2001

Scenario-based verification and validation of dynamic UML specifications

Alaa El-Sayed Ibrahim
West Virginia University

Scenario-based Verification and Validation of Dynamic UML Specifications

Alaa E. Ibrahim

Thesis Submitted to the College of Engineering and Mineral Resources at West Virginia University in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Hany H. Ammar, Ph.D., Chair
Ali Mili, Ph.D.
Vittorio Cortellessa, Ph.D.

Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2001

Keywords: Verification and Validation, UML, Simulation, Automated, Timing Analysis, Timing Constraints, Risk assessment, Performance Modeling, Fault Injection

ABSTRACT

Scenario-based Verification and Validation of UML Dynamic Specifications

Alaa E. Ibrahim

The Unified Modeling Language (UML) is the result of the unification process of earlier object oriented models and notations. Verification and validation (V&V) tasks, as applied to UML specifications, enable early detection of analysis and design flaws prior to implementation. In this work, we address four V&V analysis methods for UML dynamic specifications, namely: Timing analysis and automatic V&V of timing constraints, automated Architectural-level Risk assessment, Performance Modeling and Fault Injection analysis. For each we present: approaches, methods and/or automated techniques. We use two case studies: a Cardiac Pacemaker and a simplified Automatic Teller Machine (ATM) banking subsystem, for illustrating the developed techniques.

ACKNOWLEDGMENTS

I wish to express my deep gratitude to my research advisor, Dr. Hany Ammar for helping me define my research goals and for providing valuable guidance during this research.

I would also like to thank the members of my committee, Dr. Ali Mili and Dr. Vittorio Cortellessa for their support and review and for their time in serving on my committee.

I would like also to express gratitude to the project team at Averstar Group, especially to Dr. Jim Dabney and Dr. Khalid Lateef for their directions, support and encouragement.

Many thanks to Dr. Sherif Yacoub, who invested from his time, knowledge and effort in this work. He has been an unfailing source of support and encouragement to me during all the research period.

Special thanks to Dr. Vittorio Cortellessa, for his patience, teaching and efforts in sharing his wide knowledge in Performance Modeling.

I also thank my parents for always motivating me to pursue higher education and to expand my scientific knowledge. I offer my sisters and friends heartfelt thanks for their invaluable consideration and moral support.

I am also grateful to all my colleagues in the research lab. Thank you for your help and for creating a very positive atmosphere that makes it easier to withstand the difficulties that sometimes arise.

This work was funded by the Averstar Group research grant to West Virginia University through the Software Engineering Research Center (SERC).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

The Unified Modeling Language (UML) is becoming a widely accepted standard notation for modeling software systems. The software development industry is embracing this modeling language for requirement analysis and the subsequent phases of software development lifecycle. Its success mostly relies on few elementary characteristics: different diagrams are provided (in an integrated framework) to represent the software model from different viewpoints, so explicitly specifying software aspects elsewhere hidden; the language is supported by a graphical representation, easy to use, that is not far from the classical diagrams used before introducing UML (e.g., State Diagrams, Class Diagrams, Sequence Diagrams); no standard software development process is coupled to the notation, thus software designers may decide to use whatever subset of diagrams that can better fit their application requirements, and organize an application oriented software process. As a result of the rapid success, Verification and Validation (V&V) teams need to devise methods for evaluating UML artifacts. V&V analysis can be categorized as static or dynamic. Static analysis helps V&V teams in reviewing the structure of UML models and generating metrics such as class size, the size of the hierarchy and static complexity measures. The complex dynamic behavior of many applications, especially real-time applications, motivates a shift in interest from traditional static analysis to dynamic analysis. Dynamic analysis is performed to analyze the behavior of objects as expected at run time.

1.1 Background

UML was explicitly born as an "open" project [17], with the potential of embedding additional notations and tools to satisfy specific design requisites. Along this trace, Rational Software [21](the UML originator) and ObjecTime Limited [16](the Real-Time Object Oriented Modeling "ROOM" originator) collaborated in defining UML for Real-Time [11,25] (UML-RT), an extension of UML optimized for real-time embedded software development. ROOM was introduced to study the dynamic aspects of applications modeled as concurrently executing objects with complex dynamic behavior. ROOM models are intended for simulating the application execution scenarios and complex object behavior. UML specification provides a State Machine package as a sub package of the behavioral elements package. UML state machines formalism is a

variant of Harel Statecharts and it incorporates several ROOMcharts concepts and ROOMcharts are a valiant of ROOM modeling language [30]. Dynamic analysis can be conducted on executable design models using several tools, such as Rational Rose Real-Time (RRT) from Rational Software Inc. and ObjecTime Developer from ObjecTime Inc., and hence the dynamic behavior of applications can be verified and assessed.

1.2 Problem Statement

V&V can be conducted at various development phases. Early V&V of software specification and analysis artifacts is encouraged before large investment is made in development. V&V of UML specifications can be done at an early development phase - prior to implementation - using scenarios, requirements and simulation models. Although UML is a rich analysis and design modeling language, it does not define how to study the dynamic aspects of the models through simulation, a capability that is required to monitor and assess the expected run-time behavior of software systems. V&V teams being much smaller than development teams must use efficient techniques to perform their analysis. At present mostly manual methods are being used to analyze UML models. Given the size and complexity of the large software systems, the manual efforts are time-consuming, tedious and error prone. Therefore automated techniques for V&V of UML models need to be developed.

1.3 Research Objectives

In this work, techniques are developed to help V&V teams in performing there task in the early development stages of UML dynamic specifications. We develop methods and approaches. We extend tool support for fast and automatic deployment of the developed techniques. Four areas are investigated in this thesis:

1. Developing automated techniques and methods for the V&V of the temporal characteristics of software systems (more importantly Real-Time software systems). Temporal V&V and timing analysis are not part of UML specifications, thus studying the conformance of the UML model with the timing constraints specified in the requirements is needed.

2. The automated generation of software metrics for ordering the components, connectors and subsystems, based on well defined metrics is needed. This will help in allocating the resources during the next development phases and in assessing the software quality. Dynamic component complexity and connector coupling metrics developed in [35] and the Architectural-Risk assessment methodology developed in [33] are selected for this purpose.

3. Optimizing the number test-case scenarios required for software testing, and assessing component severity are the motives behind the third area of investigation where we develop and assess a fault model for fault injection analysis.

4. Studying the performance of software systems, where queuing networks that model the performance characteristics of software systems have been well investigated. Interest in performance modeling for UML specifications has gained an increasing acceptance in industry standard. In [2] UML sequence diagrams where used as the starting point for performance model generation. In this study we aim to utilize the simulation capabilities in studying the performance characteristics of UML-RT models through resource modeling.

1.4 Thesis Structure

Considering the four areas of investigation mentioned above and two case studies, we structure this thesis as follows (figure 1.4). Chapter 2 introduces our simulation environment and the tool extensions developed and chapter 3 presents the first case study: the software model of a Cardiac Pacemaker device. Chapter 4 discusses automated temporal V&V techniques. Chapter 5 discusses the automatic extraction of dynamic metrics and architectural-level risk. Chapter 6 presents techniques for fault injection analysis. Chapter 7 discusses performance modeling based on UML dynamic specifications in simulation environments (the fourth area of investigation) and we use a simple abstraction of the software of the Automatic Teller Machine (ATM) banking subsystem. Finally we conclude and discuss potential areas for future work.

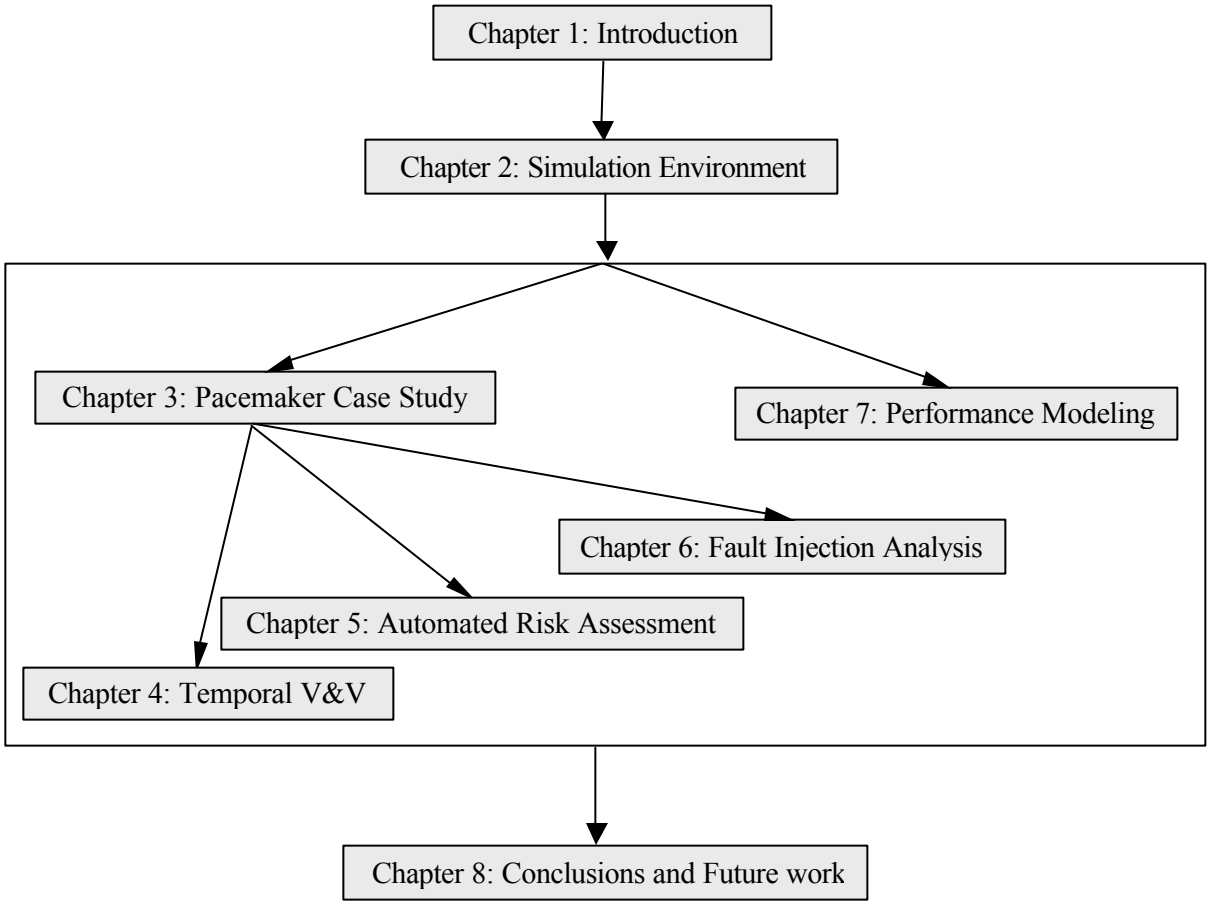

Figure 1.1 Flow chart of the thesis chapters

CHAPTER 2: SIMULATION ENVIRONMENT

Our general approach for V&V of UML models is based on simulating the dynamic specifications. Figure 2.4 shows an overall view of our environment in which we developed methods and techniques to perform the required tasks. The simulation settings for a particular scenario are adjusted by the analyst and the UML model is executed in a given simulation environment to produce simulation logs for that particular scenario. We generate the timing diagram from processing the simulation log files. The generated timing diagrams are inspected visually to determine and assess the correctness of the developed methods and techniques, and to analyze the logic behind our findings. Elements in our environment are:

1. Rational Rose Real-Time 6.0 [22] RRT as the modeling and simulation tool.
2. Simulation log files and the log analysis tool that is composed of Microsoft Excel and Visual Basic Scripts that were develop.
3. The timing diagrams are charts showing each object as a series of changes in its states versus time.

2.1 UML-RT modeling and simulation tool

In [25] the derivation of the set of architectural constructs that integrate ROOM notation in UML were presented. These architectural constructs are derived from general UML modeling concepts using UML extensibility mechanisms. Table 2.1 provides a summary for these extensions, as a brief description of the basic constructs used in modeling the system structure and component behavior. Three principal constructs; Capsules, Ports and Connectors, are used to explicitly describe the system structure. In a Capsule collaboration diagram, Capsules and Ports are stereotype roles, and Connectors are association roles. Behavior is modeled using Protocols and state machines. A Protocol specifies the desired behavior over a connector and compromises a set of participants, each participant plays a specific ProtocolRole. A Protocol state machine specifies valid communication sequence and is the standard UML state machine. Capsule behavior is defined in UML state machine where the stereotype (ChainState) is a state that is used in case of

transitions that are split into a transition that terminates on the boundary of the state and a transition that propagated into the state (in case of hierarchical state machines).

Metamodel Class	Stereotype
Collaboration	Protocol
ClassifierRole	ProtocolRole
Class	Port
Class	Capsule
State	ChainState

Table 2.1 Summary of UML Extensions for ROOM, source [25]

Figure 2.1 shows a Capsule named *Top_Level_Capsule* and its Structure Diagram. The Structure Diagram of *Top_Level_Capsule* contains two Capsules: *First_Capsule* and *Second_Capsule,* each with one port named *Port_1*. *Port_1* in *First_Capsule* is assigned a ProtocolRole *Protocol_1* and *Port_1* in *Second_Capsule* is assigned a ProtocolRole *Protocol_1~*, which is the conjugate of *Protocol_1*. As mentioned earlier a Protocol defines the flow of messages between ports. Messages are categorized into incoming and outgoing messages. In a conjugated Port the messages defined in the Protocol as incoming messages are defined as outgoing in the ProtocolRole assigned to the Port, and like wise the outgoing messages are defined as incoming messages in the ProtocolRole assigned to the Port. A connector connects the two ports and works as a media for message delivery.

Figure 2.2 shows the State Diagram of *Second_Capsule. Second_Capsule* has two states *S_1* and *S_2,* and two transition; *t_top* and the initial transition that defines the initial state. *S_1* is a macro state that can be expanded into another State Diagram shown in figure 2.3. S_1 has two states and three transition, *t_1*, *t_2* and the initial transition. *t_2* is a transition top a ChainState. Each transition is configured with a message that defines its firing conditions, except transitions from ChainStates like *t_top*.

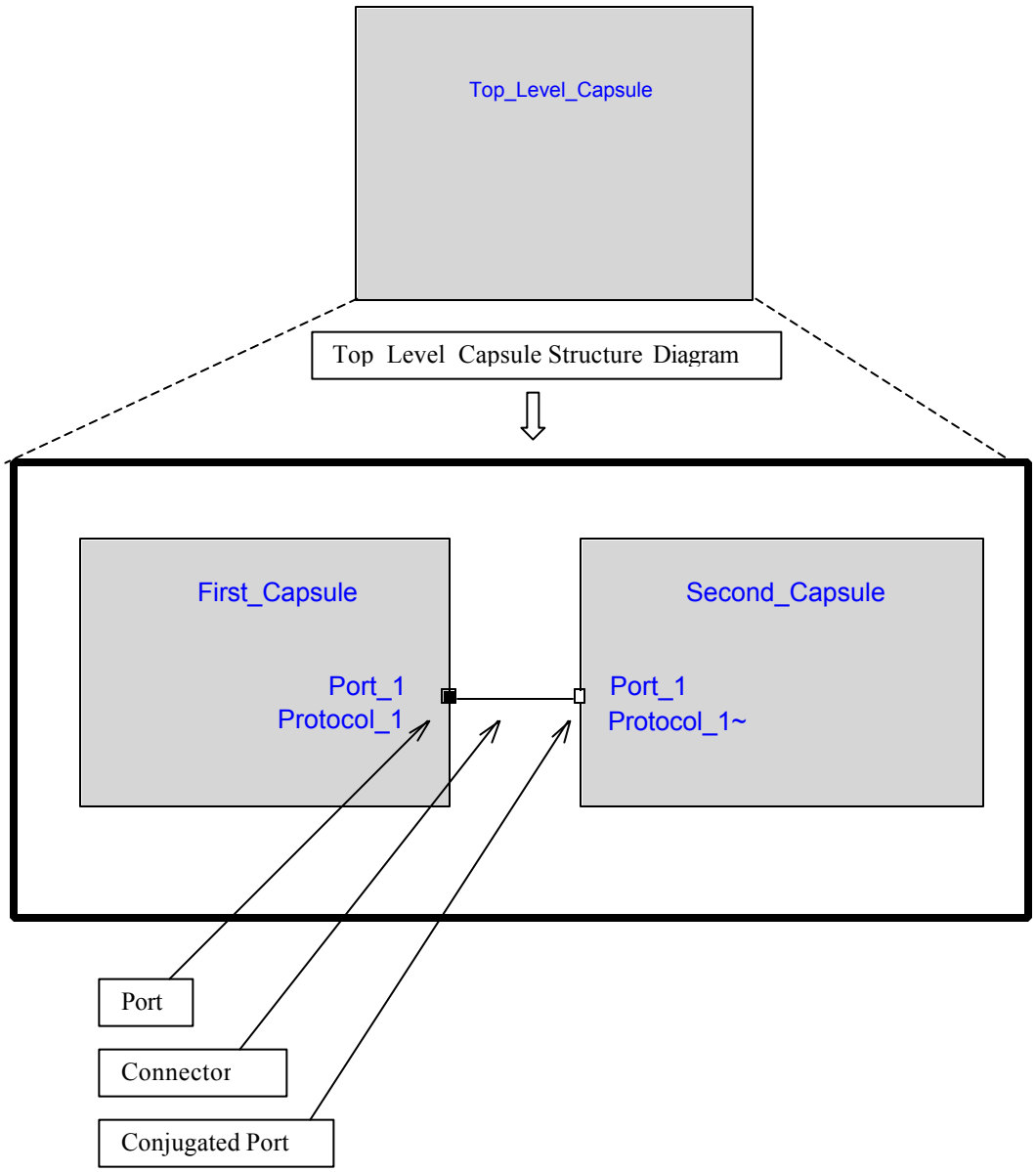

Figure 2.1 A Capsule (*Top_Level_Capsule*) and its Structure Diagram

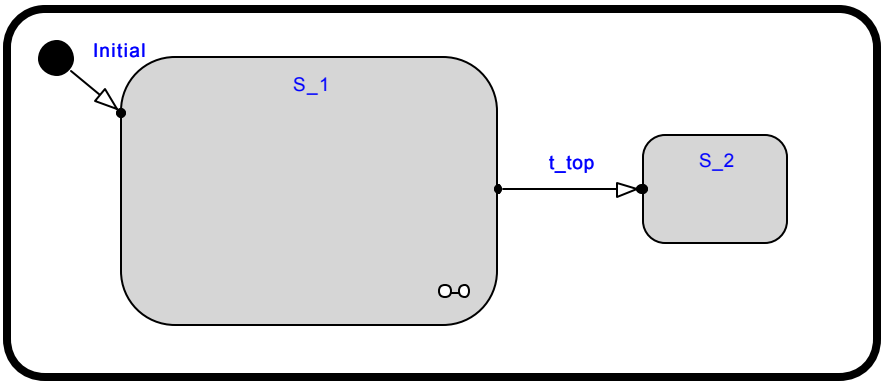

Figure 2.2 State Diagram of *First_Capsule* (top level)

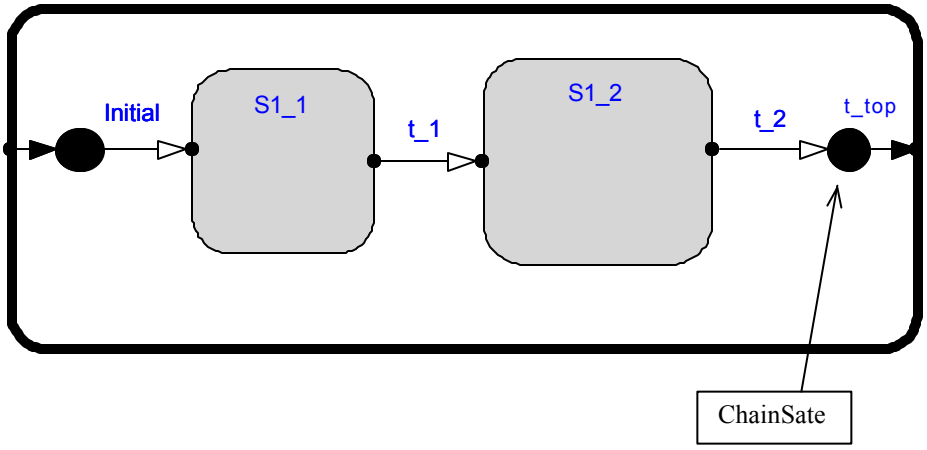

Figure 2.3 State Diagram of the macro state *S_1*

A typical early model of a software product is known as the software architecture, that is essentially a graph whose nodes represent software components and arcs represent software connectors. In order to provide to a software architecture the potential to represent the same software at different levels of detail, it can be hierarchically structured. In other words, a component can be detailed by describing its internal structure of subcomponents and connectors, while unvarying its external structure consisting of connectors with other components.

UML notation does not explicitly provide a diagram to describe a software architecture, which is in fact not necessary. The RRT tool allows building a diagram of components and connectors, where each component is represented by a Capsule and its Ports as interfaces to which Connectors are associated to exchange messages with other Capsules. The suitable hierarchical structure that such a software architecture should have is also provided, by allowing to detail the internal structure of a Capsule with other Capsules and Connectors.

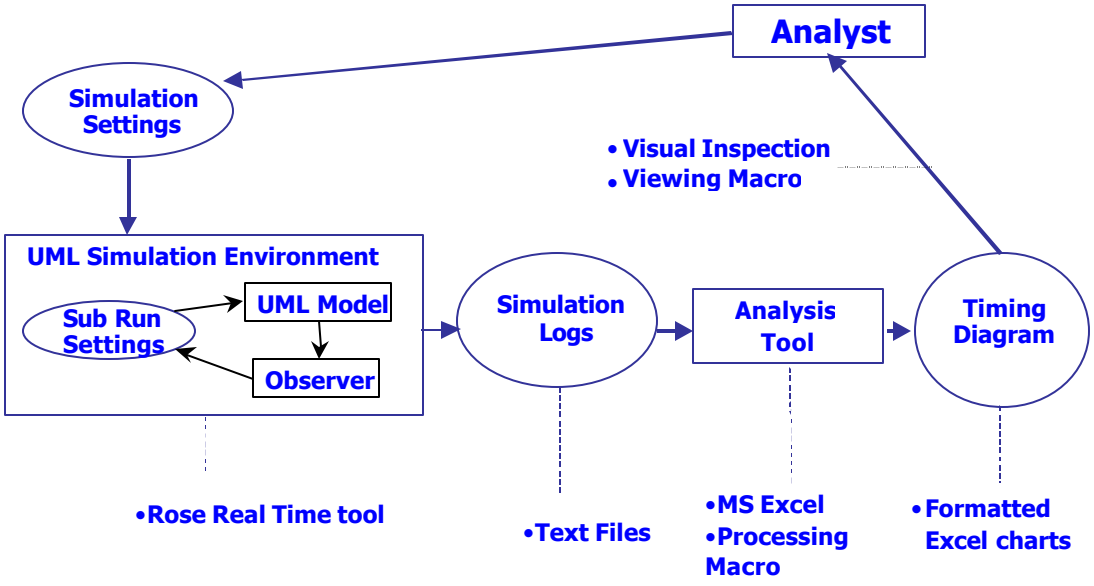

Figure 2.4 Environmental overall view

The simulative nature of this tool requires as a minimum, in order to run such a scheme, a dynamic description of the behavior of each Capsule belonging to the lowest levels of the hierarchy, that is each Capsule that does not contain other Capsules. This dynamic (behavioral) description is represented in the Capsules State Diagrams as part of the UML specifications.

Figure 2.4 shows an overview of our simulation environment, RRT as the main tool and Visual Basic Scripts running from within Microsoft Excel as tool extensions.

2.2 Log files

The log files are two text files. The first (state log file) contains an entry for each state change in each component during a simulation run, where each entry is composed of: the simulation time of the entry, the object and the new state. The second (message log file) contains an entry for each message sent in the system during a simulation run, where each entry is composed of: the send time of the message, the source object, the destination object and the message name.

2.3 Timing Diagrams

Figure 2.4 shows a sample-timing diagram from the Cardiac Pacemaker case study that will be presented later in chapter 3. The x-axis is a time series of 1 milisec with labels every 100 milisec and on the y-axis are the states of three objects. The first object named "Heart" has two states: Pulse and Waiting, the second and third objects named "Ventricle" and "Atrial" respectively each has three states: Pacing, Waiting and Refractory. For each object a series of the state changes is plotted on the timing diagram. The fields "Graph Start" and "Graph End" are used by the viewing macro to define the starting and ending values of the x-axis, which corresponds to the window of time, in a single simulation run, to be displayed.

For automatic generation of timing diagrams from simulation logs, two Visual Basic macros were developed, Processing macro and Viewing macro, within Microsoft Excel environment. First, the processing macro, which recognizes all executed objects and all their involved states, generates numeric distinct codes for all involved states in each object, adjusts values to enforce continuous vertical and horizontal line representation of state changes, configures x-axis as a time series of milliseconds, y-axis as state codes, and each object as a series, and automatically generates an Excel chart for each simulation run. Appendix A shows the Processing macro as a subroutine named "Processing_Macro()" in Visual Basic Script. Bellow we show the steps followed by the Processing macro in processing the log file.

1. Extract all the Capsules "Objects" in the log file.
2. Extract the Object names and their states coded in continuous numeric state codes. i.e. For each Object: extract all states and generate a consecutive state code for each
3. For each Object: use the state codes to generate an eleven columns log table with time as the first column and the rest as the states in state code.

4. Create continuous lines (horizontal and vertical) from the ten fragmented series representing the state changes (in state codes) of the ten Objects.
5. Size the chart and force the start to be 0 milisec and the end to be 20000 milisec.

The second macro is the viewing macro, which enables the analyst to zoom in and out of the timing diagram and adjust the window of time to be viewed. Appendix A shows the Viewing macro as a subroutine named "Viewing_Macro()" in Visual Basic Script. The basic function is to resize the chart (figure 2.5) based on the start, end and step fields.

Graph Start:	5000	Graph End:	7500	X axis label Step:	100				
Graph Name:	Concurrency based analysis. All messages between the Atrial and the Ventricular models are delayed by 10 epochs								
Series 1:	heHeart	Series 2:	VENTRICULAR_MODEL	Series 3:	ATRIAL_MODEL				
Comments:	In case of more than one unsensed consecutive heart beets, the next heart beet overlaps with the generated paces. The Pacing, Pacing timeout and the Refractory timeout messages are each delayed from expected by 10 epochs thus the waiting state starts delayed by at least 30 epochs. As well it was noticed in other delay values that queuing of messages occurs for delays larger than 20 epochs.								

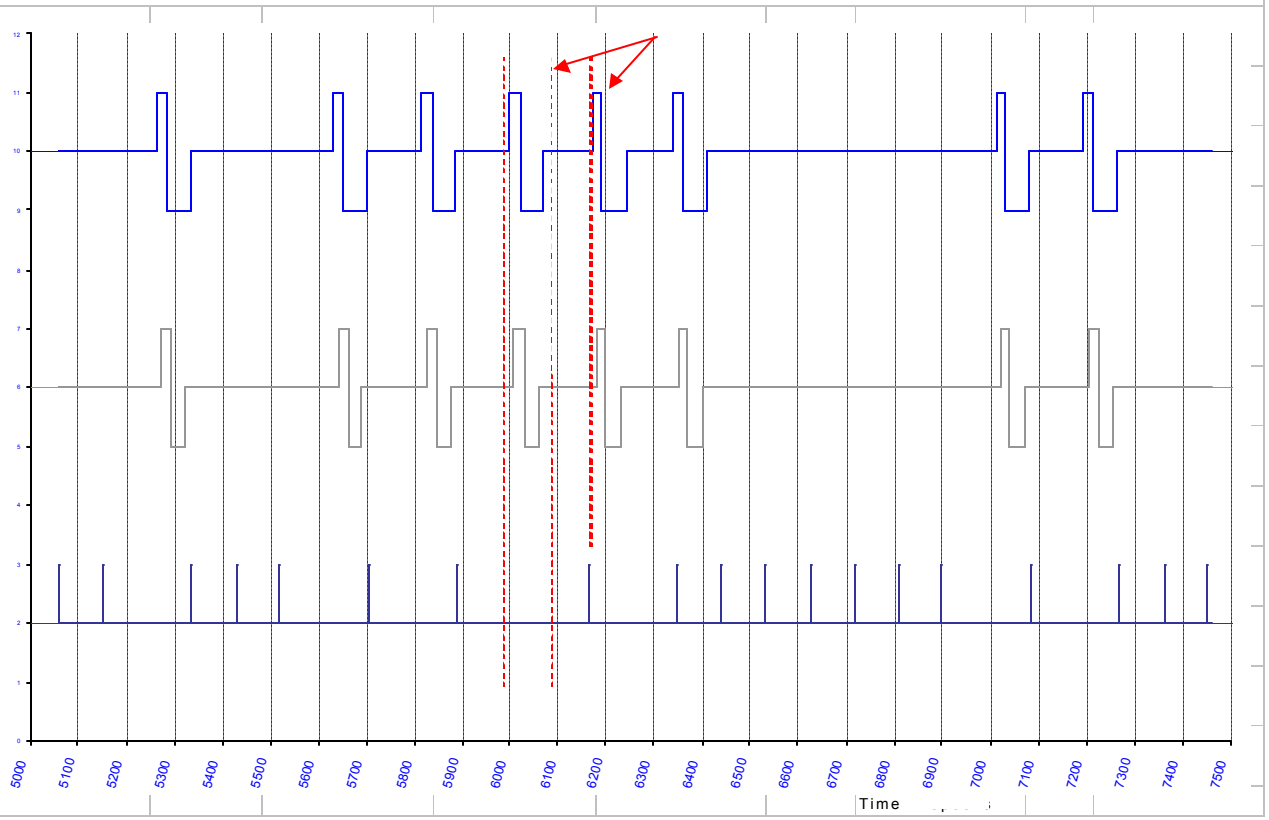

Figure 2.5 A sample-timing diagram

CHAPTER 3: THE CARDIAC PACEMAKER CASE STUDY

We have selected as a case study a cardiac pacemaker (Pacemaker) device [4, pp177] to discuss the applicability of the proposed approaches and methods. The pacemaker is a critical real-time application. An error in the software operation of the device can cause loss of the patient's life. Therefore, it is necessary to model its design in an executable form to validate its temporal behavior. We have used RRT simulation environment [22] and dynamic UML specifications [30] to model and gather simulation statistics.

A cardiac pacemaker is an implanted device that assists cardiac functions when the underlying pathologies make the intrinsic heartbeats low. The pacemaker runs in either a programming mode or in one of operational modes. During programming, the programmer specifies the type of the operation mode in which the device will work. The operation mode depends on whether the Atrium, Ventricle, or both are being monitored or paced. The programmer also specifies whether the pacing is inhibited (I) or triggered (T). For the purpose of this paper, we limit our discussion to the AVI operation mode. In this mode, the Atrial portion of the heart is paced (shocked), the Ventricular portion of the heart is sensed (monitored), and the Atrium is only paced when a Ventricular sense does not occur; i.e., inhibited (I). Figure 3.1 shows (a) the system structure diagram of the external components and the pacemaker design model. The external components are modeled for simulation purposes. In the pacemaker example the Programming device (DoctorsProgrammer) is used to configure the pacemaker's operational mode. Therefore it appears as one of the components interacting with the pacemaker components in the Programming scenario only, whereas the heart is represented by the PatientsHeart component and is interacting with the pacemaker in all the operational modes. The Observer component shown in figure 3.1 (a) is the external monitoring component that we discuss in chapter 4. The pacemaker consists of the following components: (shown in figure 3.1 (b))

Reed_Switch: A magnetically activated switch that must be closed before programming the device. The switch is used to avoid accidental programming by electric noise.

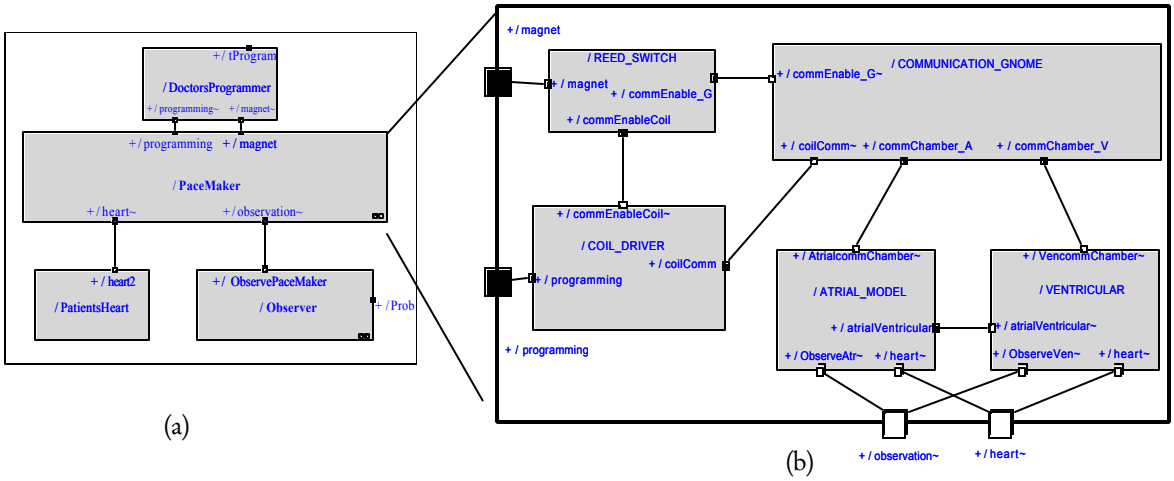

Figure 3.1 Structure diagram for the Pacemaker.

(a) Pacemaker and all external Capsules (context level).

(b) Pacemaker internal Structure Diagram

Coil_Driver: Receives/sends pulses from/to the DoctorsProgrammer. These pulses are counted and then interpreted as a bit of value zero or one. These bits are then grouped into bytes and sent to the communication gnome. Positive and negative acknowledgments as well as programming bits are sent back to the programmer to confirm whether the device has been correctly programmed and the commands are validated.

Communication_Gnome: Receives bytes from the coil driver, verifies these bytes as commands, and sends the commands to the Ventricular and Atrial models. It sends the positive and negative acknowledgments to the coil driver to verify command processing.

Ventricular_Model and Atrial_Model: These two actors are similar in operation. They both could pace the heart and/or sense heartbeats. The AVI mode is a complicated mode, as it requires coordination between the Atrial and Ventricular models. Once the Pacemaker is programmed the magnet is removed from the Reed_Switch. The Atrial_Model and Ventricular_Model communicate together without further intervention. Only battery decay or some medical maintenance reasons force reprogramming.

A hierarchical UML state machine models the behavior of each component. As mentioned earlier, a pacemaker can be programmed to operate in one of several modes depending on which part of the heart is to be sensed and which part is to be paced. The analysis of the device operation defines six scenarios. Figure 3.2 show the main Use Case diagram and all the relationships among the six Use Cases and the two actors, DoctorsProgrammer and PatientsHeart. Each scenario, in the pacemaker, maps to a Use Case, one for the programming scenario and five for the operational modes. The AAI operational scenario: in which the Ventricular_Model is Idle and the Atrial_Model is sensing and pacing the heart when a heartbeat is not sensed. The AAT operational scenario: in which the Ventricular_Model is Idle and the Atrial_Model is sensing and pacing the heart when a heartbeat is not sensed. The VVI operational scenario: in which the Atrial_Model is Idle and the Ventricular_Model is sensing and pacing the heart when a heartbeat is not sensed. The VVT operational scenario: in which the Atrial_Model is Idle and the Ventricular_Model is sensing and pacing the heart when a heartbeat is sensed or not. We only use the AVI Operational scenario: in which the Ventricular_Model senses the heart and the Atrial_Model paces the heart when a heart beat is not sensed. In all scenarios a refractory period is then in effect after every pace.

Currently UML representation of timing constraints [30] is limited to construction marks on sequence diagrams (common in blueprints), labels, and message transmission and reception on sequence diagrams. We compose the AVI timing constraints from: elements representing the time of a message transmission and reception; elements mapping to the time of entry of a state are represented by the reception of the message that fired the transition. We applied our approaches in chapter 4 to the following two timing constraints of the AVI operational scenario.

The first timing constraint is on the paces generated by the pacemaker in response to unsensed heart pulses. The time to each pace corresponding to an unsensed pulse should be less than 350 milisec.

$$\forall s_i \exists p_j \mid T(p_j)-T(s_i) < \varepsilon \text{ and } s_i \in S \text{ and } p_j \in P$$

where S is the set of all unsensed heart beats observed during a simulation run, S={s1, s2, .., sn}, P is the set of all paces generated by the pacemaker to the heart during a simulation run, P={p1, p2, .. , pm} and ε is the maximum permissible delay of pacing after a heart beat is not sensed and is equal to 350milisec. Figure 3.3 shows two cases: in the first ε was not exceeded, while in the second it was exceeded and the result was Pacing the patients heart while a pulse is naturally in place.

The second timing constraint is on the refractory period, the time in which the pacemaker stays idle after every pace. The Atrial_Model refractory time represents this period and is controlled by the Ventricular_Model refractory state which intern is controlled by the Ventricular_Model refractory timer. The Atrial_Model refractory time should be less than 350milisec.

$$\forall i_i \exists o_j \mid T(o_j)-T(i_i) < \varepsilon \text{ and } i_i \in I \text{ and } o_j \in O$$

where I is the set of all transitions from the Pace state to the Refractory state in the Atrial_Model, I = {i1, i2, .., in}, O is the set of all transitions from the Refractory state to the Waiting state in the Atrial_Model, O = {o1, o2, .. , om}, and ε is the maximum permissible refractory time for the Atrial_Model and is equal to 350milisec.

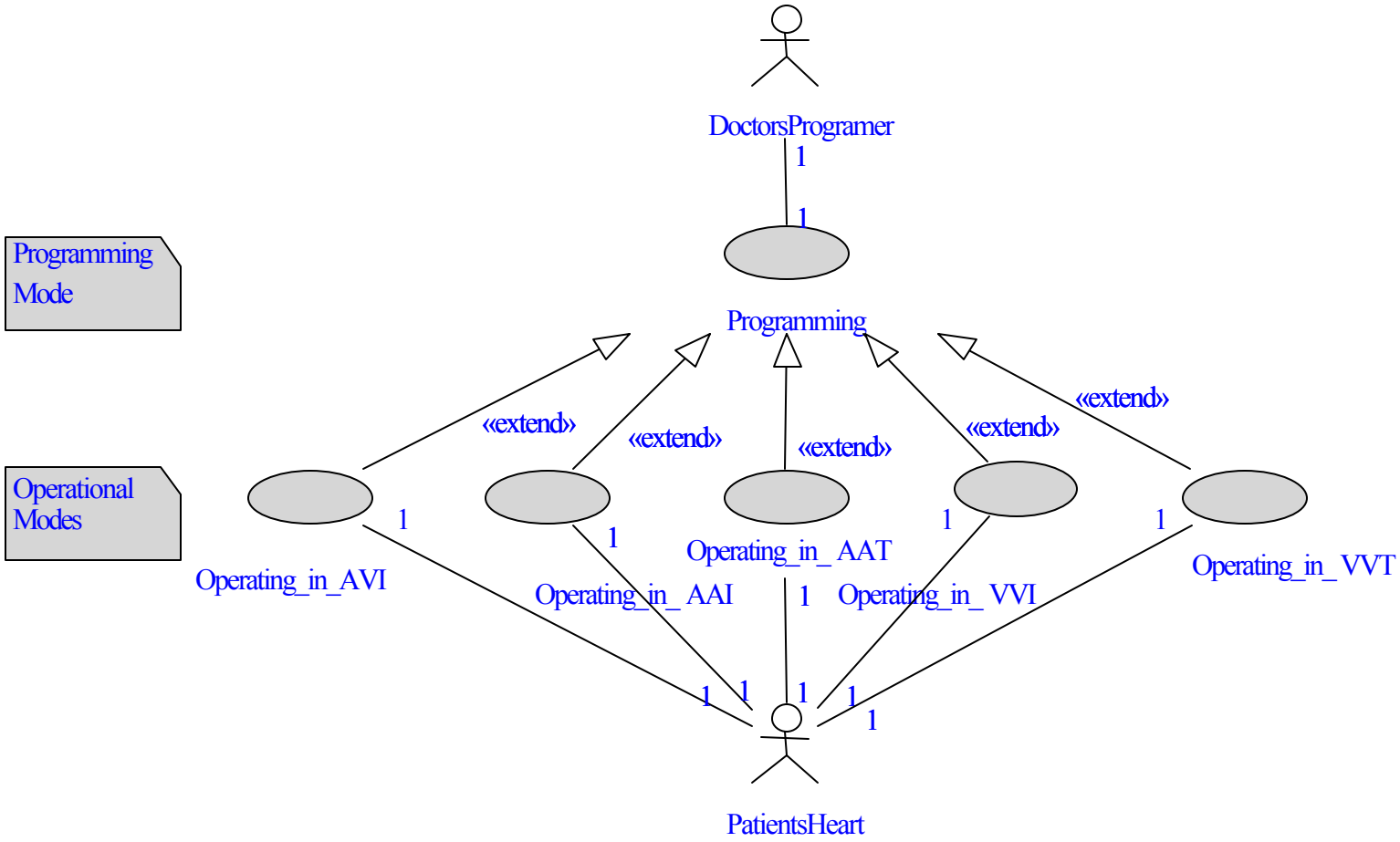

Figure 3.2 Main Use Case Diagram

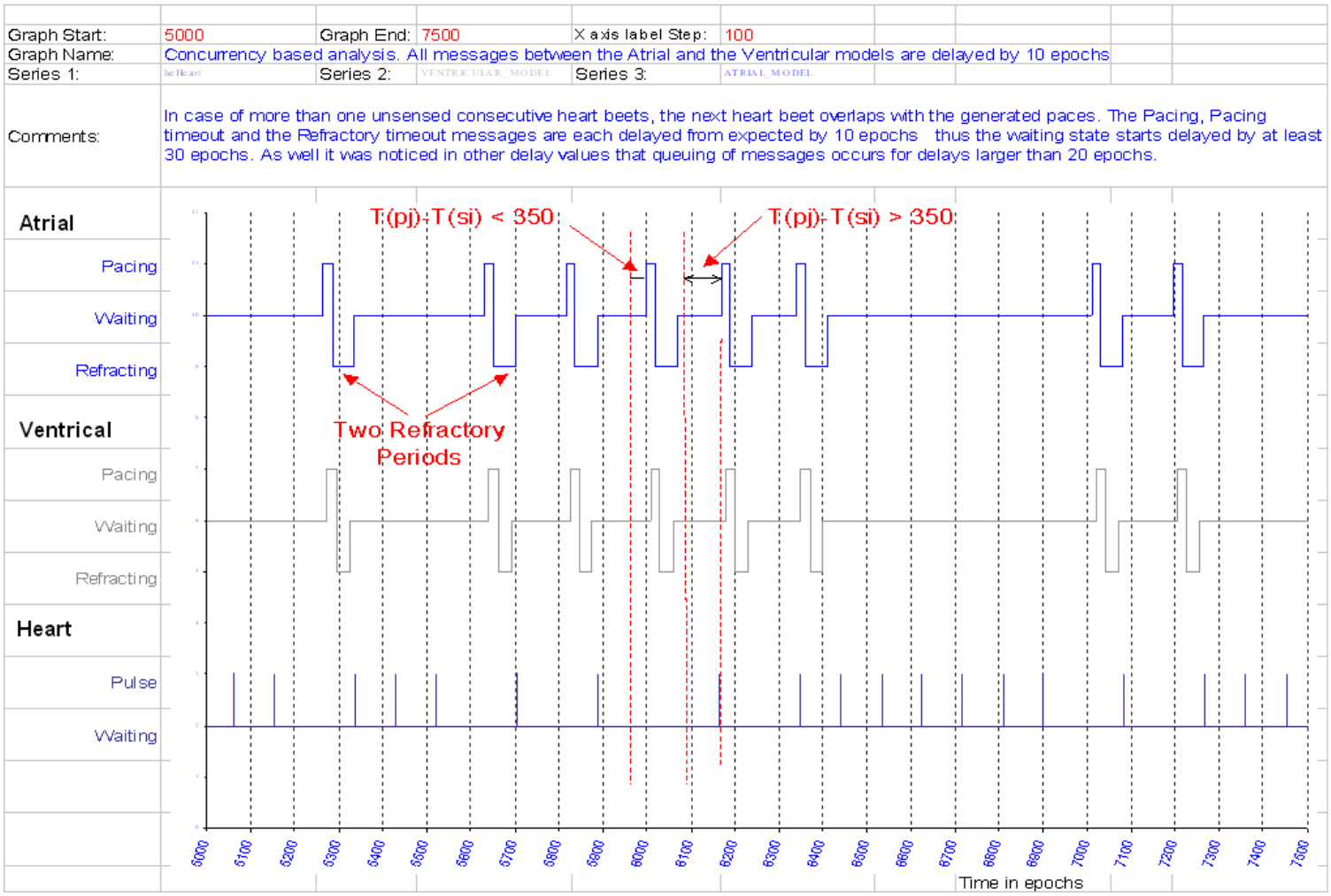

Figure 3.3 A sample-timing diagram illustrating the timing constraints

CHAPTER 4: TEMPORAL V&V

Capitalizing on the simulation environment (commercial tool "RRT, Microsoft Excel and Visual Basic Scripts" and the tool extensions "logging and automatic generated Timing Diagrams") described in chapter 2, the V&V analyst can inspect the timing diagrams to verify that the timing constraints are met. Moreover, two approaches for automatic V&V of timing constraints [8] are presented in this chapter, together with the results and the lessons learned, using the Pacemaker case study presented in chapter 3. As well as four timing analysis methods, and their deployment procedure to UML artifacts [34] are presented, together with samples of the results from the Pacemaker example.

4.1 Automated V&V of Timing Constraints

The first approach is based of processing the simulation log files in search of constraint violations. While the second approach is based on an Observer component, modeled as an external entity to the modeled system and acting as a monitoring device. Hence two methods for modeling the timing constraints in the Observer Component, namely: Constraint driven and Use Case driven, are developed. The output in both approaches is a violation table, table 4.1 is a sample of a violation table. Figure 4.1 shows a high level view (process/product view) of the Automated Timing Constraints V&V process.

4.1.1 The first approach for Automatic timing constraints verification

In this approach the violation algorithm shown below processes the message log file. The product is the violation table which is a list of violations and their time of occurrence in the simulation run. The violation algorithm consumes the message log file and the timing constraints. Each entry in the message log file contains the time of message occurrence, the message name, and the type of

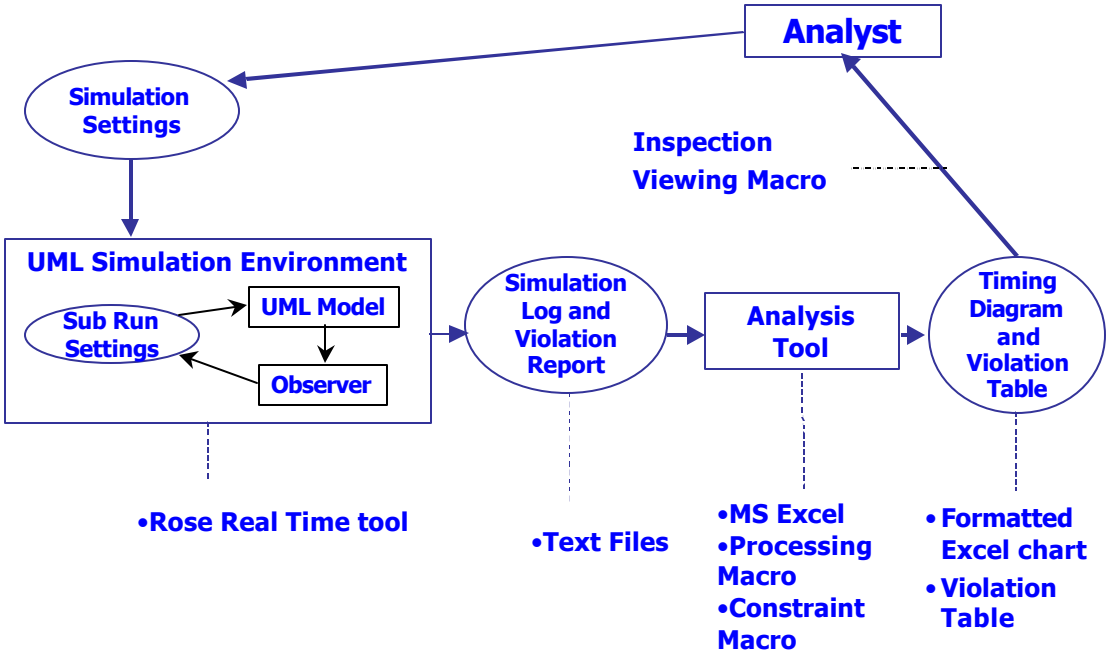

Figure 4.1 High level view of the Automated Timing Constrains V&V process

occurrence (receive_by or send_by). The timing constraints are in the form of Boolean expressions containing elements that correspond to the time of the transmission or reception of a message and a constant to which the evaluated expression is compared. The timing constraints are coded in the algorithm in the form of a two dimensional array where each row represents one timing constraint and contains: the constant time value to which the expression is compared, the total number of elements in the expression, the set of elements which represent the time of a message occurrence ordered by their expected occurrence, the set of corresponding occurrence types (transmission or reception), and the set of operators acting on the corresponding element including the Boolean operator as the last operator. Examples are shown below.

For each constraint the algorithm scans the message log file and searches for the elements in order. For each element detected, the corresponding operator is applied on the temporary variable temp_time and the element. The Boolean expression is evaluated after the last element is detected

and processed, and if it evaluates to false, an entry in the violation table is recorded in the form of the time and type of the violation.

Algorithm

Procedure Violation

Parameters

Consumes: $log\_file\_entry_i(time, message, occurrence)$, *where* $0 < i < end\_of\_log\_file$
$timing\_constrain_k(constant, no\_of\_elements, element_h, occurrence_h, operators_h(first\_operand, second\_operand))$, *where* $0 < k <= total\_no\_of\_timing\_constrains$ *and* $0 < h <= no\_of\_elements$

Produces: *Violation_Table(Stack[time,constrainID])*

Initialization:

$i = k = h = 1$

temp_time = 0

Algorithm

while $k <= total\_no\_of\_timing\_constrains$ do
  $i = h = j = 1$
  while $h < no\_of\_elements$ do
    $i = j$
    while $i < end\_of\_log\_file$ do
      if $log\_file\_entry_i.message = timing\_constraint_k.element_h$ AND
        $log\_file\_entry_i.occurrence = timing\_constraint_k.occurrence$
        $timing\_constraint_k.operator_h(temp\_time, log\_file\_entry_i.time)$
        $j = i$
        next h
      end if
      if $h = no\_of\_elements$ AND $timing\_constraint_k.operator_h(temp\_time, log\_file\_entry_i.time) = False$
        push $(log\_file\_entry_i.time, k)$
      end if

```
        next i
      end while
    end while
    next k
  end while
end Procedure Violation
```

The pacemaker constraints were composed and fed to the above algorithm and the log file, generated for a faulty simulation run in which the waiting time was increased by 50 milisec to be 1050 milisec, was processed. The parameters consumed by the algorithm are:

The two timing constraints:

- $timing\_constrain_1$(constant = 350 milisec, no_of_elements = 2,
[$element_1$ = Pace, $element_2$ = Unsensed] ,
[$occurrence_1$ = Receive_heart, $occurrence_2$ = Send_heart],
[$operators_1$(first_operand, second_operand) = "-", $operators_2$(first_operand, second_operand) = "<"])

- $timing\_constrain_2$(constant = 350 milisec, no_of_elements = 2,
[$element_1$ = APaceDone, $element_2$ = VRefractDone],
[$occurrence_1$ = Receive_Atrial, $occurrence_2$ = Receive_Atrial],
[$operators_1$(first_operand, second_operand) = "-", $operators_2$(first_operand, second_operand) = "<"])

Sample of the log file:

log_file_ $entry_{124}$ (time = 22152, message = APaceDone, occurrence = Send_Venticular)
log_file_ $entry_{125}$ (time = 22152, message = APaceDone, occurrence = Receive_Atrial)
log_file_ $entry_{126}$ (time = 22653, message = unsensed, occurrence = Send_heart)
log_file_ $entry_{127}$ (time = 23004, message = Pace, occurrence = Send_ Atrial)
log_file_ $entry_{128}$ (time = 23004, message = Pace, occurrence = Receive_Venticular)

log_file_ $entry_{129}$ (time = 23007, message = Pace, occurrence = Send_Venticular)
log_file_ $entry_{130}$ (time = 23007, message = Pace, occurrence = Receive_heart)

Results showed several violations in the first constraint:

Delayed Pace at: 10015
Delayed Pace at: 22653
Delayed Pace at: 23554
Delayed Pace at: 33469
Delayed Pace at: 34370
Delayed Pace at: 35271
Delayed Pace at: 45185
Delayed Pace at: 46087
Delayed Pace at: 58705

The drawbacks of this approach is in the fact that it resembles an open loop process, i.e. we can not stop the simulation nor change the simulation settings in response to a violation as it occurs, only when the whole simulation run is performed and the logs are available we can detect the violations and start understanding the logic behind them. This fact makes the approach less valuable to the purposes of timing analysis and the sensitivity analysis to a specific variable, delay or operation. This drawback is handled in the second approach discussed next.

4.1.2 The second approach for timing constraints verification

In this approach we designed the Observer component (figure 4.2) to act as an external monitoring object that monitors the timing constraints in the modeled system, and detects and reports all the violations as they occur. The Observer component is not part of the UML specifications nor of the tool used; it is aimed to automate the detection of timing constraints violations as they occur. The Observer responsibilities are: 1) Setting and initiating consecutive simulation runs 2) Detection of timing constraint violations 3) Production of the violation report. These violations represent detected deadline failures during the simulation run. The observer is modeled using UML hierarchical state machine based on timing constraints, use cases, sequence diagrams and the

methods presented in this section. One connector delivers the messages between the modeled system and the Observer. Messages from the system represent all the instances addressed in the timing constraints. There exist no messages from the Observer directly to the system. Several connectors can exist between the Observer and the modeled external systems. Messages from the Observer to the modeled external systems are control messages to initiate and terminate subruns. In [5], requirement verification for timed UML sequence diagrams and timed automata design representation (UML models have to be converted to timed automata), were accomplished by an Observer model within UPPAAL tool that was designed to verify timed automata requirements. The modeled observer branched to a state indicating a specific traceable timing failure, while in our model for the Observer, the reaction to a timing violation is configurable (the sub run can be forced to stop and the next can be consequently configured and started).

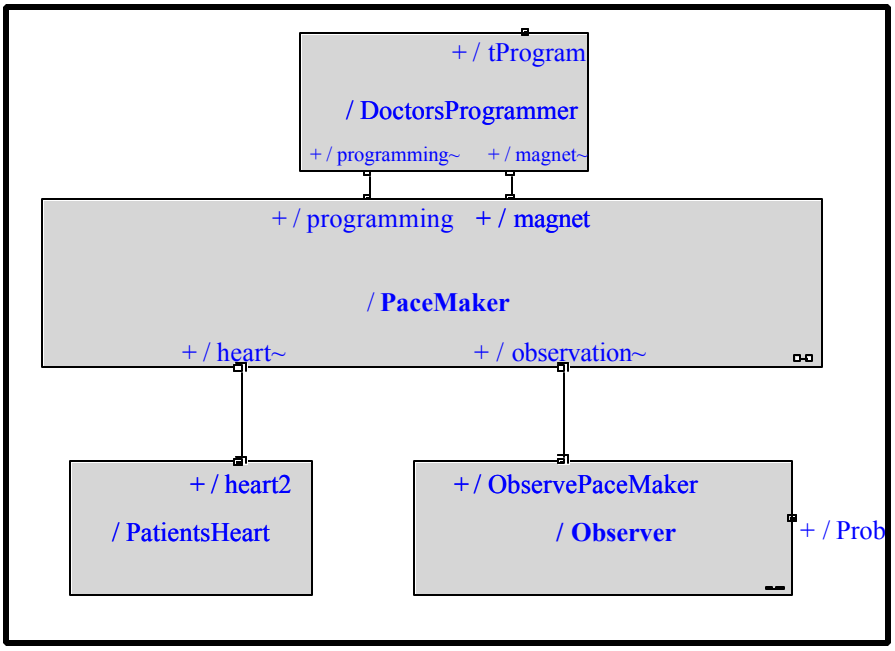

Figure 4.2 The Observer as an external object

4.1.2.1 Constraint driven Observer modeling

Our first proposal for Observer modeling is strictly based on timing constraints. Each constraint is modeled using a UML hierarchical State Machine representing the behavior of a subcomponent in the Observer component. The Observer component encapsulates all constraint components as well as an Observer controller component. The controller component is responsible for setting, initiating, terminating sub runs and controlling which set of constraint components is active at each specific time instance. The highest level of the constraint hierarchical State Machine consists of two states; on and off, and is controlled by the Observer controller component.

We modeled an Observer for the pacemaker based on the constraint driven Observer modeling and we confirmed the results with the timing diagrams. In this case the two pacemaker timing constraints mentioned in chapter 3 are modeled each in a separate component, namely: Constraint_1, Constraint_2. Figure 4.3 shows: (a) Observer component structure diagram for the pacemaker. (b) The state diagram representing the behavior of the Observer Controller (MicroObserverController in figure 4.3 (a)). (c,d) The first level of the state chart representing the behavior of constraint 2 and constraint 1 respectively. Two states are shown "Off" which is equivalent to idle and "On" which is expanded to a second level state machine, shown in (e,f), to represent the constraints.

One of the benefits of Modeling constraints in this manner is the ability to report a categorized violation of a constraint instead of just reporting the violation. This is obvious in the lower state diagram of the first constraint (figure 4.3). The violation of this constraint can imply one of two behavioral errors: a delayed pace or a skipped pace. Modeling the constraint as well as the types of violations, speeds up the analysis process performed by the analyst.

The drawbacks of the constraint driven Observer Modeling is the fact that the amount of effort spent by the analyst is directly proportional to the number of constraints modeled. This fact makes this method limited by the number of constraints to be studied. Our experience with the tool used in this work suggests that this method should only be used for a small number of timing constraints. Thus the number of components in the Observer Capsule is relatively small. This limitation is relaxed in the Use Case driven Observer modeling presented in the next subsection.

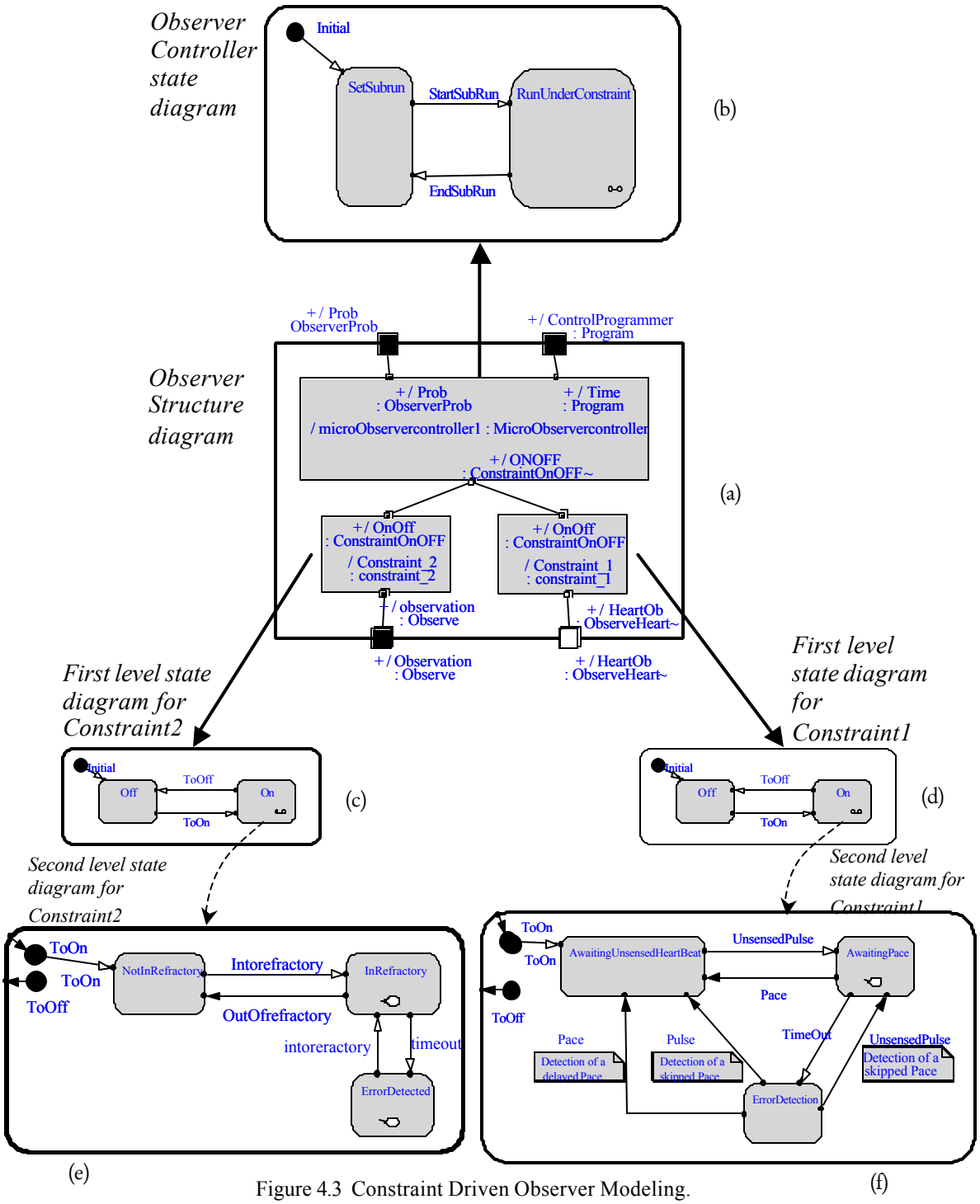

Figure 4.3 Constraint Driven Observer Modeling.
(a) Observer Structure Diagram. (b) Observer Controller State Diagram.
(c,d) First level State Diagram for Constrain 2 and Constraint 1 respectively.
(e,f) Second level State Diagram for constraint 2 and constraint 1 respectively

4.1.2.2 Use Case driven Observer modeling

Our second method for Observer modeling is based on timing constraints, use cases and sequence diagrams. In this method an adaptation of the structured sequence diagrams in [12], in which each scenario is represented as a Use Case composed of a set of Sequence Diagrams such that no loops or conditions exist within a sequence diagram, is used. This adaptation serves in mapping the sequence diagrams to state machines. In this representation the decision of the next sequence diagram is made based on the first message in the next sequence diagram. In this representation each Use Case has a set of Sequence Diagrams. For the purpose of timing constraints verification only the messages of the sequence diagram that affect one or more variables in the constraints to be verified as well as the messages on the edges of the Sequence Diagram, are mapped. Modeling the messages on the edge of the sequence diagrams is intended for modeling the messages on which the choice of the next sequence diagram is selected. Bellow we define FUC as a set of Use Cases, subset (filtered from UC) of the set of all use cases in the specification, FSD as a set of Sequence Diagrams, subset (filtered from SD) of the set of Sequence Diagrams in a Use Case that belongs to FUC and FM as the set of messages, subset (filtered from M) to the set of all messages in a sequence Diagram that belongs to FSD. In the following subsection we present the definitions of the sets mentioned above, how the filtration is performed and steps for the modeling process.

4.1.2.2.1 Definitions

The system requirements are expressed in a set of Use Cases named "UC" and each Use Case is named "UC_i" and contains a set of Sequence Diagrams named "SD_i". Each Sequence Diagram belonging to Use Case UC_i is named "SD_{ij}" and contains of a set of messages named M_{ij}. The set T is the set of timing constraints, each timing constraint is named t_l is composed of three sets: a set of messages named E_l , a set of operators named op_l and a set of constants named C_l . The sets UC , SD , M and T are presented below

$T = \{ t_l \mid 1 \le l \le n \} = \{t_1, t_2, \ldots, t_l, \ldots\ldots, t_n\}$ where $1 \le l \le n$

n is the total number of timing constraints

$t_l = < Op_l , E_l , C_l >$

$UC = \{ uc_i \mid 1 \le i \le m \} = \{ uc_1, uc_2, \ldots, uc_i, \ldots\ldots, uc_m \}$ where $1 \le i \le m$

m is the total number of Use Cases

$SD_i = \{ sd_{ij} \mid 1 \le j \le p_i \}$ $SD = {}_{i=1}\cup^{m} SD_i$

p_i is the total number of Sequence Diagrams in Use Case number i

$M_{ij} = \{ \mu_{ijk} \mid 1 \le k \le q_{ij} \}$ $M = {}_{i=1}\cup^{m} {}_{j=1}\cup^{p_i} M_{ij}$

q_{ij} is the total number of messages in Sequence Diagram ij

The set FM_{ij} is the subset of M_{ij} that contains the messages that are required for the timing constraints, and if any then, the edge messages in the Sequence Diagram SD_{ij} are included as well.

$TM_{ij} = \{ \mu_{ijk} \mid \exists l (1 \le l \le n) : \mu_{ijk} \in E_l \}$

If ($TM_{ij} = \varnothing$) then $FM_{ij} = \varnothing$ Else $FM_{ij} = TM_{ij} \cup \mu_{ij1}, \mu_{ijqij}\}$

$FM = {}_{i=1}\cup^{m} {}_{j=1}\cup^{p_i} FM_{ij}$

The set FSD_i is the subset of SD_i that contains the set of Sequence Diagrams that have messages in FM

$FSD_i = \{ sd_{ij} \mid 1 \le j \le p_i , FM_{ij} \ne \varnothing \}$ $FSD = {}_{i=1}\cup^{m} FSD_i$

The set FUC is the subset of UC that contains the set of Use Cases that have Sequence Diagrams in FSD

$FUC = \{ uc_i \mid 1 \le i \le m_i , FSD_i \ne \varnothing \}$

4.1.2.2.2 Step 1

Construct the top level/levels of the Observer state chart from the logical relationships and structure between all the Use Cases in FUC such that:

A- Each Use Case maps to a Macro state: where the Micro states of that state are constructed later in step 2 and each represent a Sequence Diagram.

B- If a Use Case is contained in another Use Case it is mapped to a Micro State inside the corresponding Macro State representing the containing Use Case.

C- Each relationship between two Use Cases in FUC is mapped to a transition triggered by the occurrence of a message from the corresponding sequence diagrams.

4.1.2.2.3 Step 2

For each element in FUC construct the state diagram that represents the logical relationships and structure between all elements in FSD such that:

A- Each Sequence Diagram in FSD maps to a Micro state in the corresponding Use Case Macro State in FUC.

B- Each relationship between two Sequence Diagrams in FSD (consecutive conditional or unconditional execution) is mapped to a transition triggered by the occurrence of the first message in following Sequence Diagram.

C- Each Use Case in FUC has a "Start_Use_Case" state representing its initial starting point from which the selection of the first Sequence Diagram to be executed, is made.

D- Selection is based on transitions triggered by the occurrence of the first message in the corresponding Sequence Diagrams causing the transition to their corresponding Macro States.

4.1.2.2.4 Step 3

For each Sequence Diagram in FSD construct the state diagram that maps all messages in FM into transitions triggered by the occurrences of messages in the observed system such that:

A- States are named by the message name "Received_messagename" and are triggered the message reception at the destination Capsule

B- Mapping is done for messages in FM only: The set FM as defined above does not contain all the messages in each Sequence Diagram in FSD. Only the messages that are related to elements of the timing constraints and the messages on the edge of the Sequence Diagrams are modeled.

C- Time stamps of messages involved in all concerned timing constraints are collected in variables “RTTimespec” using the system method “RTTimespec::getclock(Variable_Name)” , as they occur.

D- Each constraint Boolean expression is evaluated immediately after the collection of the last element (occurrence of the last message related to the constraint). Then the check is performed and the violation (if any) is logged.

4.1.2.2.5 Observer Model for the cardiac pacemaker case study

We modeled an Observer for the pacemaker based on the Use Case driven Observer modeling methodology. The two timing constraints for the AVI operational mode were used to construct the FUC, FSD and FM. Figure 4.4 shows the three level hierarchical state machine of the modeled Observer and the messages mapped from the sequence diagrams, based on the definitions and steps of the Use Case driven Observer modeling. The Programming Use Case and the AVI Use Case formed FUC. The mapping of FUC based on step 1 is shown in figure 4.4 (a). The three sequence diagrams; Refractory, Unsensed and Sensed formed FSD. The mapping of FSD based on step 2 is shown in figure 4.4 (b). Figure 4.4 (c,d) show the State Diagram of the “Unsensed” and “Refractory” Sequence Diagrams (figure 4.4 (e,f)) respectively. The timing constraints span two sequence diagrams only; Refractory and Unsensed. For this reason, the Sensed sequence diagram is modeled. In modeling the Unsensed sequence diagram, the messages “A Pace Start” and “Pace Timeout” are not elements of any of the two timing constraints and hence the are not mapped. While in the Refractory sequence diagram the message “RefTimeOut” is mapped because of being an edge message and the message “VrefractDone” is mapped because of being part of the second constraint.

One of the benefits of Modeling constrains in this manner is the ability to verify sequence diagrams, in a manner that is proportional to the amount of details modeled in the Observer.

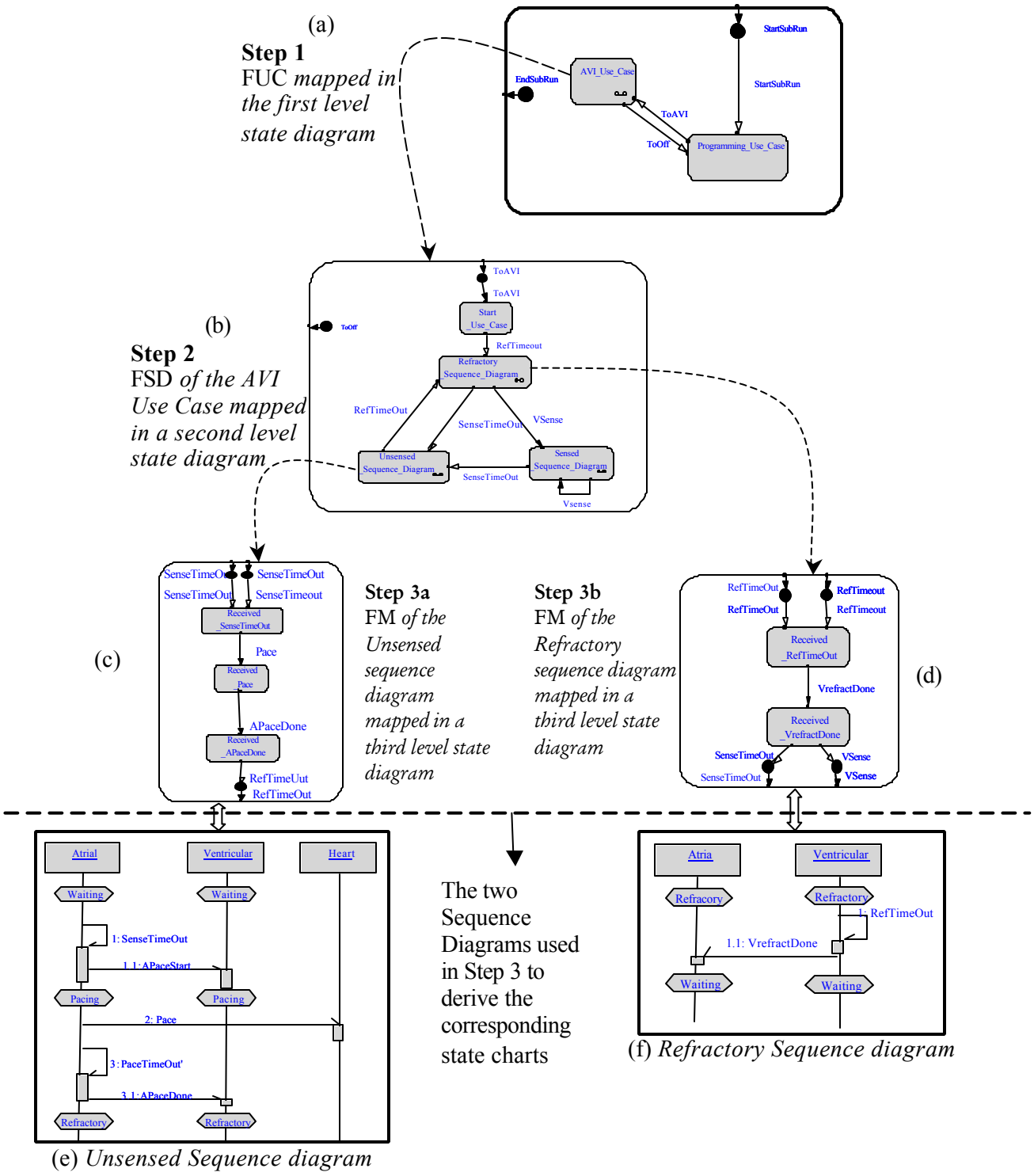

Figure 4.4 Use Case Driven Observer Modeling

4.1.3 Results and lessons learned

In this section we show our results, from applying the approaches and methods above, and confirming with visual inspection of the timing diagrams, to the AVI scenario of the pacemaker model. We injected timing faults in the pacemaker model in order to force the occurrence of violations based on the analysis methods described later in this chapter and in [34]. The timing diagrams described earlier in chapter 2 were generated and used to verify the expected logics behind the detected violations. The sample that we show below is a Time-out based timing analysis in which we study the effect of the time set for the Ventricular_Model Refractory timer (timer controlling the time spent in the Refractory state by the Ventricular_Model and the exiting transition to the waiting state) on the timing constraints, when increased by 50 milisec to be 350 milisec. We know that the Atrial_Model Refractory time (time spent in the Refractory state by the Atrial_Model) is directly controlled by the Ventricular_Model Refractory time through the messages: ApaceDone and VrefractDone from the Venticular_Model to the Atrial_Model. Thus we expect the periodic violation of the second constraint. The increase in the Atrial_Model Refractory time, being part of the cycle time, causes an increase in the delay between each generated pace and each unsensed pulse. The increase in the accumulated delay becomes significant to the first timing constraint starting from the third consecutive unsensed heart beet. We tested the presented approaches and methods and proved their correctness when the violation tables generated for the same faulty simulation run were identical. An increase in the Ventricular_Model Refractory time from 300 milisec to 350 milisec was the selected fault. Table 4.1 shows a sample of the violations from the three simulation runs where the temporal V&V was performed using the presented approaches and methods: the violation algorithm (first approach), constraint driven Observer modeling (first method in the second approach) and Use Case driven Observer modeling (second method in the second approach).

22612	Constraint 2 Violated: Refractory problem
23664	Constraint 2 Violated: Refractory problem
24165	Constraint 1 Violated: Delayed Pacing problem
24715	Constraint 2 Violated: Refractory problem
25216	Constraint 1 Violated: Delayed Pacing problem
25767	Constraint 2 Violated: Refractory problem
26268	Constraint 1 Violated: Delayed Pacing problem
26818	Constraint 2 Violated: Refractory problem
27319	Constraint 1 Violated: Delayed Pacing problem
27870	Constraint 2 Violated: Refractory problem
34339	Constraint 2 Violated: Refractory problem
35391	Constraint 2 Violated: Refractory problem
35891	Constraint 1 Violated: Delayed Pacing problem
36442	Constraint 2 Violated: Refractory problem
36943	Constraint 1 Violated: Delayed Pacing problem
37494	Constraint 2 Violated: Refractory problem
37994	Constraint 1 Violated: Delayed Pacing problem
38545	Constraint 2 Violated: Refractory problem
39046	Constraint 1 Violated: Delayed Pacing problem
39597	Constraint 2 Violated: Refractory problem
46056	Constraint 2 Violated: Refractory problem
47108	Constraint 2 Violated: Refractory problem
47608	Constraint 1 Violated: Delayed Pacing problem

Table 4.1 Sample of the violation table from simulation with 350milisec Ventricular_Model Refractory time

We argue that the three directions for automated verification of timing constraints presented above are independent, yet selecting the most suited direction is specific to the specification to verify and the V&V objectives. The first approach will be the most effective and efficient when the verification objectives do not require any response within a single simulation run. Which we described as an open loop analysis where there is no intention for stopping the simulation nor changing the simulation settings in response to a violation as it occurs. In this case the first approach is the most efficient and we perceive it to be the most scalable. This limitation is handled in using the second approach, in which a selection of the Observer modeling method should be performed. In the constraint driven Observer Modeling, the amount of effort spent by the analyst in modeling the Observer and the complexity of the Observer model is directly proportional to the

number of constraints modeled. This fact makes this method limited by the number of constraints to be studied, thus introducing the limitation on the use of the method in cases where more than four timing constrains are being verified. This limitation is eliminated when using the Use Case Observer modeling, yet the trade off when selecting the constraint driven Observer modeling over the Use Case driven Observer modeling in case of four constraints or less is in the amount of effort spent in modeling the Observer versus the inability (if required) to verify sequence diagrams nor to gather statistics that can be used in other analysis.

4.2 The Four Timing Analysis Methods

4.2.1 Methods

Using the automatic generation of timing diagrams described in chapter 2, the analyst can inspect the timing diagrams to verify that timing constraints are met. Moreover, the analyst can deploy several timing analysis methods to study the effect of delays in transmission or processing of messages. Table 4.2 summarizes four timing analysis methods that we developed to analyze UML specifications. We discuss each of the proposed methods using a Focus/Purpose/Method template.

Timing Analysis Method	Focus	Purpose
Concurrency-based	Links between objects (components)	Study the effect of delays of delivering messages between objects
Performance-based	Objects (components)	Study the effect of implementation efficiency
Timeouts-based	Objects (components)	Study effect of various timeout values.
Environment-Interactions	External Environment	Study effect of delays in recognizing hardware events

Table 4.2 Summary of Timing Analysis Methods

4.2.1.1 Concurrency-based Timing Analysis:

***Focus*:** Architecture connectors (links between objects)

***Purpose*:** Analyze the effect of delays in delivering messages from one component (object) to another.

***Method*:**

- Augment the model with delays over connectors involved in each scenario.
- Generate timing diagrams for each simulation run.
- Inspect timing diagrams to study the effects of these delays on model behavior and required deadlines.

4.2.1.2 Performance-based Timing Analysis

***Focus*:** Architecture components (objects)

***Purpose*:** Analyze the effect of inefficient implementation of state activities and actions.

***Method*:**

- Augment the model with delays in the execution of entry, exit, and activity code segments of all states involved in each scenario.
- Generate timing diagrams for each simulation run.
- Inspect timing diagrams to study the effect of these delays on model behavior and required deadlines.

4.2.1.3 Timeouts-based Timing Analysis

***Focus*:** Architecture components (objects)

***Purpose*:** Analyze the effect of timeout values of all user defined timers in the model.

***Method*:**

- Vary the values of timers used in each scenario.
- Generate timing diagrams for each simulation run.

- Inspect timing diagrams to study the effect of these variations on model behavior and required deadlines.

4.2.1.4 Environmental-Interactions Timing Analysis

***Focus*:** Interactions with the environment including hardware devices and sensors.

***Purpose*:** Analyze the effect of delay in sensing environmental events, caused by external systems and/or event recognition software (outside system boundaries).

***Method*:**

- Augment the model with delays in sensing hardware events.
- Produce timing diagrams for each simulation run.
- Inspect timing diagrams to study the effect of these delays on model behavior and required deadlines.

Later in chapter 6 the above methods are used in Fault Injections analysis.

4.2.2 The Cardiac Pacemaker Example

4.2.2.1 Concurrency-based Timing Analysis

Focus: Delay all messages on the connector between the Atrial and Ventricular components. (10 epochs is shown in Figure 4.5)

Result: Figure 4.5 shows a sample of the Concurrency-based analysis for a Cardiac Pacemaker in the AVI operational mode where all messages between the Atrial and the Ventricular models are delayed by 10 epochs (100 milliseconds). In case of more than one unsensed consecutive heart beats, the next heart beat overlaps with the generated paces.

Reason: Due to message delay, the refractory time for the Atrial increased by at least 20 epochs and the Pacing is delayed from expected by 10 epochs, thus the start of the waiting state was delayed by at least 30 epochs.

Note: We observed that Queuing of messages occurs for delays larger than 20 epochs.

4.2.2.2 Performance-based Timing Analysis

Focus: Insert delays in the execution of actions in the refractory state of the Atrial component. (10 epochs is shown)

Result: : Figure 4.6 shows a sample of the Performance-based analysis for a Cardiac Pacemaker in the AVI operational mode where the entry actions of the Atrial Refractory state is delayed by 10 epochs (100 milliseconds).In Case of 2 consecutive unsensed heart beats, the second heart pulse overlaps with the second pace.

Reason: The inserted delay added to the refractory period of the Atrial, thus causing the start of the waiting state to be delayed.

4.2.2.3 Timeout-based Timing Analysis

Focus: Increase the timeout value of the Ventricular refractory (Vrefract) timer. (5 epochs is shown)

Result: Figure 4.7 shows a sample of the Timeout-based analysis for a Cardiac Pacemaker in the AVI operational mode where the Venticular Refractory timer is increased by 5 epochs (50 milliseconds) to be 35 epochs (350 milliseconds).In Case of 2 consecutive unsensed heart beats, the 2nd heart pulse intersects with the 2nd pace.

Reason: The Refractory time-out in the Ventricular triggers the change of state to waiting in the Atrial, thus the increase in its value causes a delayed sensation period which accumulates in the in case of consecutive unsensed heart beats.

4.2.2.4 Environmental-based Timing Analysis

Focus: Delay the sensation of the heart pulses in the Ventricular component. (30 epoch is shown)

Result: Figure 4.8 shows a sample of the Environmental-based analysis for a Cardiac Pacemaker in the AVI operational mode where the sensation messages are delayed by 30 epochs (300 milliseconds). After pulse A two pulses were not sensed from the heart, thus two paces were generated but delayed by 30 epochs, this made pulse B to fall between the two paces.

Reason: The delay causes a shift in the sensed Heart beats series, thus increasing the chance for pacing while pulsing. The effect is more clear in pulse C where one pace was generated and pulse C fallen in the refractory state.

Graph Start:	5000	Graph End:	7500	X axis label Step:	100					
Graph Name:	Concurrency based analysis. All messages between the Atrial and the Ventricular models are delayed by 10 epochs									
Series 1:	heHeart	Series 2:	VENTRICULAR_MODEL	Series 3:	ATRIAL_MODEL					
Comments:	In case of more than one unsensed consecutive heart beets, the next heart beet overlaps with the generated paces. The Pacing, Pacing timeout and the Refractory timeout messages are each delayed from expected by 10 epochs thus the waiting state starts delayed by at least 30 epochs. As well it was noticed in other delay values that queuing of messages occurs for delays larger than 20 epochs.									

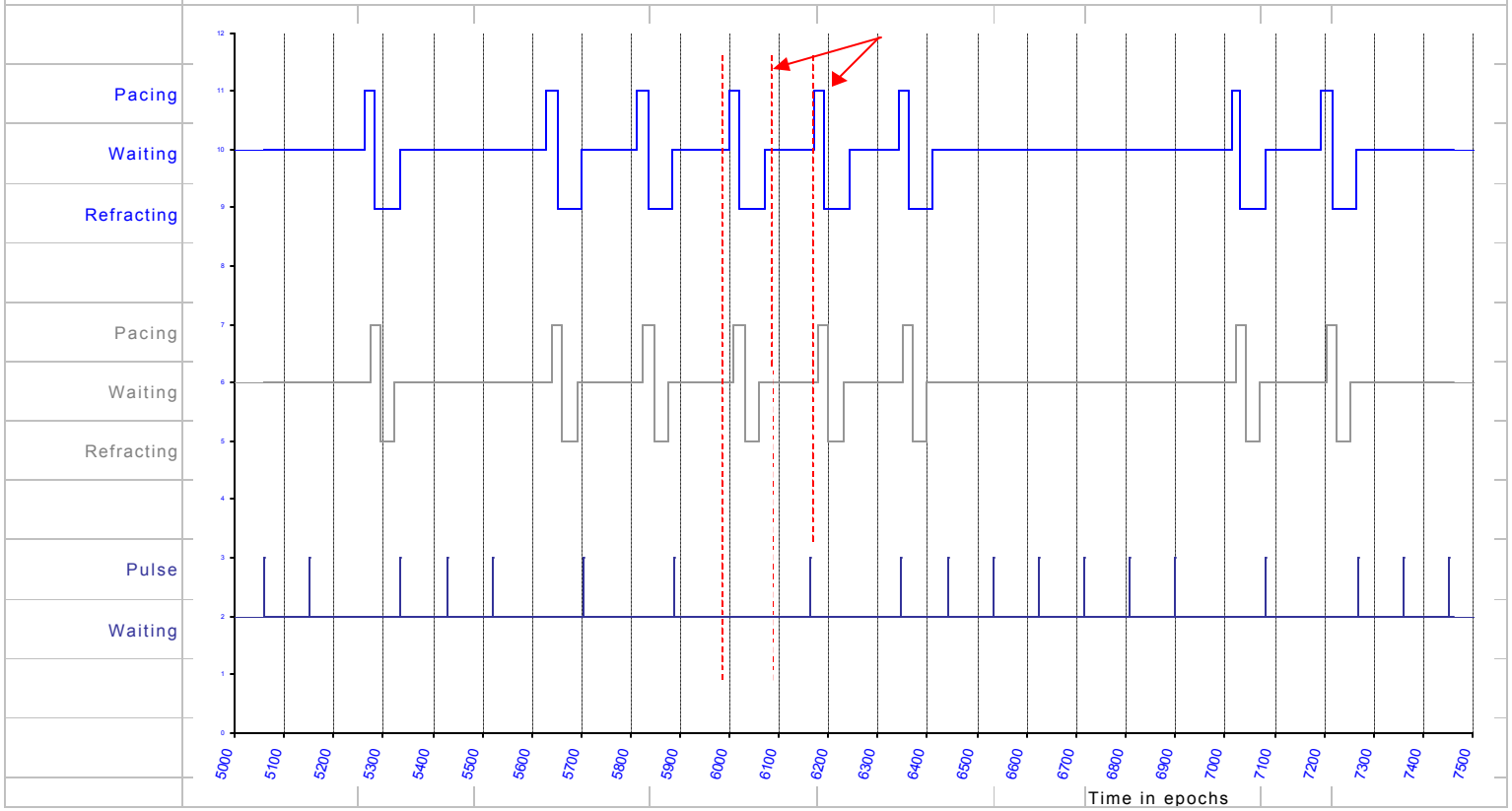

Figure 4.5 Sample of Concurrency-based Timing Analysis for a Cardiac Pacemaker in the AVI operational mode

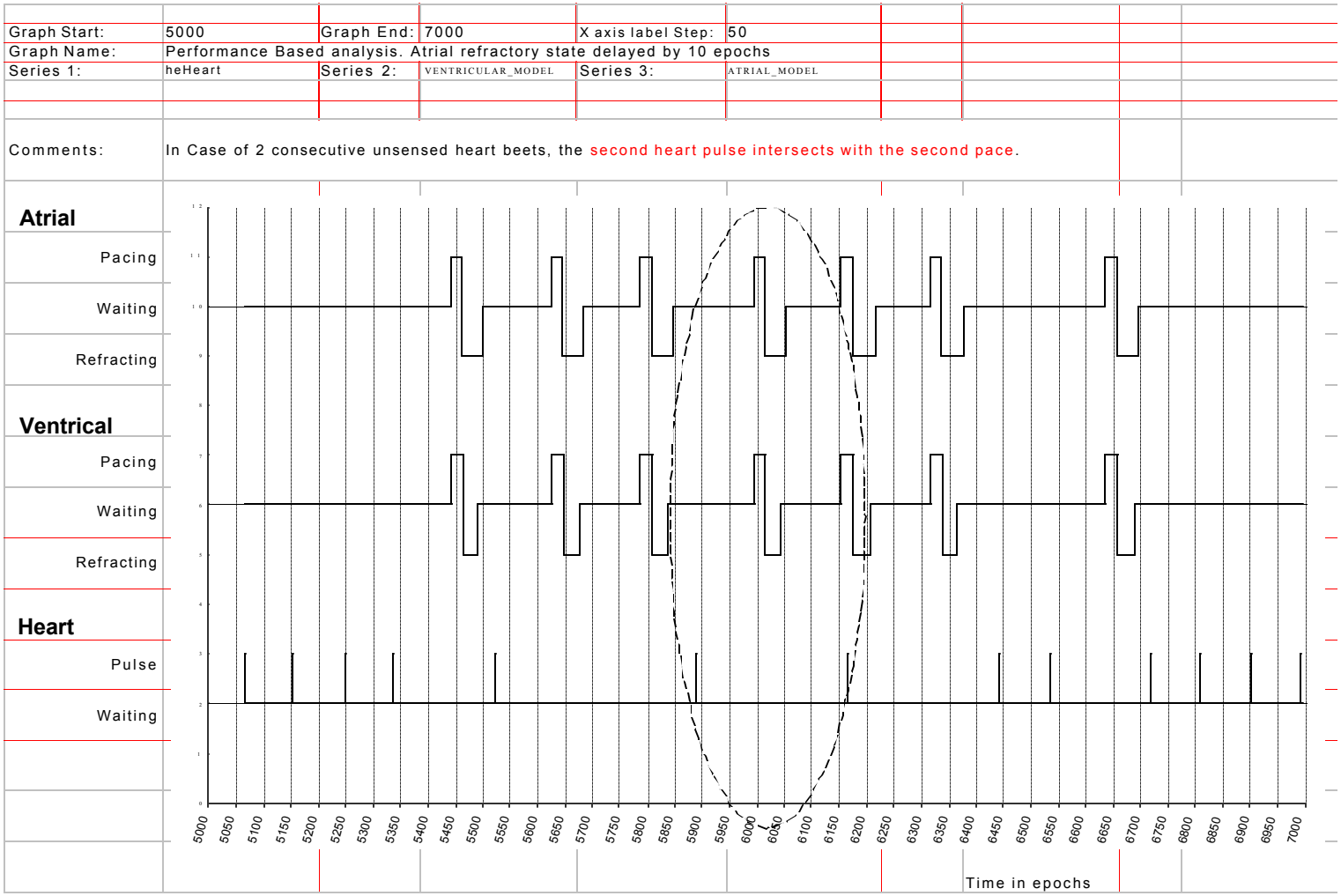

Figure 4.6 Sample of the Performance-based analysis for a Cardiac Pacemaker in the AVI operational mode

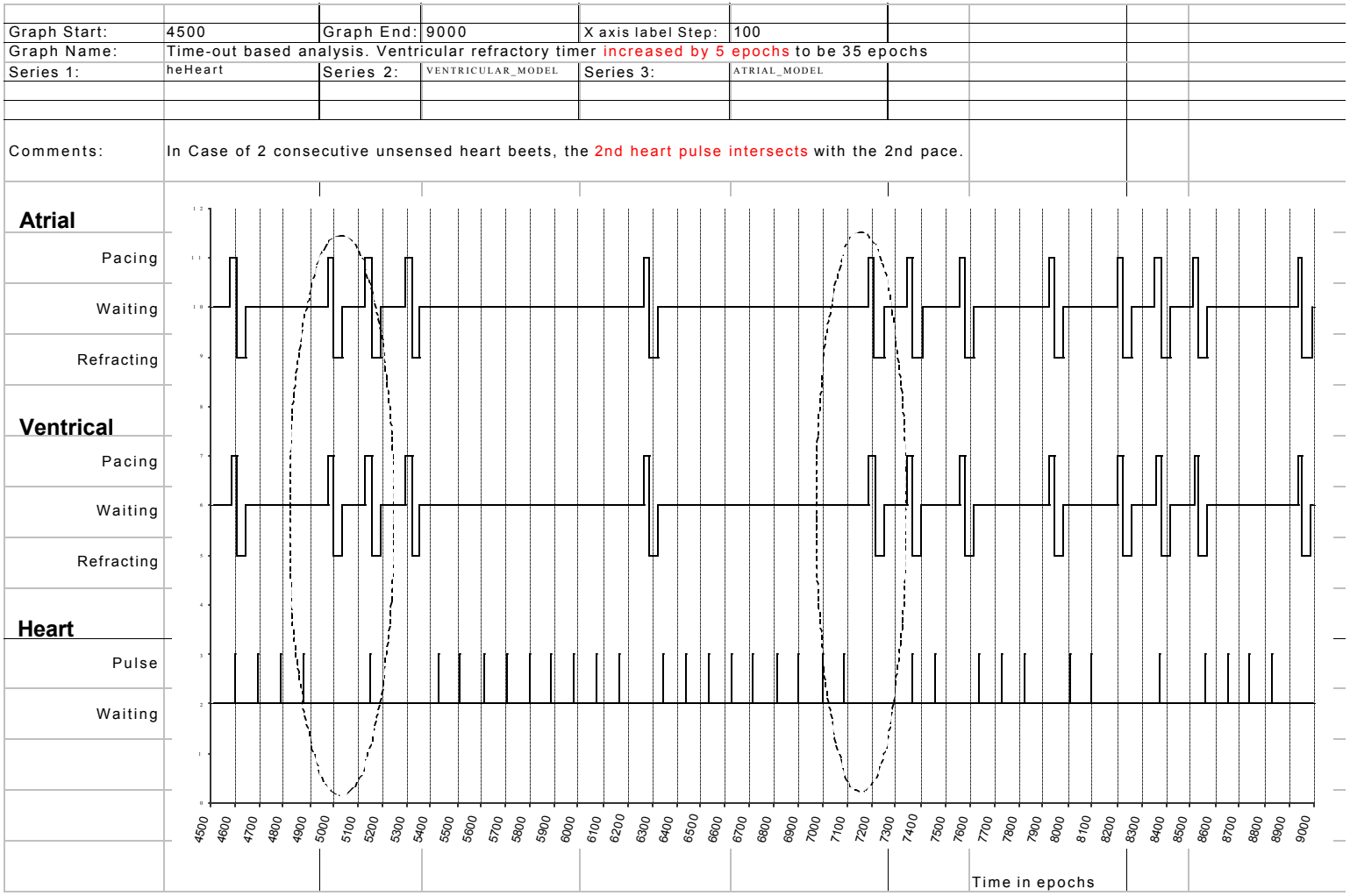

Figure 4.7 Sample of the Timeout-based analysis for a Cardiac Pacemaker in the AVI operational mode

Graph Start:	5800	Graph End:	8000	X axis label Step:	50					
Graph Name:	Environment interactions analysis. sensation messages from the heart are delayed by 30 epochs									
Series 1:	heHeart	Series 2:	VENTRICULAR_MODEL	Series 3:	ATRIAL_MODEL					
Comments:	The delay causes a shift in the Heart series, thus increasing the chance for pacing while pulsing. Indicated are two cases: After pulse A two pulses were not sensed from the heart, thus two paces were generated but delayed by 30 epochs, this made pulse B to fall between the two paces. The effect is more clear in pulse C where one pace was generated and pulse C fallen in the refractory state.									

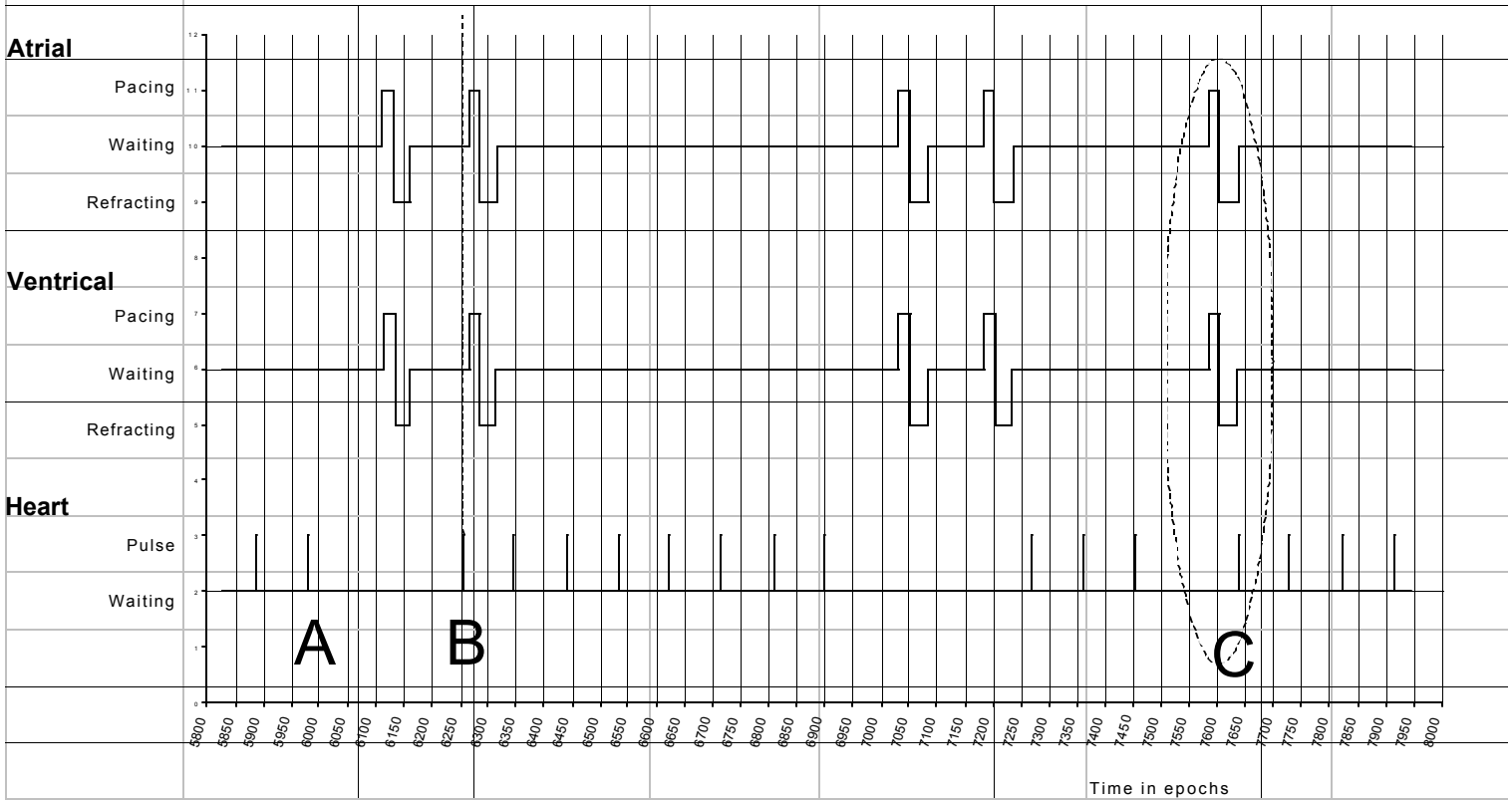

Figure 4.8 Sample of the Environmental-based analysis for a Cardiac Pacemaker in the AVI operational mode

CHAPTER 5: AUTOMATED RISK ASSESSMENT

5.1 Introduction

Risk assessment is an important process in managing software development. Performing risk assessment in the early development phases enhances the resource allocation decisions [33]. Several methodologies for risk assessment were developed, mostly based on subjective judgment. In this chapter we present how the methodology presented in [33] is automated. The methodology is based on:

1. Dynamic metrics: presented in [35] where component complexity and connector coupling factors are derived from simulating all scenarios based on the system scenario profile. A brief description is presented in section 5.1.1 of this chapter.
2. Component Dependency Graphs (CDG): introduced in [36] and adapted in [33] where a CDG Risk traversal algorithm is presented. A brief description of the CDG and the risk aggregation algorithm is presented in section 5.1.2 of this chapter.
3. Severity analysis: Based on MIL_STD_1629A where the worst case consequence of a failure is considered, and the severity is determined by the degree of injury, property damage, system damage, and mission loss that can occur. The Failure Mode and Effect Analysis (FMEA) technique is a systematic approach that details all possible failure modes and identifies their resulting effect on the system [24]. In [33] severity indices ($svrty_i$) of 0.25, 0.50, 0.75, and 0.95 were assigned to minor, marginal, critical, and catastrophic severity classes respectively.

The UML-RT model is built and simulated using RRT, from which log files are made available for extracting the required parameters. We use Microsoft Excel sheets and Macros in the development of the automated environment together with RRT tool. The methodology derives heuristic risk factors for components and connectors from dynamic metrics and severity analysis (equation 5.1), and the system/subsystem overall risk factor is obtained from the traversal of the CDG.

$$hrf_i = cpx_i \times svrty_i \qquad \textit{Eq. 5.1 (source [33])}$$

where $0 <= cpx_i <= 1$, and $0 <= svrty_i < 1$ are the normalized complexity level (dynamic complexity for components or dynamic coupling for connectors) and severity level for the architecture element respectively (source [33]). The first step in the Risk assessment methodology for dynamic specifications is to derive the complexity factors (component complexity and connector coupling) using simulation and Dynamic Metrics [35]. The next step is to derive severity factors for components and connectors using FMEA and simulation. Developing heuristic risk factors for components and connectors by using equation 5.1 is the third step. Constructing a CDGs for risk assessment purposes and traversing the graph using the risk aggregation algorithm, presented later in this chapter, is the final step where the product is the system/subsystem overall risk factor.

5.1.1 Dynamic Metrics

The complex dynamic behavior of many real-time applications motivates a shift in interest from traditional static metrics to dynamic metrics. Active components are sources of errors because they execute more frequent and experience numerous state changes. Therefore there is a higher probability that if a fault exists in an active component, it will easily manifest itself into a failure. For risk analysis at the architecture level, the risks of a failure are the interest. Hence, the motive to assess the complexity of components and connectors as expected at run-time using dynamic metrics.

In the risk analysis, the dynamic metrics defined in [35] are used to obtain complexity factors for each architecture element. A complexity factor for each component is obtained using the dynamic complexity metric for the statechart specification of that component. A complexity factor for each connector is obtained using the dynamic coupling metric for the messaging protocol of that connector.

5.1.2 Component Dependency Graphs

Component Dependency Graphs (CDGs) are introduced in [36] as probabilistic models for the purpose of reliability analysis at the architecture level. CDGs are directed graphs that represent components, component reliabilities, link and interface reliabilities, transitions, and transition probabilities. CDGs are developed from scenarios. One way to model scenarios is using UML

sequence diagrams. By using sequence diagrams, we are able to collect statistics required for building CDGs, such as the average execution time of a component in a scenario, the average execution time of a scenario, and possible interactions among components. Figure 5.1 illustrates a simple CDG example consisting of four components, C_1, C_2, C_3, and C_4.

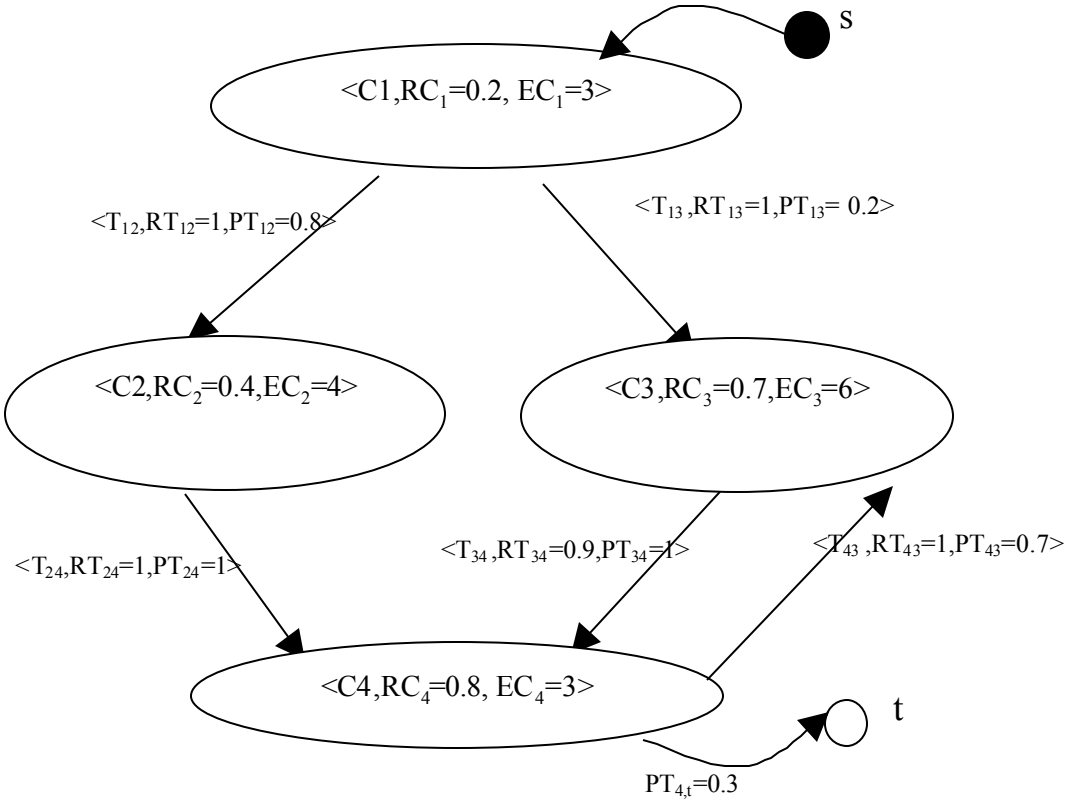

Figure 5.1 A Sample CDG *1* (source [33])

A CDG is defined as follows:

CDG=<N,E,s,t>; where *N* is set of nodes, *E* is set of edges, and *s* and *t* are the start and termination nodes, i.e. *N = {n}, E ={e},*

n = < C_i, RC_i, EC_i >; where C_i is the name of the i^{th} component, RC_i is component reliability, and EC_i is average execution time of a component C_i

$e = <T_{ij}, RT_{ij}, PT_{ij}>$, where T_{ij} is transition from node n_i to n_j in the graph, RT_{ij} is transition reliability, PT_{ij} is transition probability.

5.1.3 The Risk Analysis Algorithm

The architecture risk factor is obtained from aggregating the risk factors of individual components and connectors. Assuming that a sequence of components are executed, then the risk factor for that sequence of execution is given by:

$$HRF = 1 - p_i(1-hrf_i) \quad \textit{Eq. 5.2(source [33])}$$

Where p_i is the CDG traversal operation defined by the "while loop" in the algorithm shown in figure 5.2.

After constructing the CDG model, the risk of the application can be analyzed as the function of risk factors of components and connectors using the following risk assessment algorithm.

Algorithm
Procedure AssessRisk
Parameters
consumes CDG, AE_{appl}(average execution time for the application)
produces $Risk_{appl}$
Initialization:
$R_{appl} = R_{temp} = 1$ (temporary variables for (1-RiskFactor))
Time = 0
Algorithm
push tuple $<C_1, hrf_1, EC_1>$, Time, R_{temp}
while Stack not EMPTY do
pop $< C_i, hrf_i , EC_i >$, Time, R_{temp}
if Time > AE_{appl} or $C_i = t$; (terminating node)
R_{appl} += R_{temp} ;(an OR path)
else
$\forall < C_j , hrf_j , EC_j > \in$ children(C_i)
push ($<C_j, hrf_j , EC_j>$, Time += EC_i , R_{temp} = R_{temp}*(1-hrf_i)*(1-hrf_{ij})*PT_{ij}) (AND path)
end
end while
$Risk_{appl}$ = 1- R_{appl}
end Procedure AssessRisk

Figure 5.2 Risk Aggregation Algorithm (source [33])

The algorithm expands all branches of the CDG starting from the start node. The breadth expansions of the tree represent logical "OR" paths and are hence translated as the summation of aggregated risk factors weighted by the transition probability along each path. The depth of each path represents the sequential execution of components, the logical "AND", and is hence translated to multiplication of risk factors (in the form of $(1\text{-}hrf_i)$). The "AND" paths take into consideration the connector risk factors (hrf_{ij}). The depth expansion of a path terminates when the summation of execution time of that thread sums to the average execution time of a scenario or when the next node is a terminating node.

5.2 The Automated Environment

Figure 5.1 shows a block diagram of the products and processes in the proposed environment for automated risk assessment. Circles and ovals denote inputs/outputs to be processed/produced by the processes and activities shown.

Architecture modeling is performed using the UML simulation environment provided by RRT. The UML simulation environment consists of an Observer Capsule defined as an external observing entity. The Observer component is not part of the RRT tool; we defined this component in order to facilitate the automation process. These violations represent detected failures during the simulations. The observer is modeled using state charts based on the expected dynamic behavior of the components as depicted in the sequence diagrams.

The analyst provides simulation settings at the start of the simulation. These settings consist of variations for variables that represent timer and delay value for real-time activities on successive runs managed by the observer. They also capture the different settings for the input stimuli that simulate sequences of scenarios. The simulation Log and the violation report produced from the simulation are fed to the analysis tool (MS Excel Macro). The MS Excel Processing Macro analyzes the log file and produces timing diagrams and a violation table. The violation table consists of detected violations or failures and their occurrence time. The timing diagrams are provided to help the analyst identify the severity level of the detected failure in terms of meeting deadlines. The Excel Processing Macro also produces an Excel sheet for normalized component complexity for each component, an Excel sheet for normalized connector complexity for each connector, and an Excel sheet for the CDG. The values hrf_i and hrf_{ij} are identified in a later stage

during the execution of the Risk Macro. Severity Ranking is obtained from the severity analysis performed by the analyst using the violation table and timing diagrams as diagnostics for effect analysis and the simulation settings. Feeding the Severity ranking, complexity factors and CDG to the analysis tool, Risk factors for each component and connector are obtained and the CDG is traversed to obtain the system/subsystem overall risk factor HRF. Appendix B shows the MS Excel Risk Macro and Risk Traversal Macro, where in the Risk Macro the construction of the CDG is achieved and equation 5.1 is utilized while in the Risk Traversal Marco the CDG traversal algorithm (figure 5.2) is implemented and the product is the overall system/subsystem Risk factor based on equation 5.2.

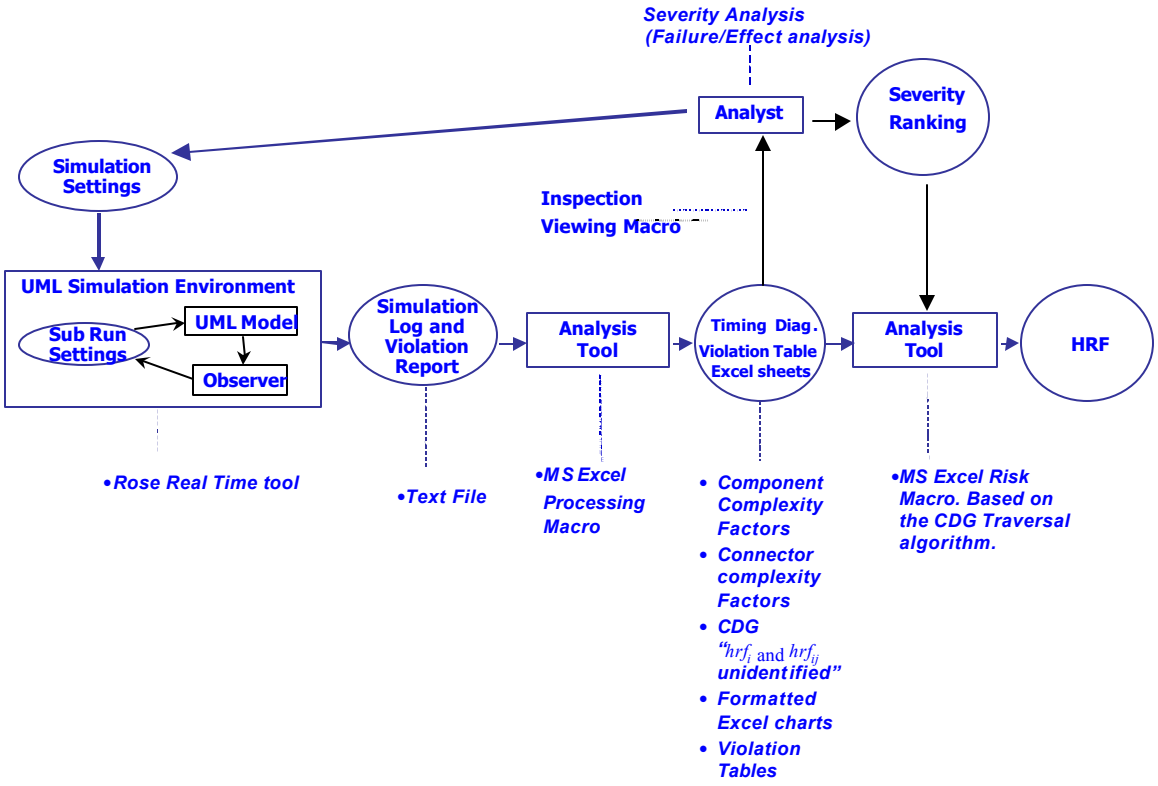

Figure 5.3 The Automation process-product diagram

5.3 Conclusion and Future Work

The methodology presented in [33] has the following benefits: it is applicable early at the *architectural-level* and hence it is possible to identify critical components and connectors early in the lifecycle. The methodology uses dynamic metrics, that covers the fact that a fault in a frequently executed component will frequently manifest itself into a failure. The methodology is based on *simulation* of UML-RT models. Simulation helps in: performing FMEA procedures and observing the timing diagrams. The presented automation environment shows how RRT tool can be used in fast and efficient deployment of the methodology.

The above methodology and its automation were applied to the Cardiac Pacemaker case study (presented in chapter 3). Yet future research could experiment with applying the methodology to larger case studies with multiple subsystems to compare the aggregated risk factors of individual subsystems. A Static Architectural-Level Risk Assessment methodology based on McCabe's Cyclomatic Complexity can be derived following the same fashion of the dynamic Architectural-Level Risk Assessment methodology. Tool support can be provided by Rose Extensibility Interface where simulation is not required. Comparing Static Risk and Dynamic Risk is required to assess the effort and time spent in applying both methods

CHAPTER 6: FAULT INJECTION ANALYSIS

Failures can occur when a software component fails, a hardware component fails, bad or corrupted input is provided to the system or/and Executing an unlikely software/hardware error (design or implementation). Fault injection is a technique for analysis and verification of systems behavior (responses) to these failures before deployment. Fault injection studies can be categorized into three types: Hardware fault injection, Software Implemented Fault Injection (SWIFI) and software simulation fault injection, which intern was studied versus SWIFI in [28]. Several studies on Fault Injection analysis were conducted, mostly on code level in case of SWIFI, on hardware prototypes in case of hardware fault injection and on simulation models in case of software simulation fault injection. Several tools were developed for fault injection analysis [9,6,7]. Software simulation are typically high level abstraction of a system, characterized by protocols, interfaces, components and function, where the typically injected faults are: miss-timings, missing or corrupted massages, and missing or corrupted message replays. Software simulation fault injection help flush out design level flaws (specially in fault tolerant systems). In this chapter we present a fault model, for conducting software simulation fault injection analysis, that we derived to be specific and optimized for UML-RT design models.

6.1 Motivations

It is our concern for this work to provide a fault model for UML-RT models in order to use it in conducting fault injection analysis. Three motives derive our study in fault injection analysis:

1. Severity Analyses: where a severity factor based on MIL_STD 1629A [24] for each component in a UML-RT model is derived. Severity factors were required in Architectural-level Risk assessment in [33].
2. Test Cases Optimization: In [1] a method for building trusted components where a component is seen as a set of: specifications, a given implementation and its embedded test cases. Later in section 5 we demonstrate the use of our fault model in optimizing the number of test cases needed.

3. Verification of Fault Tolerant software and systems. In [31] and [32] fault injection is viewed as a testing and verification tool, rather than a debugging tool.

6.2 UML-RT Model elements

The UML-RT model can be covered from two general types of elements: the Structural related elements and the Behavioral related elements. The Structural elements describe the software architecture of the model. UML-RT defines Capsules that decompose into several Capsules in a layered fashion. RRT Structure Diagram is used to view the Capsule structure. Ports and Connectors are used to connect these Capsules. Capsules are the equivalent of components in a Software Architecture while Ports and Connectors resemble the connectors. The Behavioral elements are used to describe the time related and/or dependent requirements, in essence the dynamic behavior of Components. UML-RT (and therefore RRT) defines State Charts to describe the dynamic behavior of a Component. The State Diagrams are composed from: States (Marco

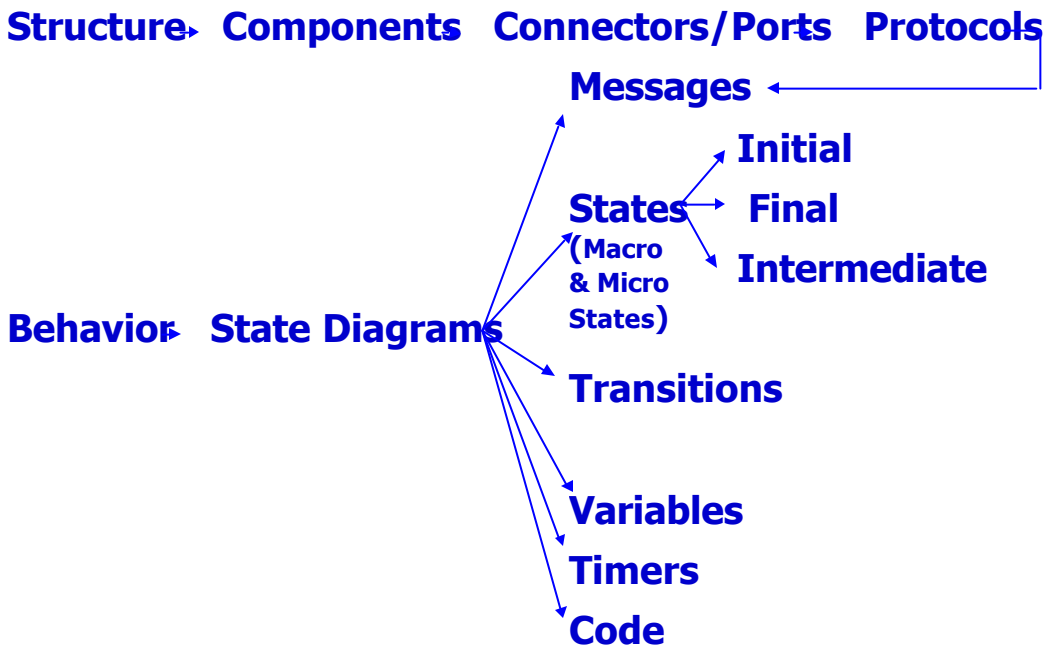

Figure 6.1 UML-RT model elements

and Micro States), Messages, Transitions (responding to the reception of a Message), Code (used in sending Messages), Timers and Variables. In [10], a full analysis for the major testing problems and their resolutions in testing state machine based models were presented. States were categorized into three types of states: Initial State, Final State and Intermediate State. The Behavioral and Structural elements are linked by the definition of Protocols that define the flow (time dependent behavior) of Messages (a behavioral element) on a Connector. Figure 8.1 summarizes the UML-RT model elements described.

6.3 Domain of faults in UML-RT Models

In this section we derive possible faults that can take place in the dynamic specification model. Based on the model elements presented above and following the Structural and Behavioral categorization, we derive faults that can exist from miss implementations.

6.3.1 Structural Faults

1. Components (that are part of the defined Software Architecture of the model):
    a. A missing component: A component that was not modeled. This makes the specification incomplete.
    b. Component class mismatch: In UML-RT Components are Capsules that are based on a Capsule Class. The Capsule should match the Capsule Class it is based on (instantiated from).
2. Connectors/ports:
    a. Misconnected ports: The connection between ports is established in the graphical interface, thus it is possible to swap (misconnect) connectors while connecting ports causing incorrect delivery of messages.
    b. Unconnected ports: A missing connectors causes two or more ports to be unconnected. This causes messages not to be received.

3. Protocol:

    a. Missing messages: An incomplete Protocol definition causes an incomplete specification.

    b. Incorrect directional configuration: For each Protocol two sets of messages are defined, incoming & outgoing, misplacement of messages between those sets can occur.

6.3.2 Behavioral Faults

1. State Diagrams:

    a. A missing State Diagram: A component without a State Diagram is a component without any behavioral representation.

    b. Interchanged diagrams: Two components with interchanged state diagrams are two components with interchanged behavioral representations.

2. States:

    a. Incorrect initial state: Default initial state is miss configured, thus causing the components Statechart to start executing from an incorrect state.

    b. Incorrect final state: In a macro state of a component with more than one *ChainState*, the transition leading to the transition exiting from the grand state (through ChainState) is miss configured. Thus leading to incorrect exit conditions from the macro state.

    c. Interchanged states: The transitions from and into a state and the entry and exit actions define the state. Interchanged states cause the state entry and exit actions to be swapped.

    d. Missing states: Incomplete description of the dynamic behavior of a component.

3. Transitions:
    a. Incorrect trigger: Incorrect transition configuration, i.e. incorrect triggering message configured.
    b. Interchanged transitions: the transitions triggering message and actions are interchanged with equivalent in another transition.
    c. Missing transitions: Incomplete description of the dynamic behavior of a component.
4. Messages:
    a. Missing sends: A message command "in code" responsible for triggering a transition in a remote component resulting in an incomplete description of the dynamic behavior of both components.
    b. Corrupted message attributes: Corrupted data carried in a message.
5. Variables:
    a. Corrupted initial value: Incorrect initial value.
    b. Corrupted dynamic value: Incorrect handling of variable value during run time.
6. Time: We refer to four timing analysis methods developed in [34] as the types of time related faults.

6.4 The Fault Model

In this section we present the set of faults derived from the domain of faults in UML-RT Models presented earlier and their deployment procedure. We claim that the selected set is generally representing the dynamic part of the domain and we assess our claim by applying the selected Fault Model to the Pacemaker case study described in chapter 3.

The Proposed Fault Model is based on the basic behavioral element in UML-RT models; *the micro State,* and is defined by the following four subsections.

6.4.1 State Selection Process

Five steps describe our process for Fault Injection analysis for UML-RT models. The first two steps are not required for Severity analysis since the severity level of each component has to be deduced. While for test case optimization all steps are required.

1. Order Components based on dynamic complexity: Our process for Fault Injection analysis for UML-RT model starts by the selection of a set of components to be analyzed based on their dynamic complexity factors (refer to [35] for details on dynamic complexity).
2. Select the set of components to be injected with faults based on highest complexities: The number of the selected components depends on the complexity threshold specified by the analyst.
3. Order states in each component based on contribution to the component complexity: Order the microstates of each component based on the degree of contribution to the dynamic complexity factor of the component. The first having the highest share in the components dynamic complexity.
4. Select the set of states and macro states to be injected with faults based on the highest contribution to the component's complexity: The number of selected states is proportional to the inverse of the quality level of the analysis and to the time spent in the whole process, which is again up to the analyst to decide.
5. Inject the three sets of faults indicated bellow for each of the selected states.

6.4.2 State faults

1. Swap the selected state with the state next in order (*State Swap*): Interchange the entry and exit action code of the selected state with equivalents in the state next in order of contribution to dynamic complexity.
2. Swap transitions out of the selected state (*Transition Swap*): If and only if the selected state has more than one outgoing transitions, interchange each two transitions, i.e. swap destination, trigger and actions.

3. If an initial state exists, force the selected state to be the initial state (*Initial State Swap*): The tool provides the ability to specify the initial state in a state diagram, and hence forces the selected state to be the initial state if it is not.

4. If a final state exists, force the selected state to be the final state: The tool provides the ability to specify the final state (or states in case of more than one exit transition from the containing macro state) in a state diagram, and hence forces the selected state to be the final state (or any of them if more than one exists).

6.4.3 State transition faults

1. Disable the transition (*Null Trigger*): Remove the triggering message (equivalent to the transition being configured to a null message).

2. Interchange trigger message with another randomly selected message (*Trigger Swap*): Change the triggering message to any other message from the same protocol.

6.4.4 Timing Faults

Listed bellow are the four timing analysis methods described in chapter 3 and summarized in table 4.2:

1. Timeouts-based

2. Concurrency-based

3. Performance-based

4. Environmental-interactions

6.5 Pacemaker case study Experimentation

We injected faults in the Pacemaker model, presented in chapter 3, based on the fault model presented above. First we conduct the dynamic complexity ordering of components [33], and we arrive to the fact that the *Atrial_Model* and the *Ventricular_Model* have the highest factors. For the purpose of this work we show results from analyzing the *Atrial_Model.* Second we analyzed the *Atrial_Model* microstates and the *Waiting* state of the AVI scenario had the highest contribution to the components dynamic complexity, followed by the *Pacing* state. We use two

sequences of heart pulses as test cases for the V&V of the AVI mode, each is a different set of heart pulses, one with three skipped pulses and the other with one skipped pulse. We use Timing Diagrams to show our results. Each fault is injected in two simulation runs with the two heart sequences as two different inputs to the system. Figure 6.2 and figure 6.3 show the expected behavior of the Pacemaker in the AVI operational mode, in case of the heart skipping three pulses consecutively (figure 6.2) and in case of the heart skipping one pulse (figure 6.3). The Timing Diagrams for the three heart pulses skipped and the one heart pulse skipped are shown for each fault. The six Timing Diagrams next to figure 6.3 are the results of applying the State Faults of the fault model while the last four are the results from applying the State Transition Faults. Below we describe the application of the fault model in Fault – Result fashion:

1. State Faults:
    a. State Swap (figure 6.4 & figure 6.5):
        i. Fault: Swap the Waiting and Pacing states of the *Atrial_Model* AVI macro state.
        ii. Result: Faulty behavior in which the *Atrial_Model* is pacing the heart periodically regardless of the existence of the pulse from the heart. This violates the AVI operational mode requirements.
    b. Transition Swap (figure 6.6 & figure 6.7):
        i. Fault: Swap the transitions "GotVSense" with "Time-Out" of the Waiting state.
        ii. Result: *Atrial_Model* and *Ventricular_Model* went out of synchronization. Thus causing the *Atrial_Model* to be stuck at the Refractory state and the *Ventricular_Model* to be stuck at the waiting state.
    c. Initial Sate Swap (figure 6.8 & figure 6.9):
        i. Fault: The waiting state is forced to be the initial state instead of the refractory state.
        ii. Results were deferent in each heart sequence:
            1. Three skipped pulses: Failure to meet the timing constrains in the first 15 seconds of operation in case of three skipped pulses

2. One skipped pulse: Successful operation in case of one skipped pulse.

2. State Transition Faults:
    a. Null Trigger (figure 6.10 & figure 6.11):
        i. Fault: The trigger of the Time-Out transition is removed.
        ii. Result: Both *Atrial_Model* and *Ventricular_Model* were stuck at the Waiting states, Pacing state never visited and the *Heart* was never paced.
    b. Trigger Swap (figure 6.12 & figure 6.13):
        i. Fault: The trigger of the Time-Out transition changed to be the Sense message from the *Ventricular_Model* and the trigger of the GotVSense transition changed to the timer's time-out message.
        ii. Result: Pacing the heart when not required while not pacing when required. Thus violating the AVI operation mode requirements.

Assuming the motive of studying the Severity of the Atrial Model, we conclude that since one or more of the faults lead to a faulty behavior that will cause the death of the patient, then its severity level is "Catastrophic", even that one of the faults "Initial state swap" did lead to a faulty initial behavior that would not cause patients death.

Assuming the motive of optimizing the number of test cases required for the testing the Atrial component, we observe that the second sequence of heart pulses with one skipped pulse does not cause the fault "Initial State Swap" to manifest into a failure, while the first sequence uncovers all the injected faults. Thus we can eliminate the second sequence from our testing process. We note that only two test cases we used demonstrate the use of our fault model in test case optimization.

6.6 Conclusions & Future Work

The proposed Fault Model is acknowledged for its applicability in early development stages and scalability. Yet further experiments should be conducted on several case studies for better assessment and enhancement. Enhancements are required in several areas:

1. In the process of component selection, the number of components selected is decided by the analyst based on the available resources (mainly time). Better criteria for this selection is required to guarantee the best results when using the fault model in test case optimization.
2. The presented model focuses on microstates, while it is applicable to Marco states as well. Thus experiments for assessing the level of effectiveness of the fault model at the macro state level.
3. In the process of state selection, the number of sates selected is decided by the analyst based on the available resources (mainly time). But the tradeoff is in the quality of the analysis, thus a criteria for this selection is required.
4. The selection of the second message to swap with in a *Trigger Swap* is random. We perceive that a selection criteria is required for better results.

Finally we stress on the fact of future work and experiments conducted on several case studies to assess and enhance the proposed model, before it is ready for industrial use.

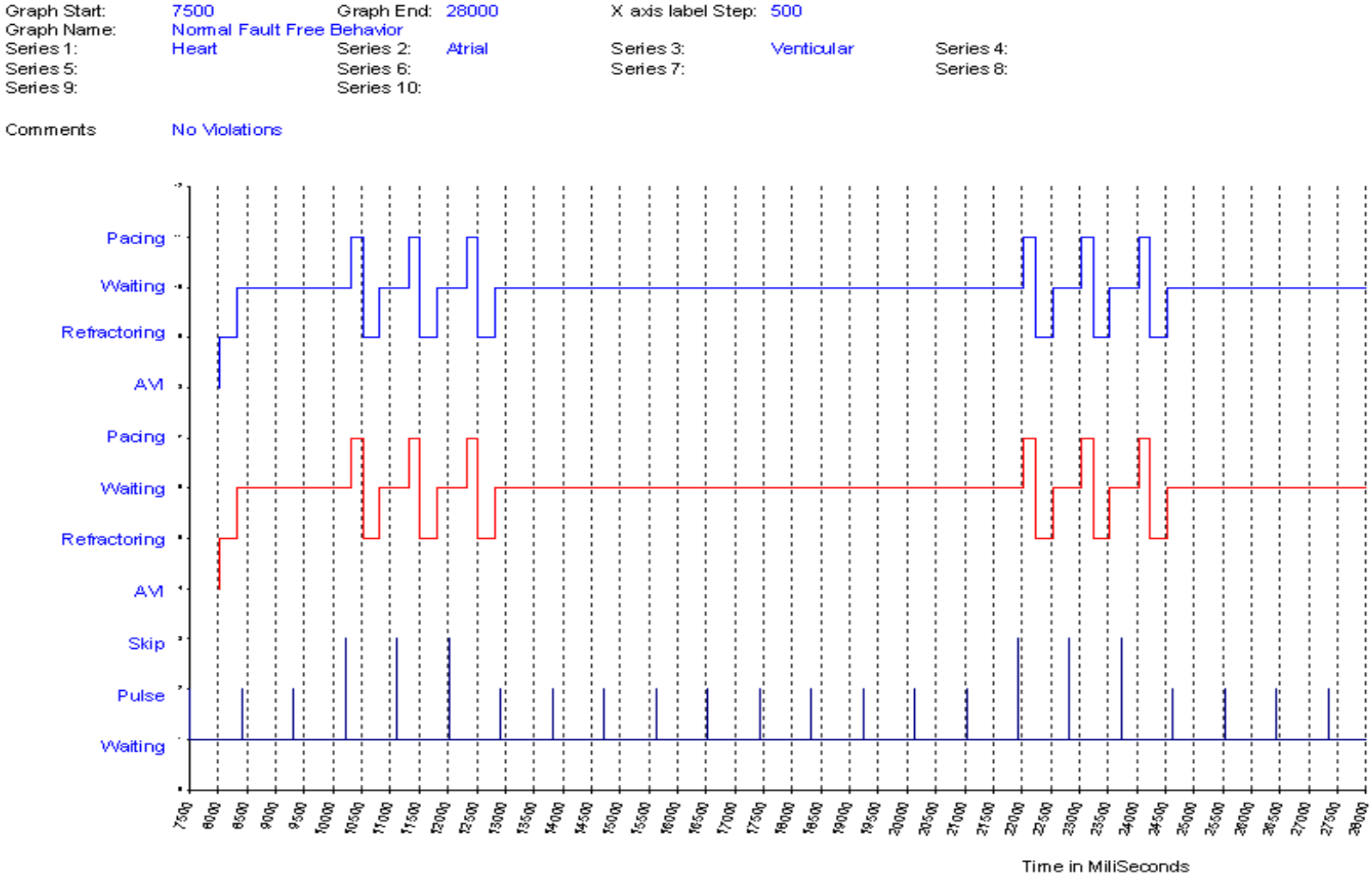

Figure 6.2 Pacemaker Expected Behavior (three pulses skipped)

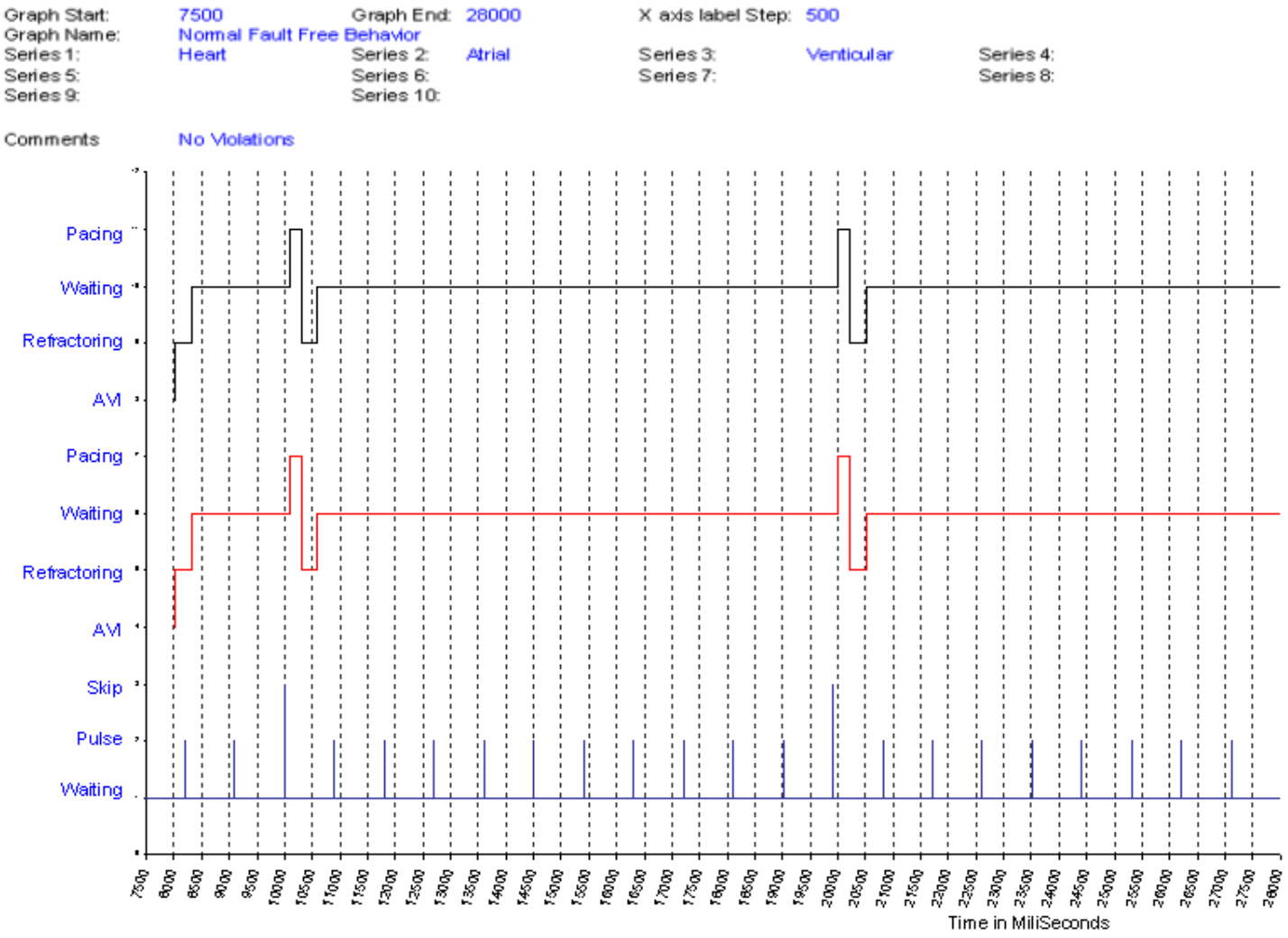

Figure 6.3 Pacemaker Expected Behavior (one pulse skipped)

Graph Start: 9000 Graph End: 23000 X axis label Step: 500
Graph Name: Atrial - AVI Waiting state swapped with the Pacing state
Series 1: Heart Series 2: Atrial Series 3: Venticular Series 4:
Series 5: Series 6: Series 7: Series 8:
Series 9: Series 10:

Comments: Faulty behavior (cyclic unconditional pacing)

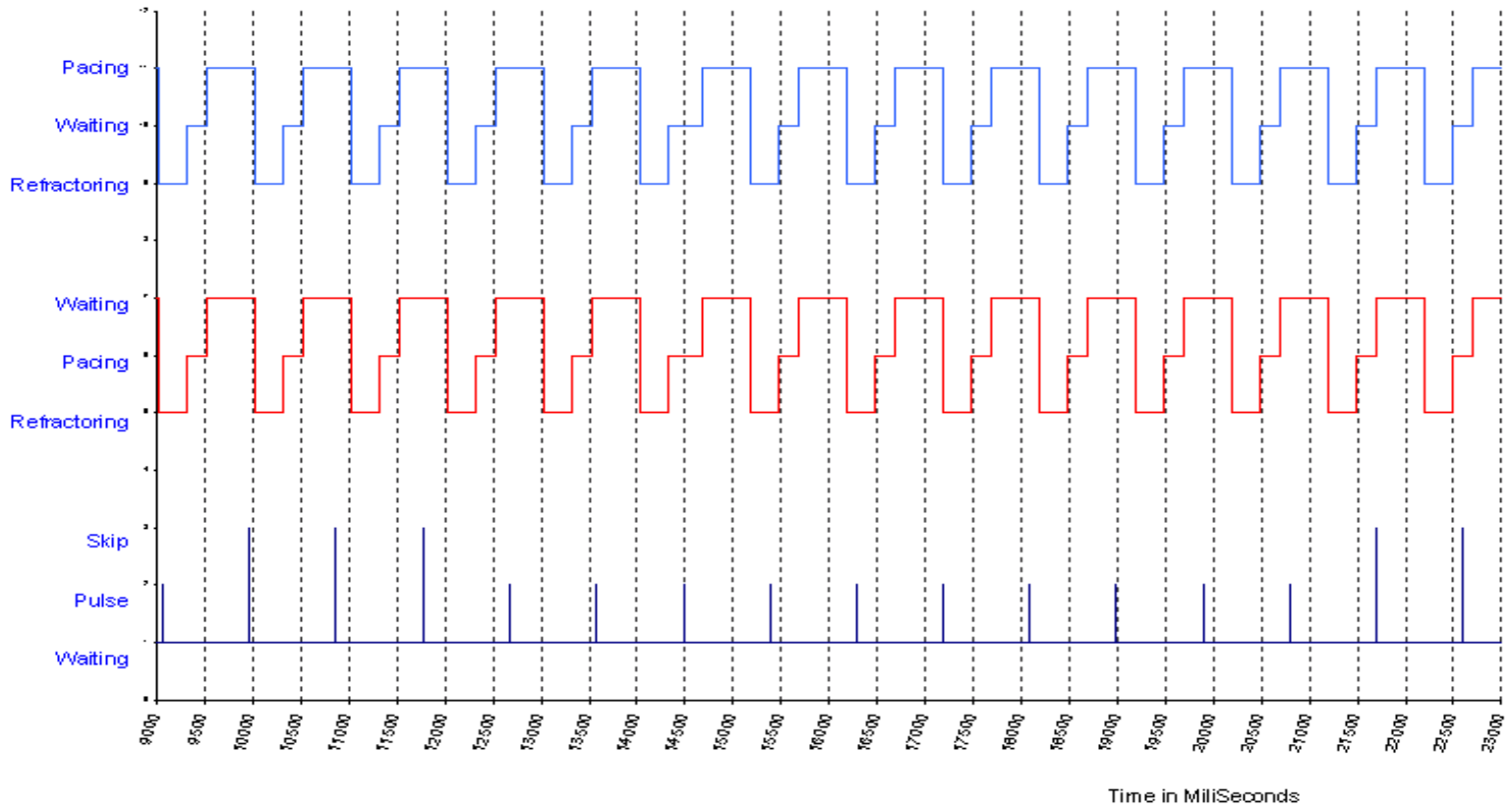

Figure 6.4 State Swap (three pulses skipped)

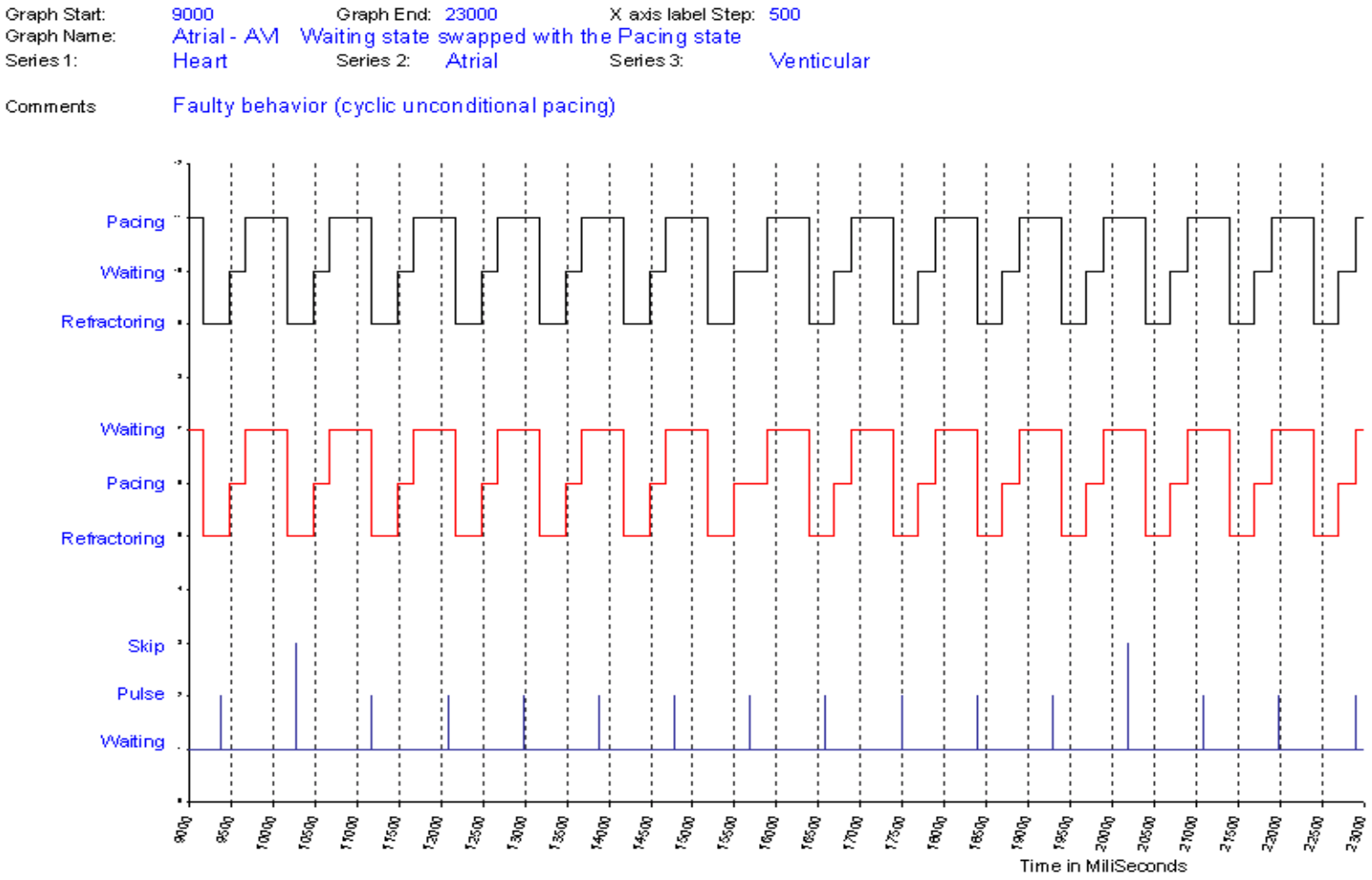

Figure 6.5 State Swap (one pulse skipped)

Graph Start: 7000 Graph End: 27000 X axis label Step: 500
Graph Name: Atrial - AVI Transitions GotVSense and Time-Out swapped
Series 1: Heart Series 2: Atrial Series 3: Venticular Series 4:
Series 5: Series 6: Series 7: Series 8:
Series 9: Series 10:

Comments: Faulty Behavior (The two components are Out synchronization

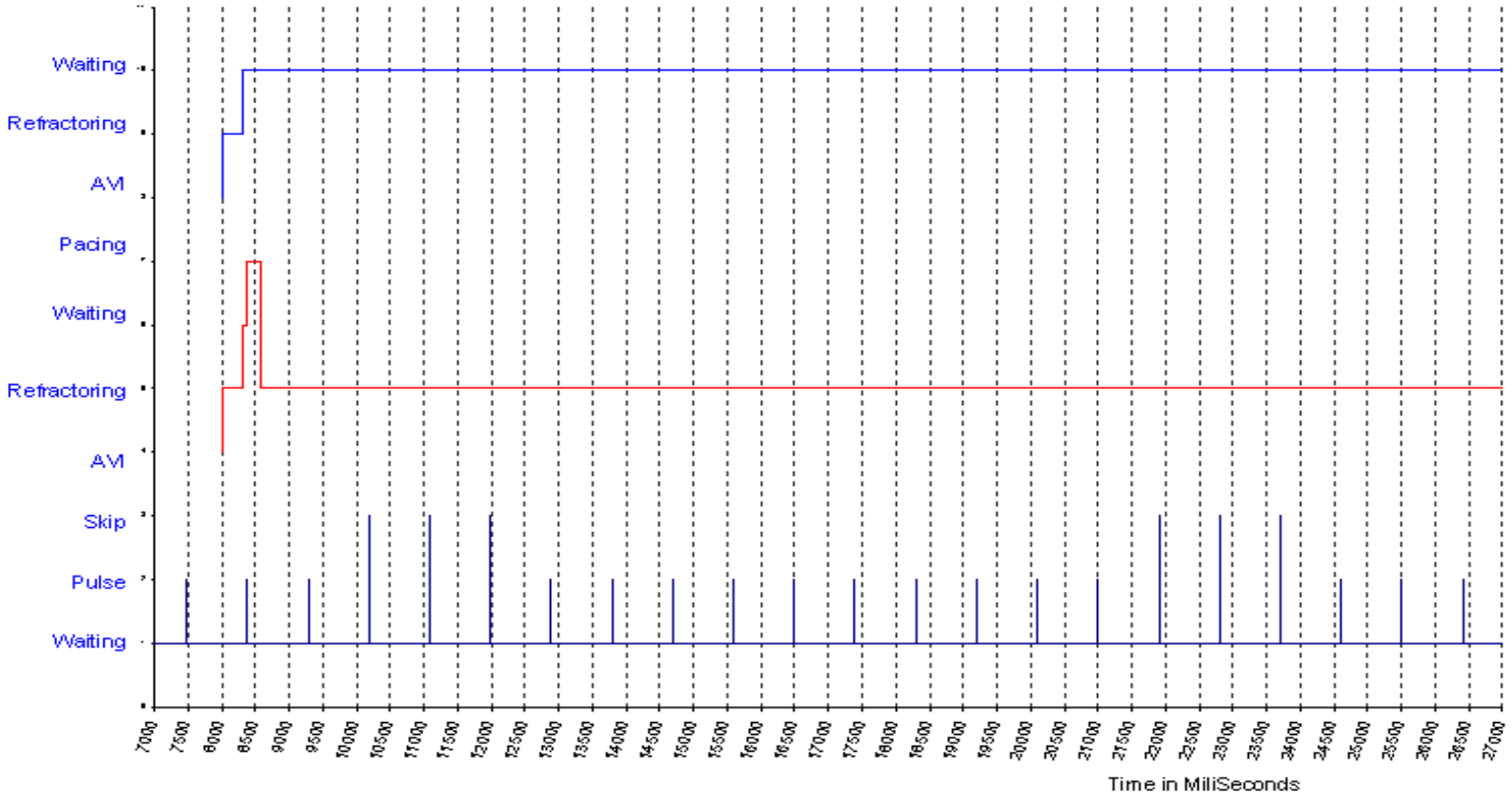

Figure 6.6 Transition Swap (three pulses skipped)

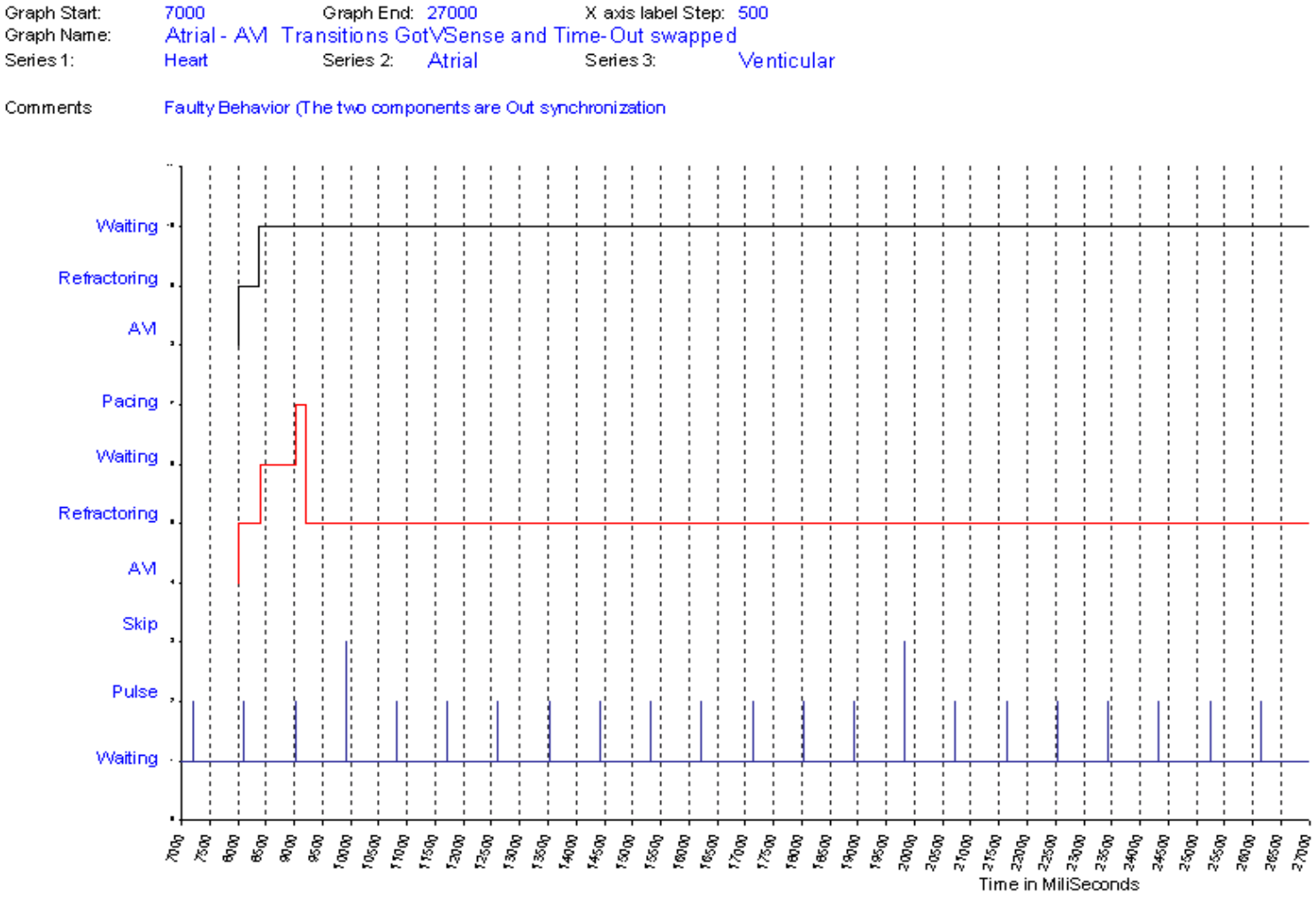

Figure 6.7 Transition Swap (one pulse skipped)

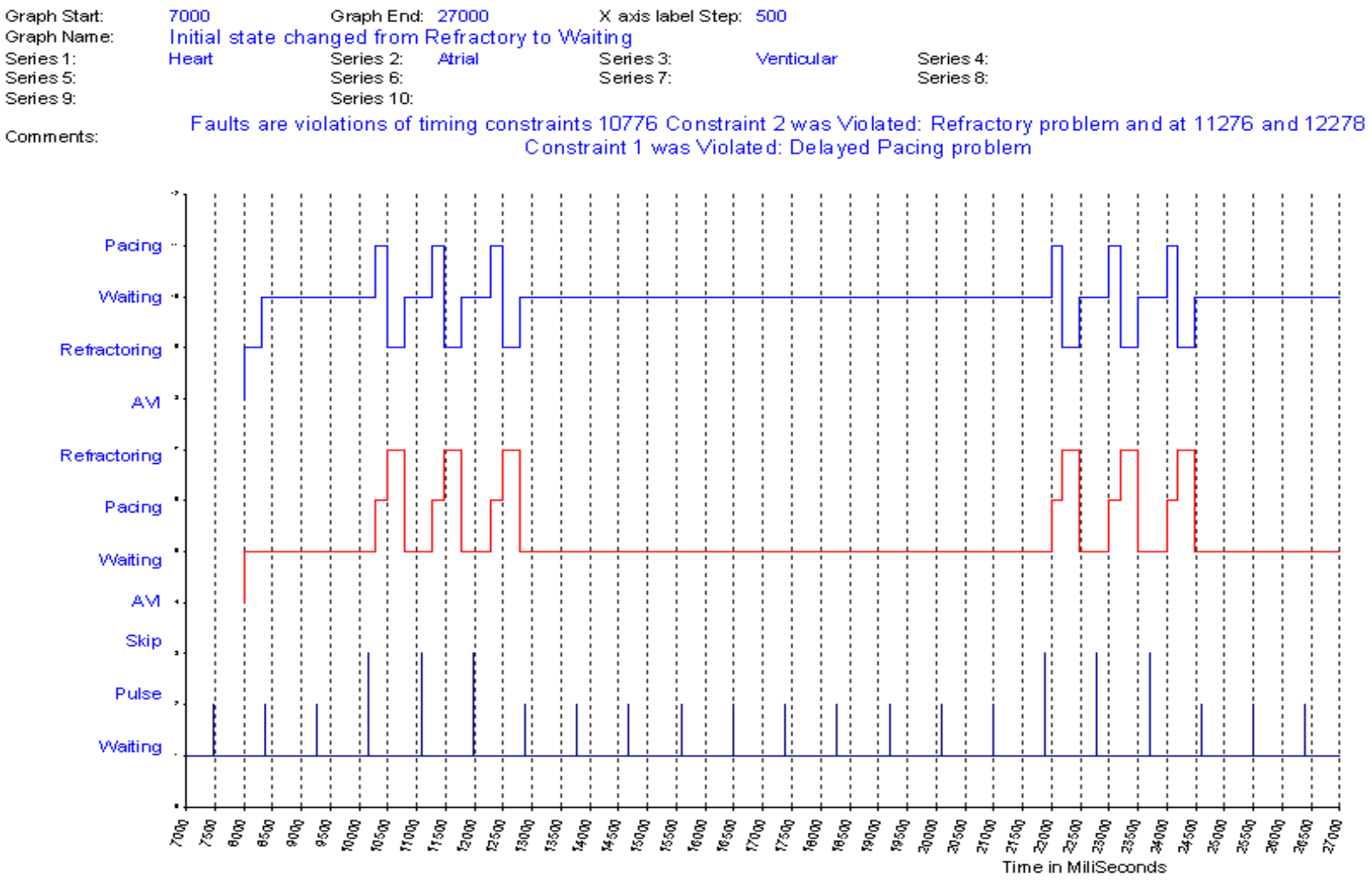

Figure 6.8 Initial Sate Swap (three pulses skipped)

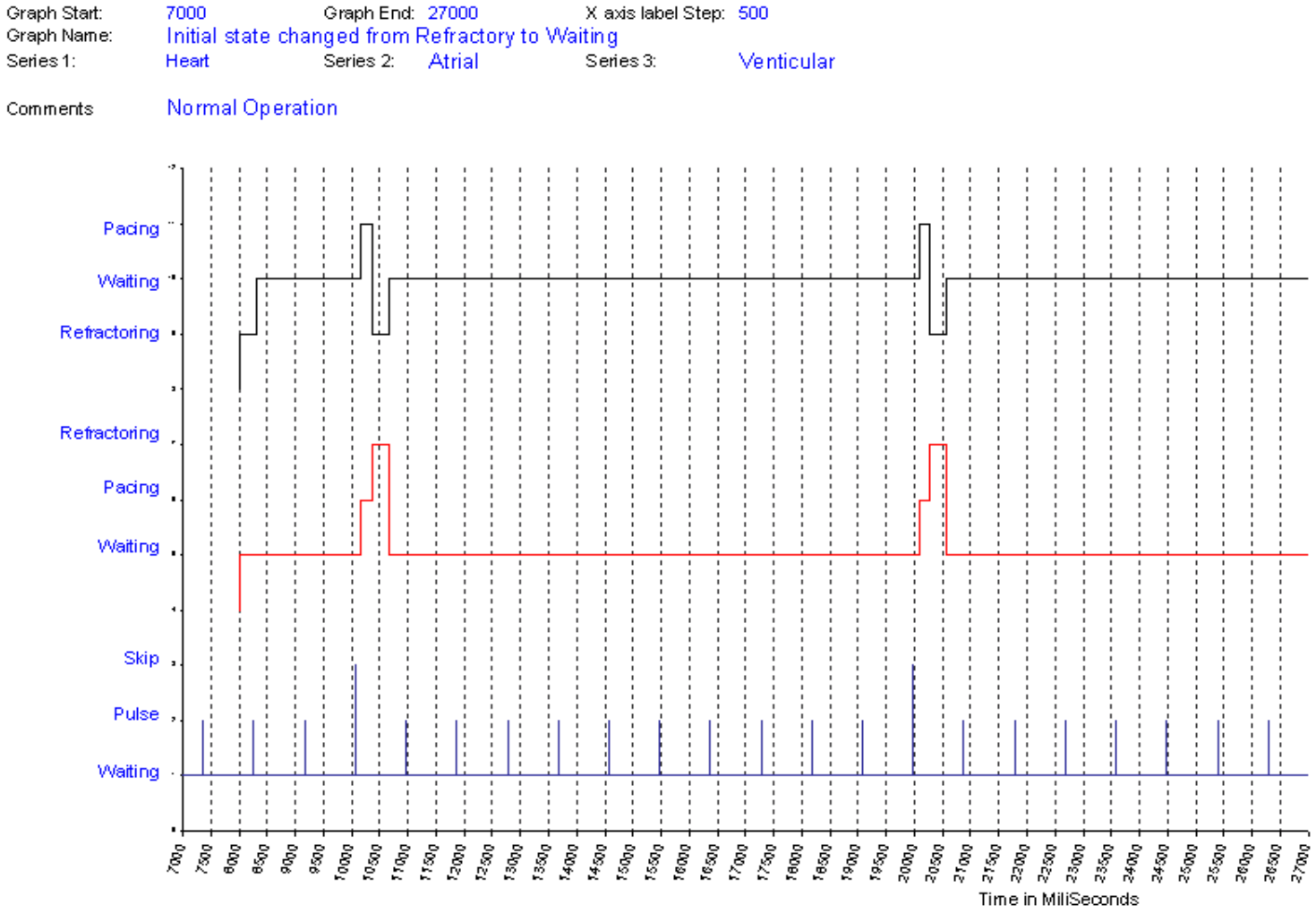

Figure 6.9 Initial Sate Swap (one pulse skipped)

Graph Start: 0 Graph End: 20000 X axis label Step: 500
Graph Name: Atrial - AVI The Trigger for the Time-Out Transition (out of Waiting State) is removed
Series 1: Heart Series 2: Atrial Series 3: Venticular Series 4:
Series 5: Series 6: Series 7: Series 8:
Series 9: Series 10:

Comments: Faulty Behavior, Pacing State is never visited

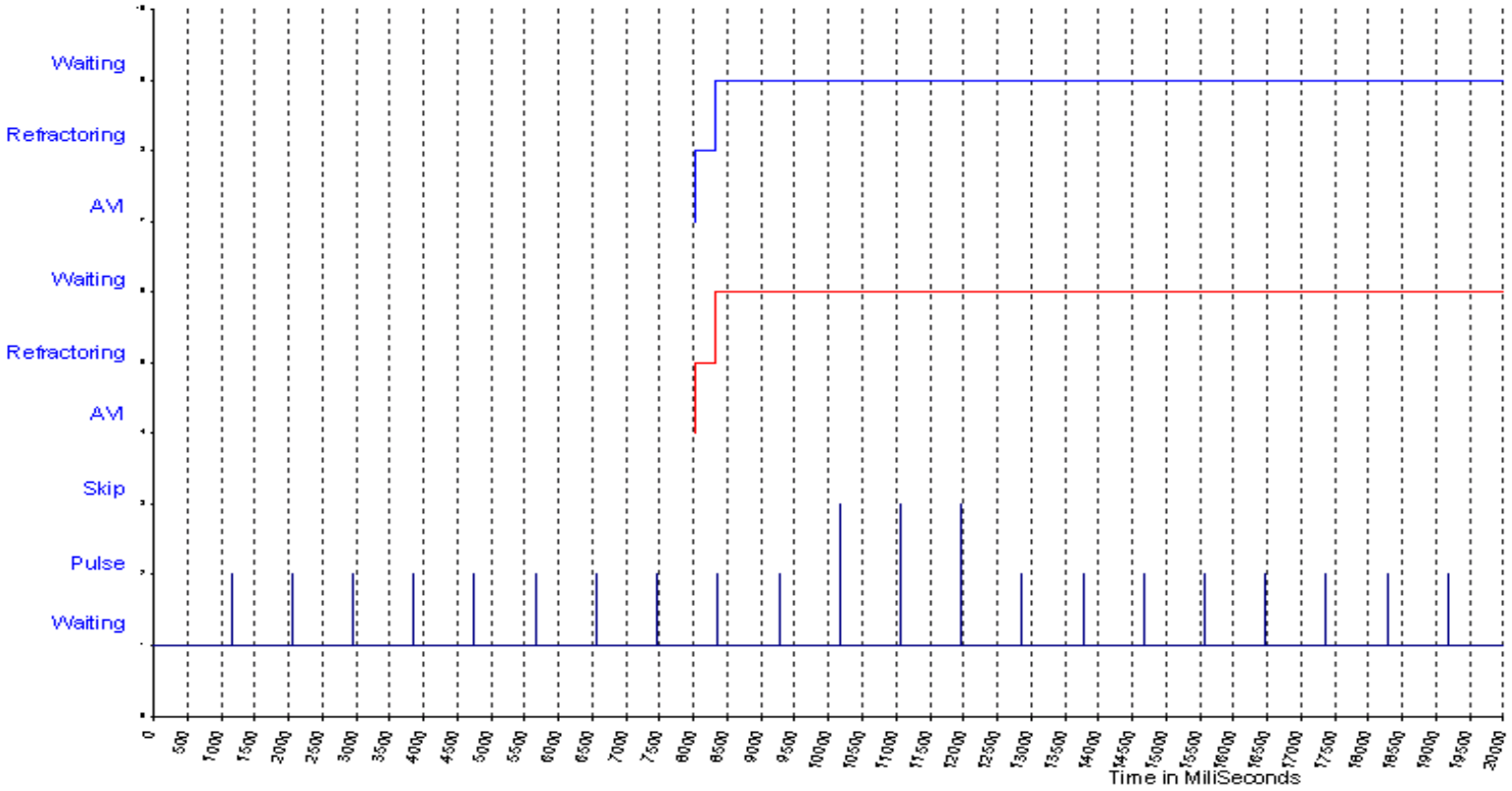

Figure 6.10 Null Trigger (three pulses skipped)

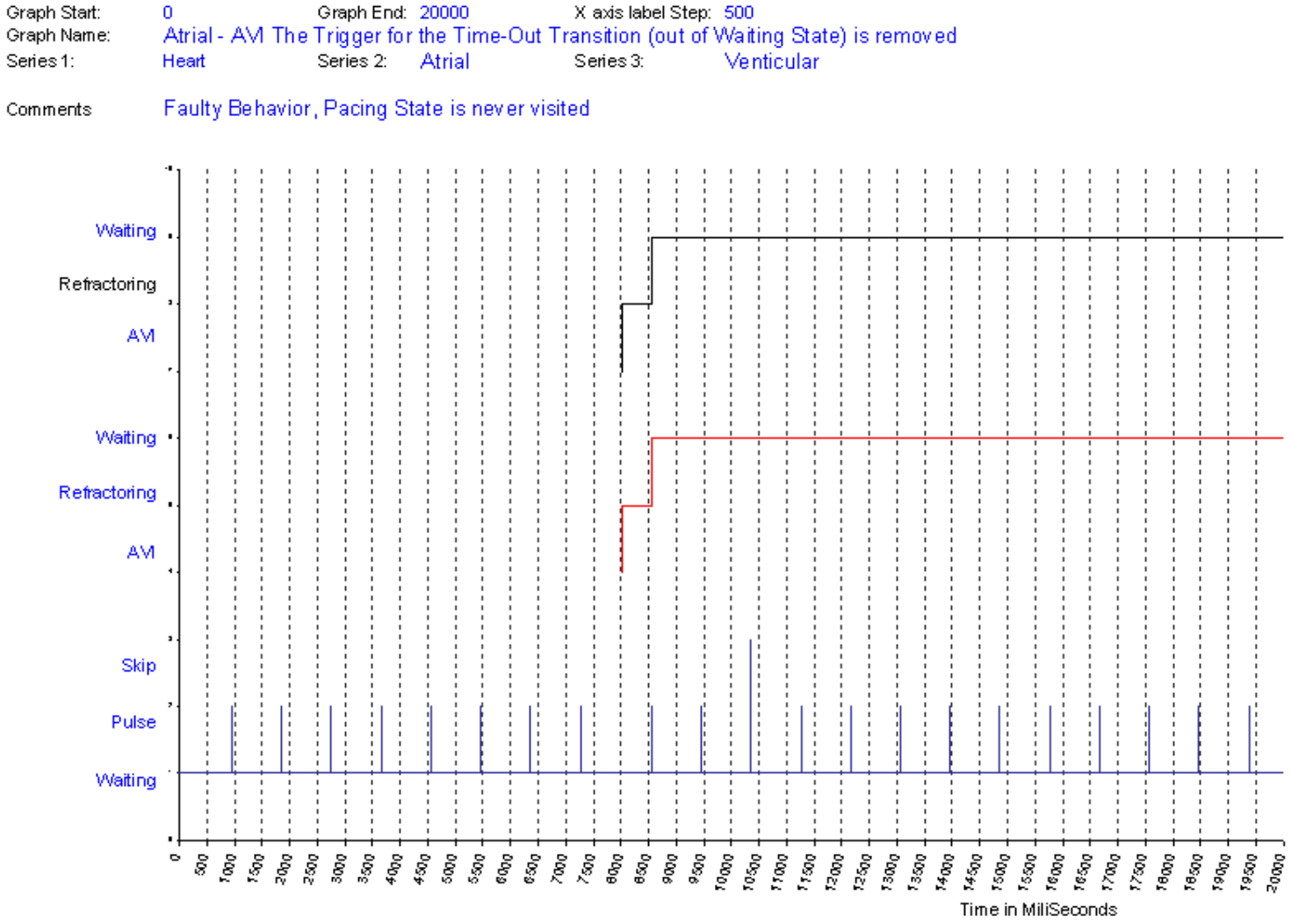

Figure 6.11 Null Trigger (one pulse skipped)

Graph Start: 5000 Graph End: 25000 X axis label Step: 500
Graph Name: Atrial - AVI the trigger for the Transition Time-Out is changed to Sense message from the Heart component
Series 1: Heart Series 2: Atrial Series 3: Venticular Series 4:
Series 5: Series 6: Series 7: Series 8:
Series 9: Series 10:

Comments: Faulty Behavior, Pacing when not required instead of when required

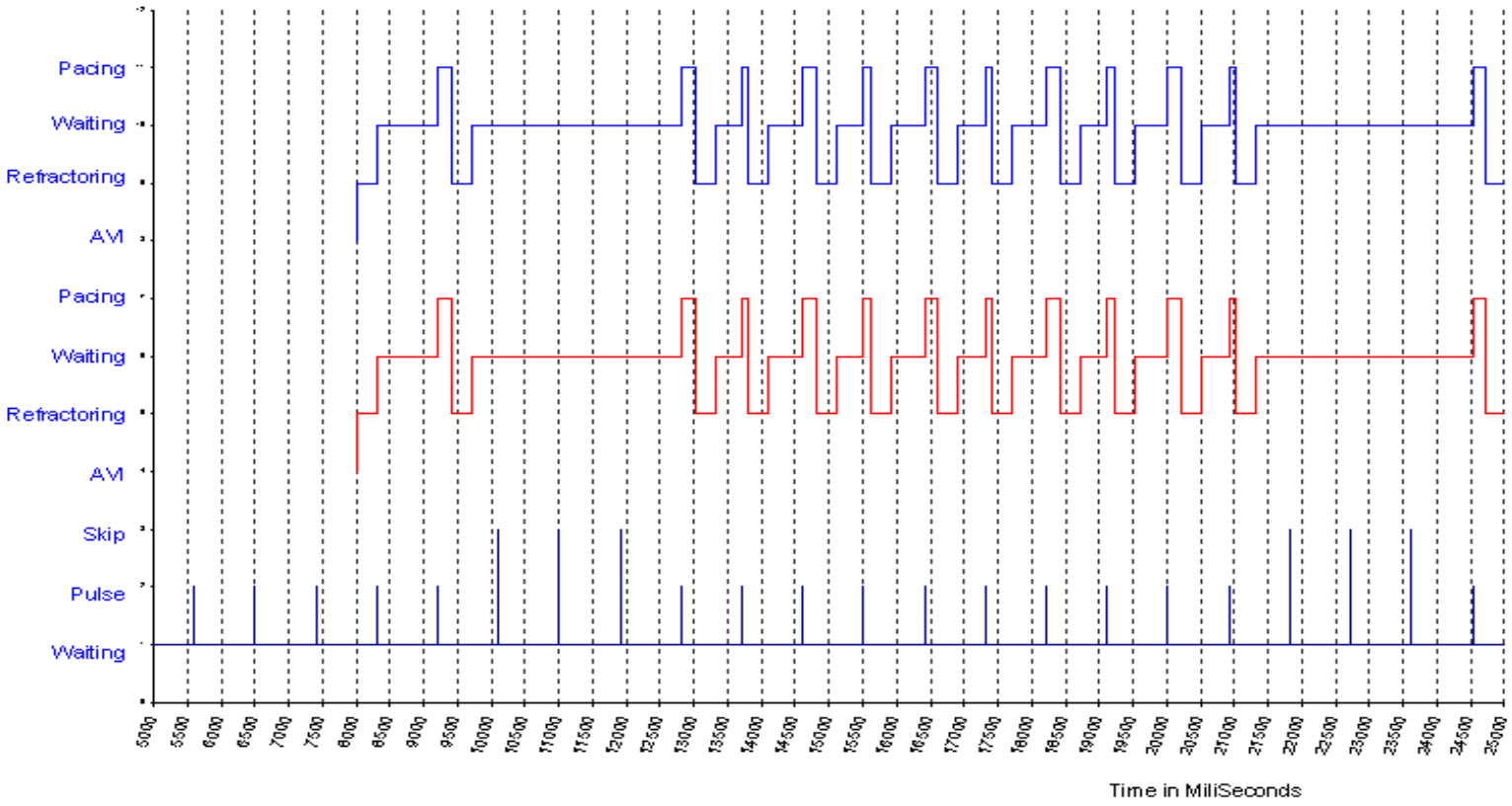

Figure 6.12 Trigger Swap (three pulses skipped)

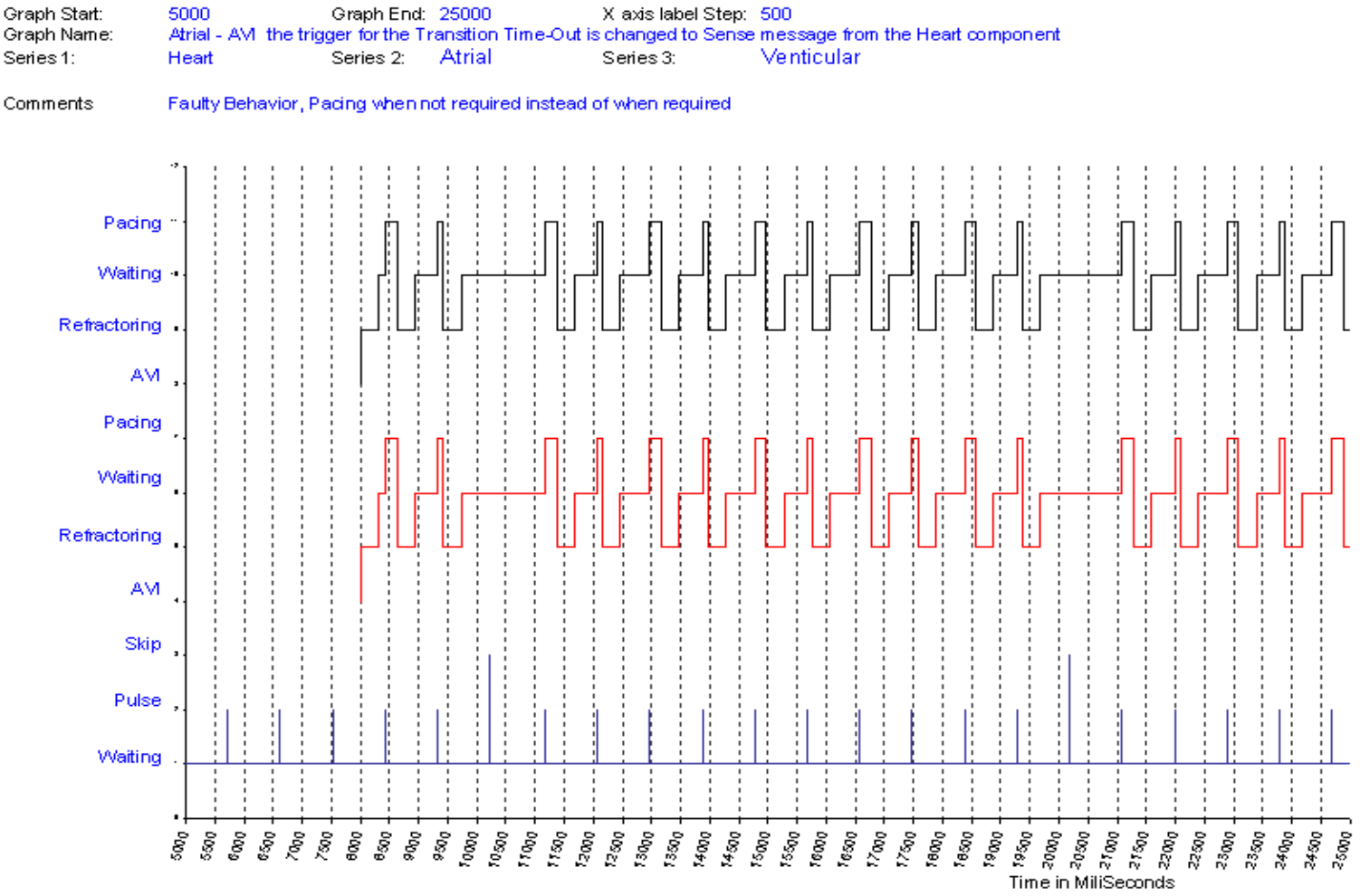

Figure 6.13 Trigger Swap (one pulse skipped)

CHAPTER 7: PERFORMANCE MODELING

7.1 Introduction

The importance of early performance assessment grows as software systems increase in terms of size, logical distribution and interaction complexity. Lack of time from the side of software developers, as well as distance among software model notations and performance model representation do not help to build an integrated software process that takes into account, from the early phases of the lifecycle, non functional requirement. From performance viewpoint, the validation of non functional requirements early in the lifecycle is an important and difficult task to accomplish. Early performance assessment allows us to build software that better fulfills performance requirements. This helps to reduce the risk of late detection of poor performance that would be hard to manage. Thus the necessity to provide a standard representation of information related to the performance (e.g., resource demand) in the UML framework is therefore ever more clear [17]. As a consequent step, this makes it easier to transfer UML models from design to performance analysis tools [27]. Several approaches for extending the UML notation to embed performance related information have been introduced

Tailoring the derivation of a performance model on a specific application domain, such as Client-Server systems, is the goal of [15], where a methodology is introduced (based on a performance engineering language developed by the authors) to make the distance between software developers and performance analysts shorter. A compiler of the language generates an analytic performance model. The derivation of performance models, based on Layered Queuing Networks (LQN), using graph transformation is presented in [18,19,20]. Specifically, the LQN model structure is derived from the software architecture description based both on informal description [20] and on UML Collaboration diagrams [19,18]. The generation of LQN model parameters is dealt with in [19] where Activity Diagrams are generated (by graph transformation) from Sequence Diagrams.

7.2 Our approach for performance modeling of Client-Server systems using the UML-RT notation

Most of above introduced approaches aim at extending the UML notation to easily translate UML models into well assessed performance tool notations. In this work we aim at filling the gap between UML model notation and performance model representation, by extending the capabilities of the environment described in chapter 2 (based on UML models), in essence we introduce an opposite process. We introduce new stereotypes representing performance related items, such as resource types and job dispatchers. They allow the software designers to homogeneously represent a software architecture integrated with a running platform, and parameterized with the resource demand that the components require. As an application example the simplified ATM banking subsystem has been considered for studying our approach. This is to prove the effectiveness in building, and simulating, software performance models. We use the simulative potential of the RRT tool to run software models that include items and parameters related to the performance of the model, so overcoming problems concerning analytical solutions of performance models. The visual notation underlying the RRT tool, that is UML-RT, has been therefore used to extend the set of stereotypes that the tool provides. The extension provides (a library of) new stereotypes that allow the representation of resource related items (such as CPUs, disks, etc.), in order to integrate in the same scheme the software structure and the resource requests of a software product. Thereafter a systematic approach has been sketched (using this additional library) to model software/hardware systems, in order to readily get insights on their performance profiles.

7.2.1 A layered software architecture

In [23] it is shown how the software architecture of a client-server application can be structured as a layered model. Components on the topmost level of the model are pure clients, the ones on the bottom are pure servers, all the other components are clients with respect to the lower level ones and server of the upper level ones. In figure 7.1 such a model is shown, where square boxes represent software components (namely tasks), with entry points, and round blocks represent resources.

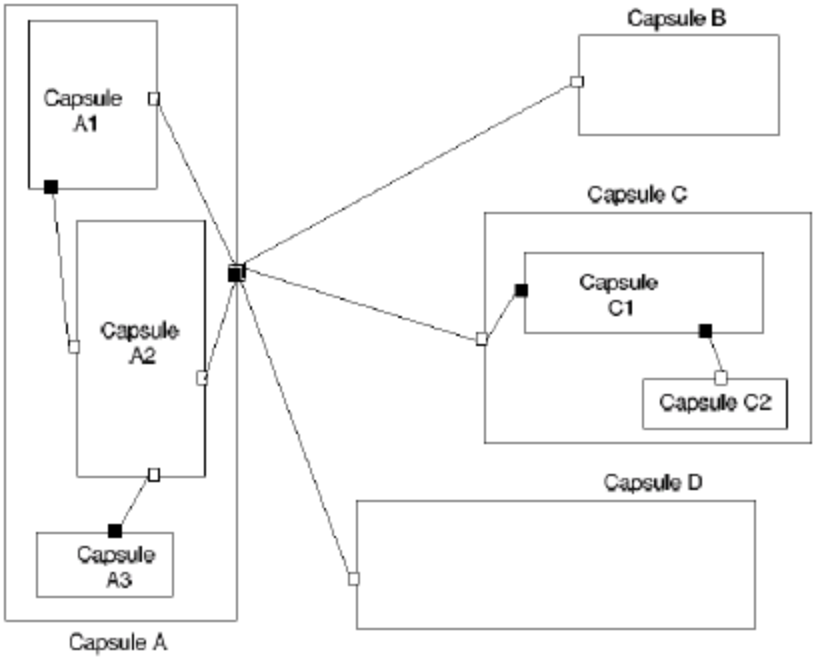

Figure 7.1 Transparent diagram of Capsules and embedded Capsules

In an UML-RT perspective such a layered structure can be obtained by merging together all the lowest level Capsules in the same diagram, that is from breaking down all the components that contain subcomponents. The resultant Capsule diagram represents the most detailed software architecture of the whole system. However a Capsule diagram presents two significant differences with respect to a layered model, that we discuss in the following:

1. The layered model is specifically designed for performance analysis and evaluation, so it contains also blocks that represent the resources. To every component a set of resources can be attached in order to represent the resource that the component requires (see figure 7.1). This is missing in a Capsule diagram, that looses the possibility to be used (as it is) for performance goals.

2. A Capsule diagram is supported by a set of State Diagrams, each describing the dynamic behavior of the component represented by a Capsule. This is missing in a layered model, that looses the possibility to simulate the dynamic internal behavior of its components.

The basic idea of our approach is providing a set of new stereotypes, based on the UML-RT notation, that can be used to represent resources in a Capsule diagram (e.g., CPUs, LANs, etc.). By embedding the appropriate set of resource instances into a Capsule diagram, the gap with a layered model is removed, and the additional value of a naturally simulative environment can be exploited to solve the performance model and get performance index insights.

7.2.2 Representing the extended software architecture

In order to represent in the same Capsule diagram the software architecture and the resources that the software components require, the diagram is conceptually split in two sides: the *Software* side and the *Resource* side (see figure 7.2). Capsules are in both sides, but while the ones in the software side represent software components, the resource side Capsules represent the resource that the considered architecture may need.

Upon the extension of the software architecture illustrated by the scheme in figure 7.2, a properly parameterized simulation of such scheme allows to evaluate the performance of the combined software architecture/resource system.

Three main issues have to be addressed to achieve this objective (and they are discussed in the following):

1. Building a basic structure of the resource side of the scheme.
2. Providing standard Capsule stereotypes to be used in the resource side.
3. Providing standard criteria to introduce the resource requests as additional items to the software side, without modifying the software architecture.

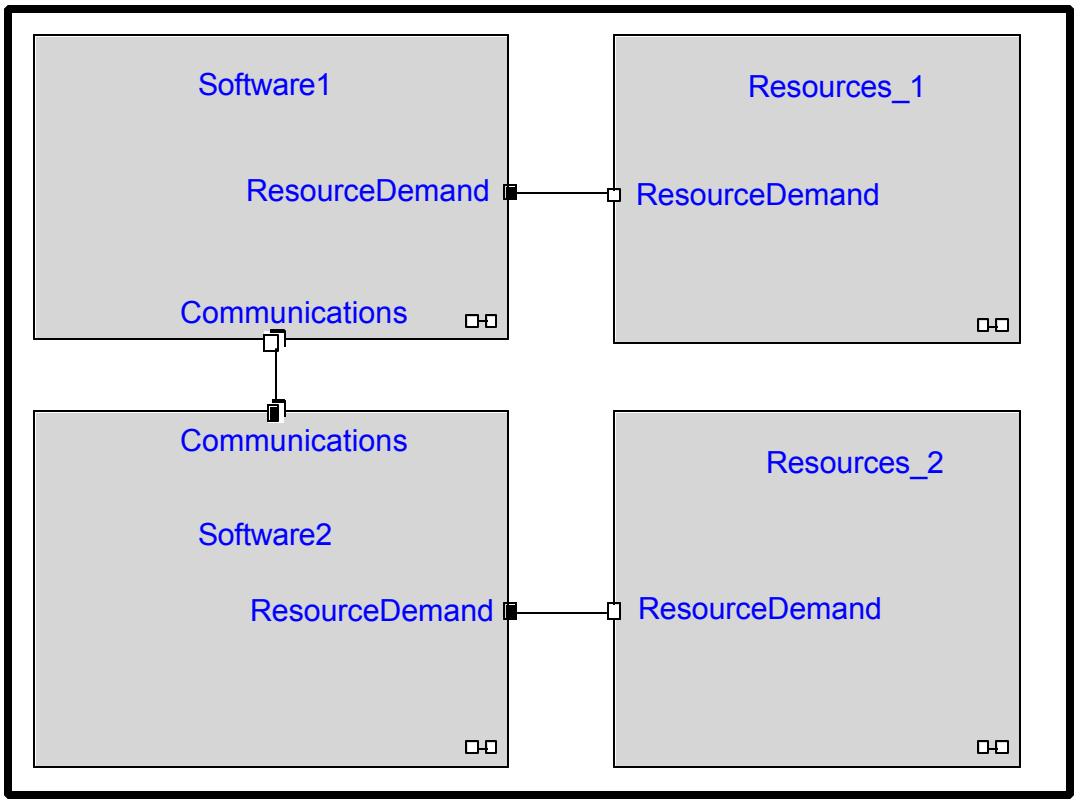

Figure 7.2 Generic two-sides Capsule diagram

In the upper side of figure 7.3 the Capsule diagram of the basic structure that we propose for the resource side of the scheme has been drawn. This basic structure is intended to be used, as it is, wherever a resource side is necessarily to be coupled to a software side. So, for example, the Capsule diagram represents the internal structure of both resource sides of figure 7.2, namely *Resource_1* and *Resource_2*. It is basically composed by a *Main Dispatcher* and a set of resource types.

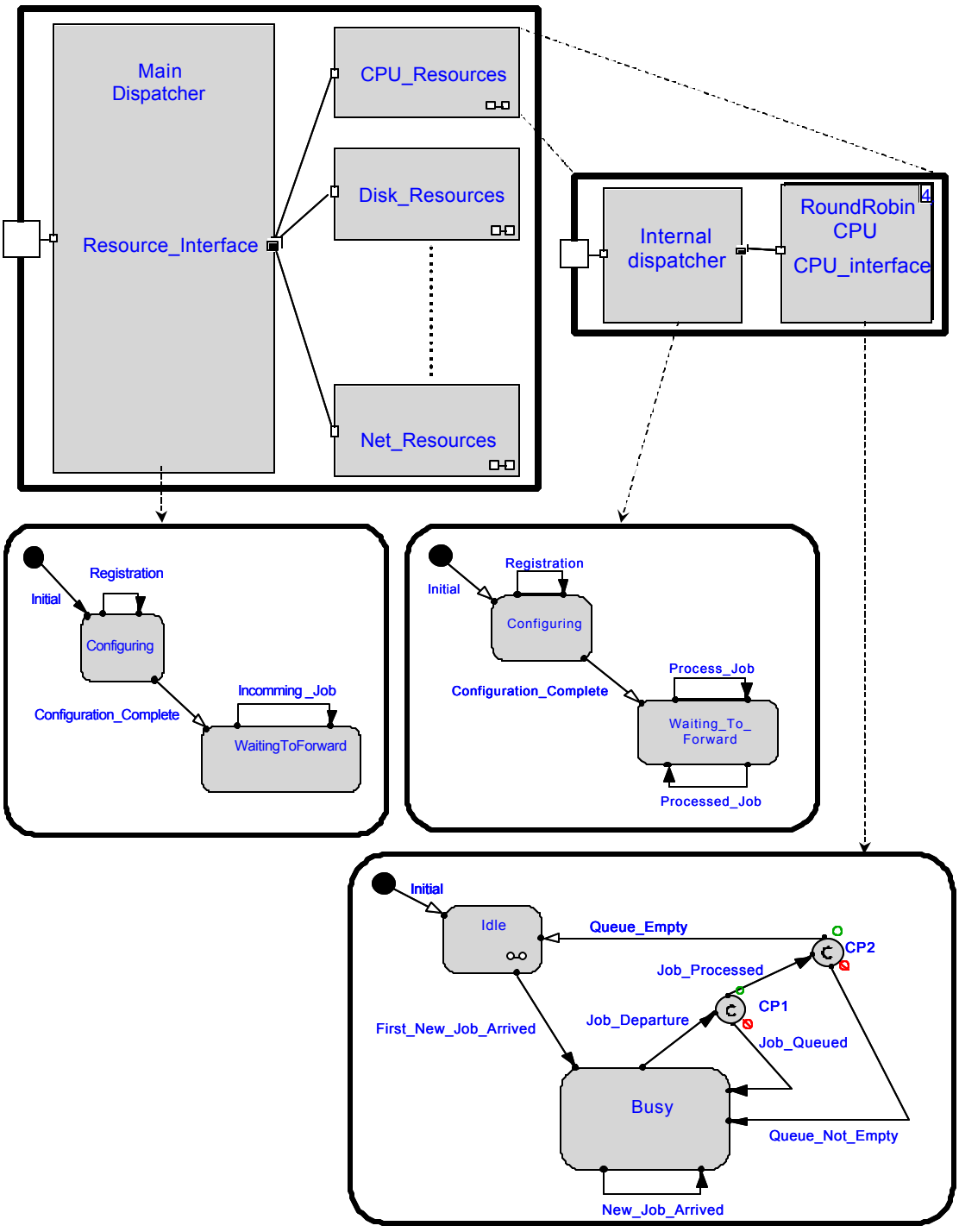

Figure 7.3 Basic structure (Capsule and State Diagrams) of the resource side.

The dispatcher is the Capsule in charge of receiving resource requests from the software side. We suppose (like in a Software Performance Engineering approach [29]) that every resource request has been produced by a software block (that is a set of operational steps), and includes the amount of every resource type needed to execute that software block (e.g., number of CPU instructions, number of disk blocks, bytes to be transferred on a network, etc.). Upon receiving a request, the dispatcher schedules, in a given order (where the order of resource consumption is here supposed do not affecting, in average, the final performance measures; however the dispatcher can be modified to take into account a specific ordering), the visits to the resource types needed. The *Resource_Interface* port in figure 7.3 is a multiport, that is a port with a given multiplicity. This contributes to the generality of our scheme with regard to the number of resource types that can be considered. Labels in figure 7.3 indicate the type of resources considered, but the implementation of this scheme allows to add (delete) a resource type by simply introducing (eliminating) a new Capsule and modifying the *Resource_Interface* multiplicity.

The internal structure of any resource type Capsule is quite standard as well. As shown in figure 7.3, where the *CPU_Resources* has been graphically expanded, every resource type Capsule contains an *Internal Dispatcher* and a set of actual resource instances. In the figure we show, as an example, the case of four CPUs, where four is the multiplicity given to the *CPU* Capsule (i.e., the number of resource instances) and the multiport connecting them to the *Internal Dispatcher*. Upon this "low level" dispatcher receiving a request of a specific amount of resource type it manages, basing on prior knowledge (e.g., speeds of different resource instances, queue lengths, previous request distribution) it schedules a job for a resource instance and notifies it by sending a message to the latter. When the requested amount has been consumed in the resource, the notification is sent back to the *Internal Dispatcher* and then forwarded to the *Main Dispatcher*; the latter checks whether the complete resource request of the software side has been satisfied or other resource types remain to be consumed. In the next section we show how to originate a resource request from the software side.

Basically in figure 7.3 have been introduced three new stereotypes (as Capsules): a high level dispatcher *Main Dispatcher*, a low level dispatcher *Internal Dispatcher*, and a *CPU* resource. In the lower side of the figure the State Diagrams of these stereotypes are shown.

For sake of conciseness and readability, we do not discuss the details of the dispatchers' State Diagrams, rather we focus on the *CPU* one. The CPU is modeled as a queued service center that extracts jobs from the queue following a quantum based round-robin strategy [14,13]. In the "idle" state the queue is supposed to be empty and no job is being served. Upon the arrival of a job, the CPU becomes "busy" and it returns to the idle state in any moment the queue is idle and no job is being served. Two state transitions originate from the busy state. In case of a new job arrival the corresponding transition only serves as update of the queue length and contents. In case of a job departure from the service center (either due to the quantum expiration or due to the end of service requested) there are two conditions to be orderly checked, namely *CP1* and *CP2*. First the residual amount of resource requested is read: if zero then the job has been completely processed and it can leave the CPU, else it has to be queued again (i.e., round-robin strategy) in order to be served later for at least one more quantum. In case of job processed an additional check is needed: if there is at least one job waiting into the queue then the first job is extracted and processed (i.e., the CPU goes again in a busy state), else the CPU returns to the idle state.

In a similar way a Capsule stereotype can be introduced for any type of resource type that contributes to build up a (possibly distributed) modern hardware platform (e.g., mass storage, wired network, etc.), provided that the corresponding State Diagram is also given. In any case the resource side of our scheme is open to represent whatever number of resource types with whatever number of instances, the only bound being the actual scalability of the modeled software/resources system.

Issue 3. aims at keeping "non-invasive" our technique, in the sense that the validation task of non functional performance requirements must be conceivable on whatever (existing or under design) software architecture, without affecting its generation process and its final structure. This means that the information related to the performance evaluation has to be fully additional to the software architecture, and therefore criteria have to be introduced to rule the addition of such information. We have described in this section how a resource request is handled from the resource side. A

resource request from the software side viewpoint is basically a message that leaves the software side and reaches the appropriate set of resources in the resource side. In the next section we show the criteria that we use to build and send such message from the software side, and to manage the associated reply.

7.3 Example: Simplified Automatic Teller Machine (ATM) banking subsystem

The application example that we have considered is a simplified ATM banking subsystem. The ATN a bank-card and requires a password for user authentication. Users can perform two transactions at the ATM: cash withdrawal, balance check. The ATM communicates with a computer server at the host bank that verifies the account and processes the transaction. At the end of the transaction some final operations are executed and user's card is returned.

7.3.1 ATM Architecture

We consider a simplified Automatic Teller Machine (ATM) banking subsystem for experimentation purposes. In figure 7.4 a nested view of modeled subsystem is shown. The topmost bold box represents the whole system built up by three types of components (i.e., the gray boxes). The first component *ServerSoftware* representing the central processing unit, at the host bank, of the subsystem and the second type of components is the *ATM* representing the remote terminal client each include the *ATM_Software* which is a UML model of the software running in the ATM and *ATM_Peripherals* representing the ATM hardware and the current user. This is more clear when viewing figure 7.4 as a two levels of nesting, where the *ATM* is shown to be built up by two components, to the left is the emulation of ATM Peripherals and users (*ATM Peripherals*) and the to the write, the *ATM_software*. The latter, in turn, contains three basic components: *BalanceTransaction, Authenticator* and *WithdrawalTransaction*

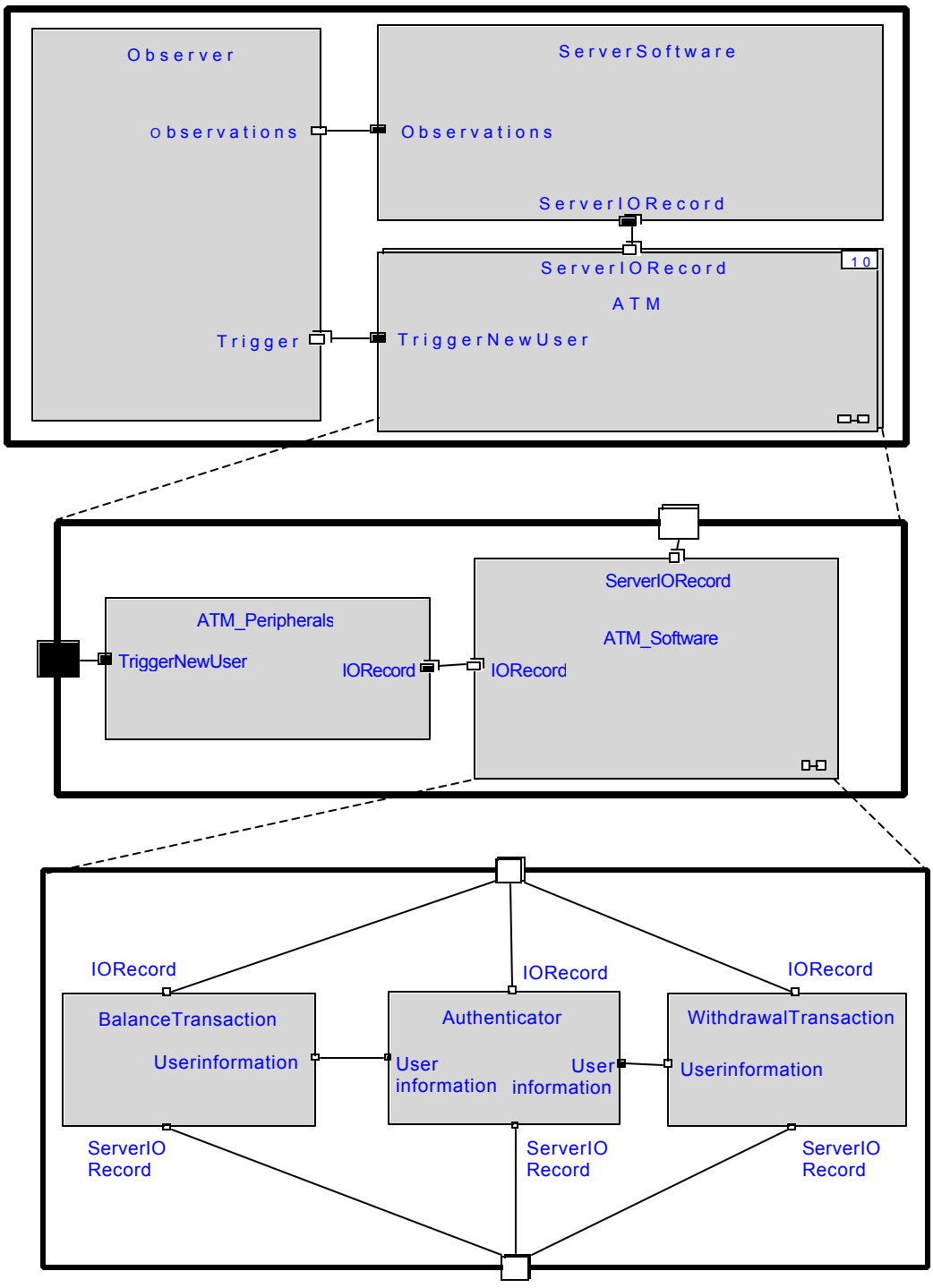

Figure 7.4 ATM software Architecture (3 level nested view)

All the accounts information and transactions are maintained and processed at the *ServerSoftware* which is modeled as an emulation (sending messages in respond to the received messages and according to the Sequence Diagrams). The basic behavior is as follows: the ATM accepts a bank-card and requires a password for user authentication. Users can perform two transactions at the ATM: cash withdrawal, balance inquiry. The ATM communicates with the *ServerSoftware* to validate the users and process the required transaction. At the end of the transaction some final operations are executed and user's card is returned. The *ATM_Software* is the component that directly interacts with the user represented as part of the *ATM_Peripheral*. Several *ATM* are instantiated using the multiplicity factor of the tool. The *ATM* has a multiplicity of ten in this case (figure 7.4 top level), meaning that ten instances of the same type of components are allocated. These components interact with the *ServerSoftware* component whenever a transaction requires access to data residing on the host bank.. On the other hand, there is a unique instance of *ServerSoftware*, meaning that all requests of service (coming from whatever ATM instance) are processed by one *ServerSoftware* component, where therefore contention can be high and performance problems are to be investigated (chapter 7).

This simple architecture allows studying the scalability of such a scheme by directly increasing the number of ATM instances. The *Observer* component is not part of the ATM subsystem, but it performs the standard function (described in chapter 4) of starting and setting simulation sub runs, as well as generating the users and the collecting the simulation statistics. It generates users in the form of trigger messages containing the user type and basing on stochastic distributions.

7.3.2 Sequence Diagrams

The ATM software architecture represents the static behavior of the system, by showing components and connectors. In order to describe the dynamic behavior of the system classical UML diagrams were built, such as Sequence and State Diagrams. Five Sequence Diagrams were derived from two scenarios: The balance scenario and the withdrawal scenario. The Five sequence diagrams are:

1. *Use_Denied*: (Appendix C, figure 1)
2. *Balance*: (Appendix C, figure 2)

3. *Balance_Print*: (Appendix C, figure 3)
4. *Withdrawal*: (Appendix C, figure 4)
5. *Withdrawal_Print*: (Appendix C, figure 5)
6. *Withdrawal_Denied*: (Appendix C, figure 6)

7.3.3 State Diagrams

It is required for the RRT tool (for simulation purposes) to have at least State Diagrams modeling the internal behavior of the lowest level components. Thus we present bellow the state diagrams of the *Authenticator* (figure 7.5), *BalanceTransaction* (figure 7.6) and *WithdrawalTransaction* (figure 7.7) components.

The lower side of figure 7.8 shows the State Diagrams of *Authenticator* and *WithdrawalTransaction components*. The upper side shows two out of the five Sequence Diagrams of the ATM subsystem. They represent a successful and an unsuccessful (without statement printing) withdrawal transaction (including and after the authentication operations). Note that the components acting in these diagrams correspond to lowest level Capsules in the ATM software architecture of figure 7.4. It is perceived (in general and applied to the ATM State Diagrams) that the overall behavior of a component can be obtained by merging the behaviors of the component in all the different Sequence Diagrams it is involved. Figure 7.8 is not complete (refer to [3] for further details), but it gives an idea on how the translation from a set of Sequence Diagrams to a set of State Diagrams describing the behaviors of the components involved.

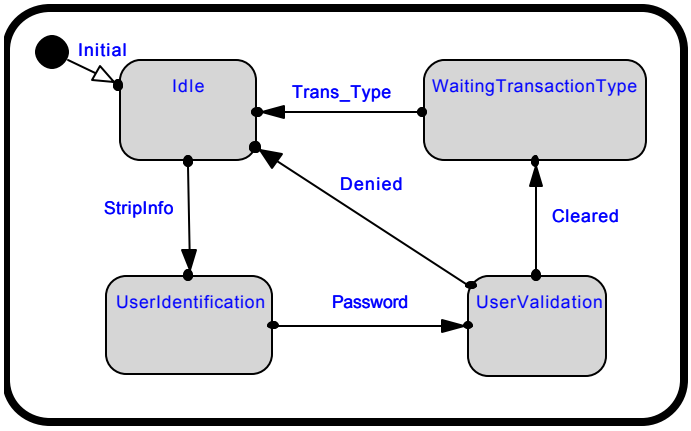

Figure 7.5 Authenticator Component State Diagram

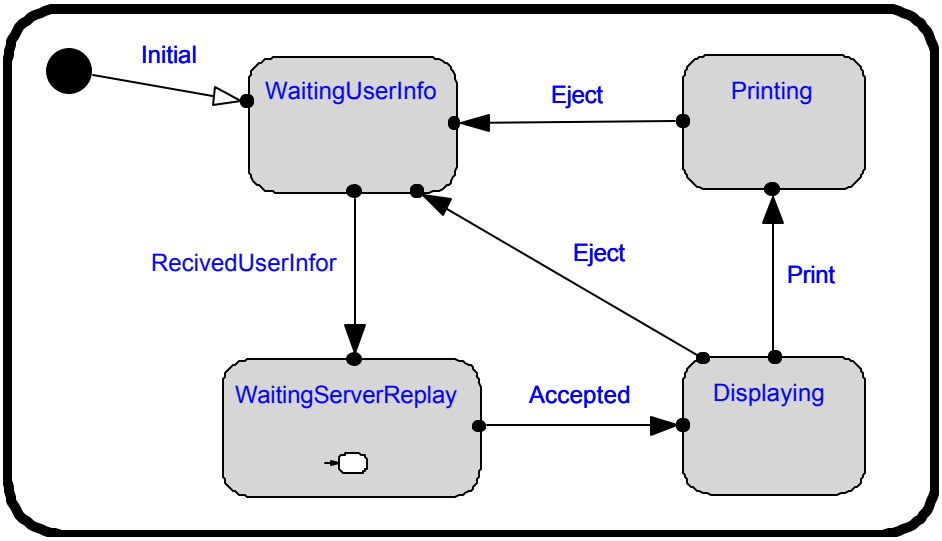

Figure 7.6 *BalanceTransaction* Component State Diagram

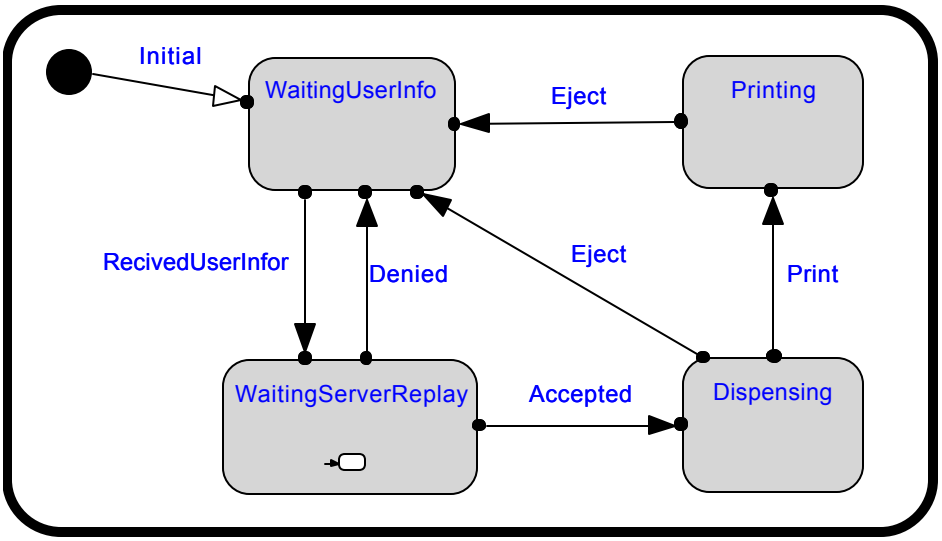

Figure 7.7 *WithdrawalTransaction* Component State Diagram

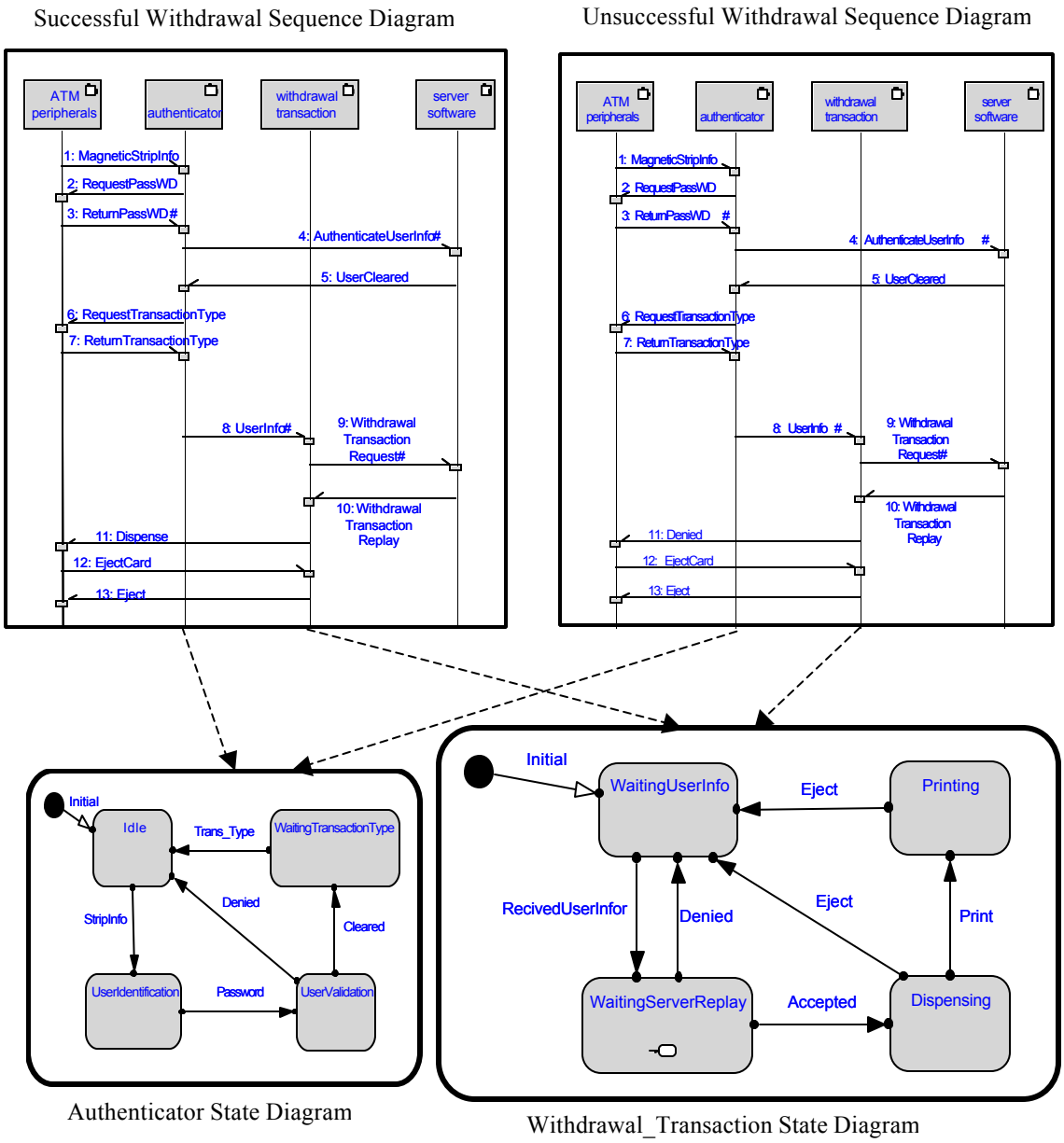

Figure 7.8 Sample of Sequence Diagram to State Diagram translation

7.3.4 Performance Modeling for the ATM Example

Applying our approach to the ATM banking subsystem adds two types of Capsules: *ServerResources and ATMResources* to the software Architecture (3 levelnested view) presented in figure 7.4, as shown in figure 7.9. These two components are the resource side (left hand side in figure 7.2 is added to figure 7.4). The *ServerResources* are the model representing the resources required/consumed during the activities of the *Serversoftware* based on the messages sent from the ATMs across the bank network where *ATMResources* are the model representing the local resources required/consumed during the activities of the *ATM_Software* based on the messages sent from the *ATM_Peripherals*.

We describe how a resource request is generated in the software side and how the associated reply (from the resource side) is handled. Let us associate each resource request to a software block. Independently of the level of detail used, in a Sequence Diagram (such as the ones in figure 7.8) a software block is the set of operations that a component performs to process an incoming interaction. From a graphical viewpoint a software block is the segment of a component lifeline that starts with an interaction entering the component and ends with the next interaction exiting the component (We are here assuming that a "service request" to a software component is always followed by either a reply to the request or a further request produced by the serving component, but this is not true in general.). In figure 7.8 all the software blocks start with a small shaded square box. In order to accomplish to the task required by an entering interaction, the component has to perform several steps, that can require the use of different resource types (e.g., CPU, disk, etc.). The resource request that corresponds to a software block is indeed a vector with each cell containing the amount of a resource type requested. This vector is built, as soon as the software block is entered, basing on prior knowledge of the designer. How many CPU instructions constitute a software block, or how many accesses to disk it needs, is a know-how that the software designer must have (at least in average) in order to fill the resource request vector. Instead, if performance of an existing software is being evaluated then the average amount of resources requested by every software block can be off-line measured. After the vector building, the request must be addressed to the appropriate component in the resource side, and this is done with a message sending. Therefore in figure 7.8, for example, the software blocks belonging to

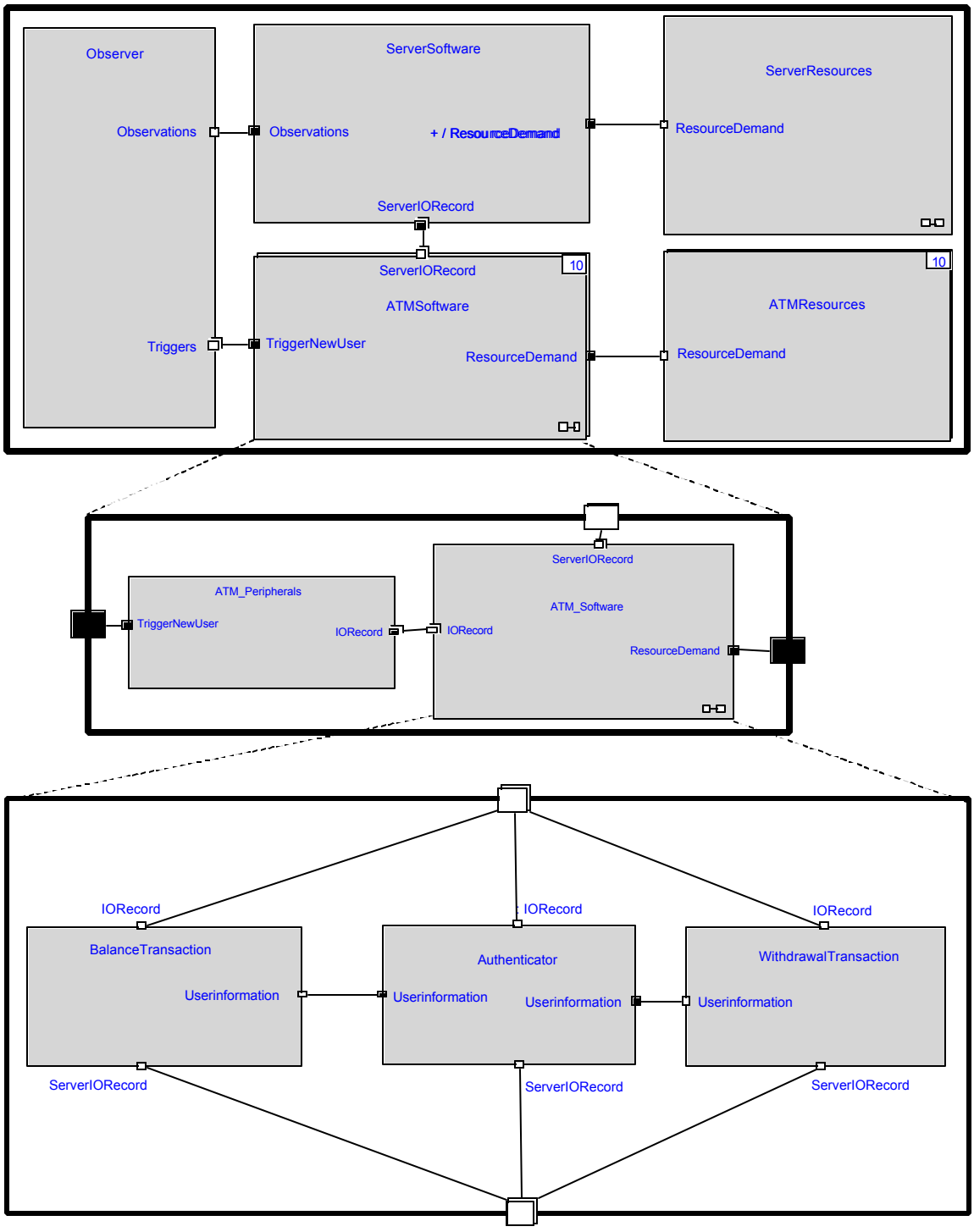

Figure 7.9 ATM software Architecture (3 level nested view).with the Resource side components

the *Authenticator* component address their requests to the corresponding *ATMResources* instance of figure 7.9, while the ones belonging to the *ServerSoftware* component address them to the *ServerResources* of figure 7.9.

In figure 7.9 an extended partial Sequence Diagram is drawn, in order to show the dynamics of a resource request. The five initial common steps of the Sequence Diagrams in figure 7.8 have been considered, and we have focused on the resource request originated by the software block delimited by steps 4 and 5 in the *ServerSoftware* component. Lifelines of Capsules belonging to the resource side have been appended and the sequence of interactions due to the resource request has been explicitly drawn. The remaining of the figure is self-explaining.

Given the close correspondence between Sequence Diagrams and State Diagrams (as shown in figure 7.8), it is straightforward that, in order to build and deliver a resource request vector (in the software side) only modifications to the State Diagrams of Capsules are necessary. In particular no additional states or transitions must be introduced, rather additional code (building and sending the vector) must be wrapped up into State Diagrams in order to fire a remote transition in the resource side that receives the request. Analogously, the termination of the request processing from the resource side originates a message that enables the requiring software Capsule to perform the next operations/interactions.

We now explore a systematic criteria to embed into a State Diagram the code corresponding to a resource request vector, building and delivery. For example, let us consider the software block, shown in both Sequence Diagrams of figure 7.8, along the *WithdrawalTransaction* component, that starts with the incoming transition labeled *8:UserInfo* and terminates with the outgoing transition labeled *9:Withdrawal Transaction Request*. In the *WithdrawalTransaction* State Diagram this software block corresponds to the actions performed while entering the *WaitingServerReplay* state. It is intuitive that code must be added to the entry point of this state aimed at building and sending the request resource vector of this software block.

The fact that above considerations imply that, as claimed in the issue 3. of section 7.2.2, no modification of the software architecture at all is required in our approach to embed information related to the performance analysis.

7.4 Experiments

The scenario used for our preliminary experimentations of the proposed performance modeling approach, is for a user entering an incorrect password. The User denied Sequence Diagram (Appendix C Figure 1) describes the interactions in this scenario. The *ATMSoftware* interacts with the ServerSoftware one time in this scenario. The message *AuthenticateUserInfo* and the replay *UserDenied* define this interaction. In the ServerSoftware a resource consumption Job is created upon the arrival of every *AuthenticateUserInfo* message. The Job is sent to the Resource side, processed (resource consumption emulated) and sent back to the *ServerSoftware*. Upon the reception of the processed Job the replay is generated and sent back to the *ATMSoftware*. The system is assumed to be configured with one CPU, hence one RoundRobin CPU is configured in the *ServerResources*. The speed of the CPU is configured through two parameters: the quantum time, set at 1 milisec and in each quantum 2000 instructions are processed. The State Diagram of the Observer shown in Figure 7.10 illustrates the start of the simulation as soon as the configuration stage finishes. The *ATM_Peripherals* generated the initial user as soon as the simulations time starts and a new User as soon as the current user finishes, Hence the user inter arrival time is 0 and the total number of Users in the system at any given point in the simulation time is equal to the number of ATMs. The simulation time is controlled by a timer that is initiated as the state *Running* is entered. In our experiments the simulation time is set to 180 seconds.

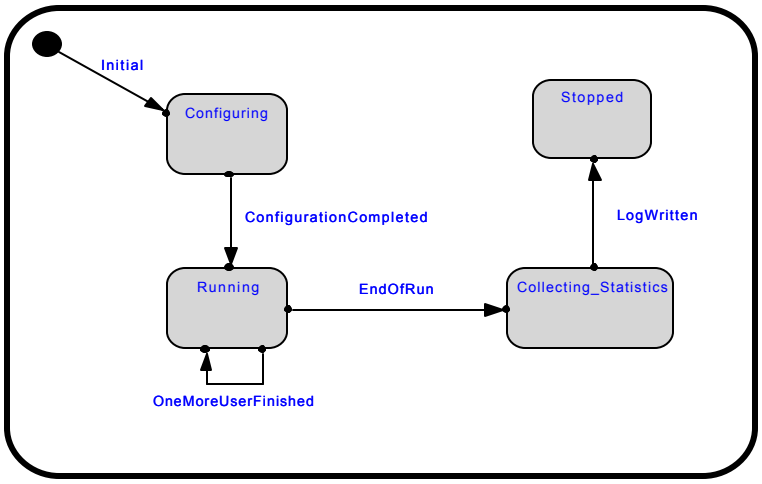

Figure 7.10 Observer State Diagram

In the first experiment we configured the number of instructions for each Job to be 100,000 instructions and the total user thinking time (while entering the password) to be 30 milisec.

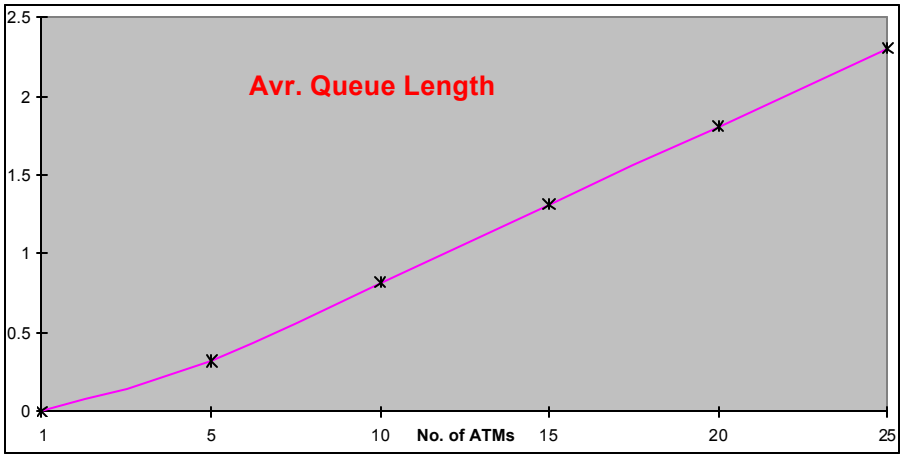

Figure 7.11 Average CPU Queue Length (first experiment)

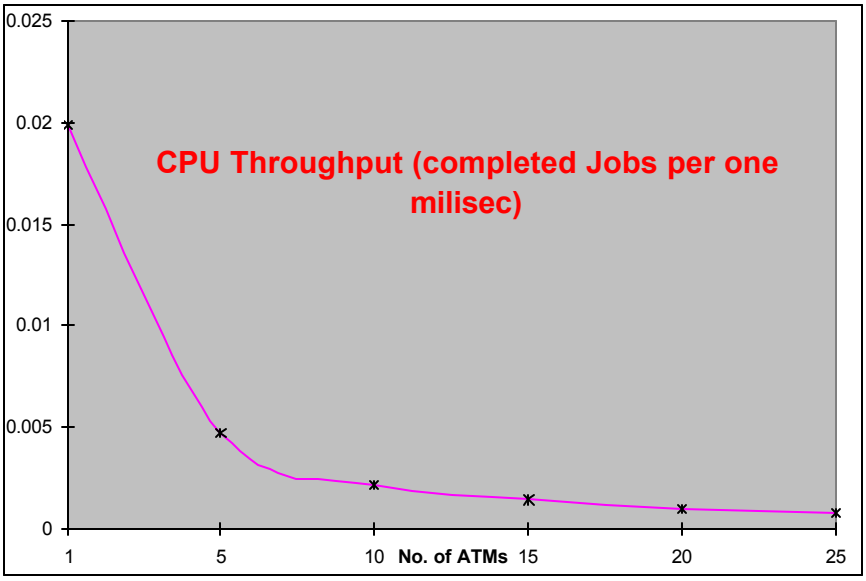

Figure 7.12 CPU Throughput (first experiment)

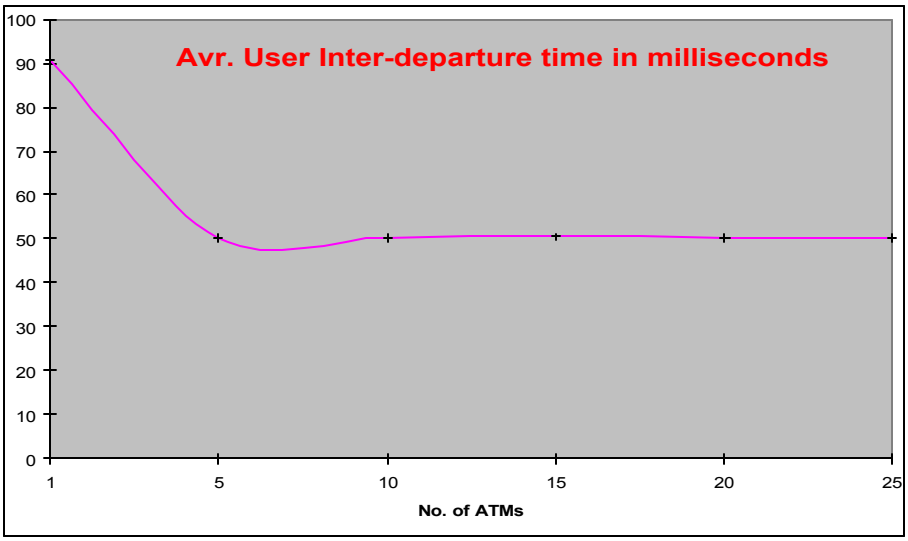

Figure 7.13 Average User Inter-departure time (first experiment)

In the second experiment we configured the number of instructions for each Job to be 20,000 instructions and the total user thinking time (while entering the password) to be 500 milisec.

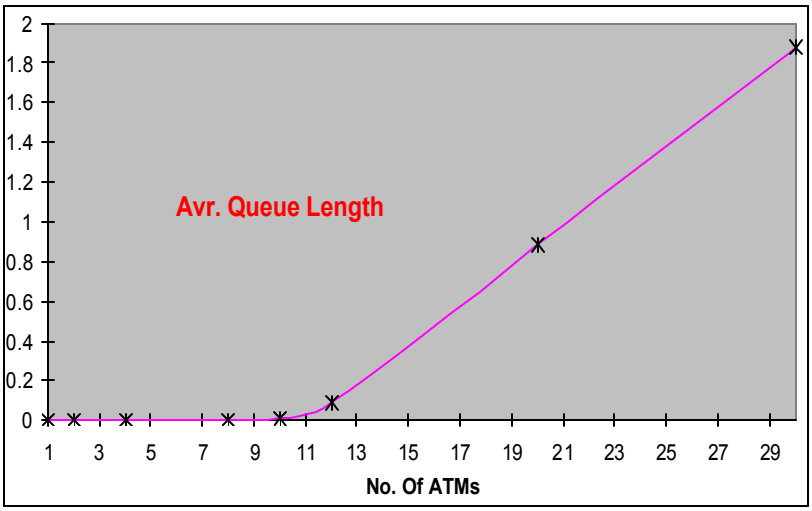

Figure 7.14 Average CPU Queue Length (second experiment)

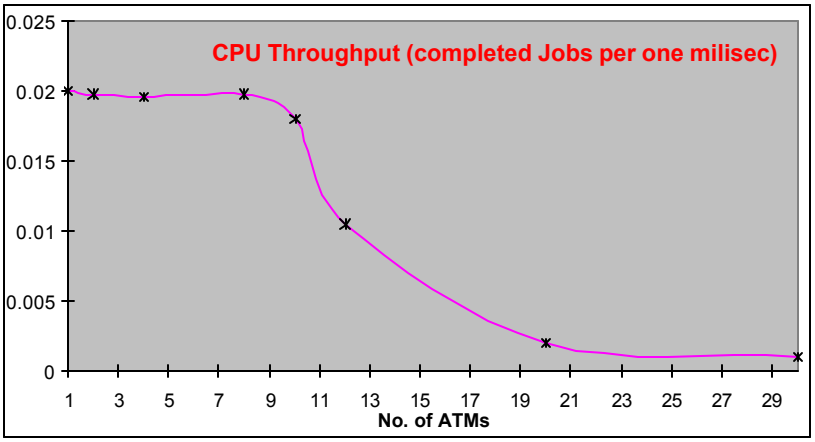

Figure 7.15 CPU Throughput (second experiment)

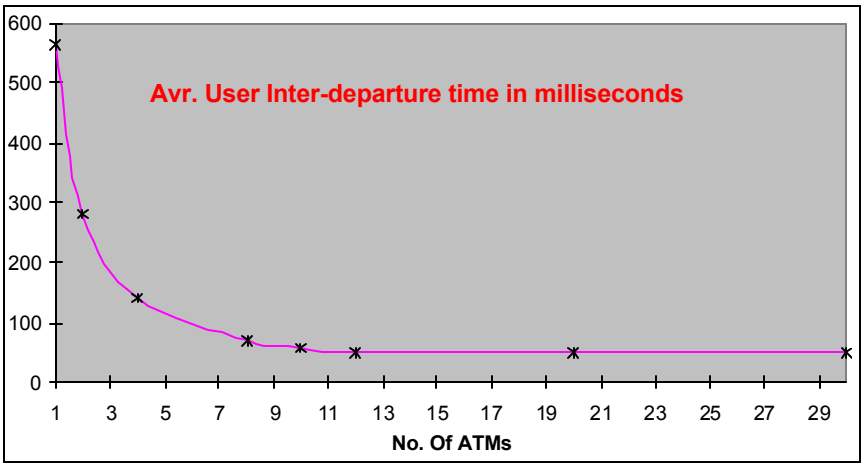

Figure 7.16 Average User Inter-departure time (second experiment)

7.5 Conclusion

We have introduced a new approach to the performance analysis and evaluation of UML based systems. The UML-RT notation has been used to build a library of stereotypes that represent resources. A software architecture modeled in UML-RT notation can thus be extended by adding a "resource side", that is the representation of a generic platform the software is supposed to run on. This uniform representation of software and resources, supported by the capability of the RRT tool (that simulates an UML-RT based model), allows to gain performance insights at the time of software architectural design. This is a preliminary study towards this approach, but we have here shown the potential scalability of our resource representation that, together with its generality, make this scheme flexible and portable.

Of the future areas of work: the automated collection of statistics using the Observer and Microsoft Excel, as well as modeling the Disk Resources and studying the effect of Database size (CPU and Disk Resources being affected simultaneously) on the overall system performance.

CHAPTER 8: CONCLUSIONS AND FUTURE WORK

Earlier we mentioned that V&V can be conducted at various development phases and that early V&V of software specification and analysis artifacts is encouraged before large investment is made in development. V&V of UML specifications can be done at an early development phase - prior to implementation - using scenarios, requirements and simulation models. Although UML is a rich analysis and design modeling language, it does not define how to study the dynamic aspects of the models through simulation; a capability that is required to monitor and asses the expected run-time behavior of software systems. V&V teams being much smaller than development teams must use efficient techniques to perform their analysis. At present mostly manual methods are being used to analyze UML models. Given the size and complexity of the large software systems, the manual efforts are time-consuming, tedious and error prone. Therefore automatable means (approaches and/or methods) for V&V of UML models need to be derived. In this work, we aim at helping and assessing V&V teams in performing there task in the early development stages of UML specifications through developing methods, approaches and extending there tool support for fast and automatic deployment of the developed means. We presented our efforts in four areas:

8.1 Temporal V&V

We discussed the automatic generation of timing violation tables from simulating UML specifications. We presented two approaches; in the first approach each simulation log is processed in search for constraint violations. In the second approach, an Observer component, acting as a monitoring object, is added as an external entity to the modeled system. We presented two methods for modeling the timing constraints in the Observer Component, namely: Constraint driven and Use Case driven. We showed results from applying the proposed approaches to the UML specifications of a cardiac pacemaker. As well we described four methods for timing analysis for assessing the degree of conformance to the timing constrains under abnormal conditions is the first area of investigation. We perceive that developing a technique for selecting scenarios, components, and connectors to which we apply the proposed timing analysis approach is a potential research area.

8.2 Automated Architectural-Risk assessment

We selected the methodology presented in [33] for automated Architectural-level risk assessment, because it has the following benefits: it is applicable early at the *architectural-level* and hence it is used to identify critical components and connectors early in the lifecycle. The methodology uses dynamic metrics, that covers the fact that a fault in a frequently executed component will frequently manifest itself into a failure. The methodology is based on *simulation* of UML-RT models. Simulation helps in: performing FMEA procedures and observing the timing diagrams. The presented automation environment shows how RRT tool can be used in fast and efficient deployment of the methodology. Future research could experiment with applying the methodology to larger case studies with multiple subsystems to compare the aggregated risk factors of individual subsystems.

8.3 Fault Injection analysis

We proposed a Fault Model, in chapter 6, that is acknowledged for its applicability in early development stages and scalability. Yet further experiments should be conducted on several case studies for better assessment and enhancement. Enhancements are required in several areas:

1. In the process of component selection, the number of components selected is decided by the analyst based on the available resources (mainly time). Better criteria for this selection is required to guarantee the best results when using the fault model in test case optimization.
2. The presented model focuses on microstates, while it is applicable to Marco states as well. Thus experiments for assessing the level of effectiveness of the fault model at the macro state level.
3. In the process of state selection, the number of sates selected is decided by the analyst based on the available resources (mainly time). But the tradeoff is in the quality of the analysis, thus a criteria for this selection is required.
4. The selection of the second message to swap with in a *Trigger Swap* is random. We perceive that a selection criteria is required for better results.

8.4 Performance Modeling

In chapter 7 we discussed how the importance of early performance assessment grows as software systems increase in terms of size, logical distribution and interaction complexity. Lack of time from the side of software developers, as well as distance among software model notations and performance model representation do not help to build an integrated software process that takes into account, from the early phases of the lifecycle, non functional requirement. In this work we aimed at filling this gap by extending the capabilities of a simulative environment developed for the UML notation. We introduced new stereotypes representing performance related items, such as resource types and job dispatchers. They allow the software designers to homogeneously represent a software architecture integrated with a running platform, and parameterized with the resource demand that the components require. As an application example a simplified Automated Teller Machine has been considered, and it has been designed also using the new stereotypes. This is to prove the effectiveness of our approach in building, and simulating, software performance models. We presented the preliminary insights gained from our study, in addition we make some considerations on the scalability of our approach. For shifting this work from research level to industrial level (being embedded in an tool and utilized by developers), more efforts in the creation of all stereotypes that covers the performance analysis needs, are required.

BIBLIOGRAPHY

[1] Baudry, B. Hanh, V.L., Jezequel J. and Traon, Y.L. "Building Trusted OO Components Using Genetic Analogy", Proc. of the 11th International Symposium on Software Reliability Engineering, ISSRE'00, IEEE Comp. Soc., October, 2000

[2] Cortellessa, V. and Mirandola R. "Deriving a Queueing Network based Performance Model from UML Diagrams", Proc. of Second International Workshop on Software and Performance, WOSP2000, September 2000, Ottawa, Canada, 2000.

[3] Cortellessa, V., Iazeolla, G. and Mirandola R. "Early Performance Validation for Object-Oriented Systems based on OMT methodology", IEE-Proceedings on Software, vol.147, issue 3, October 2000.

[4] Douglass, B. "Real-Time UML : Developing Efficient Objects for Embedded Systems", Addison-Wesley, 1998

[5] Firley, T., Huhn, M., Diethers, K., Gehrke, T. and Goltz, U., "Timed Sequence Diagrams and Tool-Based Analysis - A Case Study", The Second International Conference on The Unified Modeling Language, Beyond the Standard (UML'99), Lecture Notes in Computer Science, volume 1723, pp. 645-660, Springer-Verlag, October 1999.

[6] Goswami, K., Iyer, R. and Young, L. "DEPEND: A simulation-based environment for system level dependability analysis", IEEE Trans. on Computers, vol. 46, no. 1, pp. 60-74, 1997.

[7] Han, S., Shin, K. and Rosenberg, H. "DOCTOR: An integrated software fault injection environment for distributed real-time systems", in IEEE International Computer Performance and Dependability Symposium (IPDS'95), pp.204-213, March 1995.

[8] Ibrahim, A., Ammar, H., Yacoub, S., Dabney J. B, and Lateef, K. "Automated Verification of Timing Constraints in UML Dynamic Specifications", Submitted to Real-Time Technology and Applications Symposium RTAS'01, Taipei, Taiwan, May, 2001.

[9] Kao, W. and Iyer, R. "DEFINE: A distributed fault injection and monitoring environment", IEEE Workshop on Fault-Tolerant Parallel and Distributed System, June 1994.

[10] Lee, D. and Yannakakis, M. “Principles and methods of testing finite state machines - a survey”, The IEEE, Vol. 84, No. 8, pp. 1090-1123, August 1996.

[11] Lyons, A. “UML for Real-Time Overview”, ObjecTime, Ltd.,White Paper. http://www.ObjecTime.com/otl/technical/umlrt.html

[12] Li, X. and Lilius J. “Timing analysis of UML sequence diagrams”, The Second International Conference on The Unified Modeling Language, Beyond the Standard (UML'99), Lecture Notes in Computer Science, volume 1723, pages 661-674, Springer-Verlag, October 1999.

[13] Lazowska, E.D., Zahorjan, J., Graham, G.S. and Sevcik K.C., “Quantitative system performance : computer system analysis using queueing network models”, Englewood Cliffs, N.J., Prentice-Hall, 1984.

[14] Lavenberg, S.S. “Computer Performance Modeling Handbook”, Academic Press, New York, 1983.

[15] Menasce’, D.A. and Gomaa, H. “A method for design and performance modeling of client/server systems”, IEEE Transactions on Software Engineering, vol.26, no.11, November 2000.

[16] ObjecTime Ltd., Kanata, Ontario, Canada, http://www.ObjecTime.com

[17] Object Management Group, Inc., Needham, MA, USA. http://www.omg.org.

[18] Petriu, D. Shousha, C. and Jalnapurkar, A. “Architecture based Performance Analysis Applied to a Telecommunica-tion System”, IEEE Transaction on Software Engineering, November 2000.

[19] Petriu, D. “Deriving Performance Models from UML Models by Graph Transformations”, Tutorials, Second International Workshop on Software and Performance, WOSP2000, September 2000, Ottawa, Canada, 2000.

[20] Petriu, D. and Wang, X. “Deriving Software Performance Models from Architectural Patterns by Graph Transfor-mations”, Proc. of Theory and Applications of Graph transformations, TAGT’98, LNCS 1764, Springer Verlag, 1998.

[21] Rational Software Corporation, Cupertino, CA, USA. http://www.rational.com

[22] Rational Software Corporation, Rational Rose RealTime. http://www.rational.com/products/rosert/index.jsp

[23] Rolia, J.A. and Sevcik, K.C. “The method of Layers”, IEEE Transactions on Software Engineering, vol.21, no.8, August 1995.

[24] Software Safety, Nasa Technical Standard. NASA-STD-8719.13A, September 15, 1997 http://satc.gsfc.nasa.gov/assure/nss8719_13.html

[25] Selic, B. and Rumbaugh, J. “Using UML for modeling complex Real-Time systems”, ObjecTime, Ltd.,White Paper. http://www.ObjecTime.com/otl/technical/umlrt.html

[26] Selic, B., Gullekson, G. and Ward, P. “Real-Time Object Oriented Modeling”, John Wiley & Sons, Inc..

[27] Selic, B. “A generic framework for modeling resources with UML”, IEEE Computer, June 2000.

[28] Scott, D.T.; Ries, G.; Hsueh, Mei-Chen; Iyer, R.K. "Dependability analysis of a high-speed network using software-implemented fault injection and simulated fault injection." Proc. of the Twenty-Seventh Fault Tolerant Computing Symposium. IEEE, 1997. p. 108-119

[29] Smith, C.U. “Performance Engineering of Software Systems”, Addison-Wesley, Reading, MA, 1990.

[30] The Unified Modeling Language v1.3. http://www.rational.com/uml/resources/documentation/index.jsp

[31] Voas, Jeffrey M.; Miller, Keith W. "Using Fault Injection to Assess Software Engineering Standards", 1995 International Software Engineering Standards Symposium. IEEE, 1995. p. 139-145

[32] Voas, Jeffrey M.; Miller, Keith W. "Examining Fault-Tolerance Using Unlikely Inputs: Turning the Test Distribution Up-Side Down." Tenth Annual Conference on Computer Assurance. IEEE, 1993. p. 3-11.

[33] Yacoub, S., Ammar, H. “A Methodology for Architectural-Level Risk Assessment using Dynamic Metrics”, Proc. of the 11th International Symposium on Software Reliability Engineering, ISSRE’00, IEEE Comp. Soc., October, 2000.

[34] Yacoub, S., Ibrahim, A., Ammar, H., and Lateef, K. “Verification of UML Dynamic Specifications using Simulation-based Timing Analysis”, Proc. of 6th International

Conference on Reliability and Quality in Design, ISSAT, Orlando, Fl, August, 2000, pp.65-69.

[35] Yacoub, S., Ammar, H. and Robinson, T. "Dynamic Metrics for Object Oriented Designs", Proc. of the Sixth International Symposium on Software Metrics, Metrics'99, Boca Raton, Florida USA, November 4-6 1999, pp.50-61.

[36] Yacoub, S., Cukic, B. and Ammar, H. "Scenario-based Reliability Analysis of Component-Based Software", Proc. of the Tenth International Symposium on Software Reliability Engineering, ISSRE'99, Boca Raton, Florida USA, November 1-4 1999, pp.22-31.

APPENDIX A VISUAL BASIC MACROS

Sub Processing_Macro()

```
' Activation:  ctr g
Sheets("Process_ctrl-g").Select
totalcolumns = 200000
                    ' Extract all the Capsules "Objects" in the log file

i = 1
Objects = 1    'no of objects
While i < totalcolumns
   colB1 = "B" & i
   Oldobject = False
   j = 1
   For j = 1 To Objects Step 1      ' Set up 10 repetitions.
      colE1 = "E" & j
      If Range(colB1).Text = Range(colE1).Text Then
         Oldobject = True
      End If
   Next j
   If Oldobject = False Then
      Objects = Objects + 1
      colE1 = "E" & Objects
      Range(colE1).Value = Range(colB1).Text
   End If
   i = i + 1
Wend
```

'Dedicate columns E to Y for Object names and their states "in the form of state codes"

```
Range("E" & 1).Select
Selection.Delete Shift:=xlUp
Range("F1").Value = Range("E1").Text & " (Series 1 States)"
Range("G1").Value = "State code"
Range("H1").Value = Range("E2").Text & " (Series 2 States)"
Range("I1").Value = "State code"
Range("J1").Value = Range("E3").Text & " (Series 3 States)"
Range("K1").Value = "State code"
Range("L1").Value = Range("E4").Text & " (Series 4 States)"
Range("M1").Value = "State code"
Range("N1").Value = Range("E5").Text & " (Series 5 States)"
Range("O1").Value = "State code"
Range("P1").Value = Range("E6").Text & " (Series 6 States)"
Range("Q1").Value = "State code"
Range("R1").Value = Range("E7").Text & " (Series 7 States)"
Range("S1").Value = "State code"
Range("T1").Value = Range("E8").Text & " (Series 8 States)"
Range("U1").Value = "State code"
Range("V1").Value = Range("E9").Text & " (Series 9 States)"
Range("W1").Value = "State code"
Range("X1").Value = Range("E10").Text & " (Series 10 States)"
Range("Y1").Value = "State code"
```

'For each Object: extract all states and generate a consecutive sate code for each

```
i = 1
s = 1
If Not Range("E1").Text = "" Then
    While i < totalcolumns
        colB1 = "B" & i
        colC1 = "C" & i
        colstate = "F" & s
        colstateval = "g" & s
        If Range(colB1).Text = Range("E1").Text Then
            j = 1
            While j < s
                jcolC1 = "F" & j
                If Range(colC1).Value = Range(jcolC1).Text Then
                    j = s + 2
                Else
                    j = j + 1
                End If
            Wend
            If j = s Then
                Range(colstate).Value = Range(colC1).Text
                Range(colstateval).Value = s
                Name = Range(colstate).Value
                Object = Range("E1").Text
                Value = Range(colstateval).Value
                Sheets("Graph_ctrl-s").Select
                Graphlable = "A" & 20 - Value
                Range(Graphlable).Value = Name
                Range("B4").Value = Object
```

```
            Sheets("Process_ctrl-g").Select
            s = s + 1
          End If
        End If
        i = i + 1
    Wend
End If
NextValue = s - 1
nostatesseries1 = s - 1
i = 1
s = 1
If Not Range("E2").Text = "" Then
    While i < totalcolumns
        colB1 = "B" & i
        colC1 = "C" & i
        colstate = "H" & s
        colstateval = "I" & s
        If Range(colB1).Text = Range("E2").Text Then
            j = 1
            While j < s
                jcolC1 = "H" & j
                If Range(colC1).Value = Range(jcolC1).Text Then
                    j = s + 2
                Else
                    j = j + 1
                End If
            Wend
            If j = s Then
              Range(colstate).Value = Range(colC1).Text
              Range(colstateval).Value = s + NextValue
```

```
            Name = Range(colstate).Value
            Value = Range(colstateval).Value
            Object = Range("E2").Text
            Sheets("Graph_ctrl-s").Select
            Graphlable = "A" & 20 - Value
            Range(Graphlable).Value = Name
            Range("D4").Value = Object
            Sheets("Process_ctrl-g").Select
            s = s + 1
          End If
        End If
        i = i + 1
    Wend
End If
NextValue = s + NextValue - 1
nostatesseries2 = s – 1
i = 1
s = 1
If Not Range("E3").Text = "" Then
    While i < totalcolumns
        colB1 = "B" & i
        colC1 = "C" & i
        colstate = "J" & s
        colstateval = "K" & s
        If Range(colB1).Text = Range("E3").Text Then
            j = 1
            While j < s
                jcolC1 = "J" & j
                If Range(colC1).Value = Range(jcolC1).Text Then
                    j = s + 2
```

```
            Else
               j = j + 1
            End If
         Wend
         If j = s Then
               Range(colstate).Value = Range(colC1).Text
               Range(colstateval).Value = s + NextValue
               Name = Range(colstate).Value
               Value = Range(colstateval).Value
               Object = Range("E3").Text
               Sheets("Graph_ctrl-s").Select
               Graphlable = "A" & 20 - Value
               Range(Graphlable).Value = Name
               Range("F4").Value = Object
               Sheets("Process_ctrl-g").Select
               s = s + 1
         End If
      End If
      i = i + 1
   Wend
End If
NextValue = s + NextValue - 1
nostatesseries3 = s - 1
i = 1
s = 1
If Not Range("E4").Text = "" Then
   While i < totalcolumns
      colB1 = "B" & i
      colC1 = "C" & i
      colstate = "L" & s
```

```
        colstateval = "M" & s
        If Range(colB1).Text = Range("E4").Text Then
           j = 1
           While j < s
              jcolC1 = "L" & j
              If Range(colC1).Value = Range(jcolC1).Text Then
                 j = s + 2
              Else
                 j = j + 1
              End If
           Wend
           If j = s Then
              Range(colstate).Value = Range(colC1).Text
              Range(colstateval).Value = s + NextValue
              Name = Range(colstate).Value
              Object = Range("E4").Text
              Value = Range(colstateval).Value
              Sheets("Graph_ctrl-s").Select
              Graphlable = "A" & 20 - Value
              Range(Graphlable).Value = Name
              Range("H4").Value = Object
              Sheets("Process_ctrl-g").Select
              s = s + 1
           End If
        End If
        i = i + 1
     Wend
End If
NextValue = s + NextValue - 1
nostatesseries4 = s – 1
```

```
i = 1
s = 1
If Not Range("E5").Text = "" Then
    While i < totalcolumns
        colB1 = "B" & i
        colC1 = "C" & i
        colstate = "N" & s
        colstateval = "O" & s
        If Range(colB1).Text = Range("E5").Text Then
            j = 1
            While j < s
                jcolC1 = "N" & j
                If Range(colC1).Value = Range(jc olC1).Text Then
                    j = s + 2
                Else
                    j = j + 1
                End If
            Wend
            If j = s Then
                Range(colstate).Value = Range(colC1).Text
                Range(colstateval).Value = s + NextValue
                Name = Range(colstate).Value
                Object = Range("E5").Text
                Value = Range(colstateval).Value
                Sheets("Graph_ctrl-s").Select
                Graphlable = "A" & 20 - Value
                Range(Graphlable).Value = Name
                Range("B5").Value = Object
                Sheets("Process_ctrl-g").Select
                s = s + 1
```

```
            End If
        End If
        i = i + 1
    Wend
End If
NextValue = s + NextValue - 1
nostatesseries5 = s - 1
i = 1
s = 1
If Not Range("E6").Text = "" Then
    While i < totalcolumns
        colB1 = "B" & i
        colC1 = "C" & i
        colstate = "P" & s
        colstateval = "Q" & s
        If Range(colB1).Text = Range("E6").Text Then
            j = 1
            While j < s
                jcolC1 = "P" & j
                If Range(colC1).Value = Range(jcolC1).Text Then
                    j = s + 2
                Else
                    j = j + 1
                End If
            Wend
            If j = s Then
                Range(colstate).Value = Range(colC1).Text
                Range(colstateval).Value = s + NextValue
                Name = Range(colstate).Value
                Object = Range("E6").Text
```

```
                Value = Range(colstateval).Value
                Sheets("Graph_ctrl-s").Select
                Graphlable = "A" & 20 - Value
                Range(Graphlable).Value = Name
                Range("D5").Value = Object
                Sheets("Process_ctrl-g").Select
                s = s + 1
            End If
        End If
        i = i + 1
    Wend
End If
NextValue = s + NextValue - 1
nostatesseries6 = s - 1
i = 1
s = 1
If Not Range("E7").Text = "" Then
    While i < totalcolumns
        colB1 = "B" & i
        colC1 = "C" & i
        colstate = "R" & s
        colstateval = "S" & s
        If Range(colB1).Text = Range("E7").Text Then
            j = 1
            While j < s
                jcolC1 = "R" & j
                If Range(colC1).Value = Range(jcolC1).Text Then
                    j = s + 2
                Else
                    j = j + 1
```

```
            End If
        Wend
         If j = s Then
            Range(colstate).Value = Range(colC1).Text
            Range(colstateval).Value = s + NextValue
            Name = Range(colstate).Value
            Object = Range("E7").Text
            Value = Range(colstateval).Value
            Sheets("Graph_ctrl-s").Select
            Graphlable = "A" & 20 - Value
            Range(Graphlable).Value = Name
            Range("F5").Value = Object
            Sheets("Process_ctrl-g").Select
            s = s + 1
         End If
      End If
      i = i + 1
   Wend
End If
NextValue = s + NextValue - 1
nostatesseries7 = s - 1
i = 1
s = 1
If Not Range("E8").Text = "" Then
   While i < totalcolumns
      colB1 = "B" & i
      colC1 = "C" & i
      colstate = "T" & s
      colstateval = "U" & s
      If Range(colB1).Text = Range("E8").Text Then
```

```
            j = 1
            While j < s
               jcolC1 = "T" & j
               If Range(colC1).Value = Range(jcolC1).Text Then
                  j = s + 2
               Else
                  j = j + 1
               End If
            Wend
            If j = s Then
               Range(colstate).Value = Range(colC1).Text
               Range(colstateval).Value = s + NextValue
               Name = Range(colstate).Value
               Object = Range("E8").Text
               Value = Range(colstateval).Value
               Sheets("Graph_ctrl-s").Select
               Graphlable = "A" & 20 - Value
               Range(Graphlable).Value = Name
               Range("H5").Value = Object
               Sheets("Process_ctrl-g").Select
               s = s + 1
            End If
         End If
         i = i + 1
      Wend
End If
NextValue = s + NextValue - 1
nostatesseries8 = s - 1
i = 1
s = 1
```

```
If Not Range("E9").Text = "" Then
    While i < totalcolumns
        colB1 = "B" & i
        colC1 = "C" & i
        colstate = "V" & s
        colstateval = "W" & s
        If Range(colB1).Text = Range("E9").Text Then
            j = 1
            While j < s
                jcolC1 = "V" & j
                If Range(colC1).Value = Range(jcolC1).Text Then
                  j = s + 2
                Else
                  j = j + 1
                End If
            Wend
            If j = s Then
                Range(colstate).Value = Range(colC1).Text
                Range(colstateval).Value = s + NextValue
                Name = Range(colstate).Value
                Object = Range("E9").Text
                Value = Range(colstateval).Value
                Sheets("Graph_ctrl-s").Select
                Graphlable = "A" & 20 - Value
                Range(Graphlable).Value = Name
                Range("B6").Value = Object
                Sheets("Process_ctrl-g").Select
                s = s + 1
            End If
        End If
```

```
        i = i + 1
    Wend
End If
NextValue = s + NextValue - 1
nostatesseries9 = s - 1
‘Next Object
i = 1
s = 1
If Not Range("E10").Text = "" Then
    While i < totalcolumns
        colB1 = "B" & i
        colC1 = "C" & i
        colstate = "X" & s
        colstateval = "Y" & s
        If Range(colB1).Text = Range("E10").Text Then
            j = 1
            While j < s
                jcolC1 = "X" & j
                If Range(colC1).Value = Range(jcolC1).Text Then
                    j = s + 2
                Else
                    j = j + 1
                End If
            Wend
            If j = s Then
                Range(colstate).Value = Range(colC1).Text
                Range(colstateval).Value = s + NextValue
                Name = Range(colstate).Value
                Object = Range("E10").Text
                Value = Range(colstateval).Value
```

```
            Sheets("Graph_ctrl-s").Select
            Graphlable = "A" & 20 - Value
            Range(Graphlable).Value = Name
            Range("D6").Value = Object
            Sheets("Process_ctrl-g").Select
            s = s + 1
          End If
        End If
        i = i + 1
    Wend
End If
```

'For each Object and in columns AA though AI: use the state codes to generate an eleven columns log file without time as the first column "A" and the rest as the states in state code

```
nostatesseries10 = s - 1
i = 1
While i < totalcolumns
  colB1 = "B" & i  'actor
  colC1 = "C" & i  ' state
  colAA1 = "AA" & i
  If Range(colB1).Text = Range("E1").Text Then
     j = 1
     While j <= nostatesseries1
      jcolS1 = "F" & j
      jcolV1 = "G" & j
      If Range(colC1).Value = Range(jcolS1).Text Then
        Range(colAA1).Value = Range(jcolV1).Text
        j = nostatesseries1 + 2
      Else
```

```
        j = j + 1
      End If
    Wend
  End If
 i = i + 1
Wend
i = 1
While i < totalcolumns
  colB1 = "B" & i  'actor
  colC1 = "C" & i  ' state
  colAA1 = "AB" & i
  If Range(colB1).Text = Range("E2").Text Then
     j = 1
     While j <= nostatesseries2
      jcolS1 = "H" & j
      jcolV1 = "I" & j
      If Range(colC1).Value = Range(jcolS1).Text Then
       Range(colAA1).Value = Range(jcolV1).Text
       j = nostatesseries2 + 2
      Else
       j = j + 1
      End If
    Wend
  End If
  i = i + 1
Wend
i = 1
While i < totalcolumns
  colB1 = "B" & i  'actor
  colC1 = "C" & i  ' state
```

```
  colAA1 = "AC" & i
  If Range(colB1).Text = Range("E3").Text Then
    j = 1
    While j <= nostatesseries3
     jcolS1 = "J" & j
     jcolV1 = "K" & j
     If Range(colC1).Value = Range(jcolS1).Text Then
       Range(colAA1).Value = Range(jcolV1).Text
       j = nostatesseries3 + 2
     Else
       j = j + 1
     End If
   Wend
  End If
i = i + 1
Wend
i = 1
While i < totalcolumns
  colB1 = "B" & i  'actor
  colC1 = "C" & i  ' state
  colAA1 = "AD" & i
  If Range(colB1).Text = Range("E4").Text Then
    j = 1
    While j <= nostatesseries4
    jcolS1 = "L" & j
    jcolV1 = "M" & j
     If Range(colC1).Value = Range(jcolS1).Text Then
       Range(colAA1).Value = Range(jcolV1).Text
       j = nostatesseries4 + 2
     Else
```

```
        j = j + 1
      End If
    Wend
  End If
  i = i + 1
Wend
i = 1
While i < totalcolumns
  colB1 = "B" & i  'actor
  colC1 = "C" & i  ' state
  colAA1 = "AE" & i
  If Range(colB1).Text = Range("E5").Text Then
     j = 1
     While j <= nostatesseries5
     jcolS1 = "N" & j
     jcolV1 = "o" & j
      If Range(colC1).Value = Range(jcolS1).Text Then
        Range(colAA1).Value = Range(jcolV1).Text
        j = nostatesseries5 + 2
      Else
        j = j + 1
      End If
    Wend
  End If
  i = i + 1
Wend
i = 1
While i < totalcolumns
  colB1 = "B" & i  'actor
  colC1 = "C" & i  ' state
```

```
  colAA1 = "AF" & i
  If Range(colB1).Text = Range("E6").Text Then
     j = 1
     While j <= nostatesseries6
       jcolS1 = "P" & j
       jcolV1 = "Q" & j
      If Range(colC1).Value = Range(jcolS1).Text Then
       Range(colAA1).Value = Range(jcolV1).Text
       j = nostatesseries6 + 2
     Else
       j = j + 1
     End If
   Wend
  End If
  i = i + 1
Wend
i = 1
While i < totalcolumns
  colB1 = "B" & i  'actor
  colC1 = "C" & i  ' state
  colAA1 = "AG" & i
  If Range(colB1).Text = Range("E7").Text Then
     j = 1
     While j <= nostatesseries7
       jcolS1 = "R" & j
       jcolV1 = "S" & j
      If Range(colC1).Value = Range(jcolS1).Text Then
        Range(colAA1).Value = Range(jcolV1).Text
        j = nostatesseries7 + 2
     Else
```

```
        j = j + 1
      End If
    Wend
  End If
  i = i + 1
Wend
i = 1
While i < totalcolumns
  colB1 = "B" & i  'actor
  colC1 = "C" & i  ' state
  colAA1 = "AH" & i
  If Range(colB1).Text = Range("E8").Text Then
     j = 1
     While j <= nostatesseries8
       jcolS1 = "T" & j
       jcolV1 = "U" & j
      If Range(colC1).Value = Range(jcolS1).Text Then
        Range(colAA1).Value = Range(jcolV1).Text
        j = nostatesseries8 + 2
      Else
       j = j + 1
      End If
    Wend
  End If
  i = i + 1
Wend
i = 1
While i < totalcolumns
  colB1 = "B" & i  'actor
  colC1 = "C" & i  ' state
```

```
  colAA1 = "AI" & i
  If Range(colB1).Text = Range("E9").Text Then
     j = 1
     While j <= nostatesseries9
      jcolS1 = "V" & j
      jcolV1 = "W" & j
      If Range(colC1).Value = Range(jcolS1).Text Then
        Range(colAA1).Value = Range(jcolV1).Text
        j = nostatesseries9 + 2
      Else
        j = j + 1
      End If
    Wend
  End If
  i = i + 1
Wend
i = 1
While i < totalcolumns
  colB1 = "B" & i  'actor
  colC1 = "C" & i  ' state
  colAA1 = "AJ" & i
  If Range(colB1).Text = Range("E10").Text Then
     j = 1
     While j <= nostatesseries10
       jcolS1 = "X" & j
       jcolV1 = "Y" & j
      If Range(colC1).Value = Range(jcolS1).Text Then
        Range(colAA1).Value = Range(jcolV1).Text
        j = nostatesseries10 + 2
      Else
```

```
      j = j + 1
    End If
  Wend
 End If
 i = i + 1
Wend

'Create continuous lines (horizontal and vertical) from the ten fragmented serises
representing the state changes (in state codes) of the ten Objects
'Use 2 D array for better speed
ReDim tiarray(11, totalcolumns) As Variant
ReDim tfarray(11, 2 * totalcolumns) As Variant
Dim lastVarray(10) As Variant
i = 1

'reed from sheet into array

While i < totalcolumns
tiarray(1, i) = Range("A" & i).Text
tiarray(2, i) = Range("AA" & i).Text
tiarray(3, i) = Range("AB" & i).Text
tiarray(4, i) = Range("AC" & i).Text
tiarray(5, i) = Range("AD" & i).Text
tiarray(6, i) = Range("AE" & i).Text
tiarray(7, i) = Range("AF" & i).Text
tiarray(8, i) = Range("AG" & i).Text
tiarray(9, i) = Range("AH" & i).Text
tiarray(10, i) = Range("AI" & i).Text
tiarray(11, i) = Range("AJ" & i).Text
i = i + 1
```

```
Wend

i = 1
j = 1
While i < totalcolumns
   For x = 1 To 11 Step 1
    tfarray(x, j) = tiarray(x, i)
    If x > 1 Then tfarray(x, j) = lastVarray(x - 1)
    j = j + 1
    tfarray(x, j) = tiarray(x, i)
    If x > 1 Then
      If tfarray(x, j) = "" Then
         tfarray(x, j) = lastVarray(x - 1)
      Else
         lastVarray(x - 1) = tfarray(x, j)
      End If
    End If
    j = j - 1
   Next x
   i = i + 1
   j = j + 2
Wend
j = 1
                                    'Read from array into sheet
While j < totalcolumns
Range("A" & j).Value = tfarray(1, j)
Range("AA" & j).Value = tfarray(2, j)
Range("AB" & j).Value = tfarray(3, j)
Range("AC" & j).Value = tfarray(4, j)
Range("AD" & j).Value = tfarray(5, j)
```

```
Range("AE" & j).Value = tfarray(6, j)
Range("AF" & j).Value = tfarray(7, j)
Range("AG" & j).Value = tfarray(8, j)
Range("AH" & j).Value = tfarray(9, j)
Range("AI" & j).Value = tfarray(10, j)
Range("AJ" & j).Value = tfarray(11, j)
j = j + 1
Wend

    'Size the chart and force to start from 0 milisec and end at 20000 milisec

   Sheets("Graph_ctrl-s").Select
   ActiveSheet.ChartObjects("Chart 1").Activate
   ActiveChart.Axes(xlValue).Select
   y = ActiveChart.Axes(xlValue).MaximumScale
   ActiveChart.PlotArea.Select
   x = Selection.Height
   ActiveSheet.Range("A1").Activate
   Rows("8:50").Select
   Range("A1").Select
   xstart = 0
   Range("B2").Value = xstart
   xend = 20000
   Range("D2").Value = xend
   xstep = 500
   Range("F2").Value = xstep
   ActiveSheet.ChartObjects("Chart 1").Activate
   ActiveChart.Axes(xlCategory).Select
   With ActiveChart.Axes(xlCategory)
      .MinimumScale = xstart
```

```
        .MaximumScale = xend
        .BaseUnitIsAuto = True
        .MajorUnit = xstep
        .MajorUnitScale = xlDays
        .MinorUnit = 34
        .MinorUnitScale = xlDays
        .Crosses = xlAutomatic
        .AxisBetweenCategories = True
        .ReversePlotOrder = False
    End With
End Sub 'end of processing macro
```

```
Sub Viewing_Macro()
                ‘Resize chart based on the start, end and step in the Graph_ctrl-s
'Activation ctr s
Sheets("Graph_ctrl-s").Select
   xstart = Range("B2").Value
    xend = Range("D2").Value
    xstep = Range("F2").Value
    ActiveSheet.ChartObjects("Chart 1").Activate
    ActiveChart.Axes(xlCategory).Select
    With ActiveChart.Axes(xlCategory)
        .MinimumScale = xstart
        .MaximumScale = xend
        .BaseUnitIsAuto = True
        .MajorUnit = xstep
        .MajorUnitScale = xlDays
        .MinorUnit = 34
        .MinorUnitScale = xlDays
        .Crosses = xlAutomatic
        .AxisBetweenCategories = True
        .ReversePlotOrder = False
    End With
End Sub
```

APPENDIX B RISK MACRO

```
Sub Risk_Macro()
' Keyboard Shortcut: Ctrl+r
totalcolumns = 6950        'changeble by user
noofcomponents = 1
Dim Componentchildarray(101, 1000) As Variant
mainloop = 1
'column 1 is for component names
'column 2 is for component no of children
'column 3 is for total no of messages out of component
Componentchildarray(1, 1) = Range("A1").Text
For initnoofchildren = 0 To (UBound(Componentchildarray, 1) - 1) Step 1
    Componentchildarray(initnoofchildren, 2) = 0
    Componentchildarray(initnoofchildren, 3) = 0
Next initnoofchildren  ' Increment counter

While mainloop < totalcolumns
    j = 1
    While j <= noofcomponents
        If Range("A" & mainloop).Text = Componentchildarray(j, 1) Then
            'listed
            ' is it a new child for that component
            H = 0
            While H <= Componentchildarray(j, 2)
                If Range("B" & mainloop).Text = Componentchildarray(j, 11 + (H * 10)) Then
                    Componentchildarray(j, 13 + (H * 10)) = Componentchildarray(j, 13 + (H * 10)) + 1
                    Componentchildarray(j, 3) = Componentchildarray(j, 3) + 1
                    H = Componentchildarray(j, 2)      'listed
```

```
            Else
              If H = Componentchildarray(j, 2) Then
                 ' a new child
                 Componentchildarray(j, 2) = Componentchildarray(j, 2) + 1
                 Componentchildarray(j, 11 + (H * 10)) = Range("B" & mainloop).Text
                 Componentchildarray(j, 13 + (H * 10)) = Componentchildarray(j, 13 + (H * 10)) +
1
                 Componentchildarray(j, 3) = Componentchildarray(j, 3) + 1
                 H = H + 1    ' h starts at 0 while j starts at 1
              End If
            End If
            H = H + 1
         Wend
         j = noofcomponents      'listed
      Else
         If j = noofcomponents Then
            ' a new component
            noofcomponents = noofcomponents + 1
            Componentchildarray(j + 1, 1) = Range("A" & mainloop).Text
         End If
      End If
      j = j + 1
   Wend
   mainloop = mainloop + 1
Wend
'calculate the probabilityies & child index
For calprob = 1 To noofcomponents Step 1
   For calchildprob = 1 To Componentchildarray(calprob, 2) Step 1
      Componentchildarray(calprob, 2 + (calchildprob * 10)) = Componentchildarray(calprob, 3 +
(calchildprob * 10)) / Componentchildarray(calprob, 3)
```

```
        Componentchildarray(calprob, 4 + (calchildprob * 10)) = Componentchildarray(calprob, 3 +
(calchildprob * 10)) / totalcolumns
        'child index
        For childindex = 1 To noofcomponents Step 1
           If Componentchildarray(calprob, 1 + (calchildprob * 10)) =
Componentchildarray(childindex, 1) Then
              Componentchildarray(calprob, 0 + (calchildprob * 10)) = childindex
              childindex = noofcomponents + 1
           End If
        Next childindex
    Next calchildprob  ' Increment counter
Next calprob  ' Increment counter

'dispaly ,write to file, input complexity, severity, ET
Columns("G:R").Select
Selection.ColumnWidth = 20
Range("F1") = noofcomponents
Range("G1").Value = "Component_Index"
Range("H1").Value = "Component_Name"
Range("I1").Value = "No._of_Children"
Range("J1").Value = "Total_No._Messages"
Range("k1").Value = "Component_Complexity"
Range("L1").Value = "Component_Severity"
Range("M1").Value = "Component_Execution_Time"
Range("N1").Value = "Component_Risk"
bias = noofcomponents + 4
Range("H" & bias).Value = "Index_of_Child"
Range("I" & bias).Value = "Child_Name"
Range("J" & bias).Value = "Probability_of_Transition"
Range("K" & bias).Value = "No_Of_Messages"
```

```
Range("L" & bias).Value = "Connector_Complexity"
Range("M" & bias).Value = "Connector_Severity"
Range("N" & bias).Value = "Connector_Risk"
For displayrows = 1 To noofcomponents Step 1
   'find match
   For Match = 1 To noofcomponents Step 1
      If (Componentchildarray(displayrows, 1) = Range("D" & Match).Value) Then
        matchedindex = Match
      End If
   Next Match
    Componentchildarray(displayrows, 4) = Range("e" & matchedindex + noofcomponents +
4).Value
    Componentchildarray(displayrows, 6) = Range("d" & matchedindex + noofcomponents +
4).Value
   Componentchildarray(displayrows, 5) = Range("L" & displayrows + 1).Value
   Componentchildarray(displayrows, 7) = Componentchildarray(displayrows, 4) *
Componentchildarray(displayrows, 5)
   Range("G" & displayrows + 1).Value = displayrows
   Range("H" & displayrows + 1).Value = Componentchildarray(displayrows, 1)
   Range("I" & displayrows + 1).Value = Componentchildarray(displayrows, 2)
   Range("J" & displayrows + 1).Value = Componentchildarray(displayrows, 3)
   Range("k" & displayrows + 1).Value = Componentchildarray(displayrows, 4)
   Range("N" & displayrows + 1).Value = Componentchildarray(displayrows, 7)
   Range("M" & displayrows + 1).Value = Componentchildarray(displayrows, 6)
   childrendisplay = Componentchildarray(displayrows, 2)
   bias = bias + Componentchildarray(displayrows - 1, 2)
   While childrendisplay > 0
      Componentchildarray(displayrows, 6 + childrendisplay * 10) =
Componentchildarray(displayrows, 5 + childrendisplay * 10) *
Componentchildarray(displayrows, 4 + childrendisplay * 10)
```

```
        Range("H" & displayrows * 2 + bias + childrendisplay).Value =
Componentchildarray(displayrows, 0 + childrendisplay * 10)
        Range("I" & displayrows * 2 + bias + childrendisplay).Value =
Componentchildarray(displayrows, 1 + childrendisplay * 10)
        Range("J" & displayrows * 2 + bias + childrendisplay).Value =
Componentchildarray(displayrows, 2 + childrendisplay * 10)
        Range("K" & displayrows * 2 + bias + childrendisplay).Value =
Componentchildarray(displayrows, 3 + childrendisplay * 10)
        Range("L" & displayrows * 2 + bias + childrendisplay).Value =
Componentchildarray(displayrows, 4 + childrendisplay * 10)
        Range("M" & displayrows * 2 + bias + childrendisplay).Value =
Componentchildarray(displayrows, 5 + childrendisplay * 10)
        Range("N" & displayrows * 2 + bias + childrendisplay).Value =
Componentchildarray(displayrows, 6 + childrendisplay * 10)
        childrendisplay = childrendisplay - 1
    Wend
Next displayrows  ' Increment counter
End Sub

Sub Risk_CDG_Traversal()
'read from display
' Macro  CDG  Traversal

totalcolumns = 6950              'changeble by user
Start_Component = 1       'changeble by user
noofcomponents = 1
Dim Componentchildarray(101, 1000) As Variant
'get from display
noofcomponents = Range("F1")
bias = noofcomponents + 4
```

```
For displayrows = 1 To noofcomponents Step 1
    Componentchildarray(displayrows, 1) = Range("H" & displayrows + 1).Value
    Componentchildarray(displayrows, 2) = Range("I" & displayrows + 1).Value
    Componentchildarray(displayrows, 3) = Range("J" & displayrows + 1).Value
    Componentchildarray(displayrows, 4) = Range("k" & displayrows + 1).Value
    Componentchildarray(displayrows, 5) = Range("L" & displayrows + 1).Value
    Componentchildarray(displayrows, 6) = Range("M" & displayrows + 1).Value
    Componentchildarray(displayrows, 7) = Componentchildarray(displayrows, 4) * Componentchildarray(displayrows, 5)
    Range("N" & displayrows + 1).Value = Componentchildarray(displayrows, 7)
    childrendisplay = Componentchildarray(displayrows, 2)
    bias = bias + Componentchildarray(displayrows - 1, 2)
    While childrendisplay > 0
        Componentchildarray(displayrows, 0 + childrendisplay * 10) = Range("H" & displayrows * 2 + bias + childrendisplay).Value
        Componentchildarray(displayrows, 1 + childrendisplay * 10) = Range("I" & displayrows * 2 + bias + childrendisplay).Value
        Componentchildarray(displayrows, 2 + childrendisplay * 10) = Range("J" & displayrows * 2 + bias + childrendisplay).Value
        Componentchildarray(displayrows, 3 + childrendisplay * 10) = Range("K" & displayrows * 2 + bias + childrendisplay).Value
        Componentchildarray(displayrows, 4 + childrendisplay * 10) = Range("L" & displayrows * 2 + bias + childrendisplay).Value
        Componentchildarray(displayrows, 5 + childrendisplay * 10) = Range("M" & displayrows * 2 + bias + childrendisplay).Value
        Componentchildarray(displayrows, 6 + childrendisplay * 10) = Componentchildarray(displayrows, 4 + childrendisplay * 10) * Componentchildarray(displayrows, 5 + childrendisplay * 10)
        Range("N" & displayrows * 2 + bias + childrendisplay).Value = Componentchildarray(displayrows, 6 + childrendisplay * 10)
```

```
        childrendisplay = childrendisplay - 1
    Wend
Next displayrows  ' Increment counter
'CDG Traversal
Dim R_appl As Double
R_appl = 0
Dim SegmaTime As Double
SegmaTime = 0
Dim R_Temp As Double
R_Temp = 1
Dim AE_appl As Double
AE_appl = Range("d" & noofcomponents + 2).Value
Dim Stackindex As Integer
Stackindex = 1
Dim Currentcomponent As Integer
Currentcomponent = 0
Dim Traversalstack(100000, 3) As Double
'first push
Traversalstack(Stackindex, 1) = Start_Component
Traversalstack(Stackindex, 2) = SegmaTime
Traversalstack(Stackindex, 3) = R_Temp
Stackindex = Stackindex + 1
While Stackindex > 0
    Currentcomponent = Traversalstack(Stackindex, 1)
    SegmaTime = Traversalstack(Stackindex, 2)
    R_Temp = Traversalstack(Stackindex, 3)
    Stackindex = Stackindex - 1
    If (SegmaTime >= AE_appl) Or (Componentchildarray(Currentcomponent, 2) = 0) Then
'refer to terminal node
        R_appl = R_appl + R_Temp
```

```
    Else
        For x = 0 To (Componentchildarray(Currentcomponent, 2) - 1) Step 1
            Stackindex = Stackindex + 1
            Traversalstack(Stackindex, 1) = Componentchildarray(Currentcomponent, 10 + x * 10)
            Traversalstack(Stackindex, 2) = SegmaTime + Componentchildarray(Currentcomponent,
6)
            Traversalstack(Stackindex, 3) = R_Temp * (1 - Componentchildarray(Currentcomponent,
7)) * Componentchildarray(Currentcomponent, 12 + x * 10) * (1 -
Componentchildarray(Currentcomponent, 16 + x * 10))
        Next x  ' Increment counter
    End If
Wend
Range("f3").Value = "System Risk"
Range("f4").Value = 1 - R_appl
End Sub
```

APPENDIX C ATM SEQUENCE DIAGRAMS

1. *Use_Denied*: (figure 1)
2. *Balance*: (figure 2)
3. *Balance_Print*: (figure 3)
4. *Withdrawal*: (figure 4)
5. *Withdrawal_Print*: (figure 5)
6. *Withdrawal_Denied*: (figure 6)

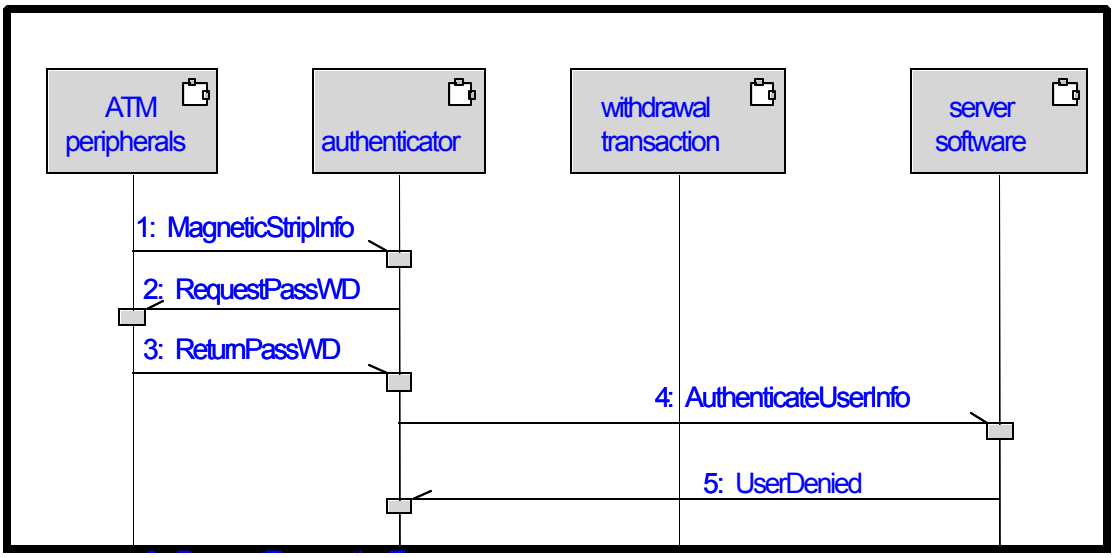

Figure 1 *Use_Denied*: Sequence Diagram for failed Authentication

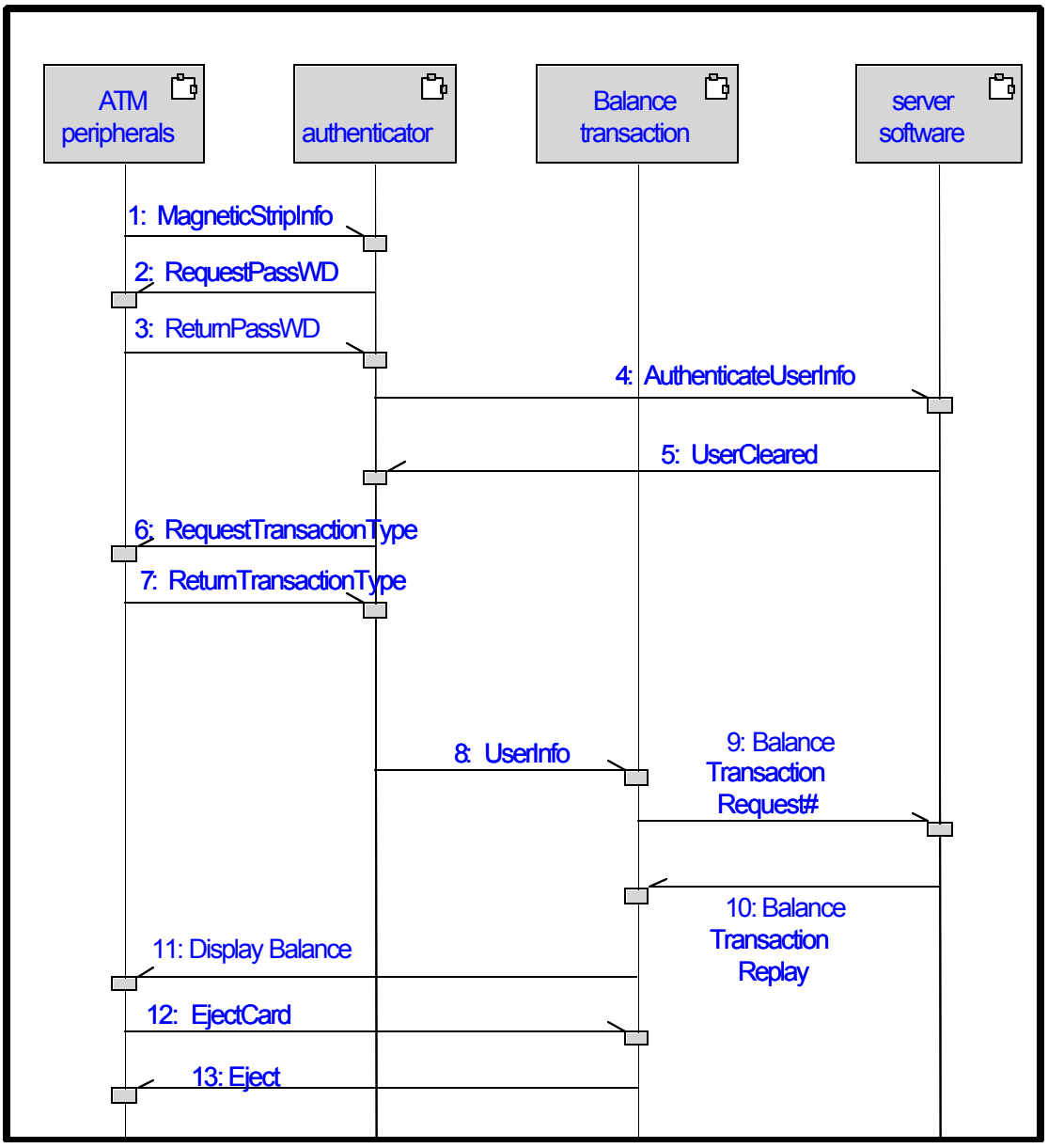

Figure 2 *Balance*: Sequence Diagram for balance inquiry transaction without statement printing

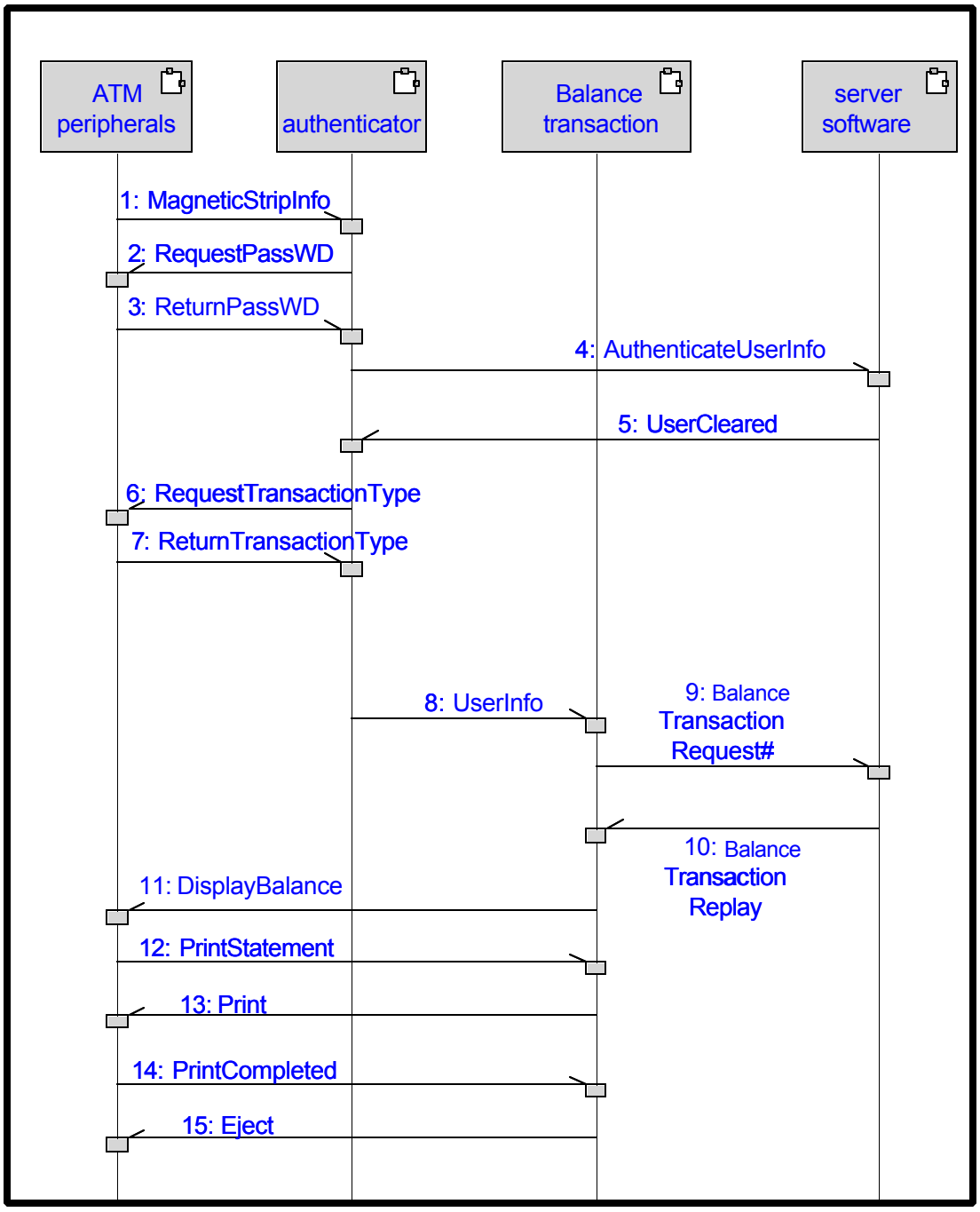

Figure 3 *Balance_Print* : Sequence Diagram for balance inquiry transaction with statement printing

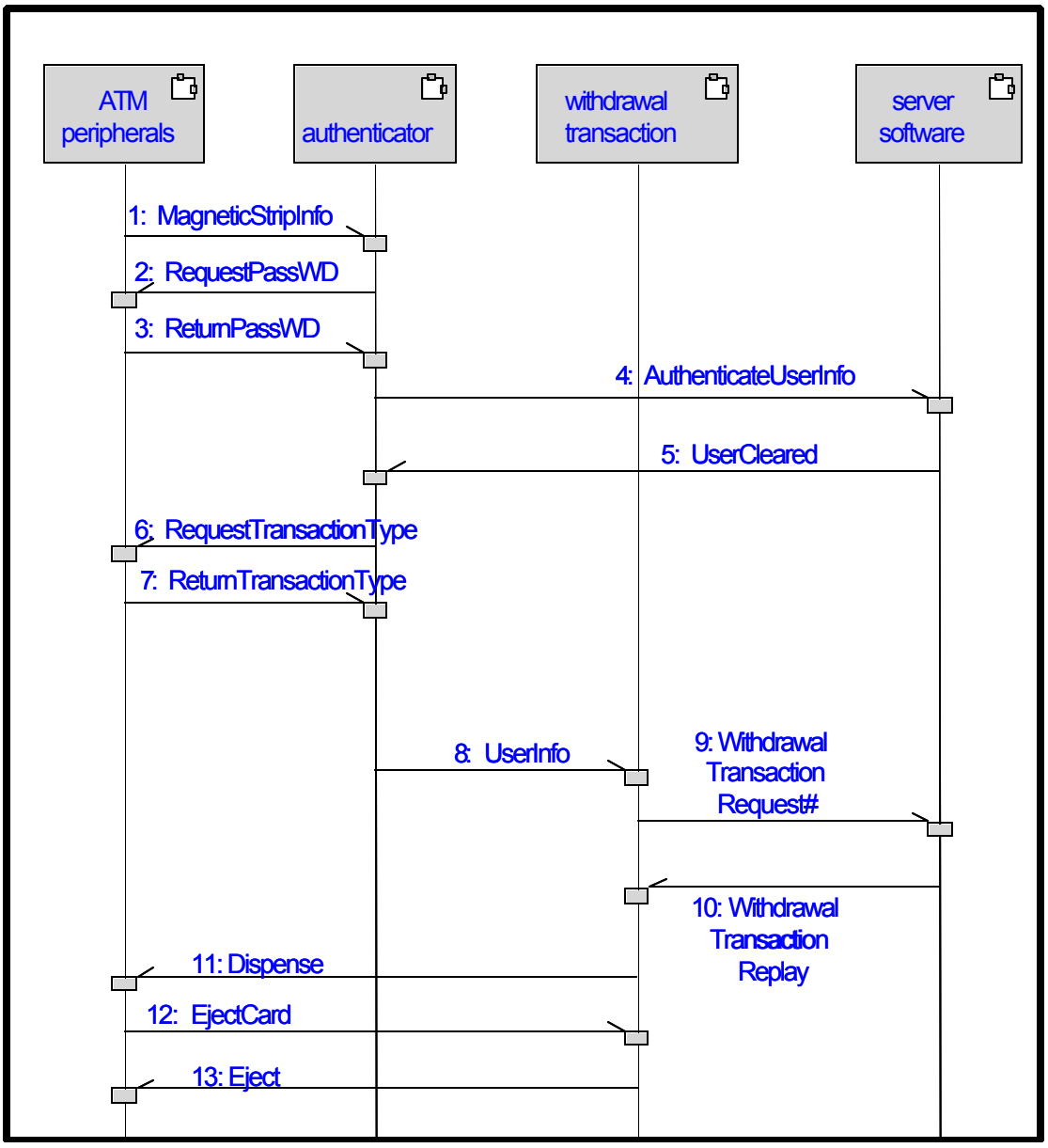

Figure 4 *Withdrawal* : Sequence Diagram for successful withdrawal transaction without statement printing

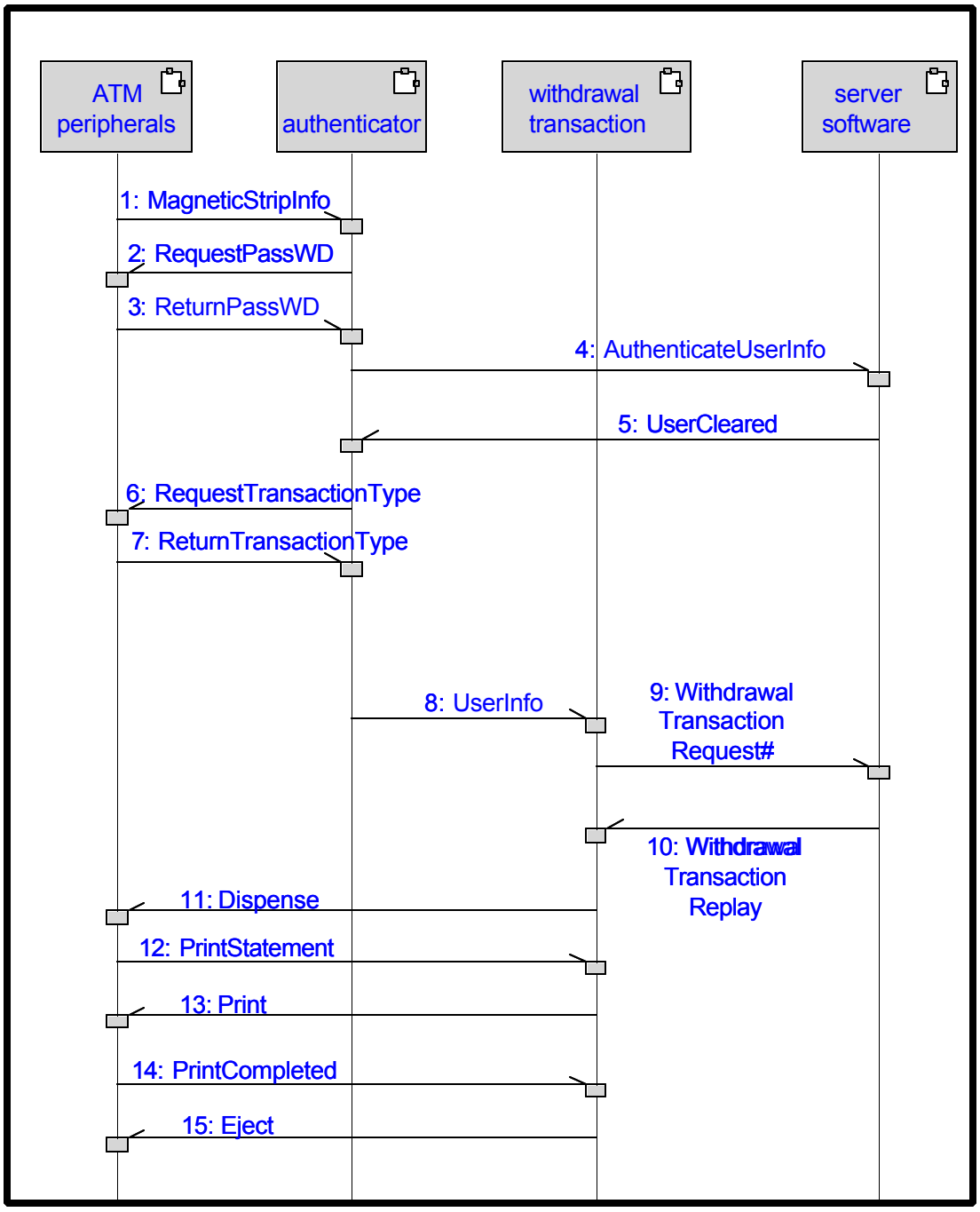

Figure 5 *Withdrawal_Print* : Sequence Diagram for successful withdrawal transaction with statement printing

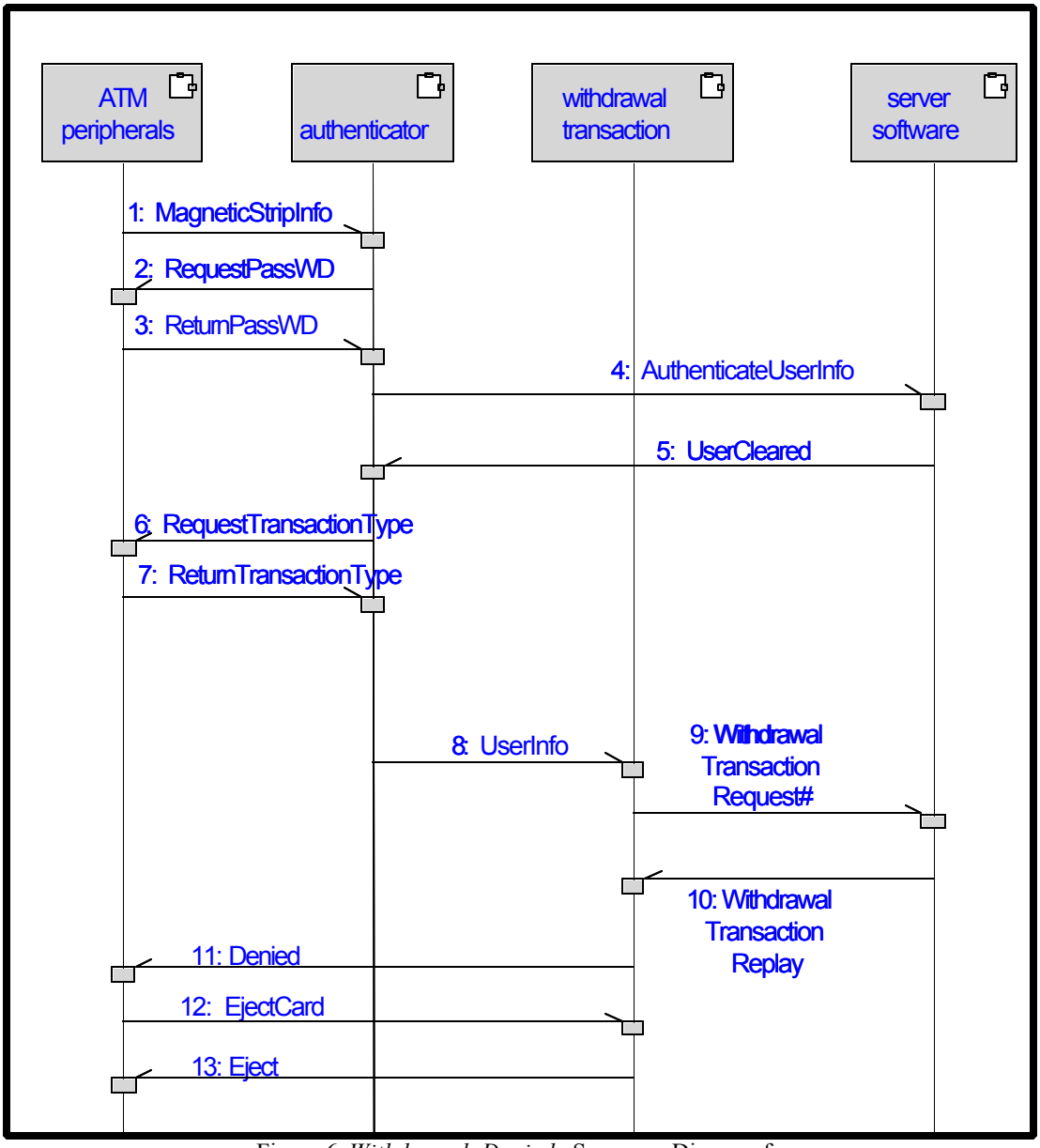

Figure 6 *Withdrawal_Denied* : Sequence Diagram for unsuccessful withdrawal transaction without statement printing

Alaa Ibrahim

Email: ibrahim@csee.wvu.edu
Department of Computer Science & Electrical Engineering,
West Virginia University. PO Box 6104
Morgantown, WV 26506-6104

Objective

Seeking a position in the design and development of real-time embedded software/systems where graduate education and 6 years of varied experiences will add great value to the organization.

Experience

2000 – Present West Virginia University

Graduate Research Assistant

Thesis: Scenario based Verification and Validation of UML Specifications. Project funded by AverStar Inc., Fairmont, WV, through the Software Engineering Research Center (SERC)

Publications:

1. Yacoub, S., Ibrahim, A., Ammar, H., and Lateef, K. "**Verification of UML Dynamic Specifications using Simulation-based Timing Analysis**", Proc. of 6th International Conference on Reliability and Quality in Design, **ISSAT**, Orlando, Fl, August, 2000, pp.65-69.
2. Ibrahim, A., Yacoub, S., Ammar, H., Dabney, J and Lateef, K. "**Automated Verification of Timing Constraints in UML Dynamic Specifications**", submitted to Real-Time Technology and Applications Symposium, **RTAS'2001**, Taipei, Taiwan, ROC, May 29-June 1, 2001.
3. Ibrahim, A., Yacoub, S., Ammar, H., Dabney, J and Lateef, K. "**Automated Verification of Timing Constraints in UML Dynamic Specifications**", Submitted to the **Journal of Automated Software Engineering**.
4. Ibrahim, A., Yacoub, S., Ammar, H. "**Automated Architectural-Level Risk Analysis for UML Dynamic Specifications**", submitted to Software Quality Management 2001, **SQM 2001**, Loughborough , UK, April 18 -20, 2001.

5. Ibrahim, A., Ammar, H. "**A Fault Model for Fault Injection analysis of Dynamic UML Specifications**", submitted to 12th International Symposium on Software Reliability Engineering, **ISSRE 2001**, Hong Kong, Nov 28- Dec 1, 2001.
6. Ammar, H., Cortellessa, V., Ibrahim, A. "**Modeling resources in a UML-based simulative environment**", Accepted by ACS/IEEE International Conference on Computer Systems and Applications, **AICCSA 2001**, Beirut, Lebanon, June 26-29, 2001.

1998 – 2000 GlobalOne Egypt Network

Operations Manager

1. Responsible for software and hardware maintenance of ALCATEL Telnet Processors "TPs for Frame Relay and X.25 Switching", MUXs "IDNX 70 from NET and DataSMART from Kentrox" and DSL modems "Paradyne and Nokia".
2. Commissioning, startup and troubleshooting "connectivity and BERT, Bit Error Rate Testing" of intentional Frame Relay & X.25 links in coordination with Egypt Telecomm, customers and GlobalOne's remote POP.
3. Analysis and breakdown of settlement, revenue and cost of backbone and services.
4. Provided sales technical support for Frame Relay & X.25 international services.

1995 – 1998 NCR Corporation Egypt Branch

System Engineer, CSS Customer Support Services

1. Installed and supported integrated information solutions for different classes of customers with Windows NT and AT&T UNIX over varied NCR server platforms, TCP/IP on Cisco routers and LANVIEW "Cabletron Systems Inc." Network Management software.
2. **Awarded Employee of the Month July 1996** for the outstanding achievements in installing amazing variety of products "S10, S40, LAN Switches and Cisco Routers" and Operating Systems "Windows NT and Novel", for a large new pharmaceutical factory.
3. Launched the 23 site WAN of Monofia University, "Paradyne Modems, Cisco Routers and SCOUNIX operating system".

1994 – 1995 Siemens

Industrial Automation Engineer

1. Preformed Maintenance and troubleshooting of Siemens S5 PLC controlled machines in several factories.
2. Replaced the obsolete PLC control unit of a cement crusher plant for the National Cement company with SIMATIC S5 115U and rewrote and tested the software in STEP 5 language.
3. Engineered the software for Hans duplex elevator control unit, "SIMATIC S5 100U".
4. Conducted Step 5 introductory training courses.

Education

2000 – Present West Virginia University, West Virginia, USA

Masters of Science in Electrical Engineering URL: www.csee.wvu.edu

Major: Software Engineering Minor: Control Systems Expected GPA: 3.89

Thesis: Scenario-based Verification & Validation of Dynamic UML SpecificationsDeveloped Methodologies for dynamic UML specifications on Timing Analysis, Early verification of timing constraints, Architectural – Level risk assessment and Performance analysis.

1997 – 1999 Maastricht School of Management, MSM, Maastricht, Netherlands

Masters of Business Administration URL: www.msm.nl

Major: International Business

Project: Competing Through Manufacturing.Studied El-Nile Clothing company's competitive edge through planning its manufacturing strategy based on John Miltenburg's framework.

1997 Microsoft Certified System Engineer

Windows NT 4.0 MCSE

1989 – 1994 Cairo University

Bachelor of Science in Electrical Engineering

Major: Computer & Control Minor : Electronics & Communication Top 10% of class

Graduation Project: Deadlock Problem in Distributed DatabasesImplemented a distributed database system and applied a deadlock prediction then detection algorithm developed at Cairo University in 1986.

Further Information

Some Tools:

Rational Rose RealTime 6.1 "UML modeling and simulation of real-time models"

ObjecTime Developer 5.2 "ROOM modeling and simulation"

Software Through Pictures STP "Computer-Aided Software Engineering (CASE) tool"

Microsoft Visual C++ and J++

Visual Basic, VB Script.

Some Courses:

CPE391, Real-Time Software Engineering, WVU.§CPE391, Object-Oriented Programming in C++, WVU.

CPE391, Fundamentals of Object-Oriented Concurrent Programming in Java, WVU.

IMSE277, Engineering Economy, WVU.