be verified on either one of many levels. In literature, verification techniques vary in their methodology, requirements, structure, complexity, and functionality. Some verification methods are listed below:

- Simulation based Verification
- Formal Verification
- Semi-Formal Verification
- Assertion-based Verification
- HW/SW Co- Verification
- Coverage-based Verification.

2.1.2 **Simulation-based Verification**

Simulation-based verification is predicting how digital circuits interact and behave using simulation software packages and hardware description languages. Such simulation can be performed at varying degrees of physical abstraction, namely at the transistor level, gate level, register-transfer level (RTL), electronic system-level (ESL) or behavioral level. Interestingly, simulation-based verification allows the user to interact directly and get feedback on their design which is a very effective way to highlight issues and design problems. A simulator exercises the given implementation with a stimulus in the form of a manually or automatically generated test vector an-d verifies the results afterwards as depicted in Figure 2-1. [1]

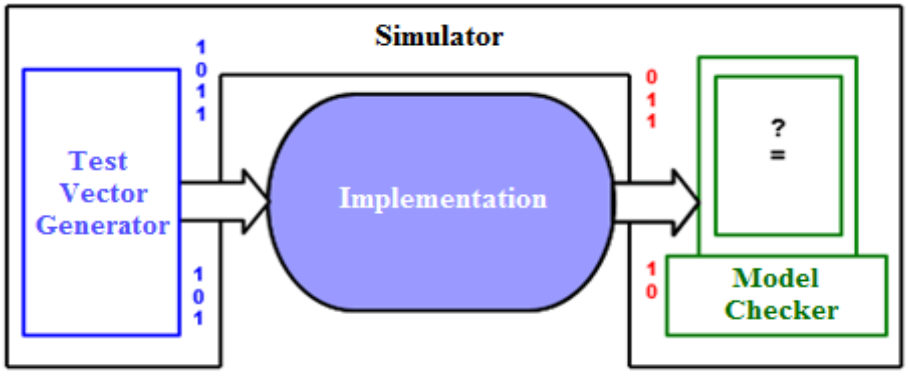

Figure 2-1: Simulation-based Verification [1]

2.1.3 Formal Verification

Formal verification is the method by which we prove or disprove a design implementation against a formal specification or property using mathematical reasoning. Formal verification covers exhaustive state space, which is difficult to achieve by simulation. In addition, it does not require generation of input stimulus since exhaustive stimuli can be generated automatically. There is also no need to generate expected output sequences because the correctness of the design is guaranteed mathematically. [1]

On the other hand, formal verification algorithmically and exhaustively explores all possible input values over time. Hence, it works well for small designs where the number of inputs, outputs, and states is relatively small. It is very susceptible scalability since an increase in any of the variables mentioned previously would grow the state space exponentially. Some methods require that the specification be translated into properties to verify that these properties hold under all inputs and all states. Unfortunately, deciding on which properties to verify is not an easy task and requires a lot of effort and trials. [1]

2.1.4 Semi-formal Verification

Semi-formal verification takes advantage of the two previously mentioned verification methods, simulation-based verification and formal verification. It uses simulation to reach interesting states in the design and then uses formal verification to perform exhaustive analysis around the interesting state to uncover deep bugs in the system. [1]

2.1.5 Assertion-based Verification

Assertion-based verification can be achieved by using static assertions in formal verification or using assertions in dynamic simulations. The former type of assertions formally specifies all functional requirements and behavior of a design and can use procedures to prove that all the specified properties hold true. The latter type of assertions is useful for monitoring assertions in a dynamic simulation and hence improving observability that reduces the time involved in debug. An advantage of assertion-based verification is moving from ambiguous natural language forms to forms that are mathematically precise and verifiable, and these forms lend themselves to automation and improved verification quality. [1]

2.1.6 HW/SW Co-verification

The idea of partitioning the verification into hardware and software is that simulation performance is improved by moving time consuming parts of the design to hardware. As shown in Figure 2-2, one of the modules of an example system design runs slowly on SW based simulations and needs to be accelerated, therefore it is moved to a hardware accelerator, synthesized and compiled into Field Programmable Gate Arrays (FPGAs). In this mode, system designs involves both hardware and software development, since hardware and software components are verified separately but need to work together in the real product. The main advantage of co-verification is to accelerate the completion of projects with higher confidence in the verification process. [1]

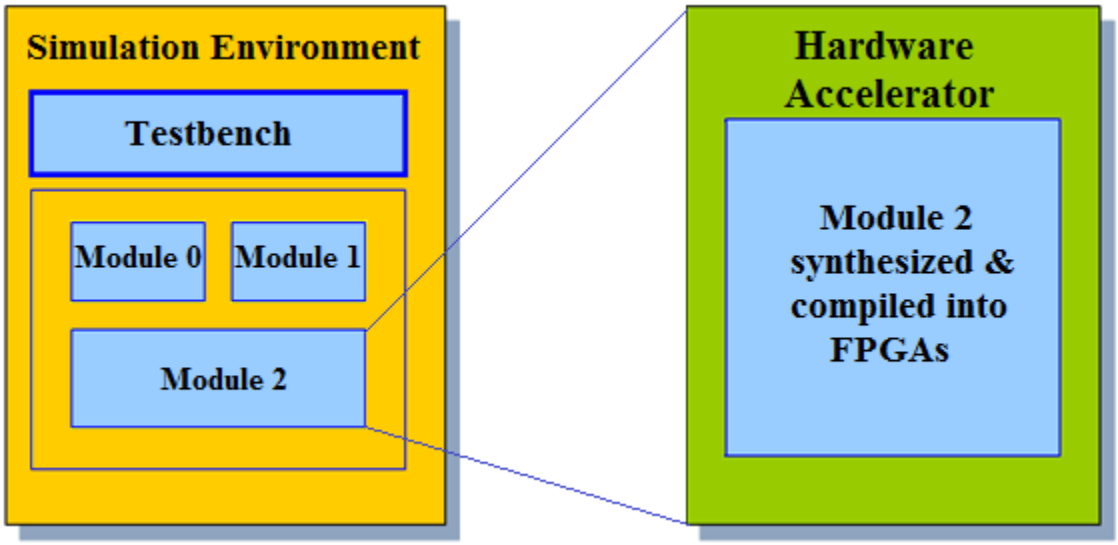

Figure 2-2: HW/SW Co-verification [1]

2.1.7 Coverage-based Verification

Coverage is a metric of the completeness of verification and defines the quality of the verification process. A system with low verification coverage is more susceptible to undetected bugs and design flaws compared to a system with high coverage. A system with high coverage has most of its source code executed during testing, hence it has a lower likelihood of containing bugs. There are two types of coverage: code coverage and functional coverage. [1]

Code coverage is simply a measure of what percentage of code has been executed by a test suite [1]. This includes:

- **Statement coverage:** Has each statement of the source code been executed?
- **Branch coverage:** Has each control structure been evaluated to both true and false including if-conditions, switch cases, for, and while loops?
- **Condition coverage:** Has each Boolean subexpression been evaluated both to true and false?
- **Expression coverage:** Has the right hand side of an assignment statement been exercised with all possible values?

  For example, has an expression like **x <= a xor (not b)** been evaluated with all possible values for *a* and *b*?
- **Toggle coverage:** Has each bit toggled back and forth at least once?

  For example, has each bit of these wires and registers changed its value from 0 to 1 and back from 1 to 0 again during simulation?

  These toggles could be standard transitions such as transitions from a low state to a high state and vice versa, or extended transitions which additionally includes transitions from Z states to either high or low states.
- **Finite State Machine coverage:** It represents the coverage of all states, being visited and traversed through all arcs providing state coverage as well as arc coverage.

On the other hand, functional coverage is about covering the functionality of the Design Under Test (DUT). It is often derived from design specifications and requirements. Typically, functional coverage covers DUT inputs, outputs and internals.

- **DUT Inputs:** Are all input operations injected?
- **DUT Outputs:** Are all responses seen from every output port?
- **DUT Internals:** Are all interested design events being verified including FIFO queues, arbitration mechanisms, etc.

Coverage driven verification is a process that starts with creating initial coverage metrics and specifying a coverage method, followed by generating random tests, and running these tests to collect coverage results. If the coverage goals are met, then the verification process is

complete. Otherwise, verification engineers should identify coverage holes, add tests to target these holes, and possibly enhance stimulus generation and coverage metrics if needed. They should re-run the tests and repeat the process again and check if the coverage goals are met as shown in Figure 2-3 [1].

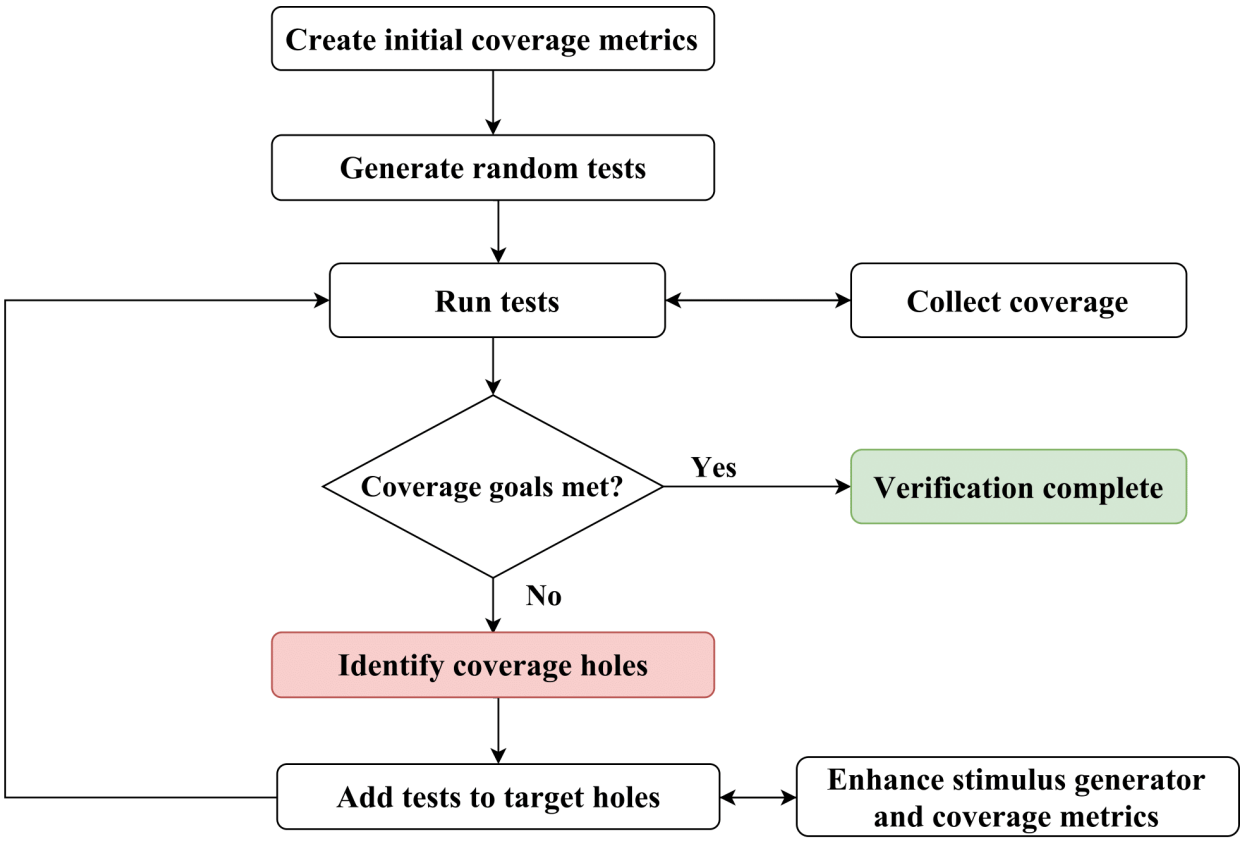

Figure 2-3: Coverage driven verification [1]

According to Rashinkar [1], 70% of design cycles are spent on verifying designs compared to 10% on system design and 20% on actual implementation. Because of its ultimate importance, Mentor Graphics provides the Questa Verification Solution with formal and simulation engines to provide a comprehensive platform to verify complex system on chip (SoC) designs.

2.2 Questa Verification Run Manager Background

2.2.1 Introduction to Questa VRM Features and Use Models

Verification Run Manager (VRM) is a tool developed by Mentor Graphics within Questa Verification Solution. The tool automates various verification tasks allowing for more efficient regressions while maintaining flexibility of deployment in different environments. Figure 2-4 shows a simple block diagram that indicates VRM's interaction with its inputs and outputs.

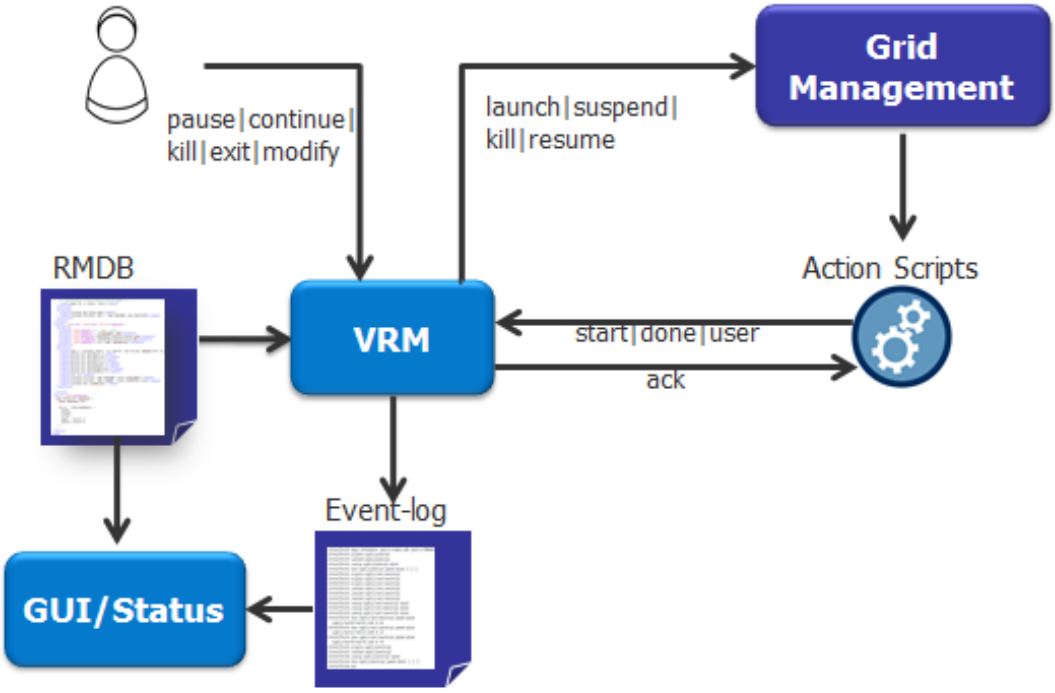

Figure 2-4: VRM Block Diagram

VRM offers many features such as regression configuration, task automation, resource management, tool management, and result analysis. VRM can run in both batch mode and GUI mode.

1. Regression Configuration

VRM regressions can be easily configured through the Run Manager Database (RMDB) file. The RMDB file offers an easy way to control the automation of regressions by allowing the user to specify:

- Tasks to execute.
- Tasks dependencies and regression flow.

- Resources used for execution.
- Parameters such as jobs timeouts, test plan file, merge files, etc.

The RMDB file is easily maintained and highly reusable due to its simplicity and modularity.

2. Task Automation

VRM offers the ability to automate most verification tasks such as:

- Running tests according to the flow specified in the configuration file.
- Importing test plans and using it to guide the overall process.
- Determining passing and failing tests and logging the results to the user.
- Rerunning failed tests.
- Creating/deleting output files and data.
- Managing seeds.
- Merging coverage data.

3. Resource Management

VRM can control different resources during execution. This includes the local machine where VRM was invoked or one or more grid management systems connected to remote machines. VRM manages these resources efficiently to execute its tasks according to the specification in the RMDB file. As shown in Figure 2-4, VRM schedules the scripts to be executed on each resource and controls the execution by launching, suspending, killing, or resuming scripts. Scripts running on each resource send control messages to VRM indicating starting, ending, and other information. VRM logs all the information mentioned in its event log.

4. Tool Management

VRM is designed as a flexible automated solution that can be used in different environments. VRM offers the ability to integrate with any scripting environment used by the customer. This allows users with homegrown systems an easy way to migrate to VRM in order to exploit its many features.

5. Result Analysis

VRM gives the user to ability to utilize Questa Results Analysis functionality. This offers the users the ability to analyze regression results to speed up problem detection and resolution. VRM presents the results of different verification runs while offering functionalities such as grouping, sorting, triaging, and filtering results.

6. Use Models

VRM can be deployed into any verification project, automating its flow and increasing its overall efficiency in the process. Here are some examples where VRM can be used:

- ***Nightly Regressions:***

Automated test suites, possibly consisting of smaller test subgroups, which run on a daily basis. An RMDB file controls the project. The regression runs in batch mode unattended and VRM controls a grid management system.

- ***Automated rerun in debug mode:***

Failed tests from a nightly regression can be automatically rerun in debug mode to gather extra information.

- ***Rerun of failing tests after a source code fix:***

The user can specify a set of failing tests from the GUI and choose to rerun them manually. A custom suite is generated and the tests are run interactively.

- ***Search for high-coverage random seeds:***

A random test can be run a number of times with different seeds (possibly overnight). At completion, coverage data from all tests can be collected and the seeds with better coverage could be selected.

2.2.2 VRM Terminology

In this section, the building blocks and main terminology of VRM are described. The core concepts of how VRM handles input and executes tasks are discussed in brief.

1. Runnables

Runnables are the main building block of VRM execution. Each runnable represents a task to be executed. Non-leaf nodes are known as group runnables and are used to control the flow of the regression. There is not limit on the nesting of runnables.

A Leaf runnable represents a task. It has an execution script known as *execScript*. This is the script that corresponds to the test initiated by this runnable. A leaf runnable can have other types of scripts that execute other verification tasks.

Group runnables can either be sequential or parallel. Each group runnable usually begins with a *preScript* that runs before its children runnables. Children runnables are then launched either in parallel or in sequence according to the mode specified. Each group usually has a *postScript* at the end after all children finish execution.

VRM enables the user to repeat a runnable multiple times in a regression run. This can be achieved by specifying a repeat count or a test list possibly with different seed values.

2. Actions and Commands

Actions represent the actual jobs run by VRM on its local machine or grid controlled machines. There are different types of actions used within VRM. Basic actions used within runnables are *execScript*, *preScript*, and *postScript*. Each of these actions is associated with a script in the RMDB file. Each script consists of zero or more commands. On execution, the commands are copied to a script file before execution. This script can be defined in the OS's shell or in any other scripting language. By default the language is assume to be TCL. There are other types of action scripts such as *mergeScript*, *triageScript*, *tplanScript*, and *trendScript*.

3. Grouping and Inheritance

VRM allows for inheritance between its elements such as runnables using the *base* attribute. Elements and parameters in a parent element can be accessed from an inheriting element. This applies to runnables, methods, and user defined TCL. For runnables, a member runnable in a group runnable can access parameters and elements of the defining group. A set of rules define the order in which parameters and elements are searched within any element and its parents.

2.2.3 **VRM Execution**

As illustrated in the simplified block diagram in Figure 2-4, VRM's execution is controlled by the RMDB file. The RMDB file specifies the configuration of regression tasks. VRM manages the working directory and creates any necessary scripts. VRM then controls the underlying resources to automate the workflow and run the scripts. VRM logs the entire regression events to the event log and outputs data files to the working directory.

- ***Regression Execution***

VRM takes the name of the runnable to be run from the command line. It then expands the execution graph from the information provided in the RMDB file into a list of actions. The sequence of execution of actions depends on the grouping in the RMDB file. Each action is executed and VRM reports the outcome of the run. Actions are expanded to a graph whose nodes represent the actions to be executed. Some examples of these actions are *preScript*, *execScript*, and *postScript*.

Task runnables appear as leaves in the graph and have nodes representing scripts running under them such as *execScript*, and *mergeScript*. For groups, a *preScript* and a *postScript* are added to the graph. A *preScript* runs before all children scripts and a *postScript* runs after all children scripts finish execution.

2.3 Process Mining

Process mining is one of the fastest growing research fields. Although process mining is categorized under data mining due to its handling of large input data, its true power surpasses that of many other data mining techniques. The data mining field is much wider and more generic. It is mostly concerned with collecting, analyzing, and interpreting data from a variety of resources and transforming data into values. Process mining is largely process orientated. It can help model any system to provide better understanding of the system's design and workflow; better supervision and monitoring of resource usage and efficiency; and better analysis and performance checking.

Applying process mining techniques may improve the overall performance, but in order to do so, there must exist a model and an event log. A model is a representation of a system as a set of processes on which one can perform the process mining techniques. An event log is the actual flow of data on these processes. They are traces of the model, or how the process flow was executed. Here appear three main concepts, play-out, play-in, and replay [2].

- **Play-out** is extracting some possible event log traces from the model. This, of course, requires a model representing the system to be ready. Tracing the model with different paths and combinations may help in having better insights about the possible paths the process flow may follow.
- **Play-in** is generating a model from the already known event log traces. This is a step which can be done before analysis in order to discover or create the model.
- **Replay** is tracing the event log traces on the model representing the system. This helps much in conformance checking and making sure that the model fully represents the system. It also helps in performance analysis, identifies the bottlenecks in the system and increases the efficiency of resource allocation in the system.

In order to create a model, one must first choose a suitable representation for the system. There are various ways to represent a system such as **petri nets**, **causal nets**, **business process model and notation**, ***BPMN***, **dependency graphs**, etc. Then, after choosing the proper representation technique, the process mining steps may be applied in order to analyze the model more efficiently [2].

Process mining mainly consists of three steps, ***process modeling***, sometimes referred to as process discovery, ***conformance checking*** and ***performance analysis***.

2.3.1 Process representation

There are various ways to represent a system. Those various representations emerged due to the different functionalities, audience and fields in which process mining are used. Here we discuss one of the most widely used representations, petri nets.

1. Petri nets:

Petri nets are a graphical and mathematical modeling tool applicable to many systems. It helps graphically notate the model and visually analyze it. Petri nets can be used by both practitioners and theoreticians; this makes it a very powerful tool used in both industry and research [3].

- ***Petri nets notations, and transition firing***

A petri net is a particular type of directed graphs which mainly consists of two types of nodes, called **places** and **transitions**. The initial place is called the **initial marking**. In graphical representation, places are drawn as circles and transitions are drawn as rectangles. Arcs are assigned between a places and a transition, or vice versa, which indicates the direction of the flow [3].

In modeling, places represent states and transitions represent events. A transition has a number of input and output places representing the preconditions and post conditions respectively. The state of the system is identified by the tokens' marking in places. For a token to change its place, i.e. for the system to change its state, a transition must be fired. There are certain rules that are associated with the applicability of transition firing [3]:

1) A transition is said to be enabled if each input place is marked with, at least, the number of tokens needed by the weighted arc to fire.
2) An enabled transition may or may not be fired depending on whether the transition, representing the event, actually happened or not.
3) A firing of the enabled transition consumes the input places' tokens used and produces new tokens, depending on the weights of the arcs, in the output places.

These rules are illustrated in Figure 2-5.

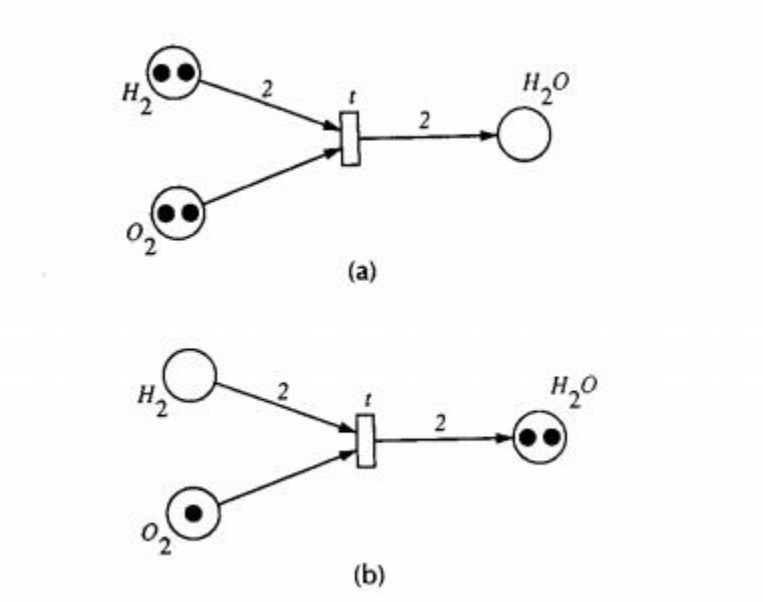

Figure 2-5: Example of transition firing. (a) Model before firing (b) Model after firing [3]

We can see that for transition *t* to fire, two tokens much be present in the H_2 place and one token must be present in the O_2 place. This causes these three tokens to be consumed in order to produce two tokens in the H_2O place [3].

2.3.2 **Process Modeling:**

The second step, after specifying the representation method, is to generate a model that represents the system. A model is a simplified mapping of reality to serve a specific purpose. This mapping helps identify the system's states, events, and how these states and event are related. This can be achieved by either ***process discovery***, or ***direct model generation***.

1. ***Process Discovery:***

Process discovery is one way to generate a descriptive model for the system. Since an event log describes how the model might behave, it can be used in order to generate the model.

- ***Algorithms for process discovery:***

There are several algorithms used for generating the model of a system from its event log. These algorithms allow the discovery process to be systematic. Different algorithms are

implemented to perform process discovery such as; ***Alpha algorithm***, and ***inductive mining algorithm*** [2].

2. ***Direct model generation:***

This method depends on model generation by analyzing the system itself. It focuses on determining the system's states and possible events, then, constructing relations between them in order to reach the correct process flow.

- ***Algorithm used:***

A very simple algorithm is used in order to generate the system's model:

- Determine the system's states. These states are modelled as places in the petri net. They indicate when the system is running, paused, or stalled waiting for an action or an event to occur in order to proceed to another state.
- Determine the system's events. These events are modelled as transitions in the petri net. They help the system proceed from one state to another.
- Determine the relations between the set of places and the set of events. These relations add logic to the model. They show what new state the system enters after an event happens, the dependencies between events, and whether the conditions for an event firing are satisfied or not.
- Connect between each place and the set of transitions that are allowed to fire given the system's state.

Naturally, the algorithm becomes very computationally expensive with systems that are more complex. A method to mitigate this is to decompose your system into smaller blocks or systems that are independent of each other; model these blocks independently; and connect between these blocks using the appropriate events. For a better understanding of the algorithm here is an example on a simple communication protocol.

- ***Simple communication protocol example:***

Suppose we need to implement a communication protocol between a sender and a receiver. The sender can send one packet at a time to the receiver. It must wait for the receiver to receive this packet, and sends a new packet only after receiving an acknowledgement that the

receiver has received it correctly. No more than one packet can be sent at once and no more than one packet can be received at once.

To be able to model this protocol we have to specify the states of the system:

- The sender is ready to send.
- The receiver is ready to receive.
- The sender is waiting for an ACK from the receiver.
- The receiver receives the message.
- The sender receives the ACK.
- The receiver is waiting for the sender to receive the ACK.

Now we need to specify the events that may occur in the system:

- The sender sends a message.
- The receiver receives a message.
- The receiver sends an acknowledgment.
- The sender receives an acknowledgment.

Now, after specifying the system's states and events, the only task remaining is to specify the connections between them. These relations are modeled as connections between transitions and places. We can see the complete model of the system in Figure 2-6.

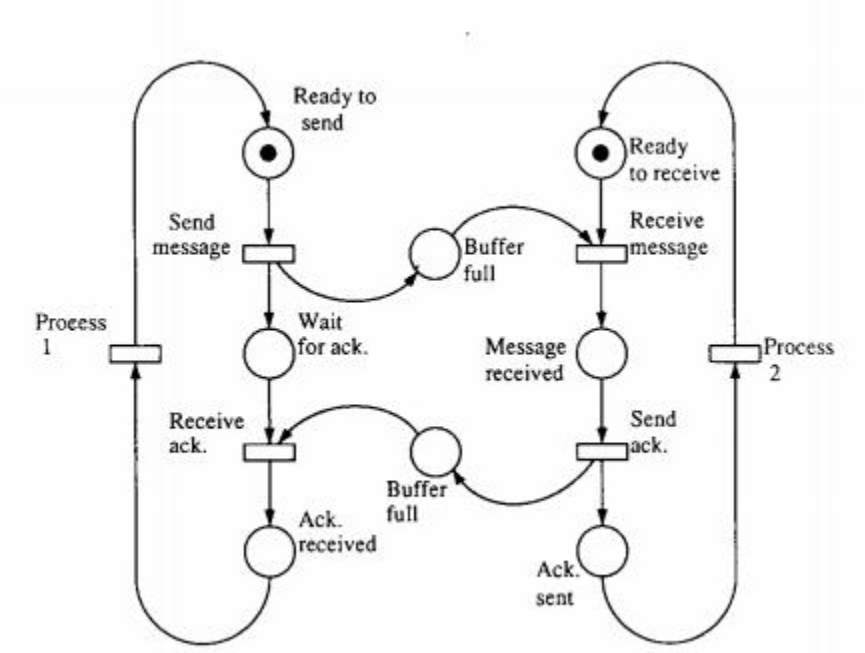

Figure 2-6: A simplified model of a communication protocol [3]

2.3.3 Performance Analysis:

It mainly depends on analyzing the performance of your system, using a generated model. Several performance key indicators help evaluate a system's efficiency. It uses both the process model and the event log to replay the data on the model to calculate performance metrics.

3. Performance analysis goals:

There are several goals for analyzing the performance of a system using its model:

- ***Identify bottlenecks in the system:***
  This can help in understanding why the system takes more time than expected when performing tasks.

- ***Identify resource utilization in the system.***
  This allows us to explore when the system resources are fully utilized. It also shows how the system reacts with a large task load.

- ***Identify the intervals where the system waits in each state and the time transitions take to fire.***

  This data can help us recognize where the system spends most of its time.

4. Performance Analysis metrics:

There are several metrics that can be used in analyzing performance. These metrics can be place metrics, one-transition metrics, two-transition metrics, and activity metrics [4].

- ***Place metrics:***

The place-related metrics are as follows:

- *Frequency*: the number of visits of tokens to a place.
- *Arrival rate*: the rate at which tokens arrive to a place per unit time.
- *Wait time*: the time that passes from the full enabling of the following transition to its actual firing.
- *Synchronization time*: the time that passes from the partial enabling of a transition until full enabling.

- *Sojourn time*: the total time a token spends in a place during a visit.

- ***One-transition metrics:***

The one-transition-related metrics are as follows:

- *Idle time*: the time that passes from the complete enabling of a transition and its actual firing.

- ***Two-transition metrics:***

The two-transition-related metrics are as follows:

- *Frequency*: the number of process instances in which both transitions fire at least once.
- *Time in between*: the time between the first firing of the first transition and the first firing of the second transition.
- *Average time in between*: the average of the time in between, if both transitions fired consecutively several times.

Chapter 3 :

Market Survey

3.1 Real World Process Mining Applications

3.1.1 Customer Order Handling (SAP ERP)

A case study was made on applying Process Mining techniques using the ProM framework to a practical case. The actual business model was extracted from a SAP ERP customer order handling system by cleaning, transforming, and feeding business transactional data into Process Mining algorithms as shown in Figure 3-1. A method for removing bad and inconsistent data was applied. Various Process Mining algorithms were applied and compared. The final model was output to reporting software for use by the end user. The conclusion of the endeavor was that process discovery is indeed feasible and that system analysis through Process Mining is indeed possible with a grain of salt – the application must be specific to the domain under question to receive maximal gains.

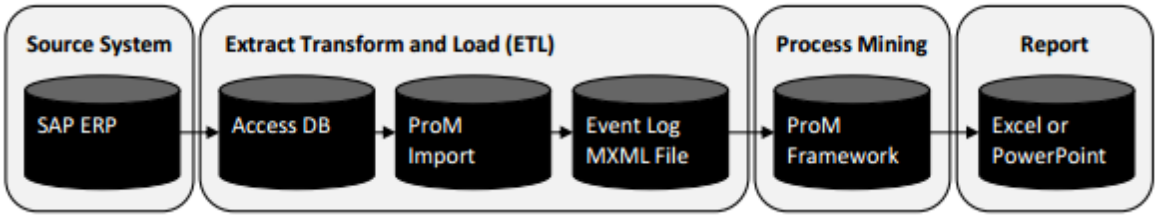

Figure 3-1: Flow form ERP to ProM

3.2 Commercial Process Mining Software

3.2.1 Celonis Pi

Celonis Pi is an off-the-shelf Process Mining tool for IT professionals. It uses AI and Machine Learning powered Process Mining to discover business processes and display them to the user. It can connect with various business process software to extract data and log files. It is advertised as being able to discover a process model, detect vulnerabilities, and analyze bottlenecks. It does not however provide any means by which to check conformance of the current system. It is mostly advertised as a monitoring tool.

3.2.2 Fluxicon Disco

Fluxicon Disco is a general-purpose commercial Process Mining tool advertised as being able to bring the benefits of Process Mining to any domain. However, it is largely focused on

aiding IT professionals in businesses to better understand their systems. It can take in logs in any format, discover a process model, and then provide analysis on conformance as shown in Figure 3-2 and performance.

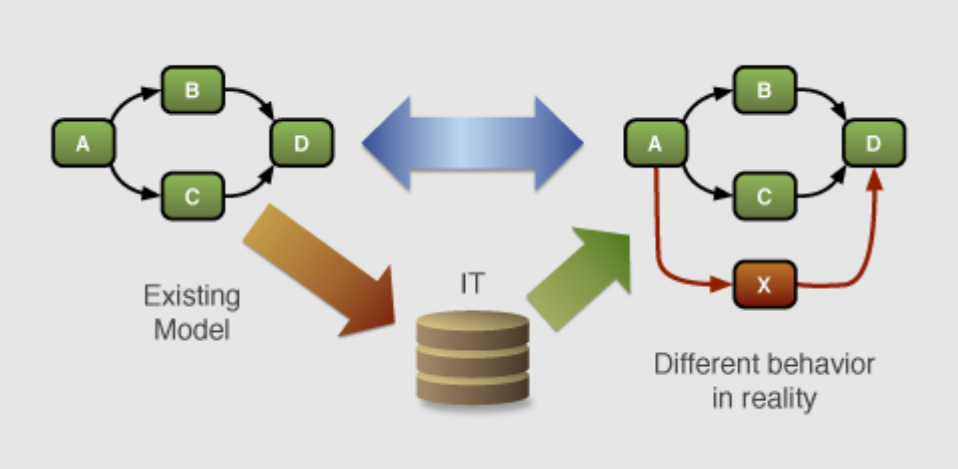

Figure 3-2: Conformance in Disco

Chapter 4 :

VRM Performance Analyzer Design

4.1 Overview

VRM Performance Analyzer's main goal is to mine bottlenecks and find performance issues within VRM itself. This goal is achieved by parsing, modeling, replaying, grouping, analyzing, and visualizing a VRM regression process.

4.2 Block Diagram

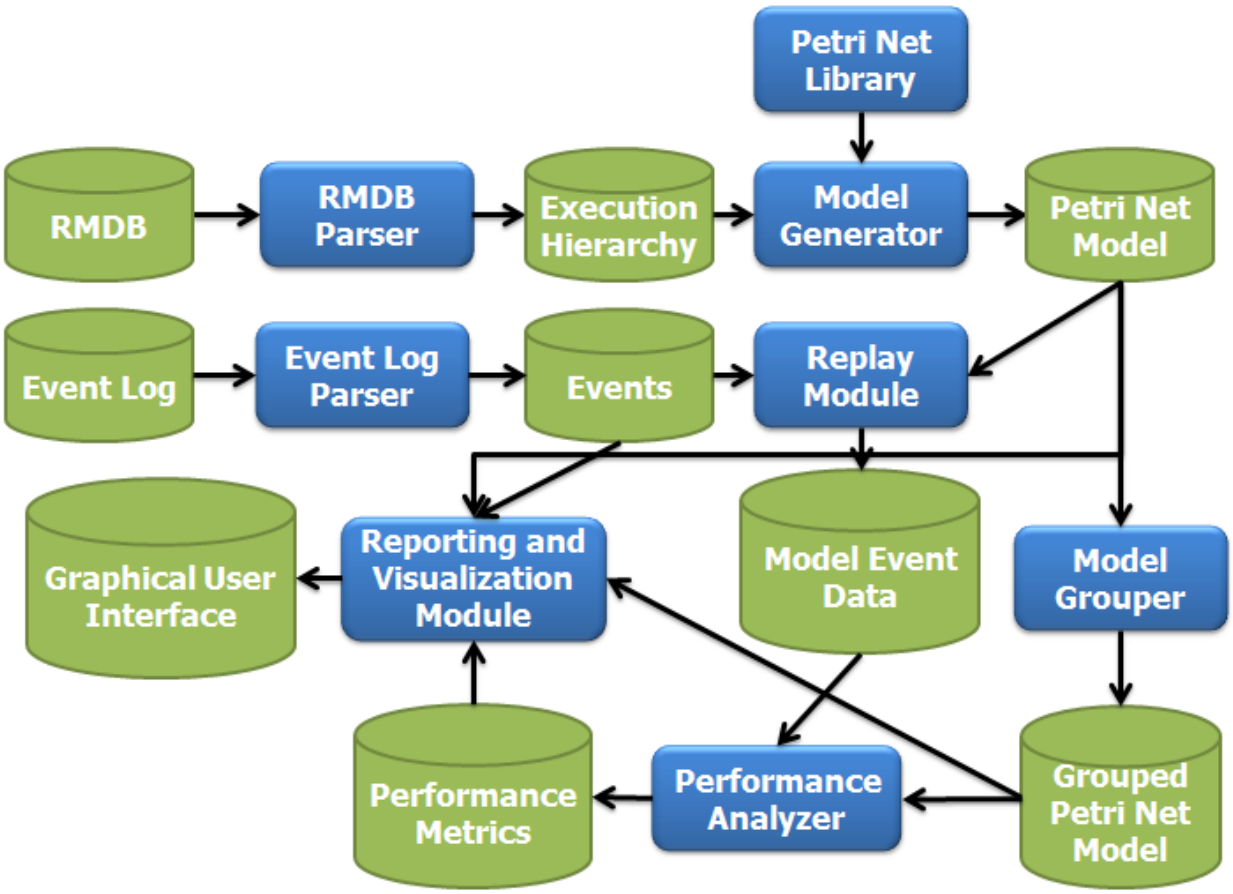

Figure 4-1: VRM Performance Analyzer Block Diagram

Figure 4-1 illustrates the six main modules of VRM Performance Analyzer along with input and output of each module. First, we parse the configuration file and the event log and generate the necessary data structures. Second, we model the execution of a regression using the generated data structures using a petri net representation. Third, we replay the regression's event log on top of the generated petri net model. Fourth, we group the petri net into an abstracted

grouped petri net. Fifth, performance analysis techniques are applied to the results of the replay process. Finally, the model, the replay process, and the performance analysis results are presented to the user through the UI.

4.3 Parser Module

The parser module is responsible for taking raw files from a given VRM regression and generating the data structures necessary for analysis. It has two submodules; **RMDB parser**, and **event log parser**. Figure 4-2 illustrates the block diagram of the parser module.

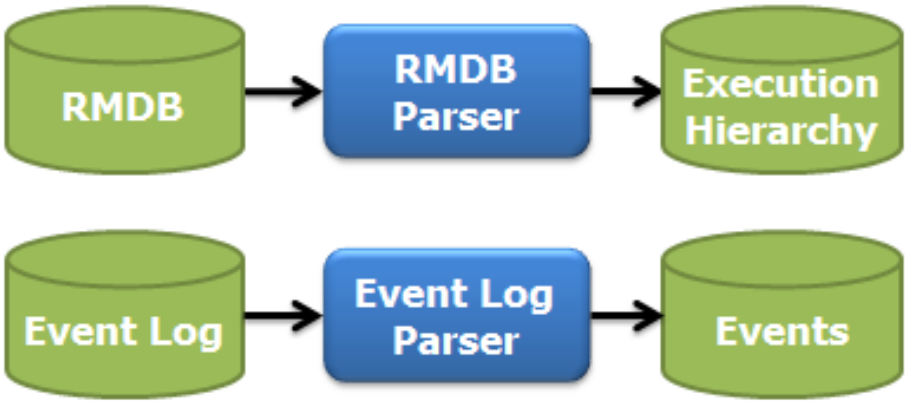

Figure 4-2: Parser Block Diagram

4.3.1 Inputs

- ***RMDB***

RMDB is an xml based configuration file for a given regression. It specifies the execution hierarchy required along with specific options for the run.

Figure 4-3 shows an example of an RMDB configuration file. This example shows a parent runnable named **Nightly**. Nightly has a *postScript* that runs after children runnables are executed. Nighlty has two child runnables, **Directed** and **Random**. Both Directed and Random are also group runnables each having three child runnables. Random has a *preScript* that runs before its children.

```
<rmdb version="1.1">
    <runnable name="nightly" type="group">
        <parameters>
            <parameter name="mergefile">merge.ucdb</parameter>
        </parameters>
        <members>
            <member>directed</member>
            <member>random</member>
        </members>
        <postScript launch="vsim">
            <command>vcover attr -name ORIGFILENAME -name TESTSTATUS (%mergefile%)</command>
        </postScript>
    </runnable>
    <!-- ============== -->
    <!-- DIRECTED TESTS -->
    <!-- ============== -->
    <runnable name="directed" type="group">
        <parameters>
            <parameter name="ucdbfile">../(%INSTANCE%).ucdb</parameter>
        </parameters>
        <members>
            <member>dirtest1</member>
            <member>dirtest2</member>
            <member>dirtest3</member>
        </members>
        <execScript>
            <command>vlib work</command>
            <command>vlog -sv +define+dirtest=\"(%INSTANCE%)\"(%RMDBDIR%)/src/top.sv</command>
            <command>vsim -c top</command>
            <command>run -all</command>
            <command>coverage attribute -name TESTNAME -value (%INSTANCE%)</command>
            <command>coverage save (%ucdbfile%)</command>
        </execScript>
    </runnable>
    <runnable name="dirtest1" type="task" />
    <runnable name="dirtest2" type="task" />
    <runnable name="dirtest3" type="task" />
    <runnable name="random" type="group">
        <parameters>
            <parameter name="ucdbfile">../test(%seed%).ucdb</parameter>
        </parameters>
        <members>
            <member>randtest1</member>
            <member>randtest2</member>
            <member>randtest3</member>
        </members>
        <preScript>
            <command>file delete -force work</command>
            <command>vlib work</command>
            <command>vlog -sv (%RMDBDIR%)/src/top.sv</command>
        </preScript>
        <execScript>
            <command>vsim -lib ../work -sv_seed (%seed%) +SEED=(%seed%) -c top</command>
            <command>run -all</command>
            <command>coverage attribute -name TESTNAME-value randtest(%seed%)</command>
            <command>coverage save (%ucdbfile%)</command>
        </execScript>
    </runnable>
    <runnable name="randtest1" type="task">
        <parameters>
            <parameter name="seed">123</parameter>
        </parameters>
    </runnable>
    <runnable name="randtest2" type="task">
        <parameters>
            <parameter name="seed">456</parameter>
        </parameters>
    </runnable>
    <runnable name="randtest3" type="task">
        <parameters>
            <parameter name="seed">789</parameter>
        </parameters>
    </runnable>
</rmdb>
```

Figure 4-3: RMDB File Example

The RMDB file is a very flexible way to configure a regression with the capability of generating a wide range of execution heirarchies with further custimization as a result of different options. It is curcial to extract the intended execution hierarchy from the given RMBD to be used later within the tool.

- ***Event Log***

The event log is a raw text file outputted by VRM where each line maps to an event. Each event has a timestamp indicating when it happened. There are various types of events that can occur. Table 4-1 shows some of the events that can occur in the event log along with their description.

Table 4-1: Events Types

Event	Description
begin	Indicates that the regression execution has begun.
eligible	Indicates that an action script is eligible to run
launched	Indicates that an action script was launched on the chosen machine or grid management system.
running	Indicates that an action script started running on the chosen machine or grid management system.
done	Indicates that an action script finished running along with the reason.
merge	Corresponds to merging of one or multiple UCDB files into a common UCDB file
triage	Corresponds to triaging of a test result into a common triage file.
trend	Corresponds to performing trend analysis on a merge file.
rerun	Indicates that an action script was rerun.
end	Indicates the end of a regression.

Figure 4-4 shows the event log corresponding to the RMDB example file in Figure 4-3. Each line corresponds to an event. Action specific events consist of a timestamp in host computer's local time format, followed by the event type, the name of the action script being run, and event specific data.

```
20170512T210326 eligible nightly/preScript {}
20170512T210326 done nightly/preScript empty
20170512T210326 eligible nightly/directed/preScript {}
20170512T210326 eligible nightly/random/preScript {}
20170512T210326 done nightly/directed/preScript empty
20170512T210326 eligible nightly/directed/dirtest1/execScript {}
20170512T210326 eligible nightly/directed/dirtest2/execScript {}
20170512T210326 eligible nightly/directed/dirtest3/execScript {}
20170512T210326 launched nightly/random/preScript
20170512T210326 launched nightly/directed/dirtest1/execScript
20170512T210327 launched nightly/directed/dirtest2/execScript
20170512T210327 launched nightly/directed/dirtest3/execScript
20170512T210327 queues {S:running, Q/R:(<none>:0/4), L:0, D:0, K:0}
20170512T210337 running nightly/random/preScript mohammed-pc
20170512T210338 running nightly/directed/dirtest1/execScript mohammed-pc
20170512T210338 running nightly/directed/dirtest2/execScript mohammed-pc
20170512T210338 running nightly/directed/dirtest3/execScript mohammed-pc
20170512T210340 done nightly/random/preScript passed mohammed-pc {} {} {} {} {}
20170512T210340 eligible nightly/random/randtest1/execScript {}
20170512T210340 eligible nightly/random/randtest2/execScript {}
20170512T210340 eligible nightly/random/randtest3/execScript {}
20170512T210341 launched nightly/random/randtest1/execScript
20170512T210342 launched nightly/random/randtest2/execScript
20170512T210342 launched nightly/random/randtest3/execScript
20170512T210342 queues {S:running, Q/R:(<none>:0/6), L:0, D:0, K:0}
20170512T210352 running nightly/random/randtest1/execScript mohammed-pc
20170512T210352 running nightly/random/randtest3/execScript mohammed-pc
20170512T210352 running nightly/random/randtest2/execScript mohammed-pc
```

Figure 4-4: Event Log Example

The event log file is important to the performance analysis process as it shows how VRM has run a given regression. It is important to parse this file in order to perform replay and performance analysis in later modules.

4.3.2 Outputs

- ***Execution Hierarchy***

The execution hierarchy is a data structure in memory that is used later by the modeler in order to generate the regression model. It is a directed acyclic graph that corresponds to the intended execution of the runnables along with other data and options found in the RMDB.

4.3.3 Module Functionality

Parser module has two submodules; **RMDB Parser**, and **Event Log Parser**.

1. RMDB Parser

As shown in Figure 4-2 the input to the RMDB Parser is an RMDB file. The parser uses the RMDB API shipped with VRM to read and parse the RMDB file. It outputs the execution hierarchy as a directed acyclic graph whose nodes represent runnables. The parser uses the RMDB API to extract all the information about all runnables and relations between them. Data associated with each runnable in the output graph consists of parameters, methods, attributes, and action scripts. The parser resolves base and group inheritance for each runnable. After the RMDB parser finishes execution, the execution hierarchy graph is generated and loaded in memory for use by other modules.

2. Event Log Parser

As shown in Figure 4-2 the input to the Event Log Parser is an event log file. The parser reads the event log line by line and transfers it into a list of records. This corresponds to the events generated by a given regression execution along with all extra information provided by the log. Examples of the extra information provided are the timestamps, the name of the machine running a given script, and command line options used to execute VRM. After the Event Log parser finishes execution the events list is generated and loaded in memory.

4.4 Modeler Module

The modeler module is responsible for generating the petri net corresponding to a given regression. It has two submodules; petri net library, and model generator. Figure 4-5 illustrates the block diagram of the modeler module.

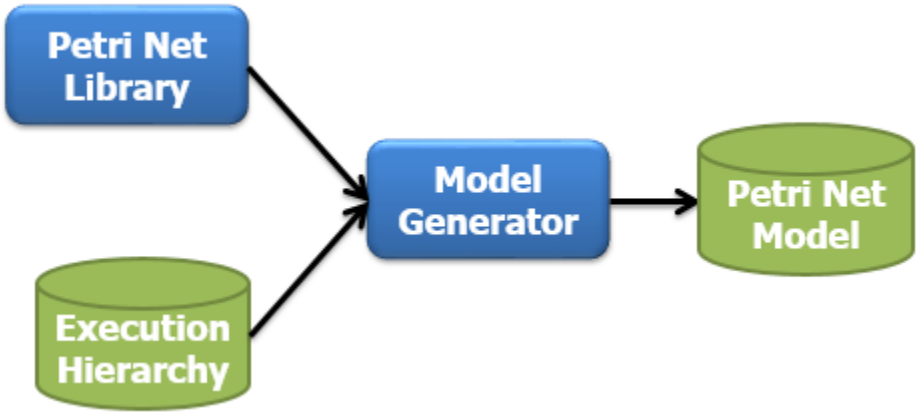

Figure 4-5: Modeler Block Diagram

4.4.1 Inputs

- ***Execution Hierarchy***

The modeler traverses the execution hierarchy generated by the parser module in order to create the corresponding petri net.

4.4.2 Outputs

- ***Petri Net Model***

We used petri net as our main model representation. Each transition in the petri net maps to a given expected event in the event log. For example there are eligible, launch, and running transitions.

Each transition and place has information related to the context they model, for example the parent runnable and action script. This information is used by the Performance Analyzer.

4.4.3 Module Functionality

Modeler module has two submodules; **Petri Net Library**, and **Model Generator**.

1. Petri Net Library

In order to perform modeling, we built a customized petri net library to use as the basis for modeling. The library aims at being flexible and capable of representing any given petri net scenario. It provides components and methods that facilitate generating a model, executing transitions and generating statistics. Components and their descriptions are listed below.

Transitions and places are designed to have extra information related to the context they are created in such as the runnable name and the script name.

The main components of the library are:

- **Place:**

  It is the first component offered by the library. It represents one of the two main building blocks of a petri net. A place is a node that can hold tokens. Distribution of tokens across places indicates the current state of the petri net. On adding and consuming tokens the library uses the timestamp of the executing transition to add performance statistics related to the process.

- **Transition:**

  It is the second main building block of a petri net. A transition is mapped to events that can execute. Transition execution is also called firing. A transition has a group of input places and a group of output places. Each of these connections has a restriction on the number of tokens that should be consumed or added. When a transition fires, it consumes tokens from all input places according to the cardinality of the arc between each input place and the given transition. It then outputs tokens to all output places depending on the cardinality of the arc connecting between each output place and the given transition. Place-transition interaction and transition firing are discussed in more detail in section 2.3.1.

  The library offers two types of transitions; instantaneous transitions and lambda transitions. Instantaneous transitions are transitions which execute in response to an event occurring in the event log. For example, an *Eligible* event in VRM's event log is mapped to an instantaneous transition. Whenever an *Eligible* event is encountered in the log, the corresponding transition in the petri net is fired instantaneously. Lambda transitions are a special type of instantaneous transitions. They are silent in the event log; they have no trace in the event log or in performance statistics. They are used for modularity and are abstracted away from the user. They are executed only when a dependent transition needs to be executed.

Each transition has an execute method that is used to fire the transition when enabled. A given transition recognizes when it is enabled and takes as an input the execution time to be used later in performance analysis.

- **Subnet:**

  It is a logical grouping of places and transitions. It is used for modularity in building the entire petri net. Each subnet has a parent. Initial and final transitions are used to connect this subnet to the rest of the model. It is used to facilitate abstracting away the complexity of underlying runnables and scripts. For example, each script will be created as a separate subnet abstracted away from the rest of the petri net. Subnets also contain children subnets. Subnet usage and the hierarchy of children subnets will be explained later in model generation.

- **Petri Net:**

  It is the entry point of the library. It holds all places, transitions, and subnets created. It has an initial transition called the emitter which indicates the start of execution and ends with a final transition called the collector which indicates the end of execution. While traversing a petri net, one must start at the emitter and end at the collector.

2. Model Generator

The model generator is responsible generating a petri net model using the execution hierarchy generated previously. It utilizes the petri net library illustrated above. As mentioned before, the execution hierarchy is a directed acyclic graph whose nodes represent a runnable. The model generator traverses the graph with a depth-first approach and creates a subnet for each runnable. The root runnable is created under the initial petri net with the emitter and collector as the initial and final transitions. The model generator iteratively processes the execution hierarchy graph until all runnables have been processed. The final petri net is then output that represents the complete regression execution and can be used for further analysis. We will now discuss the subnets created by the model generator in more detail.

- ***Action Script Subnet:***

Action script subnet is the base subnet that all action scripts inherit from. PreScript, postScript and execScript subnets completely follow the same structure provided by action script

subnet. MergeScript and triageScript subnets each has a branch with the same structure and adds further functionalities. A summary of all places and transitions included in the subnet follows. The listing includes all variations depending on whether the scripts are sequential or not, executed, empty, or rerun. Figure 4-6 illustrates an example for an action script subnet.

Places included in the action script subnet:

- **Started**: This is the initial place of the action script.
- **Ready**: This is the waiting place after the eligible event and before launched event.
- **Launched**: This is the waiting place after the launched event and before the running event.
- **Running**: This place indicates that the action script is running.
- **Finished**: This is the final place of the action script after it has finished execution

Transitions included in the Action script subnet:

- **Eligible**: An instantaneous transition that represents the *Eligible* event. An *Eligible* transition fires if there is a token in the *Started* place. When the *Eligible* transition fires, a token is produced in the *Ready* place.
- **Launch**: An instantaneous transition that represents the *Launched* event. A *Launch* transition fires if there is a token in the *Ready* place. When the *Launch* transition fires, a token is produced in the *Launched* place.
- **Run**: An instantaneous transition that represents the *Running* event. A *Run* transition fires if there is a token in the *Launched* place. When the *Run* transition fires, a token is produced in the *Running* place.
- **Done**: An instantaneous transition that represents the *Done* event. A *Done* transition fires if there is a token in the *Running* place. When the *Done* transition fires, a token is produced in the *Finished* place.
- **Rerun**: An instantaneous transition that represents *Rerun* event in case of failed scripts. A *Rerun* transition fires if there is a token in the *Running* place. When the *Rerun* transition fires, a token is produced in the *Launched* place.

- **Done Empty**: An instantaneous transition that represents the *Done* event in case of empty scripts. A *Done Empty* transition fires if there is a token in the *Launched* place. When the *Done Empty* transition fires, a token is produced in the *Finished* place.
- **Lambda**: A *Lambda* transition is placed at the end of sequential scripts and is set as its final transition. Such a transition is used to link sequential scripts together and ensure coherence of the petri net.

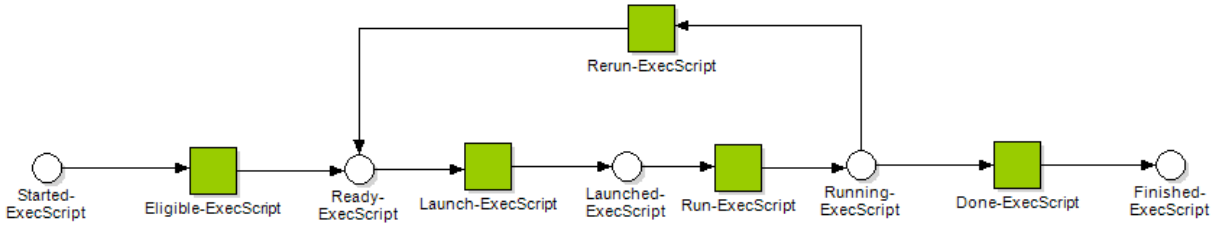

Figure 4-6: Action Script Subnet Example

This subnet applies to *execScript*, *preScript*, and *postScript*.

- **ExecScript Subnet**: *ExecScript* subnet is considered sequential in case it is followed by another script in the task or when the parent task runnable is in a sequential group.
- **PreScript Subnet**: *Prescript* subnet is always a sequential action script subnet at the beginning of all group scripts. When it is empty, launch and running are replaced by a *done empty* transition and the launched place is omitted.
- **PostScript Subnet**: *PostScript* subnet is considered sequential in case the parent runnable is in a sequential group. When it is empty, launch and running are replaced by a *done empty* transition and the launched place is omitted.

- ***MergeScript and TriageScript Subnet:***

Each of the two subnets has a branch that follows the same structure mentioned in action script subnet. However, each has two extra parallel branches and one extra enable place.

To map the dependency of different scripts of the same file on each other, one common place called enable is created per merge file or triage file. This place initially has one token. When a script linked to a merge file or a triage file starts executing it consumes this token disabling other scripts to run. When the running script finishes execution it adds a token back to

the enable place enabling other merge scripts or triage scripts to run. Figure 4-7 and Figure 4-8 illustrate two examples for mergeScript and triageScript subnets.

There are two transitions from started to finished place; merged for merge scripts or triaged for triage scripts, and did not run.

- **Merged**: An instantaneous transition that represents the *Merge* event. A *Merged* transition fires if there is a token in the *Started* place. When the *Merged* transition fires, a token is produced in the *Finished* place. It indicates that the ucdb file of the execScript task was merged.
- **Triaged**: An instantaneous transition that represents the *Triage* event. A *Triaged* transition fires if there is a token in the *Started* place. When the *Triaged* transition fires, a token is produced in the *Finished* place It indicates that the results of the execScript task were triaged.
- **Did not run**: A lambda transition between *Started* place and *Finished* place. In case of mergeScript subnet it indicates that the UCDB file was not merged because the execScript failed or any other condition the user specifies. In case of triageScript subnet it indicates that no triaging occurred for this task.

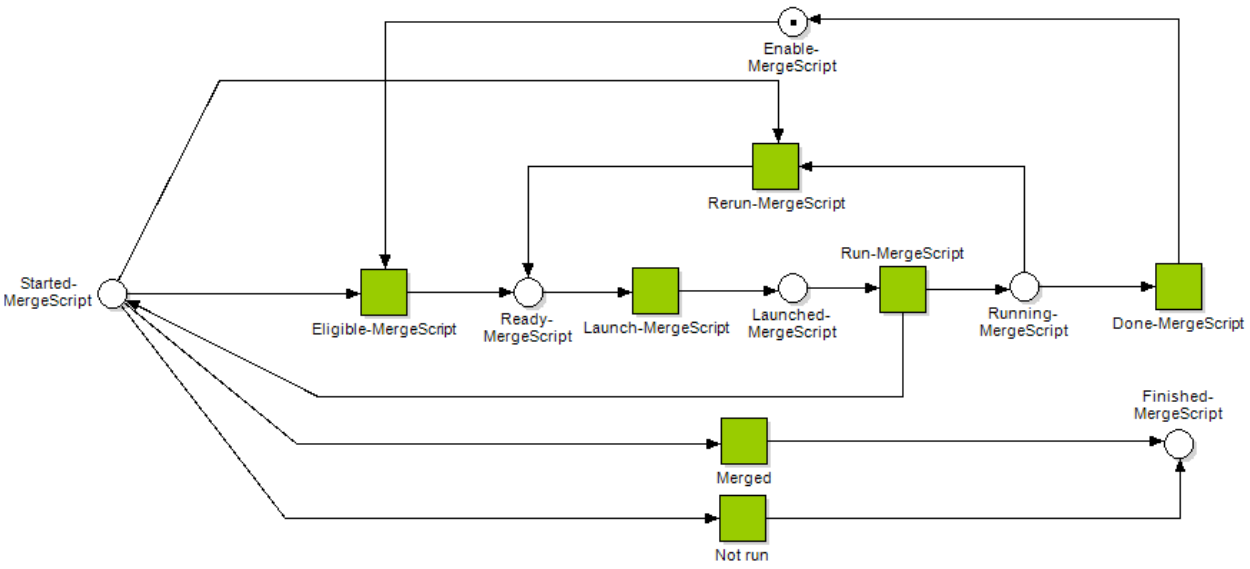

Figure 4-7: MergeScript Subnet Example

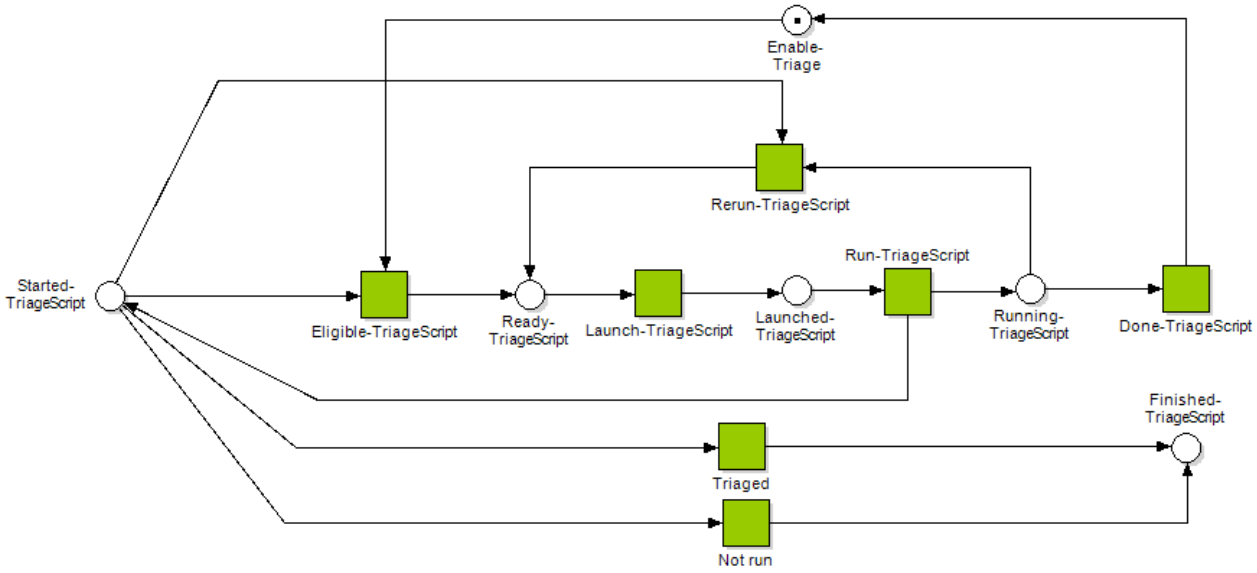

Figure 4-8: TriageScript Subnet Example

- ***Group Runnable Subnet***

A group runnable subnet is a parent subnet that holds children subnets of the following types:

- Group runnable subnet.
- Task runnable subnet.
- Prescript subnet.
- Post script subnet.

It starts with the preScript subnet, and then lists the children runnable subnets – either task or group – in parallel or sequential order according to the group configuration. Finally, it creates the postScript subnet as the final subnet.

In parallel groups, the initial transition of each child runnable is the final lambda transition returned by the prescript subnet. The final transition of each child is created inside the group as a lambda transition called a *branch collector*. Figure 4-9 illustrates a parallel group example.

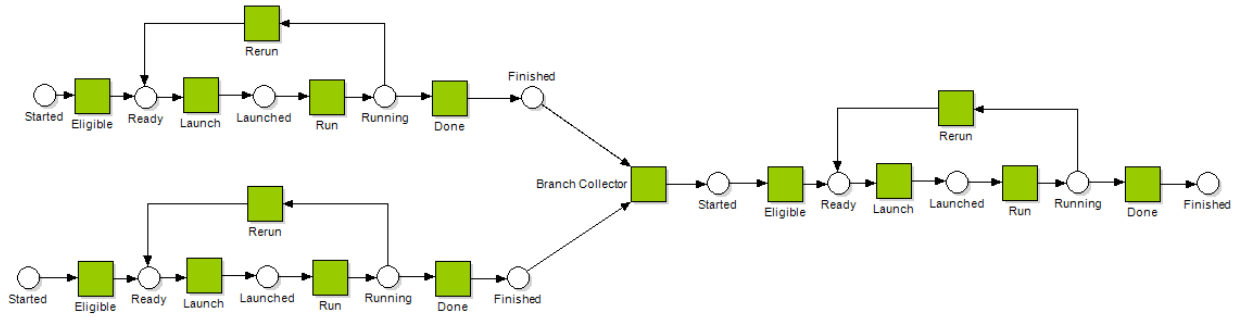

Figure 4-9: Parallel Group Example

In sequential groups, the initial transition of each child runnable is the final transition returned by the previous child. The first child's runnable initial transition is the final lambda transition returned by the prescript subnet. The second child's runnable initial transition is the final lambda transition returned by the previous subnet and so on. The final transition of each child runnable subnet is its own final lambda transition. Figure 4-10 illustrates a sequential group example.

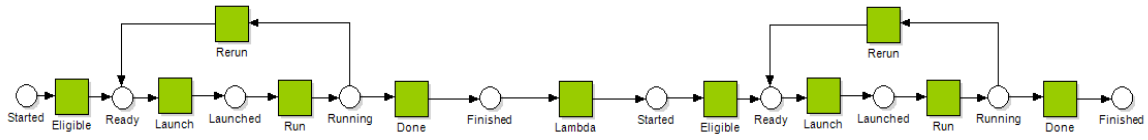

Figure 4-10: Sequential Group Example

- ***Task Runnable Subnet***

A task runnable subnet groups together action scripts executing under the runnable. Subnets under a task runnable subnet are of the following types:

- ExecScript subnet.
- MergeScript subnet.
- TriageScript subnet.

The task runnable always starts with an ExecScript. If merge file and/or triage file parameters are available, a mergeScript and/or triageScript subnets are created. If both subnets are available they are created in parallel. The final places are connected to the final transition, or if the parent runnable is sequential, a common lambda transition is set as final transition.

4.5 Replay Module

The replay module is responsible for replaying events on the generated petri net model. Figure 4-11 illustrates the block diagram of the replay module.

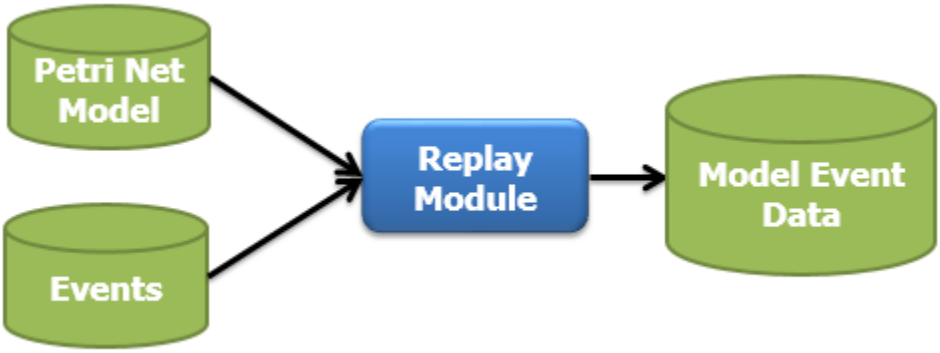

Figure 4-11: Replay Module Block Diagram

4.5.1 Inputs

- ***Events***

The replay module uses the events generated by the parser module in order to fire the corresponding transition for each event.

- ***Petri Net Model***

The replay module replays events on top of the petri net model generated by the modeler module.

4.5.2 Outputs

- ***Model Event Data***

The replay module uses the generated petri net model along with the events parsed from the event log to generate useful data specific to each place and transition. Such data is important for analyzing the process's performance in the performance analysis stage.

The model event data can be divided into two groups: transition data and place data. Transition data includes the enabled time and the execution time of a given transition. Place data

includes the entry time, the exit time, and the enabling time of the first enabled transition connected to this place. They can be summarized as follows:

- **Enabled time of a transition**: The time at which a transition is eligible to fire but has not fired yet.
- **Executing time of a transition**: The time at which a transition is actually fired.
- **Entry time of a place**: The time at which a token is first emitted in that place.
- **Exit time of a place**: The time at which all tokens are consumed from that place.
- **Enabling time of a place**: The time at which the transition connected to that place is enabled.

4.5.3 Module Functionality

The replay module takes as an input the petri net model generated by the model generator as well as the events parsed from the event logs in the parser module. The replay module then replays these events consecutively on the generated model by mapping events to transitions. In other words, an occurrence of an event in the log is equivalent to firing the corresponding transition in the model. When a transition fires, the replay module uses the event timestamp to generate event model data. This data is used later in the performance analysis module.

For each replay step, the replay module fires a single transition, which in turn changes the distribution of tokens in the petri net. A replay process starts by firing the emitter transition whenever a begin event is encountered in the event log; this marks the start of the replay process. With each following event, the associated transition consumes tokens from its inbound places and produces tokens in its outbound places according to the cardinality of the arcs connecting these places and the given transition. This represents a step in the replay process. The replay process is concluded when the collector transition fires.

4.6 Model Grouper Module

The Model Grouper module is responsible for taking the petri net model as input and outputting the abstracted grouped petri net model. Figure 4-12 illustrates the block diagram of the grouper module.

Figure 4-12: Model Grouper Block Diagram

4.6.1 Inputs

- ***Petri Net Model***

The grouper module uses the petri net generated by the in order to create the abstracted grouped model.

4.6.2 Outputs

- ***Grouped Petri Net Model***

The grouped petri net model is an abstraction layer added on top of the petri net model. A place or transition in the grouped model groups multiple places or transitions from the initial petri net.

4.6.3 Module Functionality

There are four options grouping options: grouping by executing machine, grouping by script type, grouping by runnable iterations, or no grouping where the model is left as is. When a grouping option is specified all places/transitions that should be grouped together, are placed in one associated place/transition.

1. ***Grouping by executing machine***

When this option is specified all places/transitions that share the same name (for example place “ready”), host machine, and script type are grouped into one place/transition.

2. ***Grouping by script***

When this option is specified all places/transitions that share the same name (for example place "ready") and script type are grouped into one place/transition.

3. ***Grouping by runnable iterations***

When this option is specified all places/transitions that share the same name (for example place "ready"), iterative parent runnable, and script type are grouped into one place/transition.

4. ***No specified grouping option***

When no grouping option is specified, each place/transition maps to one place/transition.

4.7 Performance Analyzer Module

The performance analyzer module is responsible for generating the necessary performance metrics reported to the user. Figure 4-13 illustrates the block diagram of the performance analyzer module.

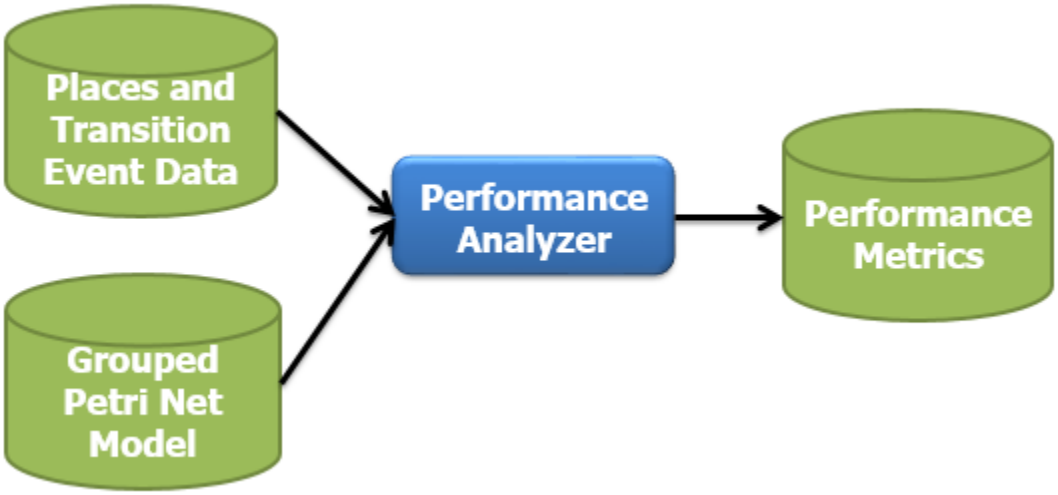

Figure 4-13: Performance Analyzer Block Diagram

4.7.1 Inputs

- ***Grouped Petri Net Model***

The performance analyzer module uses the grouped petri net model in order to generate the required performance metrics.

- ***Model Event Data***

The model event data generated by the replay module is used by the performance analyzer to calculate performance metrics.

- *Sojourn time* is the time spent by a token in a place from entering to exiting.
- *Waiting time* is the time taken by a token between enabling of the transition connected to the place to exiting the place.
- *Synchronization time* is the time taken by a token from entering a place to enabling of the transition connected to it.

4.7.2 Outputs

- ***Performance Metrics***

The performance analyzer module generates performance metrics that are used in the reporting and visualization module. The generated metrics are place time metrics, transition waiting time, place total time, resource utilization and split frequency.

- **Place performance metrics** are specific to a grouped place. Some of the metrics included are *sojourn time*, *waiting time*, and *synchronization time*. Each grouped place has a mean and associated standard deviation calculated for each metric.
- **Transition waiting time** is the time taken from enabling of the transition until the firing of said transition. The mean and standard deviation are calculated for each grouped transition.
- **Place total time** is the total time spent in a grouped place with consideration given to the parallelism of different tokens. It differs from sojourn time in that it is not specific to each token execution. The mean and standard deviation are calculated using the total time data from every place group.
- **Resource utilization** is calculated for each *running* grouped place. For each possible number of tokens running concurrently in the place, a percentage distribution of the total time spent is calculated and presented the user. Running place is discussed in more depth in section 2.

- **Split frequency** is a relation between each grouped place and each grouped transition connected to it. It is the number of times a token from a given grouped place fired a corresponding grouped transition.

4.7.3 Module Functionality

The performance analyzer operates on the grouped petri net model in order to generate five necessary metrics that give insights to different aspects of the given regression: place time metrics, transition waiting time, place total time, resource utilization and split frequency.

- **Place time metrics** are sojourn time, waiting time, and synchronization time. These metrics show how tokens behave at each state in the system. Mean and standard deviation of each metric is used to find execution place instances with low performance.
    - *Sojourn time* shows place instances inside a grouped place where tokens have spent more time.
    - *Waiting time* shows place instances inside a grouped place where execution was enabled but stalled.
    - *Synchronization time* shows place instances inside a grouped place where time was spent in parallel branches waiting for each other to finish execution.
- **Transition waiting time** shows transition instances inside a grouped place which were enabled but did not execute. Mean and standard deviation of each transition is used in order to find stalling transition instances.
- **Place total time** shows grouped places taking most of the execution time.
- **Resource utilization** shows how a *running* grouped place was utilized. If more parallel tokens were present concurrently in a running grouped place this place is said to be well utilized.
- **Split frequency** shows how tokens took different paths in the grouped model. This metric gives insights to how frequently different scenarios occurred in the regression.

4.8 Reporting and Visualization Module

The reporting and visualization module is responsible for visual representation of the data. The main goal of this module is to help end users analyze the generated model clearly and efficiently and present performance metrics in a more accessible, understandable and usable way. The generated petri net is visualized such that users can understand the structure and configuration of the RMDB file. In addition, the event logs are also visualized to enable users to proceed step by step with the event logs and figure out the distribution of tokens at every step.

Moreover, this module reports and visualizes the performance statistics generated from the performance analysis module for every single place and transition. The visualization of the model performance represents a guiding tool to figure out where bottlenecks may exist easily and quickly. The block diagram of the reporting and visualization module is illustrated in Figure 4-14

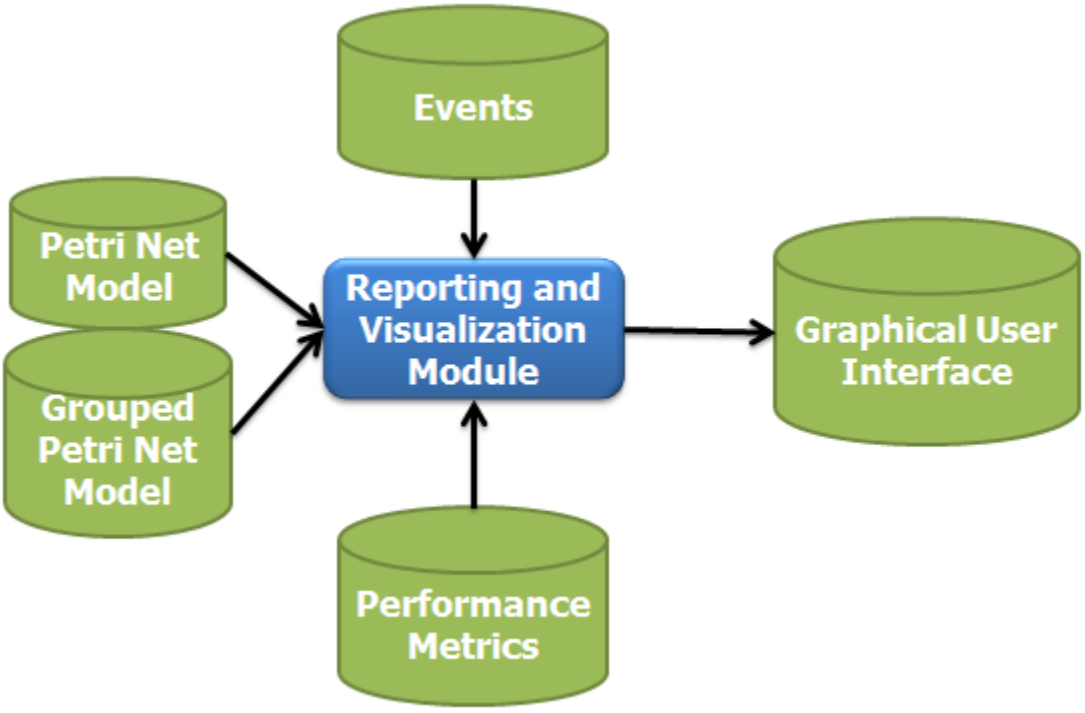

Figure 4-14: Reporting and Visualization Module Block Diagram

4.8.1 Inputs

- ***Petri Net Model:***

This module takes as an input the generated petri net model from the modeler module to visualize it and represent it in a clear and concrete form.

- ***Grouped Petri Net Model***

This module takes as an input the grouped petri net model. If a grouping option is specified by the user, the model is displayed at the associated abstraction level.

- ***Events***

This module uses the events generated from the parser module and interacts with the replay module in order to visualize the replay of these events on top of the petri net model.

- ***Performance Metrics***

This module uses the performance statistics generated from the performance analyzer module to display these statistics and overlay performance color for each place and transition in the petri net model.

4.8.2 Outputs

- ***Graphical User Interface***

This module provides a graphical user interface including buttons, menus and status bar that allows the user to interact easily with the generated model. The user interface is designed to be user friendly to enable users to learn the system quickly and use it efficiently with great accessibility and higher productivity.

4.8.3 Module Functionality

1. ***Visualize the generated petri net model***

This feature visualizes the abstract petri net model. When the user loads a given RMDB file and its corresponding RMEV log file, the generated petri net model is rendered with all places, transitions, and connecting arcs. Places, transitions, and arcs are represented as yellow circles, blue rectangles and black arrows respectively to make it easier for the user to grasp the

structure of the petri net. Lambda transitions are colored in purple. The user can trace the model easily and efficiently and explore any specific regions in the model as illustrated in Figure 4-15

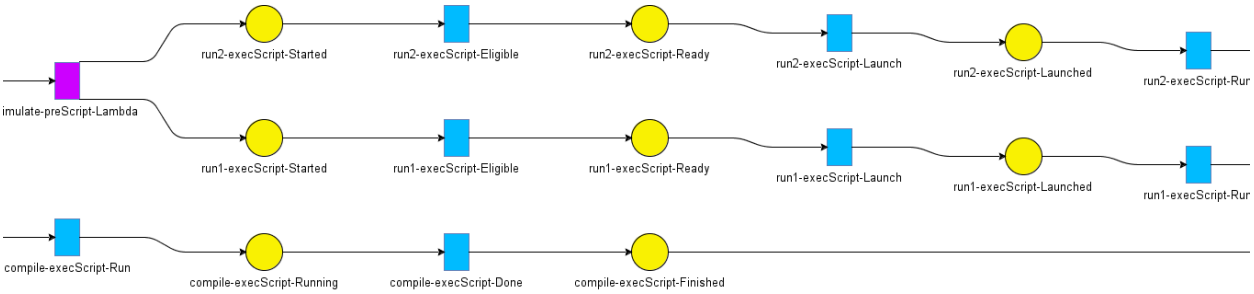

Figure 4-15: Generated Petri Net Model

2. ***Prune unnecessary branches***

The petri net model can become too complex and too large for the user to trace and explore. This feature enables the user to prune the model and remove unnecessary branches. These branches represent unvisited branches in the model. When these branches are removed, the model becomes simpler and easier to debug and analyze as illustrated in Figure 4-16. For example, all the scripts may run successfully and hence there will be no need to include the ***rerun*** branches or ***triageScript*** branch in the model.

3. ***Group the petri net model***

This feature enables the user to view the model from different perspectives. The user can choose a grouping option by which related places are grouped into a single place and related transitions are grouped into a single transition. As explained in section 4.6, the user can group the petri net model by host machine, script, or by runnable iterations. Once a grouping option is specified, the new grouped model is generated and rendered. This feature is powerful as it adds an abstraction layer on top of the model and shows the relevant perspective of the model to the user as illustrated in Figure 4-17.

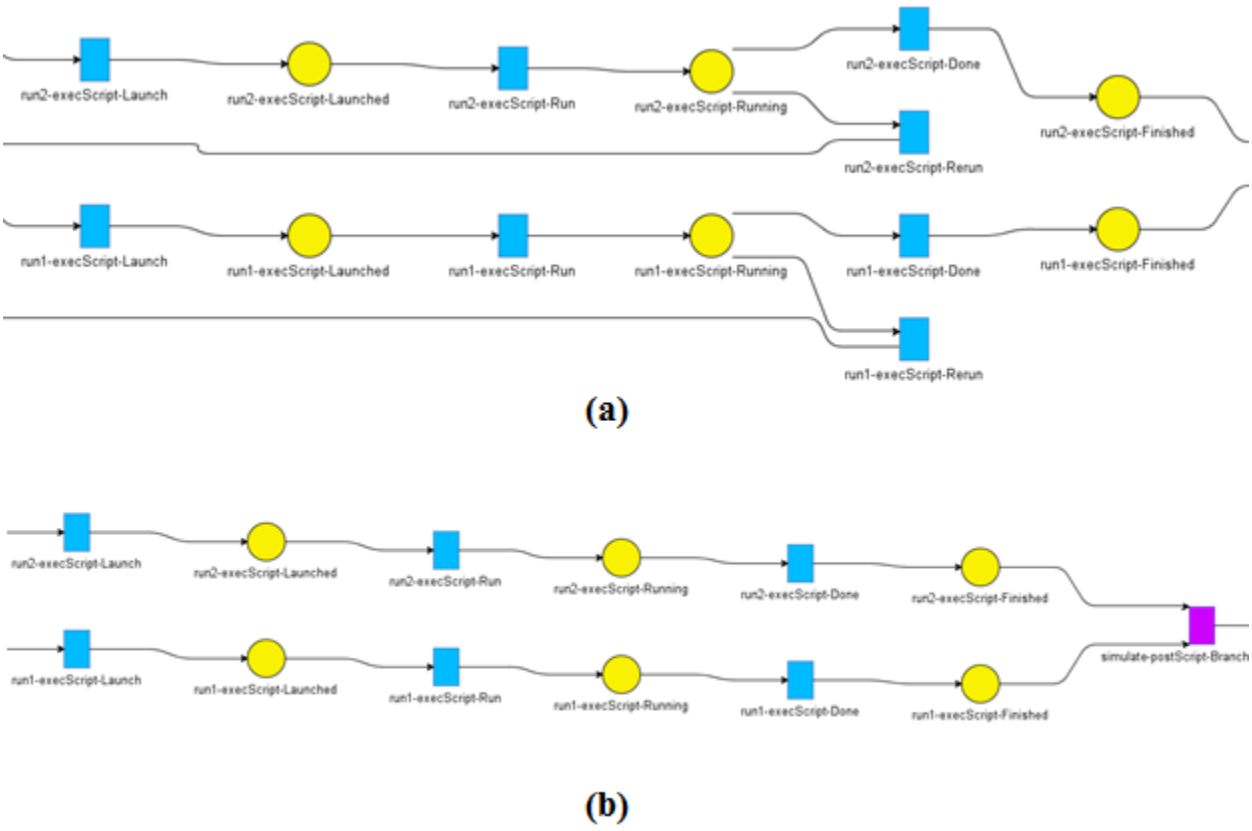

Figure 4-16: Pruning rerun branches. (a) Unpruned model (b) Pruned model

4. Automatic replay of event logs

This feature enables the user to replay the event log on top of the generated model automatically. The user can visualize how the distribution of tokens changes at every time step from the very beginning when the emitter fires until the collector fires. The user can pause the automatic replay of events, analyze the marking of the petri net at this current state, and then resumes the replay process. While replaying, the replay path is highlighted to help the user visualize the progress of events.

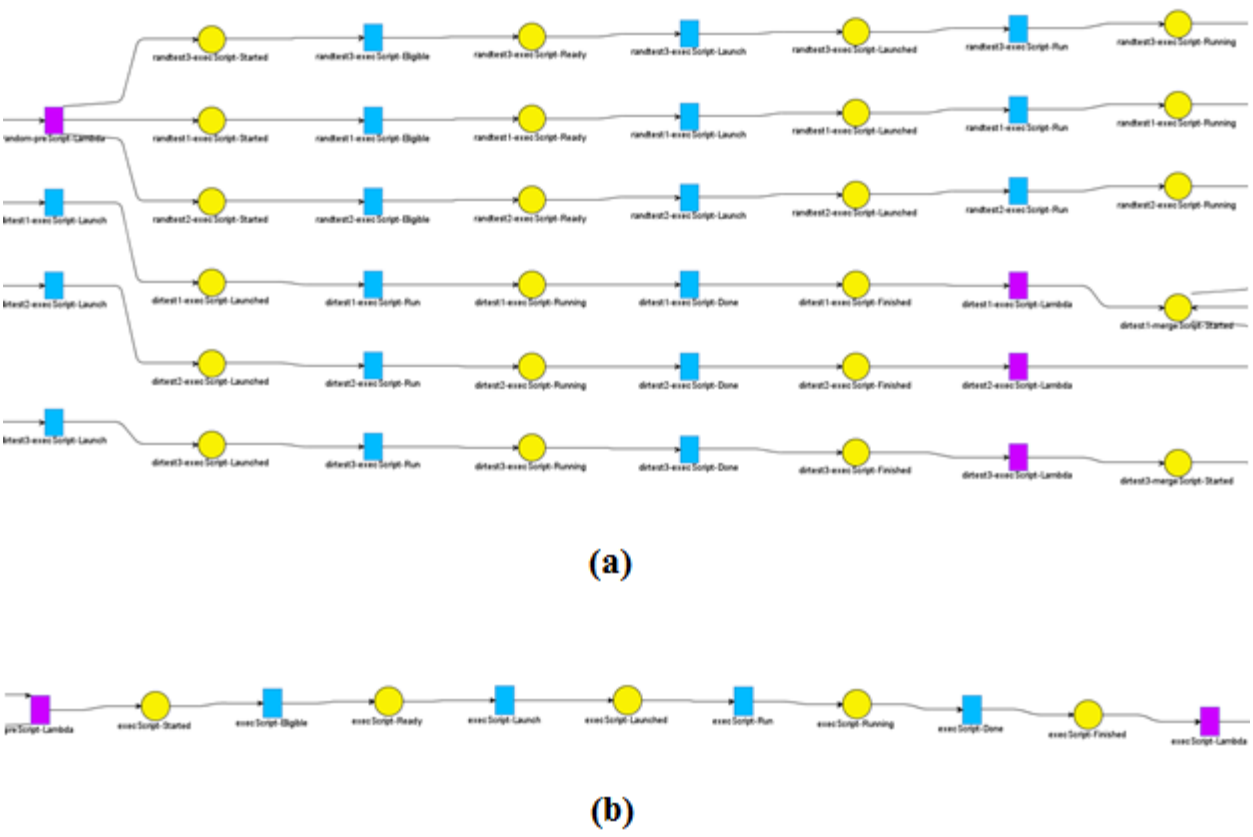

Figure 4-17: Grouping by Execute Scripts. (a) Ungrouped model (b) Grouped Model

5. Manual replay of event logs

This feature enables the user to manually replay the event logs on top of the generated model. The user can replay events successively step by step. At each step, one event is processed and an associated transition is fired. The user can then visualize the marking of the petri net after this transition has fired. While progressing, the replay path is highlighted in order to help the user visualize the progress of events as illustrated in Figure 4-18.

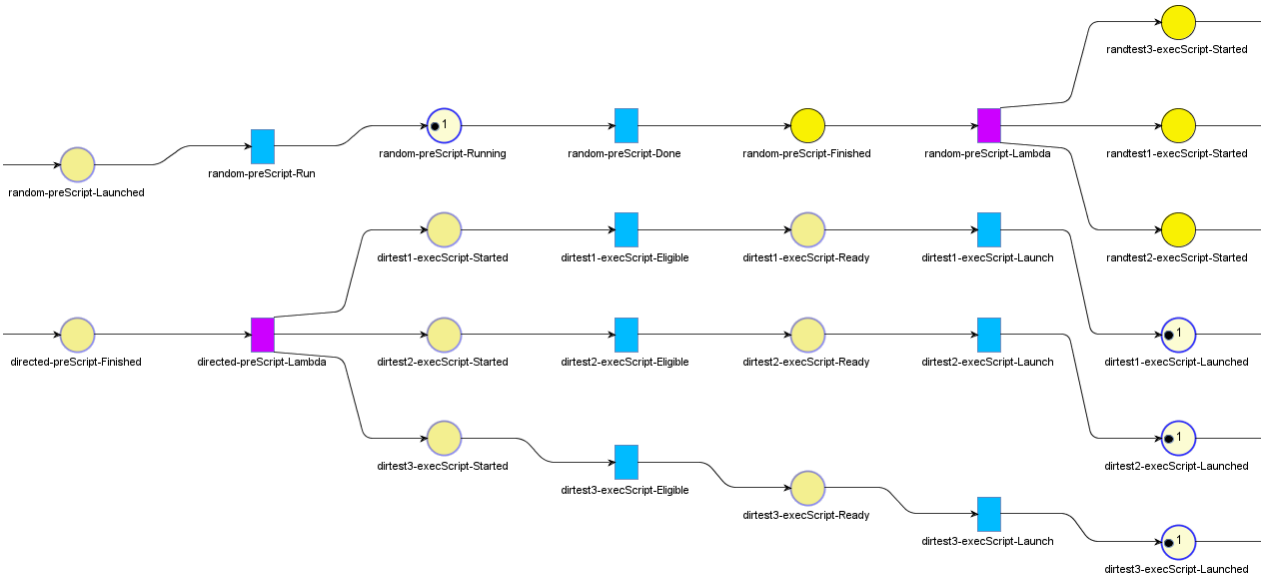

Figure 4-18: Manual Replay of Events

6. Speed control of automatic replay

This feature enables the user to control the speed at which the automatic replay progresses. The user can choose to increase the speed of automatic replay when they are not interested in inspecting certain parts of the model. On the other hand, the user can choose to decrease the speed of automatic replay when they want to investigate a certain property or behavior in a certain region in the model.

7. Debugging mode

This feature enables the user to set breakpoints and debug the petri net model. Initially, one or more places are set as breakpoints and automatic replay is done in the background until a breakpoint is encountered. The places with breakpoints are displayed with red borders to distinguish them from normal places. At this step, the debug mode is initiated and the user can inspect the model as illustrated in Figure 4-19. The user can also choose to show performance statistics or toggle performance color overlays for all places and transitions while debugging. At any time, the user can clear one or all of the breakpoints. The debug mode is terminated when all breakpoints are cleared.

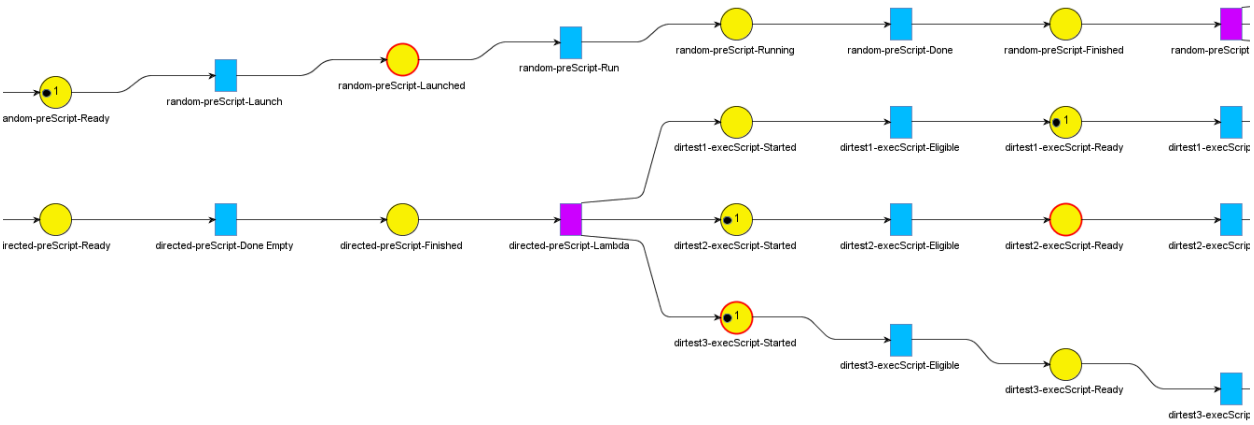

Figure 4-19: Debugging with breakpoints

8. Display performance statistics

This feature enables the user to show performance statistics such as waiting time, sojourn time, and synchronization time for places as well as waiting time for transitions. In the grouped model, the same statistics are displayed for grouped places and transitions according to the chosen grouping option. This feature can be toggled in normal mode, debug mode, replay mode, and grouping mode. This feature helps the user grasp statistics for each place and transition in a clear and organized representation as shown in Figure 4-20.

9. Toggle performance color overlay

This feature enables the user to further visualize the performance of the model by changing the color of each place according to amount time spent in the place compared with all other places. In the grouped mode, the color indicates the total time spent by one token or more in the grouped place(s). Places where tokens spend the longest time are colored in red and places where tokens spend the least time are colored in green. The color scale is applied to all places with blending to show the various times between the two extremes as shown in Figure 4-20

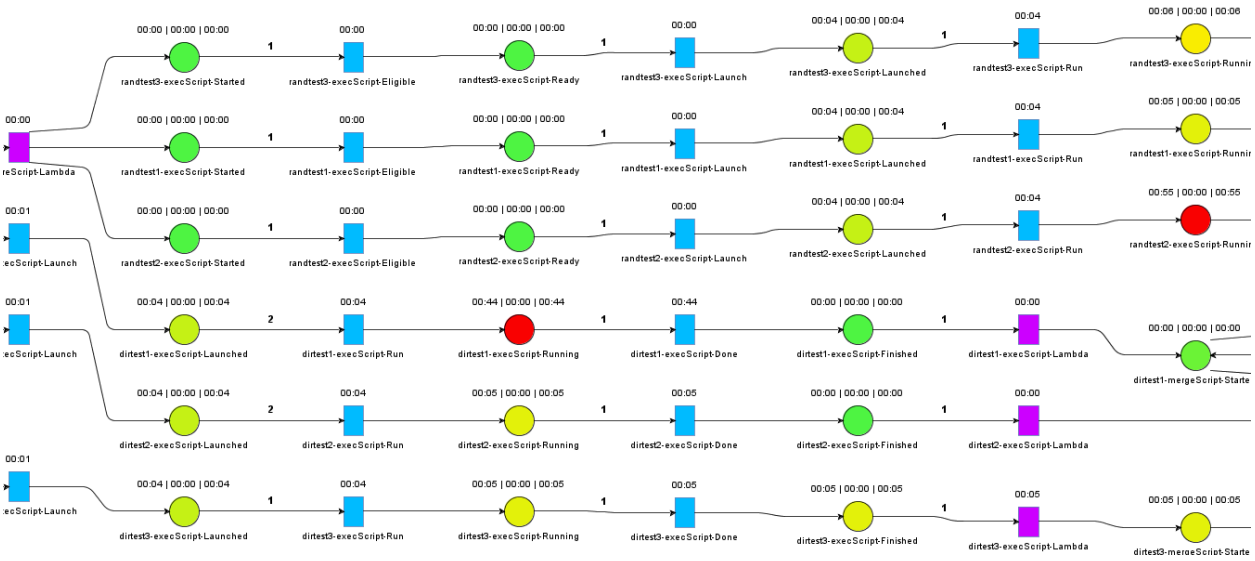

Figure 4-20: Performance color and statistics overlay

10. Performance panel for places and transitions

This feature enables the user to click on a grouped place or transition and show the performance metrics for it. For a place, sojourn time, waiting time, and synchronization time of each token passing through the place is shown together with the mean and standard deviation. For the *running* place, resource utilization is also shown for each number of tokens in the place. For a transition, the waiting time for each token firing the transition is shown together with the mean and standard deviation. Coloring is also applied to each metric to highlight outliers as illustrated in Figure 4-21. Performance metrics are discussed in more depth in section 4.7.

Place Statistics

Heirarchy	Sojourn Time	Sync Time	Waiting Time
nightly/random/randtest2	00:55	00:00	00:55
nightly/directed/dirtest1	00:44	00:00	00:44
nightly/random/randtest3	00:06	00:00	00:06
nightly/directed/dirtest3	00:05	00:00	00:05
nightly/random/randtest1	00:05	00:00	00:05
nightly/directed/dirtest2	00:05	00:00	00:05
Average	00:20	00:00	00:20
Std. Deviation	00:21	00:00	00:21

Resource Utilization

Tokens in place	% of total time
1	26.67
2	55.00
3	10.00
4	8.33

Figure 4-21: Performance Panel of a place

11. Zooming-in and zooming-out

This feature enables the user to zoom in the model to inspect specific parts more clearly and zoom out the model to get an overall view of the model.

Chapter 5 :

Languages, Tools and Libraries

In this chapter, languages, tools, and libraries used in creating VRM performance analyzer are discussed.

5.1 Languages

5.1.1 Java

Description	Java is a general-purpose high level programming language that is class based, object oriented, and platform independent. One of the main advantages of Java is that Java programs are portable; any system with a java virtual machine can run compiled Java code. [2]
Usage	VRM performance analyzer is implemented in Java to utilize its portability. VRM itself can run in both Windows and Linux environments. Therefore, VRM performance analyzer needs to be platform independent.

5.1.2 Tcl

Description	Tcl (Tool command language) is a dynamic programming language that is suitable for a wide range of uses such as web applications, desktop applications, networking, and testing. [3]
Usage	Tcl is used to interface with VRM APIs in order to retrieve information from RMDB files.

5.2 Tools

5.2.1 Yasper

Description	Yasper (Yet Another Smart Process EditoR) is a tool for modeling stepwise processes. Yasper uses petri nets as the process representation tool. It facilitates analysis and simulation of processes. [4]
Usage	Yasper was used to perform preliminary modeling experiments of different VRM processes.

5.2.2 ProM

Description	ProM is an extensible framework developed in Java that supports a wide variety of process mining techniques. [5]
Usage	ProM was used to do perform process mining experiments on manually created models.

5.2.3 Eclipse

Description	Eclipse is an integrated development environment (IDE) mainly used for Java development. Eclipse is built mostly in Java and is based on plugin-development. All features in Eclipse are developed as a plugin first then integrated into Eclipse. There are numerous distributions of Eclipse to fit the needs of most users. Such Eclipse flavors include Java development and C/C++ development.
Usage	Eclipse is the IDE used to develop VRM performance analyzer due to its portability, ease of use, and flexibility in integrating with various development software such as Git.

5.2.4 Git

Description	Git is a free and open source distributed version control system. It is designed to handle small and large projects with speed and efficiency. [6]
Usage	Git is used as the version control system for this project due to its ease of use, integrity, and scalability. It allowed us to develop features of the tool in parallel.

5.3 Libraries

5.3.1 Swing

Description	Swing is a Java GUI widget toolkit developed to provide a native platform user interface. It is part of the Java foundation classes and it is platform independent. [7]
Usage	Swing is used as the basis for the VRM Performance Analyzer GUI.

5.3.2 **JGraphX**

Description	JGraphX is a Java swing diagramming library. Its core functionality is visualizing graphs.
Usage	JGraphX is used to draw the models of the system generated by VRM performance analyzer.

Chapter 6 :

Testing Plan

In this chapter, the testing plan used is specified. To begin with, unit testing for each individual module is performed to indicate whether or not the module functions accurately and completely. Results are specified in Table 6-1, Table 6-2, Table 6-3, and Table 6-4.

Black box testing is then added to make sure that every module works perfectly performing its function. It is applied on the UI module as it is the one that black boxes all other modules. Results are specified in Table 6-5.

6.1 Unit testing

Table 6-1: Modeler Module Unit Testing

Modeler Module			
#	Input	Expected Output	**Passed**
1	Single task runnable	The model of a task runnable containing its prescript, postScript, and execScript.	**Yes**
2	Parallel group runnable	The model of a group runnable, with multiple parallel task runnables, showing its preScript, postScript, mergeScript, triageScript, and each task runnable's execScript.	**Yes**
3	Parallel Sequential runnable	The model of a group runnable, with multiple sequential task runnables, showing its prescript, postScript, mergeScript, triageScript, and each task runnable's execScript.	**Yes**
4	PreScript	The model of the preScript with only one branch showing its execution.	**Yes**
5	PostScript	The model of the postScript with only one branch showing its execution.	**Yes**
6	ExecScript	The model of the execScript with only one branch showing its execution.	**Yes**

7	MergeScript	The model of the mergeScript with three branches. One for the generation of a merge file, another for merging the current UCDB file into the merge file, and a third for skipping the script.	**Yes**
8	TriageScript	The model of the triageScript with three branches. One for the generation of a triage file, another for triaging the current UCDB file into the triage file and a third for skipping the script.	**Yes**

Table 6-2: Replay Module Unit Testing

Replay Module			
#	Input	Output	**Passed**
1	RMDB file and log file*.	The flow of executing log entries on the model generated without any deviations.	**Yes**
	* *Several RMDB files and log files were tested and none generated a deviation.*		

Table 6-3: Model Grouper Module Unit Testing

Model Grouper Module			
#	Input	Output	**Passed**
1	Model to be grouped by running machine.	A model where all places/transitions with the same name, the same script type, and the same host machine, are grouped and modeled as one place/transition.	**Yes**
2	Model to be grouped by script.	A model where all places/transitions with the same name and the same script type are grouped and modeled as one place/transition.	**Yes**
3	Model to be grouped by iterations.	A model where all places/transitions with the same name, script type, and whose scripts are iterations of the same runnable are grouped and modeled as one place/transition.	**Yes**

Table 6-4: Performance Analyzer Module Unit Testing

Performance Analyzer Module			
#	Input	Output	Passed
1	An RMDB file whose performance statistics are manually calculated.	The performance analysis calculated from the module should be similar to those manually calculated.	Yes

6.2 Black Box testing

Table 6-5: UI Module Black Box Testing

UI Module			
#	Input	Output	**Passed**
1	Click on **Generate Model** button without importing an rmdb and/or an rmev files.	Error Message: "Please import a valid rmdb/rmev file"	**Yes**
2	Click on the **Import** option	Import menu appear. Upon choosing an option a directory window appears to choose the desired file.	**Yes**
4	Click on the **Generate Model** button.	A model for the specified rmdb/rmev file is generated.	**Yes**
5	Click on the **Group By Iterations** button	A model, where places/transitions are grouped by iterations, is generated.	**Yes**
6	Click on the **Group By Machine** button	A model, where places/transitions are grouped by machine, is generated.	**Yes**
7	Click on the **Group By Script** button	A model, where places/transitions are grouped by script type, is generated.	**Yes**

8	Click on the **Debug** button	The model enters replay mode and advances to the nearest breakpoint.	**Yes**
9	Click on the **Step** button	The model enters replay mode and advances one step forward.	**Yes**
10	Click on the **Replay Automatically** button	The model enters replay mode and advances step by step with the specified run speed until replay finishes.	**Yes**
11	Click on the **Pause** button	All replay actions are paused.	**Yes**
12	Click on the **Reset Tokens** button	All replay actions are abandoned.	**Yes**
13	Click on the **Zoom Out** button	The screen zooms in.	**Yes**
14	Click on the **Zoom In** button	The screen zooms out.	**Yes**
15	Click on the **Performance Color Overlay** button	The colors of places change depending on their respective places' performance statistics.	**Yes**
16	Click on the **Performance Statistics Overlay** button	The performance metrics for each place appears above it. Also, the execution split frequencies appear on arcs outbound from a place.	**Yes**
17	Click on the **Toggle Full Model Display** button	The full model appears including branches that were never visited.	**Yes**
18	Click on the **Set Breakpoints** button	When a place is clicked it is assigned a breakpoint and its border changes to red.	**Yes**
19	Click on the **Clear Breakpoints** button	All breakpoints are cleared.	**Yes**

20	Click on the **Terminate** button	The application closes.	**Yes**
21	Click on any place	A performance panel appears showing the place's performance statistics and resource utilization.	**Yes**

Chapter 7 : Conclusion and Future Work

7.1 Conclusion

Our main proposal was focused on analyzing the performance of a large complex process that is Mentor Graphics Questa VRM. In order to complete this task, we explored how Process Mining could help in the analysis of VRM. We used this knowledge to develop a tool for debugging VRM regression runs.

During the development of our tool, we constructed a process model for VRM using a customized version of Petri Nets. We also defined a set of metrics for analyzing the performance of VRM regression runs using this model; these metrics include Wait Time, Synchronization Time, Sojourn Time, and Resource Utilization. We developed software that dynamically generates a process model for each VRM regression run. This tool allows the user to replay a regression run and visualize its performance.

This tool gives Mentor Graphics and VRM end users the ability to analyze the performance of VRM regression runs and ultimately make better decisions when configuring VRM to gain the utmost in regression performance.

7.2 Future Work

A few residual elements of the VRM process were not included in our implementation of the tool. Although unimportant to almost all realistic situations involving the use of this tool, their inclusion into the process model and tool combination we have created for VRM would be an added benefit.

7.2.1 VRM's Remaining Scripts

A few scripts – *tplanScript*, *trendScript*, and *deleteScript* – have not been included in our model. *deleteScript* is an automatically generated pseudo script used by VRM to clean up execution data. *trendScript* and *tplanScript* are user defined scripts that allow the user to override certain VRM housekeeping tasks. The inclusion of these scripts in our model would allow the user to debug certain edge cases associated with their use.

7.2.2 User Provided Tcl Code

User provided Tcl code or *userTcl*, is a special feature VRM provides for users to inject specific Tcl scripts into a regression. They allow the user to override certain VRM functionality to suite their highly specific needs. VRM accommodates most users out of the box; however, this provides extra functionality for those inclined. As with the scripts mentioned above, the inclusion of *userTcl* would allow the user to uncover any performance issues related to their *userTcl* usage.

7.2.3 Web-Based UI

In its current iteration, the tool must run on a local machine containing the required input files and libraries. As is the case with most regression projects, the machines that run VRM are *headless*, i.e. they are not connected to conventional display and input controls; they are mostly controlled through web applications. In order to conform to this paradigm, our tool would also have to be developed into a web application to ease its deployment.

Chapter 8 :

Bibliography

[1] P. Rashinkar, P. Paterson and L. Singh, System-on-a-Chip Verification: Methodology and Techniques, Springer, 2002.

[2] W. M. P. van der Aalst, Process Mining Discovery, Conformance and Enhancement of Business Processes, Busan: Springer, 2011.

[3] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE,* vol. 77, no. 4, pp. 541 - 580, 1989.

[4] Process Mining Group, "Performance Analysis with Petri Net," Eindhoven University of Technology, 26 05 2009. [Online]. Available: http://www.processmining.org/online/performanceanalysiswithpetrinet. [Accessed 15 5 2017].

[5] J. Gosling, B. Joy, G. Steele, G. Bracha and A. Buckley, The Java® Language Specification, 2014.

[6] Tcl Community, [Online]. Available: https://www.tcl.tk/. [Accessed 25 5 2017].

[7] TU Eindhoven, [Online]. Available: http://www.yasper.org/. [Accessed 25 5 2017].

[8] TU Eindhoven, [Online]. Available: http://www.promtools.org/doku.php. [Accessed 25 5 2017].

[9] Git Community, [Online]. Available: https://git-scm.com/. [Accessed 5 25 2017].

[10] SSS IT Pvt Ltd, "Java Swing Tutorial," [Online]. Available: https://www.javatpoint.com/java-swing. [Accessed 30 5 2017].

Appendix A:

User Manual

This guide explains how to setup and use **VRM Performance Analyzer** in steps:

1. Start **VRM Performance Analyzer** as shown in Figure A-1.

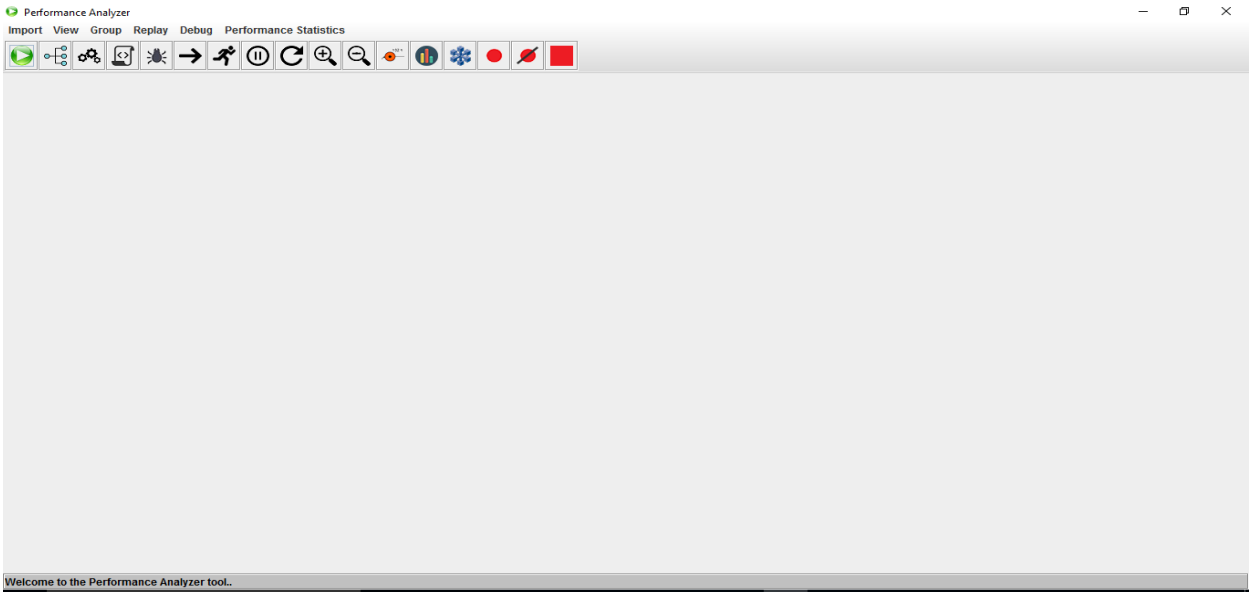

Figure A-1: VRM Performance Analyzer

2. Choose **Import** from the menu bar and select **Import RMDB file** as shown in Figure A-2.

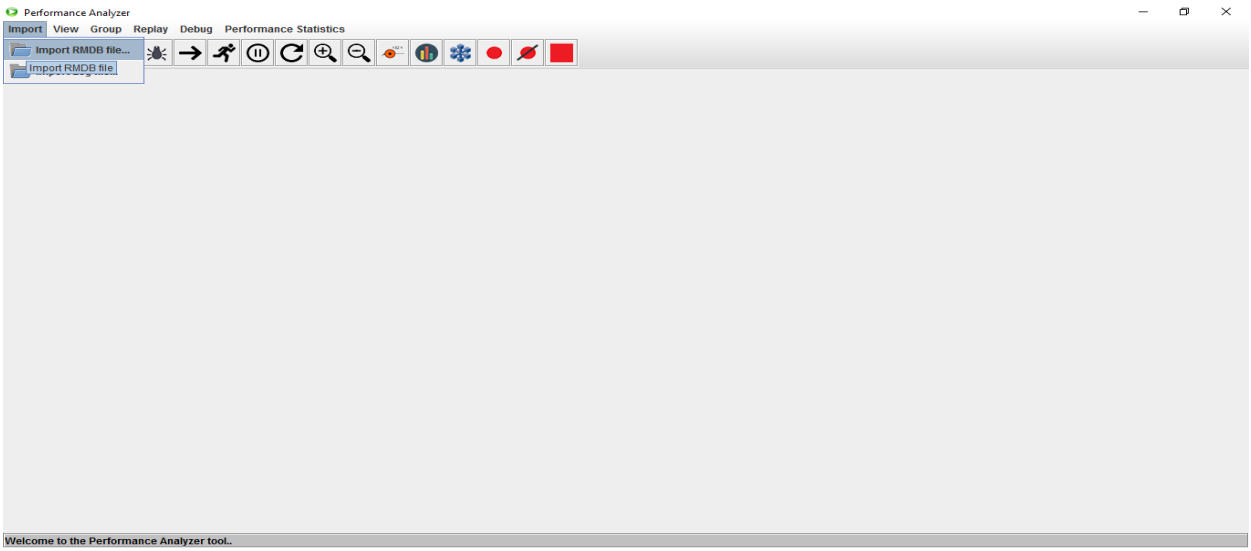

Figure A-2: Import RMDB File

3. Choose a valid *rmdb* file to import and press **OK**.
4. Choose **Import** from the menu bar and select **Import Log File**.
5. Choose a valid *rmev* file to import and press **OK**.
6. Click on the **Generate Model** button in the toolbar.

7. After the model is loaded, the corresponding petri net graph is displayed as Figure A-3.

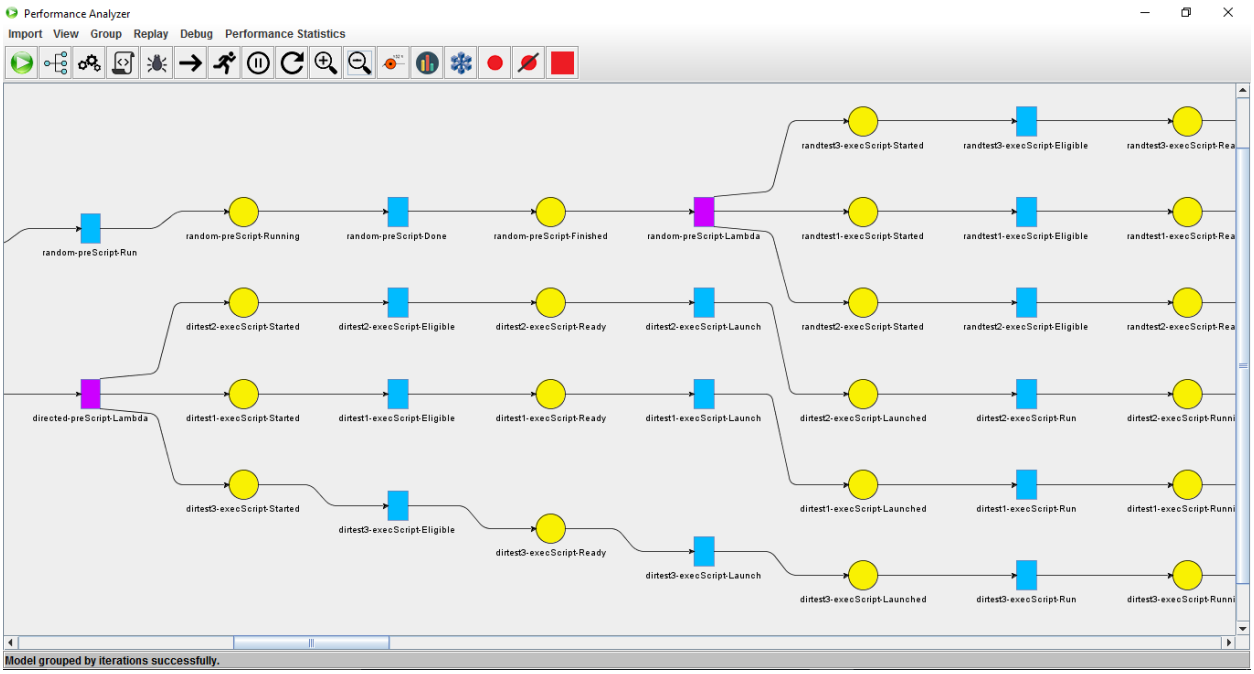

Figure A-3: Generated Petri Net Model

8. Choose **Replay** from the menu bar and select **Replay Automatically** for automatic replay functionality.
9. Click on **Pause** button to pause the automatic replay.
10. Click on the **Step** button for manual replay as shown in Figure A-4.

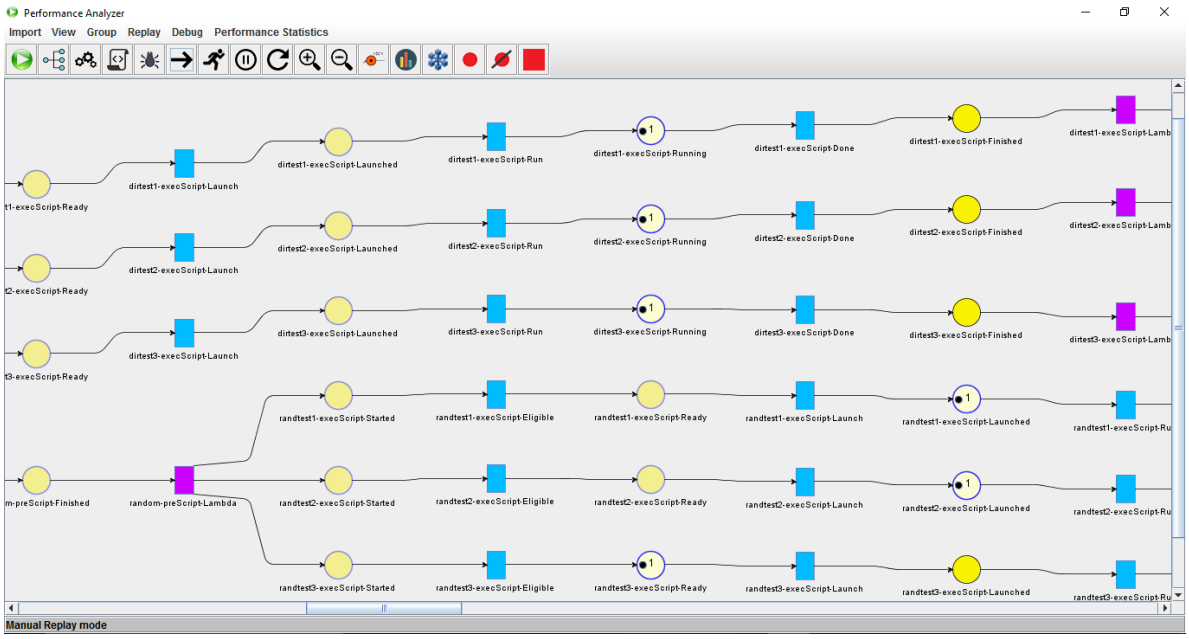

Figure A-4: Manual Replay Mode

11. For grouping by machine name, click on **Group by Machine** button. A grouped petri net by machine name will be displayed as Figure A-5.

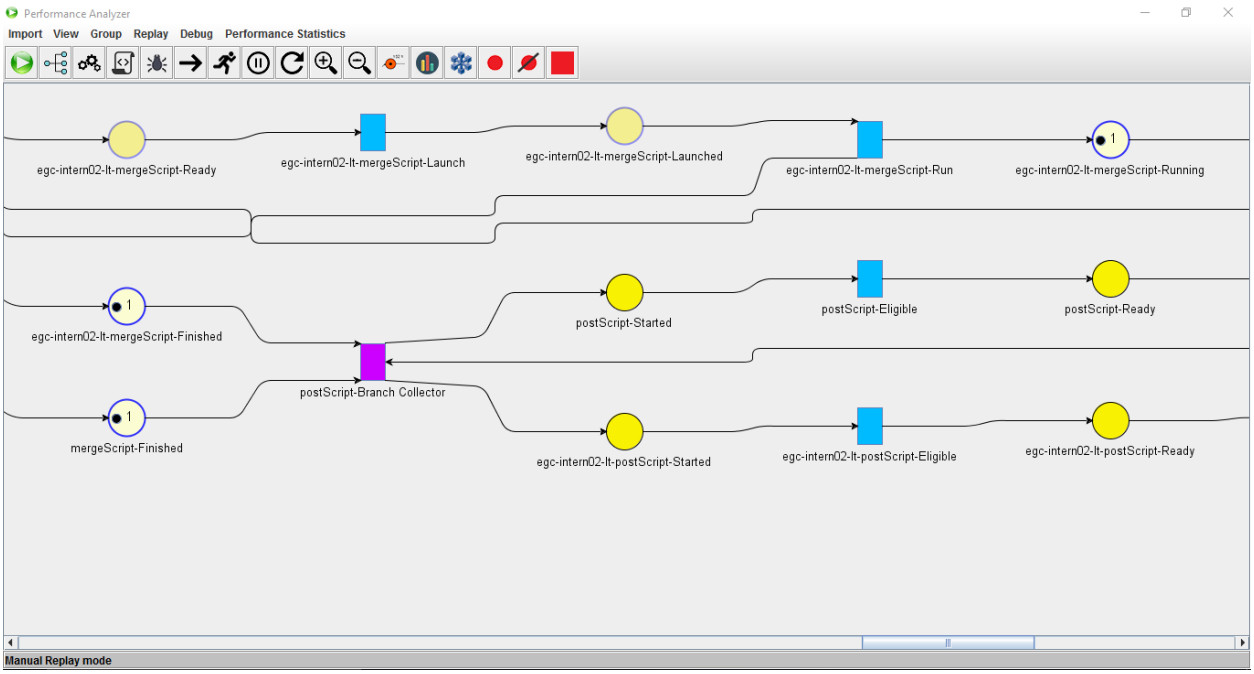

Figure A-5: Grouping by Machine

12. For grouping by script name, click on **Group by Script** button. A grouped petri net by script name will be displayed as Figure A-6.

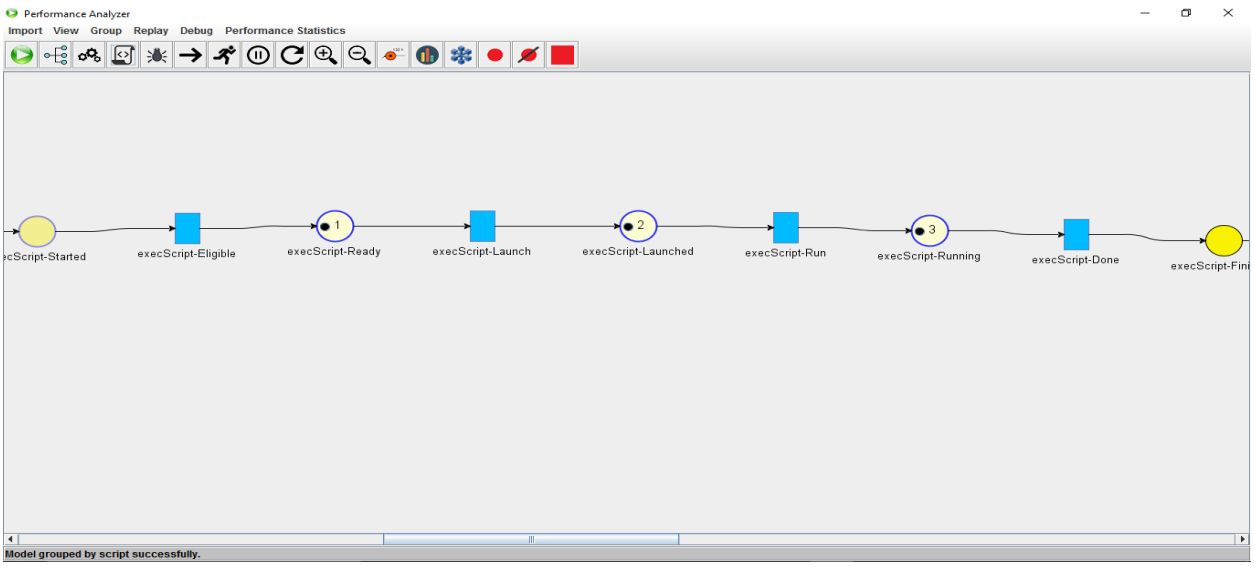

Figure A-6: Grouping by Scripts

13. For grouping by iterations, click on **Group by Iterations** button. A grouped petri net by iterations will be displayed similar to the previous grouping methods.

14. Click on **Reset Tokens** to clear all places and terminate replay mode.
15. Select **Performance Statistics** from the menu bar and then choose **Performance Color Overlay**. A performance graph will be displayed as shown in Figure A-7.

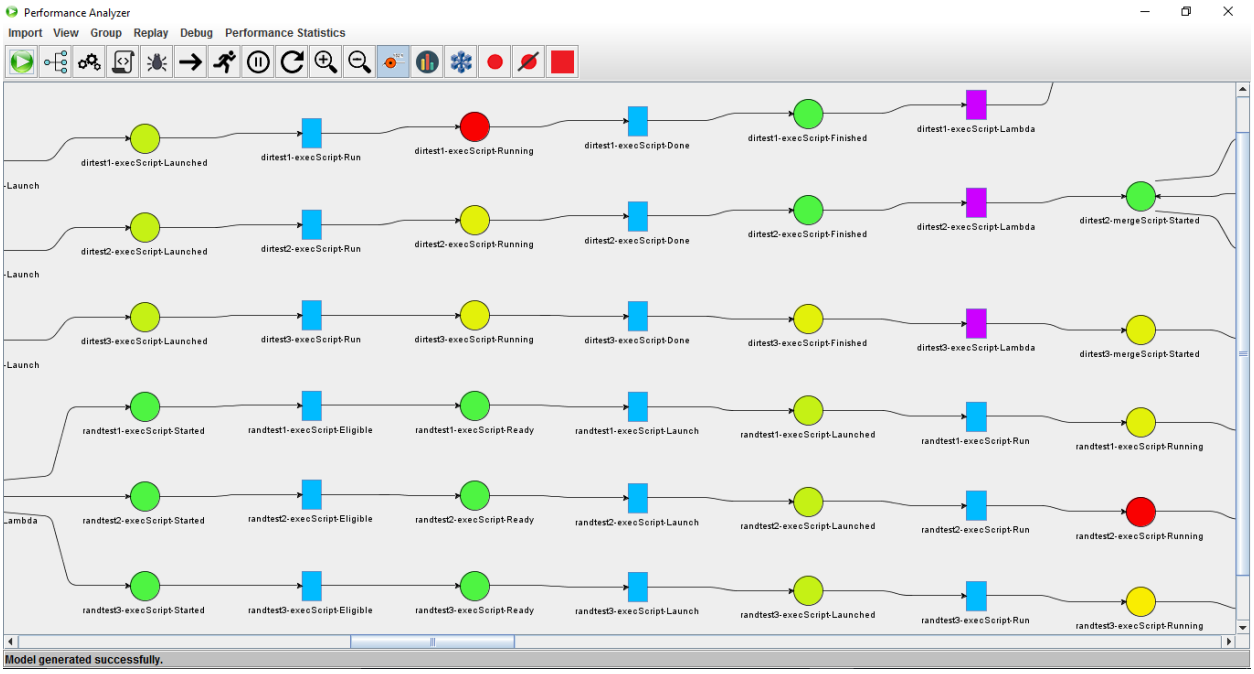

Figure A-7: Performance Color Overlay

16. Select **Performance Statistics** from the menu bar and then choose **Performance Statistics Overlay**. Statistics for every place and transition are displayed as shown in Figure A-8.

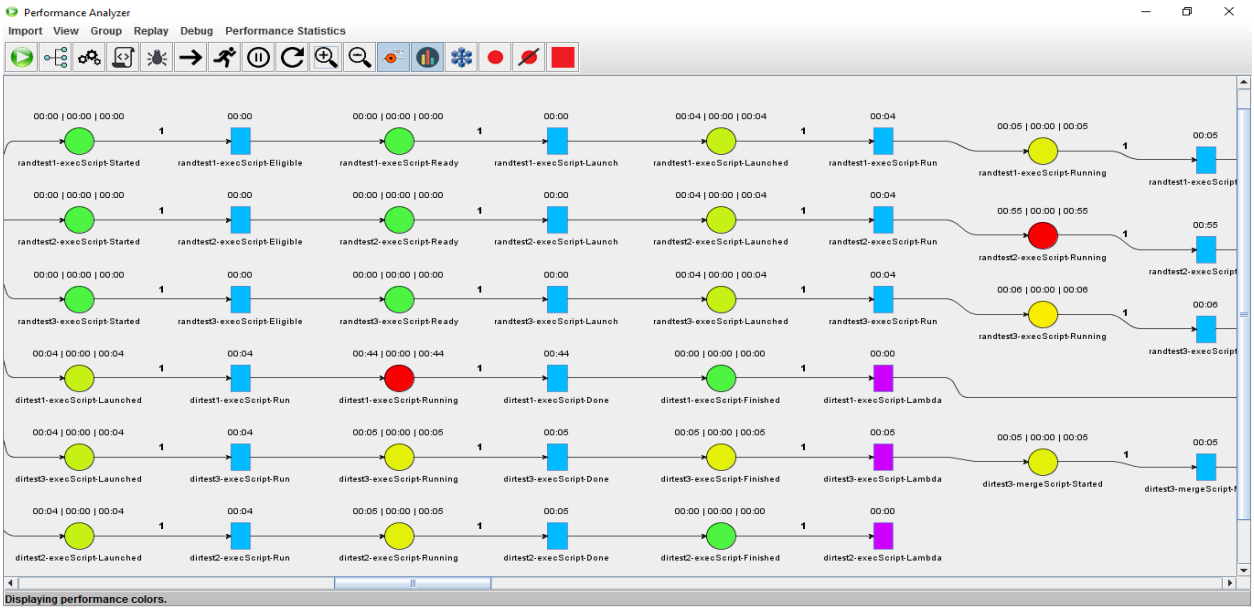

Figure A-8: Performance Statistics Overlay

17. For full model view, click on **Toggle Full Model Display** button from the toolbar. A full model will be displayed as Figure A-9.

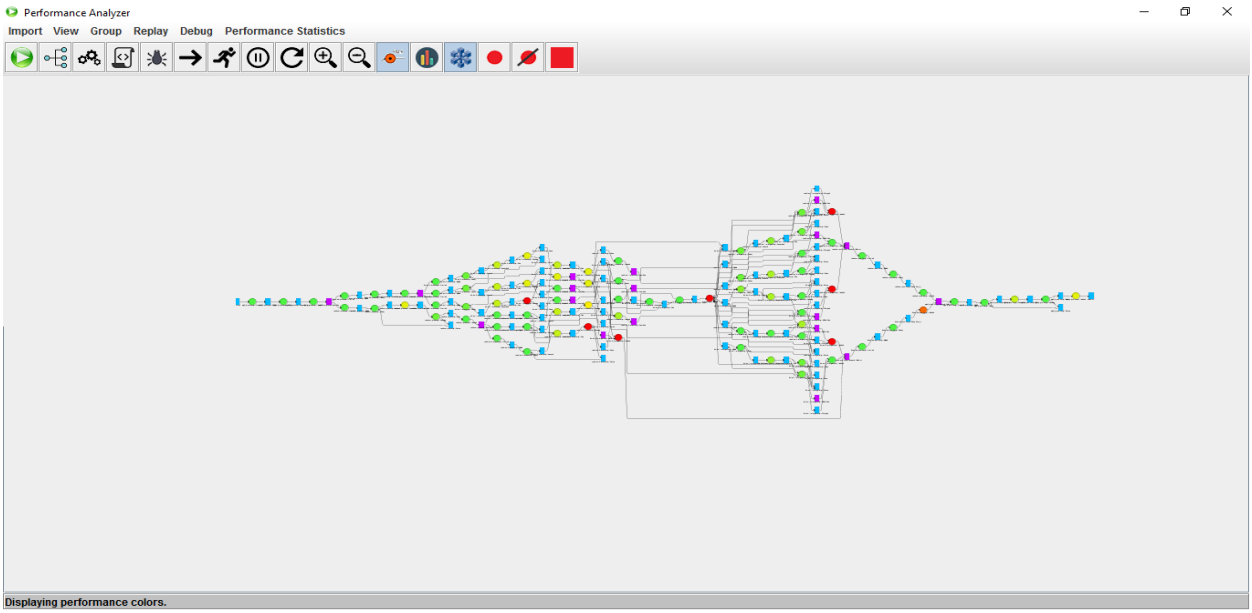

Figure A-9: Full Model Display

18. Return back to the normal graph by clicking on the **Toggle Full Model Display** button.
19. For detailed performance statistics on places, click on any place in the model. A performance panel for the selected place will be displayed as illustrated in Figure A-10.

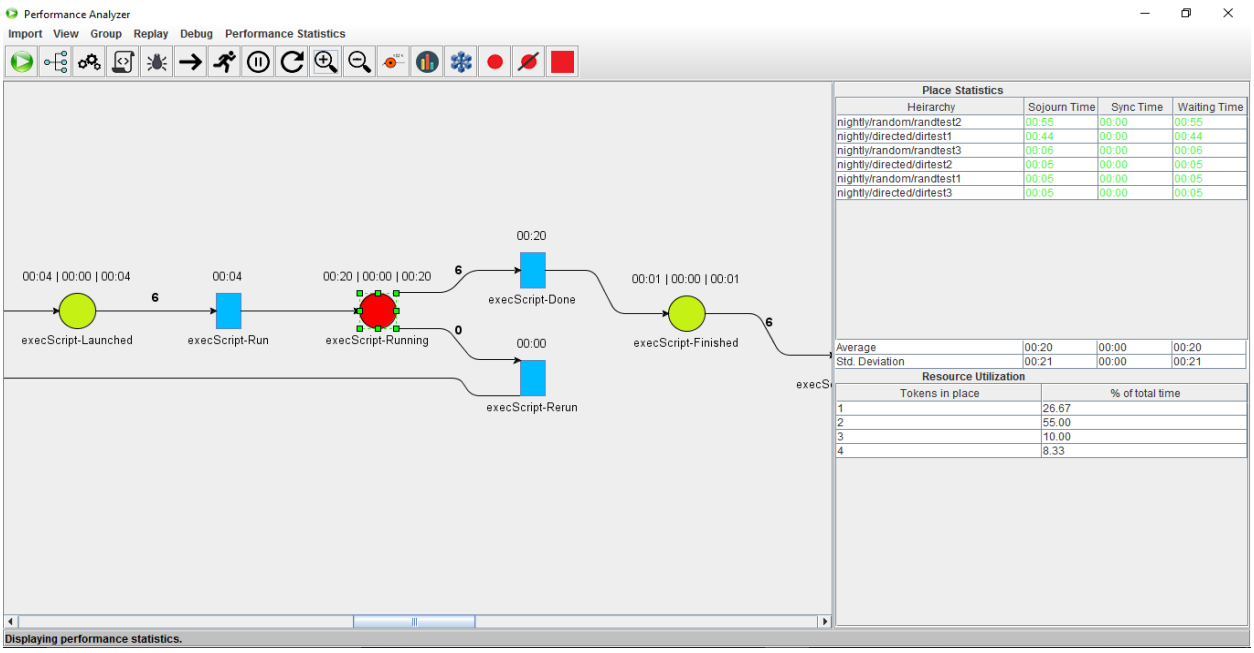

Figure A-10: Performance Statistics of a place

20. For detailed performance statistics on transitions, click on any transition in the model. A performance panel for the selected transition will be displayed as illustrated in Figure A-11.

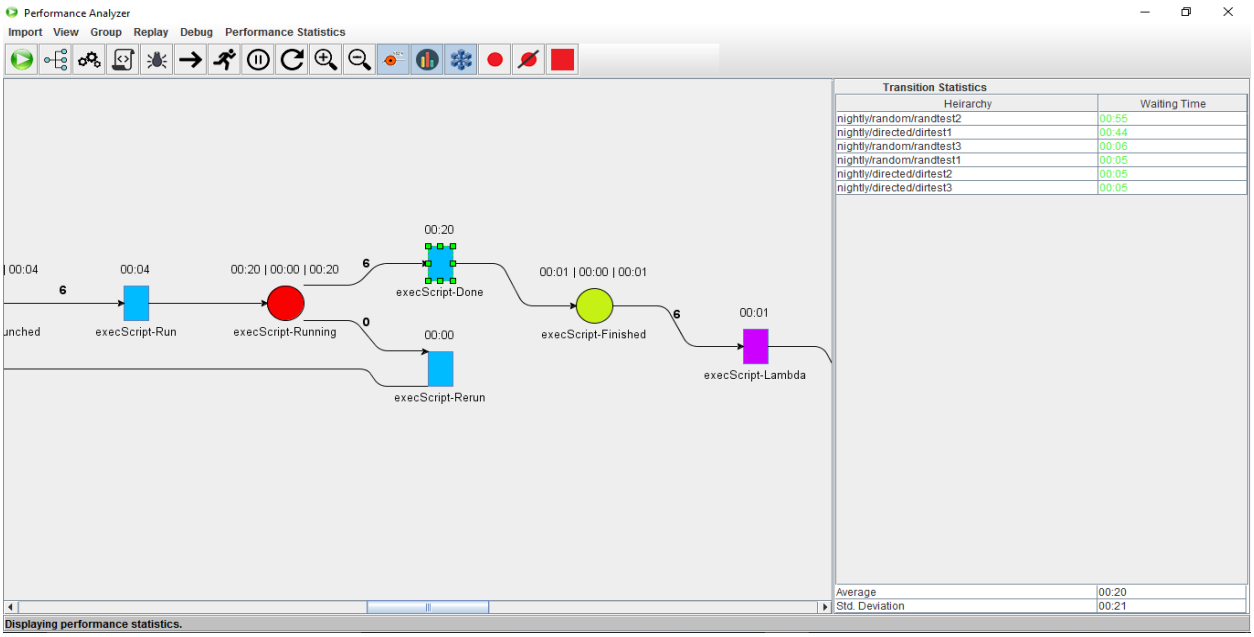

Figure A-11: Performance Statistics of a transition

21. For **debugging**, click on **Set Breakpoints** buttons from the toolbar.
22. Select on any place to set a breakpoint at that place as shown in Figure A-12. Places with breakpoints will be displayed with a red border. Normal places are shown with a black border.

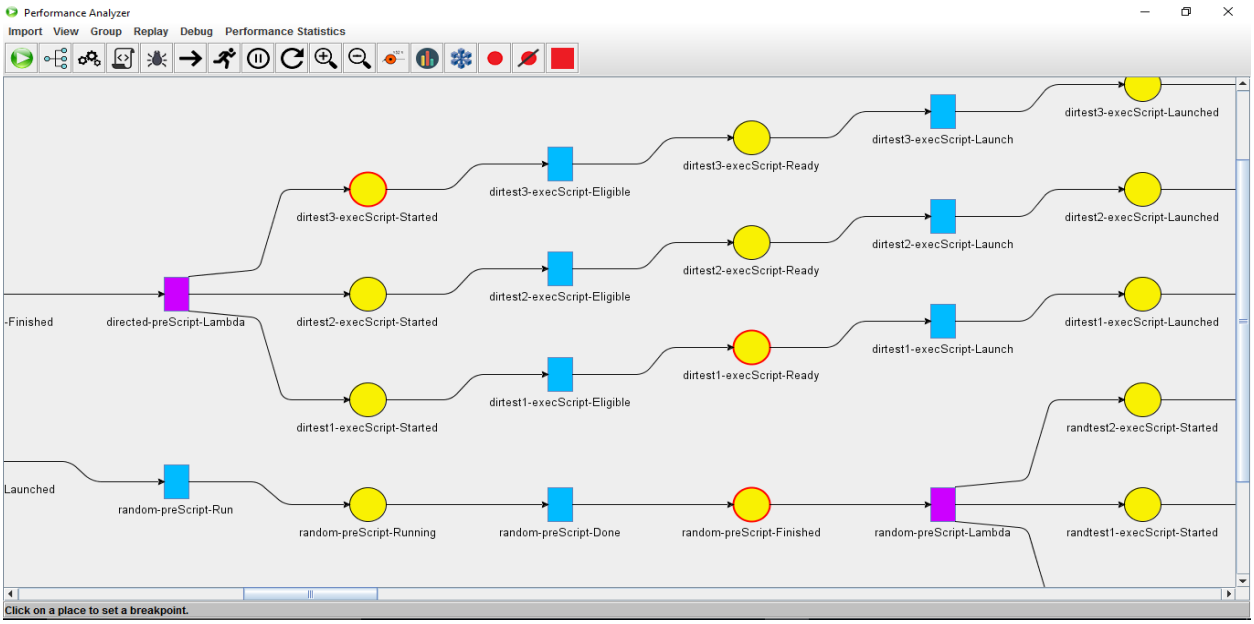

Figure A-12: Debug Mode

23. Click on **Debug** button from the toolbar. The replay process is run in background and is paused when one of the specified breakpoints is encountered as shown in Figure A-13.

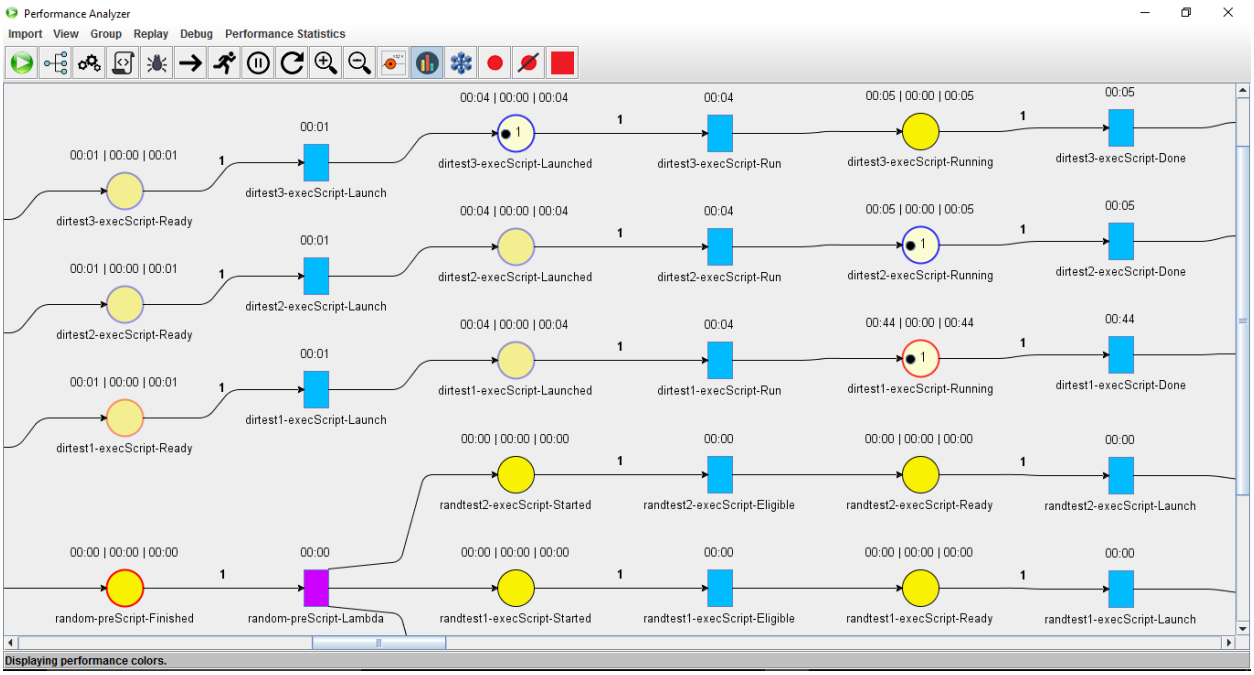

Figure A-13: Debug Mode with Replay

24. To terminate debug mode, click on the **Clear breakpoints** button from the toolbar.
25. To zoom in the model at any region, click on the **Zoom In** button from the toolbar.
26. To zoom out the model at any region, click on the **Zoom Out** button from the toolbar.
27. To exit the application at any time, select the **Terminate** button from the toolbar.