DESIGN OF LOW COMPLEXITY FAULT TOLERANCE FOR LIFE CRITICAL SITUATION AWARENESS SYSTEMS

BY

DHASHRATH RAGURAMAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Advisor:

Professor Lui Sha

ABSTRACT

In cyber-human-medical environments, coordinating supervisory medical systems and medical staff to perform treatment in accordance with best practice is essential for patient safety. However, the dynamics of patient conditions and the non-deterministic nature of potential side effects of treatment pose significant challenges. This work covers my contribution to one such system in development of its low complexity workflow which enhances situation awareness and in the design and implementation of it fault tolerance.

In the first part of this document, we cover a validation protocol to enforce the correct execution sequence of treatments, preconditions validation, side effects monitoring and checking expected responses based on pathophysiological models. The proposed protocol organizes the medical information concisely and comprehensively to help medical staff validate treatments.The proposed protocol dynamically adapts to the patient conditions and side effects of treatments. A cardiac arrest scenario is used as a case study to verify the safety properties of the proposed protocol.

In the second part of this document, we describe the integration of some well understood fault tolerance strategies in context of safety critical systems. We list out the requirements of our system and explore the traditional Active/Standby in context of certain guiding design principles to fit our specific requirement. Like any software engineering project, we design test suites to ensure QOS[1]. We go a step further and try to make this design verifiable using model checking tools like UPPAAL to demonstrate the correctness of our system architecture under conditions of normal operation and failure.

[1]Quality of Service

To Dharma Paati,my grandmother,you are sorely missed.

ACKNOWLEDGMENTS

I would like to briefly thank a number of people who contributed to my successful completion of this project. I am immensely grateful to Prof. Lui Sha, my thesis advisor. His encouragement convinced me to pursue this project and he offered much guidance along the way.
I am grateful to Dr. Jeonghwan Choi for indulging me with multiple brainstorming sessions that helped me iron out some of the finer details.
Thanks also to Po−Liang, Min Young, Maryam and Andrew who mentored me throughout the initial stages and bailed me out without a second thought whenever I got stuck.
Special thanks to Dr. Berlin for his invaluable medical expertise during our group meetings. I would also like to thank the other team of doctors, Dr. White, Dr. Johnson and Dr. Hill for their useful insights on the requirements of the project.
My parents and siblings also require much thanks for their patience, support, and encouragement throughout my education. I would also like to thank Sreedevi for her help in editing and proofreading this document. Finally, I would like to thank all of my friends who have encouraged me to pull through tough times and made my graduate experience enjoyable.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Cardiac Resuscitation System: Overview 2
1.2 Cardiac Resuscitation System: Requirements 3

CHAPTER 2 WORKFLOW MANAGER 7
2.1 Objective 7
2.2 Controller Design 8
2.3 Case Study 12
2.4 Verification 13

CHAPTER 3 FAULT TOLERANCE MANAGER 15
3.1 Introduction 15
3.2 System Requirements 22
3.3 System Overview & Design 24
3.4 Verification 28
3.5 Iterative Design: Combining Testing & Model Checking 33
3.6 Future Work 35

CHAPTER 4 CONCLUSION 36

REFERENCES 37

APPENDIX A TEST SUITES 40

APPENDIX B VERIFIED PROPERTIES 43

CHAPTER 1

INTRODUCTION

Improving the safety and efficacy of healthcare infrastructure is an important issue for cyber-physical medical systems. Statistics indicate that the fraction of preventable medical errors is highest in the ICU when compared to other hospital units[1, 2]. Many preventable medical errors are result from concurrent and uncoordinated treatment actions leading to unintended deviations from the best practice medical guidelines. This is primarily because medical staff are under tremendous pressure and overloaded by the great amount of unorganized information[3]. In order to reduce preventable medical errors, treatment validation is an important aspect. In the first part of this work, we focus on reducing preventable medical errors by validating treatments and monitoring patient's physiological responses based on the pathophysiological models[4, 5]. Validation includes checking preconditions of the treatment, monitoring potential side effects, and checking physiological responses.
On the whole, there is motivation to develop a comprehensive situation awareness enhancement system that is central to the clinical production environment which aids doctors in reducing preventable medical errors. The Cyber−Physical Systems group at University of Illinois, Urbana−Champaign headed by Prof. Lui Sha, which I am part of, are working on one such project. The Resuscitation System[1] aims at improving patient care by introducing human in the loop automation. Introduction of automation in this space requires a certain reliability of the system. This was our motivation in designing a Fault Tolerance Manager that ensures reliable operation of the system[2]. My thesis research deals primarily with the design and implementation of the fault tolerance manager.

[1]The project in question
[2]With great power comes great responsibility

1.1 Cardiac Resuscitation System: Overview

1.1.1 Main UI

The MainUI acts as a integrated situation awareness display that is primarily operated by the physicians enabling them to:

- Get information on important patient vitals giving doctors a holistic idea of patient state.
- Get detailed visualization of actions taken till now as well as logging.
- Get risk oriented display which provides a prioritized display of information based on the risk to the patient.
- Clustering information based on organ systems.

1.1.2 Workflow Manager(WM)

The workflow manager drives the user interface to help physicians follow the resuscitation treatment guidelines and prevent safety hazards. In addition, when patient adverse events occur, Workflow manager will highlight the next steps to be taken[6].

1.1.3 Medical Order Manager(MOM)

The Medical order manager is the primary input device operated by the nurses. It automates the process of data input and eases the ordering of different actions issued by the nurse during patient care.

1.1.4 Fault Tolerance Manager(FM)

An auxiliary manager that handles distributed fault tolerance strategies and enables correct and continuous operation of above three components.[3]

[3]The Main UI, MOM & WM are pre-existing software components that were independently developed by members of my research group that I helped improve. My primary contribution is towards the Fault Tolerance Manager.

1.2 Cardiac Resuscitation System: Requirements

To enable full potential, our research group built the Resuscitation System for enhancing situation awareness in collaboration with Carle Foundation and the MDPNP team heralded by the Massachusetts General Hospital(MGH). The Resuscitation System has potential to be generalized to any organ system model. This system is driven by the workflow that was described about in the previous section.
Such systems need to meet some important requirements:

- Distributed
- Realtime
- Safety Critical

We will touch on each of these aspects before moving on to the later part of my work which involves designing a fault tolerance strategy for the system enabling it to become robust and deploy-able in real world life critical situations.

1.2.1 Distributed Requirement

A useful summary of distributed computing concerns is included in Deutsch's Eight Fallacies of Distributed Computing[7]. All of these are useful to consider in realtime system design; each is a departure point for essential design and implementation concerns:

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero

- The network is homogeneous

In our production environment, our system is designed to enhance situation awareness of physicians during cardiac resuscitation. In such an environment, we can realize how network reliability and latency are critical components and fundamental requirements. Two such scenarios are elucidated below:

- To maintain patient privacy in compliance with HIPAA, it is essential that all communication with in the system are secure.
- In our fault tolerance strategy, we deploy the active standby principle which involves cut over to a backup system and introducing replicas. This incurs partial topology changes in the system.

1.2.2 Realtime Requirement

A real-time system is a system in which the timeliness of operation completion is a part of the functional requirements and correctness measure of the system. In reality, nearly all systems might be considered 'soft' real-time, in that there are usually unspoken requirements/expectations for the timeliness of operations. We reserve the realtime term, for systems which are incorrect when time constraints are not met. Note that many of the concerns summarized in the fallacies above intersect with timeliness. In our system where latency and failure are real and non-trivial factors, the explicit management of computing and communication resources to effect timeliness and other design requirements becomes more important, and the separation of these two dimensions becomes important. Consider the following scenario:

- During the operation of the Medical Order Manager and MainUI, messaging latencies can cause incorrect logging of orders and risk the patient healthcare. Hence messaging must follow strict ordering and time constraints.

1.2.3 Safety Critical Requirement

Safety-critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment. There are many well known examples in application areas such as medical devices, aircraft flight control, weapons, and nuclear systems. The Boeing 777 is described by Boeing as one of the most technologically advanced airplane in the world. Many different technologies have contributed to the aircraft including safety-critical computer systems. There are six primary flat-panel displays and several other smaller displays in the cockpit. The aircraft has several major computerized systems to aid the pilot including flight management and enhanced ground proximity warning. Much of the traditional mechanical and hydraulic equipment is obviated by the use of a fly-by-wire control system. Such systems require specific verifiable design requirements and fault tolerance strategies. The field of medicine is no different. Computers are used in medicine far more widely than most people realize. The idea of using a microprocessor to control an insulin pump is quite well known. The fact that a pacemaker is largely a computer is less well known. The extensive use of computers in surgical procedures is almost unknown except by specialists. Computerized equipment is making inroads in procedures such as hip replacement, spinal surgery, and ophthalmic surgery. In all three of these cases, computer controlled robotic devices are replacing the surgeons traditional tools, and providing substantial benefits to patients. In our the cardiac resuscitation project, we take this a step further. We make computers human centric, aiding physician make decisions and avoid preventable medical errors which puts the patient's life is at risk. Hence correct, robust and fault tolerant operation becomes the highest priority.

1.2.4 Composition of Realtime and Distributed

Listed below are some general guiding principles when dealing with such composite systems.

- Explicit timeliness requirements: Representing the time constraints explicitly in the design and implementations, and detecting and recovering from failures.
- Time synchronization: Requirements and mechanisms for achieving clock synchrony. Many applications require only NTP.
- Synchrony requirements: This is connected to clock synchrony, but not

identical. Emphasis on partial event ordering.

- Design patterns for the requirements.

- Middleware: Encoding the distributed aspects of the system. Examples include Real-Time CORBA.

- Time Constraints: Documentation, measurement and enforcement of time constraints in the system.

- Partial Failure: A real-time system typically has reliability requirements. One of the unique aspects of distributed systems is the potential for whole classes of failures called 'partial' failures, due either to true crash/communications failures or timeliness errors that must be treated as failures.

- RTOS: Working with real time operating systems.

In the next chapter we go over the design of the Workflow Manager's Treatment Validation protocol[6][4] as well as the Fault Tolerance Manager with these above mentioned aspects in mind. My main contribution and focus has been towards the Fault Tolerance Manager. My first part of the contribution was towards the Workflow Manager. This was much more as a role to ramp me up to understand all the components of the system and its design complexity. I next focused on designing and implementing the Fault Tolerance Manager for the resuscitation system.

[4]This work was my initiation into the team and I worked with PHD Candidate Po−Liang Wu on this

CHAPTER 2

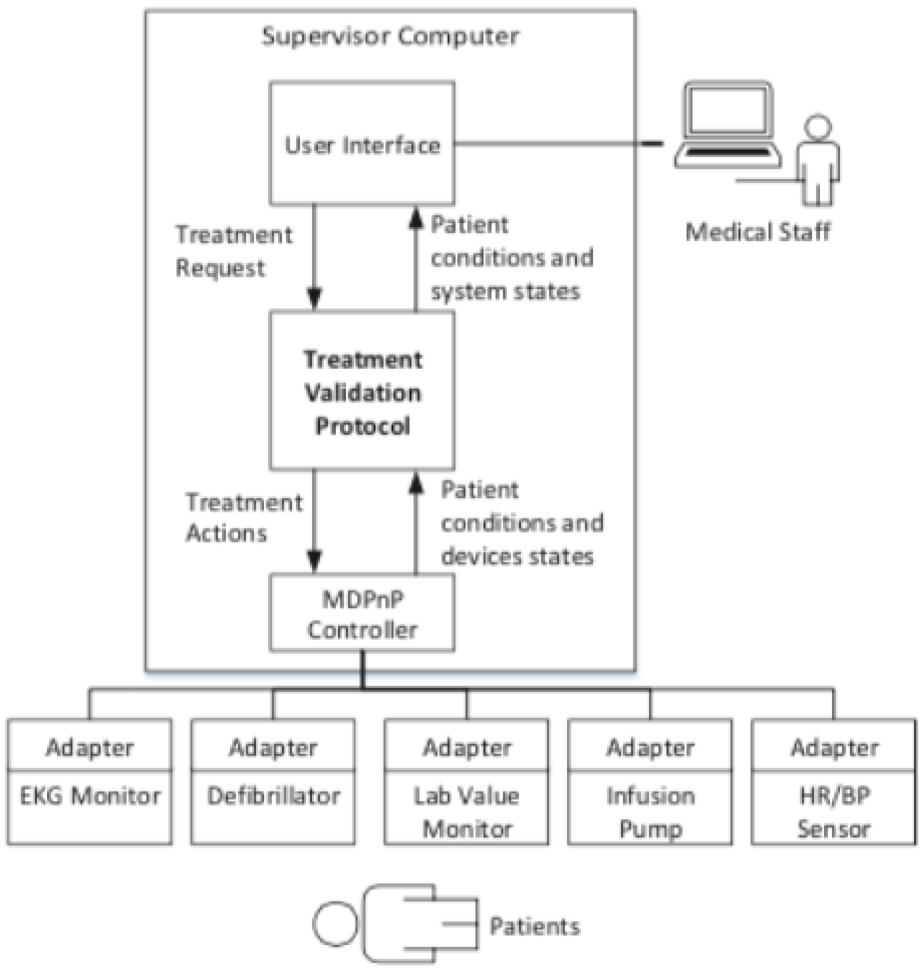
WORKFLOW MANAGER

2.1 Objective

In cyber-physical-human medical environments, coordinating supervisory systems and medical staff to perform treatments in accordance with best practice is essential for patient safety. However, the dynamics of patient conditions and the non-deterministic nature of potential side effects of treatments pose significant challenges. We model the system from previously and currently researched systems[8, 9, 10, 11, 12, 13] and the controller design derived from them is presented. We propose a validation protocol to enforce the correct execution sequence of performing treatment, regarding preconditions validation, side effects monitoring, and expected responses checking based on the pathophysiological models. The proposed protocol organizes the medical information concisely and comprehensively to help medical staff validate treatments. Therefore, the proposed protocol dynamically adapts to the patient conditions and side effects of treatments. The protocol is only advisory and can be overridden by the physician. During such an override the workflow manger readjusts the protocol to fit in the new requirements or overrides specified by the medical personnel. A cardiac arrest scenario is used as a case study to verify the safety and correctness properties of the proposed protocol.

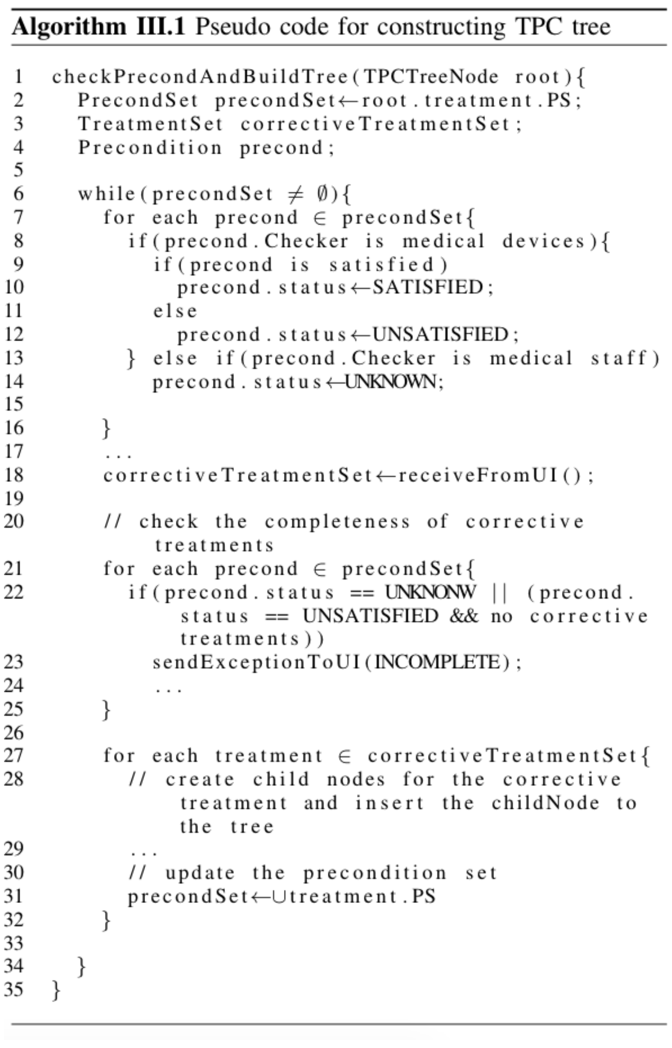
2.2 Controller Design

Here is a hierarchical design of our preexisting controller[5, 4, 14, 15] on which the algorithm is designed to work:



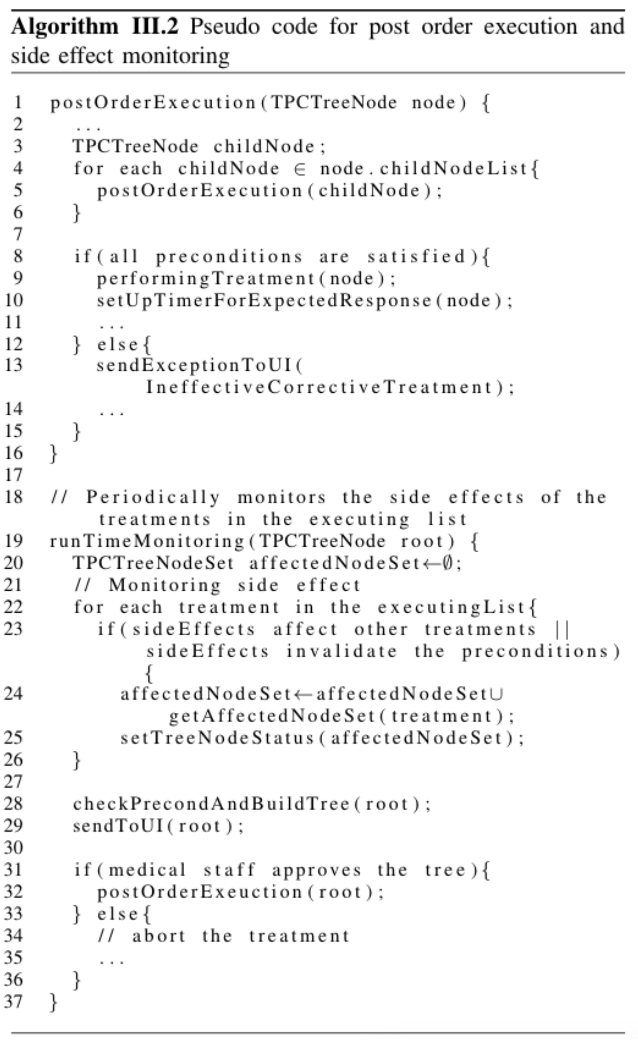
2.2.1 Resuscitation Workflow Algorithm

The following pseudocodes outline the algorithms involved in the treatment validation protocol:



Algorithm III.1 Pseudo code for constructing TPC tree

```
checkPrecondAndBuildTree(TPCTreeNode root){
  PrecondSet precondSet←root.treatment.PS;
  TreatmentSet correctiveTreatmentSet;
  Precondition precond;

  while(precondSet $\neq \emptyset$){
    for each precond $\in$ precondSet{
      if(precond.Checker is medical devices){
        if(precond is satisfied)
          precond.status←SATISFIED;
        else
          precond.status←UNSATISFIED;
      } else if(precond.Checker is medical staff)
        precond.status←UNKNOWN;

    }
    ...
    correctiveTreatmentSet←receiveFromUI();

    // check the completeness of corrective
        treatments
    for each precond $\in$ precondSet{
      if(precond.status == UNKNONW || (precond.
          status == UNSATISFIED && no corrective
          treatments))
        sendExceptionToUI(INCOMPLETE);
        ...
    }

    for each treatment $\in$ correctiveTreatmentSet{
      // create child nodes for the corrective
          treatment and insert the childNode to
          the tree
      ...
      // update the precondition set
      precondSet←∪treatment.PS
    }

  }
}
```



Algorithm III.2 Pseudo code for post order execution and side effect monitoring

```
postOrderExecution(TPCTreeNode node) {
  ...
  TPCTreeNode childNode;
  for each childNode ∈ node.childNodeList{
    postOrderExecution(childNode);
  }

  if(all preconditions are satisfied){
    performingTreatment(node);
    setUpTimerForExpectedResponse(node);
    ...
  } else{
    sendExceptionToUI(
        IneffectiveCorrectiveTreatment);
    ...
  }
}

// Periodically monitors the side effects of the
    treatments in the executing list
runTimeMonitoring(TPCTreeNode root) {
  TPCTreeNodeSet affectedNodeSet←∅;
  // Monitoring side effect
  for each treatment in the executingList{
    if(sideEffects affect other treatments ||
        sideEffects invalidate the preconditions)
        {
      affectedNodeSet←affectedNodeSet∪
          getAffectedNodeSet(treatment);
      setTreeNodeStatus(affectedNodeSet);
  }

  checkPrecondAndBuildTree(root);
  sendToUI(root);

  if(medical staff approves the tree){
    postOrderExeuction(root);
  } else{
    // abort the treatment
    ...
  }
}
```

2.2.2 Protocol Design

- **TPC Tree**[1][6] **construction phase**

  The system receives a treatment from the medical staff and starts to build a TPC tree in a breath-first manner. The system checks the preconditions of the received treatment. If any precondition is not satisfied or must be checked by the medical staff, the system sends the tree to the medical staff and requests them to check the preconditions and specify the corrective treatments, as shown in the line 7-19 [2] of Algorithm III.1. After getting the input from the medical staff, the system checks

[1]Treatment Precondition and Correction - Formal name for the data structure used to handle treatment validation

[2]Discussed in the previous subsection

if each unsatisfied precondition has a corresponding corrective treatment. If the corrective treatments are incomplete, an exception is sent to the medical staff, as shown in the line 21-24 of Algorithm III.1. The system then adds the corrective treatments as child nodes to the TPC tree. Since the corrective treatments may introduce a new set of preconditions, the system checks the preconditions and expands the tree. If there are no further preconditions to check or all the preconditions are satisfied, TPC tree is sent to the medical staff for approval. If the medical staff approves the TPC tree, the system enters the execution and monitoring phase.

- **Execution and monitoring phase**
  The system executes the treatments in the TPC tree in a post order. In order to keep track of all the ongoing treatments, the system maintains an executing treatment list. Since patient conditions dynamically change, the system checks the preconditions of the treatment again before performing it. If the preconditions are satisfied, the system inserts the treatment into the executing list and requests the medical devices to perform the treatment. The system needs to check the expected response after a time interval, specified by the medical staff, as shown in the line 8-15 of Algorithm III.2. The details of checking expected responses will be explained in the next phase. In addition, the system periodically monitors or requests the medical to check the potential side effects of the treatments in the executing list. The side effects may lead to the following situations:

  1. The side effects of a treatment interfere the other ongoing treatments. Specifically, the side effects cause the patients physiological measurements changing in an opposite direction to the expected responses of other treatments.
  2. The side effects invalidate the previously satisfied preconditions in the TPC tree.

  In both cases, the system will highlight the interfered treatments and the corresponding preconditions in the TPC tree and send an exception to the medical staff, as shown in the line 22-25 of Algorithm III.2. The medical staff can adjust the existing treatments, such as increasing or decreasing the drug dosage, or specifying alternative treatments. The system then updates the tree, as described in the previous phase. After the system informs the side effects to the medical staff and updates the TPC tree with their approval, the system restarts the post order execution.

- **Checking expected responses phase**
  As explained in the previous phases, the system must check patients conditions against the expected responses of the treatment when the timer fires. If the patient conditions are as expected, the system removes the corresponding treatment node from the TPC tree and executes the next treatments based on the post order of the TPC tree. If the patient conditions do not improve as expected, the system highlights the unsuccessful preconditions and the corresponding corrective treatments on the TPC tree for the medical staff. The medical staff can specify an alternative corrective treatment, and the system updates the TPC tree accordingly and restarts the post order execution.
  By following the above procedures, the system preforms the treatments and corrects the preconditions in a bottom-up manner. Even if the side effects adversely affect other treatments or invalidate the preconditions, the system is capable of updating the TPC tree and let medical staff change the treatments.

2.3 Case Study

Cardiac arrest is the abrupt loss of heart function and can lead to death within minutes. The American Heart Association (AHA) provided resuscitation guidelines for the urgent treatment of cardiac arrest [16]. The general procedures of resuscitation consist of the following steps:

1. Cardiopulmonary resuscitation (CPR): The medical staff perform CPR for at least two minutes. In the mean time, other medical staff try to access intravenous vein (IV therapy) and inject drugs.

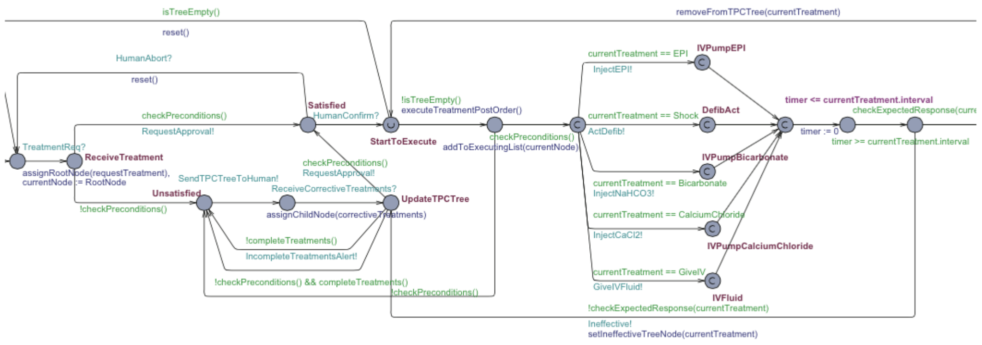
2. Check heart rhythm: The medical staff check the heart rhythm. If the rhythm is non-shockable, the medical staff should keep performing CPR

and giving drugs, such as vasopressin and epinephrine. If the rhythm is shockable, they proceed to the defibrillator step.

3. De-fibrillation: If the medical staff determine the heart rhythm is a shockable one, they should activate a defibrillator to deliver electrical energy to the heart to regulate the heart rhythm. If the patients heart rhythm is still abnormal, the medical staff should perform CPR again (back to step 1). Furthermore, the side effects of the treatments may cause adverse interactions. Therefore, the medical staff should closely monitor the patients conditions and preform alternative treatments if the side effects occur.

2.4 Verification

We model the proposed protocol in UPPAAL[17, 18]. The system consists of the following models: user interface, validation protocol, side effect monitor, EKG monitor, defibrillator, IV Pump, blood pH monitor, and urine flow rate sensor. The models communicate using UPPAAL synchronization channels and shared variables. The user interface models follows the three step resuscitation procedure and contains a list of predefined preconditions and treatments. The medical devices send the patients physiological measurements, which are modeled as non-deterministic transitions, to the validation protocol. In addition, the medical devices also receive the treatment requests from the protocol and change the states accordingly.



isTreeEmpty()
reset()
HumanAbort?
reset()
TreatmentReq?
ReceiveTreatment
assignRootNode(requestTreatment),
currentNode := RootNode
checkPreconditions()
RequestApproval!
Satisfied
HumanConfirm?
StartToExecute
!isTreeEmpty()
executeTreatmentPostOrder()
checkPreconditions()
addToExecutingList(currentNode)
Unsatisfied
!checkPreconditions()
SendTPCTreeToHuman!
ReceiveCorrectiveTreatments?
assignChildNode(correctiveTreatments)
UpdateTPCTree
checkPreconditions()
RequestApproval!
!completeTreatments()
IncompleteTreatmentsAlert!
!checkPreconditions() && completeTreatments()
!checkPreconditions()
removeFromTPCTree(currentTreatment)
currentTreatment == EPI
InjectEPI!
IVPumpEPI
currentTreatment == Shock
ActDefib!
DefibAct
IVPumpBicarbonate
currentTreatment == Bicarbonate
InjectNaHCO3!
currentTreatment == CalciumChloride
InjectCaCl2!
IVPumpCalciumChloride
currentTreatment == GiveIV
GiveIVFluid!
IVFluid
timer <= currentTreatment.interval
timer := 0
timer >= currentTreatment.interval
checkExpectedResponse(curre
!checkExpectedResponse(currentTreatment)
Ineffective!
setIneffectiveTreeNode(currentTreatment)

The evaluation environment is shown in 2.1

The medical safety properties(tabulated above) capture the safety requirements of the resuscitation scenario, which are given by the medical staff. The

Model Checker	UPPAAL 4.0.13
Processor	Intel I7
RAM	4 GB
Parameters	Number of hierarchical layers and queue size
Metrics	State space consumed, Verification time

Table 2.1: Evaluation environment

	Verified Properties
Safety	P1: Defibrillator is activated only if the EKG rhythm is a shockable one and airway and breathing are normal
	P2: Epinephrine is injected only if the blood PH value is larger than 7.4 and urine flow rate is higher than 12 mL/s
	P3: Sodium bicarbonate is injected only if calcium chloride is not currently being injected.
	P4: If epinephrine does induce a shockable, the TPC tree is updated with a new treatment node for injecting vassopressin
Protocol	P5: There are no deadlock in the system
	P6: A treatment is performed if and only if all its preconditions are satisfied
	P7: If side effect does not occur, the root node of the TPC tree is added to the execution list
	P8: If the side effects invalidate a precondition, the TPC tree is updated and well formed

Table 2.2: Verified properties of the resuscitation scenario

protocol properties guarantee the correctness of the proposed protocol.

CHAPTER 3

FAULT TOLERANCE MANAGER

The following section outlines our primary work in designing and implementing the Fault Tolerance Manager for the Resuscitation Project.

3.1 Introduction

Complex safety critical systems currently being designed and built are often difficult multi-disciplinary undertakings. In order to ensure that these systems perform as specified, even under extreme conditions, it is important to have a fault tolerant computing system. A number of recent trends, such as harsh environments, novice users, larger and more complex systems, and downtime costs etc. have accelerated interest in making general purpose computer systems fault tolerant. The primary goals of fault tolerance are to avoid downtime and to ensure correct operation even in the presence of faults. System performance, minimally defined to be the number of results per unit time times the uninterrupted length of time of correct processing, should not be compromised. In real systems, however, price/performance trade-offs must be made; fault tolerance features will incur some costs in hardware, in performance, or both. Fault-tolerant computing can hence be loosely defined as the correct execution of a specified algorithm in the presence of defects. The effect of defects can be overcome through the use of temporal redundancy (repeated calculations) or spatial redundancy (extra hardware or software). These systems are usually classified as either highly reliable or highly available. As in all system design, the system goals and specifications constrain the design space and consequently the design techniques that may be used. At the highest level of specification, fault tolerant systems are categorized as either highly available or highly reliable.

- Availability A(t): The availability of a system as a function of time is the probability that the system is operational at any instant of time t. The limiting availability is the expected proportion of the time that the system is available to run useful computations. Activities such as preventive maintenance and repair reduce the time the system is available to the user.

- Reliability R(t): The reliability of a system as a function of time is the conditional probability that the system has survived the interval [0, t], given that it was operational at time t = 0. Reliability is used to describe systems in which (1) repair cannot take place or is too costly (e.g., a satellite computer); or (2) the computer is serving a critical function and cannot be lost even for the duration of a repair (e.g., a flight computer on board an aircraft, or the control of a power distribution network).

Availability is frequently used as a figure of merit in systems for which service can be delayed or denied for short periods without serious consequences. For a system in which downtime costs tens of thousands of dollars per minute(e.g. airline reservation system) an increase of only .1% availability makes a substantial difference. In general, highly available systems are easier to build than highly reliable systems because of the more stringent requirements imposed by the reliability definition. Our emphasis in incorporating fault tolerance into the resuscitation project is exactly directed to the safety and life critical aspects of the system and we are focused on building a reliable architecture. In our system, reliability takes way more precedence than availability.

3.1.1 Background

To understand our approach lets first discuss the commonly used terms Failures,Faults and Errors.

1. Classification based on state of system.

    - Failure: Changes in hardware that produce unacceptable results or behaviors leading to requirement violations.
    - Fault: Erroneous perceived state of hardware or software resulting from failures of components, physical interference from the environment, operator error, or incorrect design.
    - Error: Manifestation of a fault within a program or data structure. The error may occur some distance from the fault site. Could be by design.

2. Temporal classification applicable to each of above

    - Permanent: Describes a failure, fault, or error that is continuous and stable. In hardware, permanent failure reflects an irreversible physical change. The word 'hard' is used interchangeably with the word permanent.
    - Intermittent: Describes a fault or error that is only occasionally present due to unstable hardware or varying hardware or software states(e.g. as a function of load or activity).
    - Transient: Describes a fault or error resulting from temporary environmental conditions. The word 'soft' is used interchangeably with transient.Transient faults and intermittent faults are the major source of system errors. The distinguishment between these two types of faults are ability of repair. We consider transient faults are not repairable, and intermittent ones as repairable. The manifestations of transient and intermittent faults and of incorrect hardware or software design are much more difficult to determine than permanent faults

.

Lets address these one at the time.

- In our system, errors typically would lead to incorrect medical information being propagated. Main UI described in 1.1.1 is an integrated situation awareness display that the doctors will rely on to make diagnosis. The whole point of the system is to reduce preventable medical errors caused by physicians having wrong mental model of patient state. If errors are allowed creep into such data, the purpose is defeated. To prevent such errors there are well studied implementations in information theory like Cyclic Redundancy Check codes (CRC) that guarantee the absence of these errors. Over and above, the workflow can validate abnormal values and flag them.

- We recognize faults and failures as very similar entities in the semantics of our safety critical system. Since the system also has the real-time and distributed properties, we chose to employ a human in the loop active/standby system incorporating a human in the loop hot swap as well to add more replica standbys later.

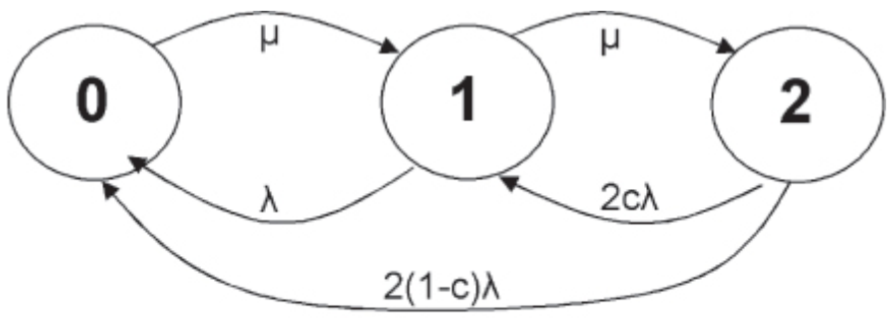
- Our system deals with at most one permanent failure. The system is heterogeneous with the Medical Order Manager being used for issuing treatment and logging by the nurses and being stateless while the MainUI+Workflow are operated by doctor and is stateful. We care only about recovering and failures involved with MainUI. The Fault Tolerance manager allows the hot swap of Medical Order Manager at anytime. Hence we go with the architecture where the Medical Order Manager can potentially become the hot standby for the MainUI+Workflow. We use the hot swap feature provided by the Fault Tolerance manager to deploy more replicas as need arises.

- Regarding Transient or Intermittent failures which we deal together as partial failure scenarios, Our low complexity safety critical architecture requires us to avoid dealing with specific partial failures. We will touch base on this in the later sections.

3.1.2 Reliability Engineering and Analysis

Software Reliability Engineering is the practice of monitoring and managing the reliability of a system. By collecting fault, error, and failure statistics during development, testing, and field operation, monitoring and managing the parameters of reliability and availability is possible. The Handbook of Software Reliability Engineering[19] contains a number of articles on topics related to Software Reliability Engineering. A useful reliability testing technique is Reliability Growth Modeling, which graphs the cumulative number of faults corrected versus time. Prediction methods calculate the cumulative number of faults expected, which enables comparison with the measured results. This, in turn, enables the determination of the number of faults remaining in the system. Another widely used technique for predicting the reliability of a system is Markov modeling. These models enable analysis of redundancy techniques and prediction of MTTF[1].Markov models are constructed by defining the possible system states. Transitions between the states are defined and are assigned a probability factor. The probability indicates the likelihood that the transition will occur. An important aspect of the model is that the probability of a state transition depends only on the current state; history is not considered. Below fig shows a simple Markov model for a duplex system in which either system may fail with probability (λ) and be restored to service with probability (μ) and a coverage factor c. The failure rate, (λ) , is the inverse of the MTTF, and the repair rate ,(μ), is the inverse of MTTR[2].

[1]Mean Time To Fail
[2]Mean Time To Repair



States: number of redundant units

* λ failure rate
* μ repair rate
* c coverage factor

$$Unavailability = (1-c)\frac{2\lambda}{\mu} + 2\left(\frac{\lambda}{\mu}\right)^2, \ \mu >> \lambda$$

3.1.3 Low Complexity Requirement and the Fail-Fast Model

Complexity analysis is wide spread in algorithm design as well as software system design. In very basic terms, if we compare two programs of same size, the one with more decision-making statements will be more complex as the control of program jumps frequently. From this basic definition to more complex metrics like the Cyclomatic Complexity Number(CCN) and corresponding Complexity Adjustment Factors (CAFs), there are in depth studies to understand complexity. There are various benefits of having low complexity. We will touch on some that are relevant to our cause. For a safety critical system, design verification as well as source code verification of the fault tolerance manager is very essential. Being able to say that our system can be model checked and simulate faults enables us to create better system designs. Low Complexity systems have this added advantage of being easier to verify as well as providing backward compatibility on updated software version. To achieve this we incorporate the fail fast model. Under the traditional development model,making your software robust by working around problems automatically and setting up fail safe assertions leads to the software 'failing slowly'. The program continues working right after an error but fails in strange ways later on. The Fail-Fast paradigm addresses partial failure tolerance from the viewpoint of design complexity. Under this model, when a partial failure scenario is encountered, the system fails immediately, visibly and quickly. Failing fast is a non-intuitive technique : 'failing immediately and visibly' sounds like it would make your software more fragile, but it actually makes it more robust. Bugs are easier to find and fix, so fewer go into production. In safety critical aspects its all the more important to know exactly in what scenario the system would fail. Doing away with partial failures, not only helps us avoid the state space explosion problem which makes verification a difficult aspect of the module, but also makes the software development phase more robust minimizing the iterative needs of deploying hot fixes once in production. It also helps avoid unexpected corner case behaviors that traditional testing and quality assurance practices do not catch.

3.2 System Requirements

While setting out to build our fault tolerance strategy, we need to first list out the requirements for our system.

1. Failure Detection:

    - Failure detector should be pessimistic. By pessimistic we mean, it may permitted to mis-predict a currently 'up' process to be failed due to reasons like message latencies(especially in Asynchronous communication between Medical Order Manager and Main UI).
    - Failure detector should be complete. This is a more important factor than accuracy since you expect every failure to be caught.
    - Heartbeat channel should be independent of Messaging Channel- In our system we separate the functional message passing between Medical Order Manager and Main UI by introducing a independent third component on each of devices capable of monitoring each of the processes. Hence it is essential that the heartbeat channel remains decoupled from the messaging channel.
    - Intra Device Heartbeat- Another feature of the Failure detector which applies to partial failure is to be able to tell if individual software components/processes (Main UI/Workflow Manager/Medical Order Manager) monitored by the Fault Tolerance Manager stop responding. For this purpose the Fault Tolerance Manager maintains a internal heartbeat of each of the components. We address this in the next section.

2. Reliable Wired vs unreliable Wireless network and input consistency.

3. State:

    - Stable state storage: Our crash recovery model is based to ability to recover from a saved state. This requires the process to log their state after every action. There is necessity for these operations to be atomic which we will call as a stable storage operation. We place some restrictions for example, the medical order manager should not process cannot store a variable in stable storage and then send a message or issue an external output in a single atomic step. This restriction is guiding principle and requirement for building each stateful component that operate with Fault Tolerance Manager.

- State Sync: The state of the various processes should be recoverable the stable storage. Towards this end, the Fault tolerance manager should be periodically be able to sync the state of system from one device to another enabling crash recovery.

4. Misc Monitoring Functionality: In close relation to the state sync, the Fault tolerance manager should be able to attribute the following functionality

    - Ability to Manually start/restart one or more processes on the device.
    - Ability to Manually terminate one or more processes on the device.
    - Ability to Manually sync the state across the Active Device and Standby device.
    - Ability to Automatically restart one or more processes upon failure and restore state in case we are dealing with partial failures[3].

5. Verification requirement: The fault tolerance manager model should be verifiable.

In the next section we will see how each of these requirements are satisfied.

[3]Note: we choose not to deal with partial failures as we shall see in the later sections

3.3 System Overview & Design

The fault tolerance manager is the central service running on the distributed components of the resuscitation system. Reliable continuous operation of our system is crucial for safety critical requirements. The fault tolerance manager ensures to satisfy QOS requirements for our system.
The Fault Tolerance Manager attributes the following functions in its implementation:

- Heartbeat
  Heartbeat failure detectors and Heartbeat messages are a widely accepted method of checking if all the functioning components of the system are alive or not. The Fault tolerance manager has a built in heartbeat detector that detects the up-time for each component and upon failure invokes the recovery module.
  Heartbeating ensures two properties of the system[20]:

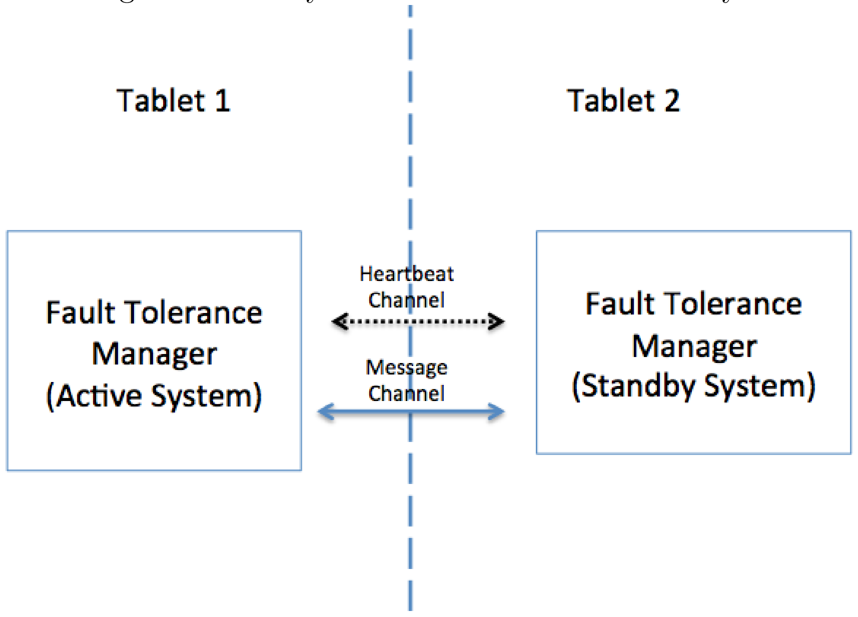
  – Proof of liveliness property: Between our two Fault Tolerance Manager Processes, each process p_i receives a HEARTBEAT message, upon receipt of which it increments the sequence in its own book keeping (could be as simple as maintaining a hash table or array with suitable inputs). If the heartbeat refuses to increase for the TIMEOUT period, the detector marks the process as failed and executes recovery routine.

  – Proof of completeness property: In the safety critical environment, we cannot leave things to chance. Once a process is suspected of failure(whether it be device failure or latency in heartbeats), the process is marked failed. Hence we avoid missing failure by being over zealous in our definition of what is assumed to be failed satisfying the completeness property i.e. all failures are potentially detected.

  The heartbeating property is verifiable using our UPPAAL model satisfying the verification requirement. Moreover, the heartbeat channel is independent of fault tolerance satisfying our other requirement mentioned in 3.2.

- Control Messages

The Fault Tolerance manager adds a layer of messaging between other similarly functioning managers in other components, that helps process control messages such as 'System Init' and 'Start Recovery'.


Tablet 1
Tablet 2
Fault Tolerance Manager (Active System)
Heartbeat Channel
Message Channel
Fault Tolerance Manager (Standby System)

- Process Control

  The Fault tolerance manager has complete control of the various processes and services running within the device, enabling it to check status of every process and issue suitable actions. For ex: When it detects a process is hung, it can either send out a control message to enable recovery or kill the hung process and restart it with suitable configuration. To implement this feature we have the following four checks:

  1. Internal Heartbeat of each process with the Fault Tolerance Manager. This could potentially detect partial failure but we treat them as a complete failure to avoid verification overhead.
  2. The Process.Responding [21] windows API call will indicate if a process that is executing a windows message loop is responding or not.
  3. Poll the memory usage for the process at intervals and if it doesn't change after enough time, assume that it is hung. One can do this with Process.WorkingSet64 API[22]. The drawback for this method is it can generate false negatives which is compliant with

our completeness requirement for safety critical systems[4].

- State Integrity Check
  A critical aspect of state recovery in the Active/Standby model is that the state recovered must be sane[5]. The fault tolerance manager incorporates validations(syntactic) that ensures the recovered state is not gibberish. Further human in the loop validation is necessary to check the semantics. Hence it helps in rollback/crash recovery.

- State Transfer

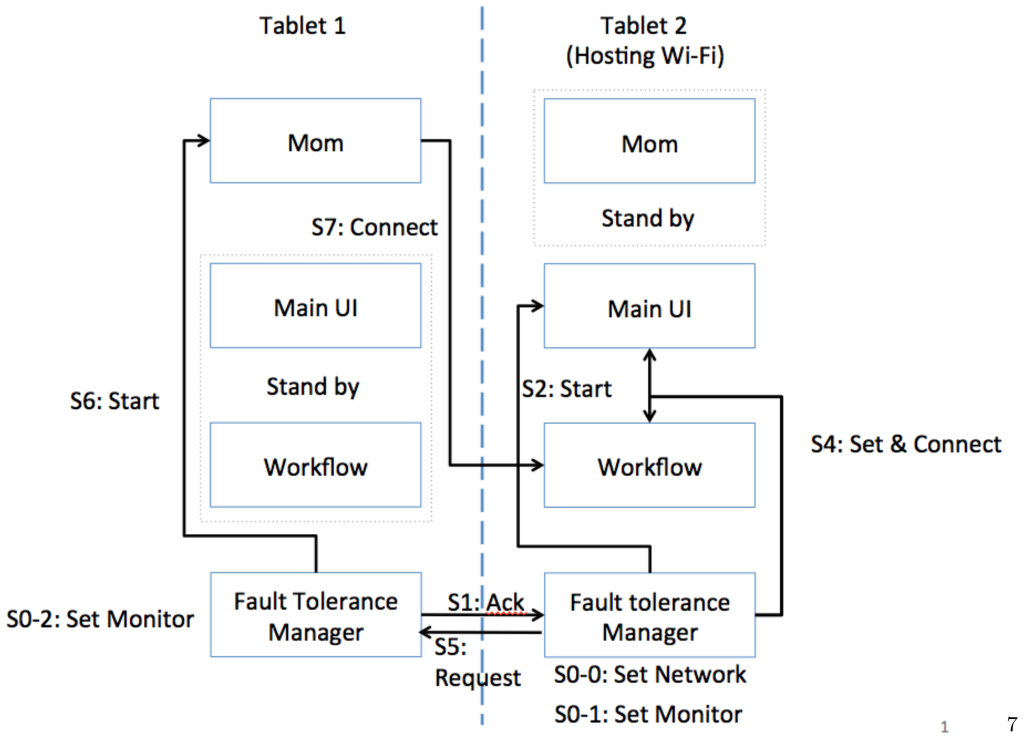
  1. The Fault tolerance manager also attributes the periodic syncing of state and logs from the active system to the standby system over the wireless network. This state transfer channel operates on the same channel as the control messaging channel which is independent of the Application messaging channel and heartbeat channel.
  2. Since above network communication is wireless,it adds a certain aspect of unreliability.to mitigate this, we add the writing of the logs to a stable storage using atomic operations. This stable storage in our system is a flash based plug and play storage that can be introduced to the standby system upon resumption. The required software hooks on the standby side would prioritize this over the network relayed state. The state integrity checker should be capable of picking up the newer version based on the timestamps in the logs [6].
  3. Input consistency management:Though this is not available in the current implication. The standby system can potentially have a buffer that serializes messages with a sequence number. A missed message is cached (i.e if the system sees a application layer message sequence 'n' and then one that's greater than 'n+1', it caches it in the buffer) and flushed accordingly. At present we rely on the human operator to catch any missed messages.

[4]Better safe than sorry

[5]In context of quality assurance sanity check

[6]This helps the standby come up to speed and handle any messaging latencies caused during the state transfer

3.3.1 System Launch Flow



[7]Mom is Medical Order Manager

3.4 Verification

3.4.1 Introduction

Testing and verification are essential to the creation of fault tolerant computing systems because they ensure that fault prevention and quality efforts are successful. Testing and verification also provide the data needed by a projects software reliability engineers to compute the expected reliability of a system.

A useful kind of testing for fault tolerant systems is operational profile testing. An operational profile describes the usage of the system in quantitative terms and the most typical scenarios that the system will process. This information helps define the most appropriate tests to be run, and how to focus testing efforts. Operational profiles are the scenarios that are used in design, development, and test. To test the reliability and performance of the system the operational profile adds quantitative information to the descriptions of typical scenarios

Software fault insertion testing is done by providing erroneous inputs to the system. One way is to alter the normal input to determine the systems behavior to incorrect values. This is commonly called boundary testing. The next level beyond this is to place hooks into the software, or use a debugging tool, to enable internal values and state to be altered. Testers insert known faults into the system and the systems behavior is monitored. This testing serves the dual purpose of identifying faults in the systems error handling processes and of providing data for the computation of coverage factors. Fault insertion testing is the only way that a systems coverage factor can be determined[8]. Known faults are introduced into the system, which is then observed to see if the system was able to handle the faults automatically. We explore each of these aspects bundled with model checking using the UPPAAL model checking tool.

[8]The coverage is computed as the percentage of cases in which recovery was successful

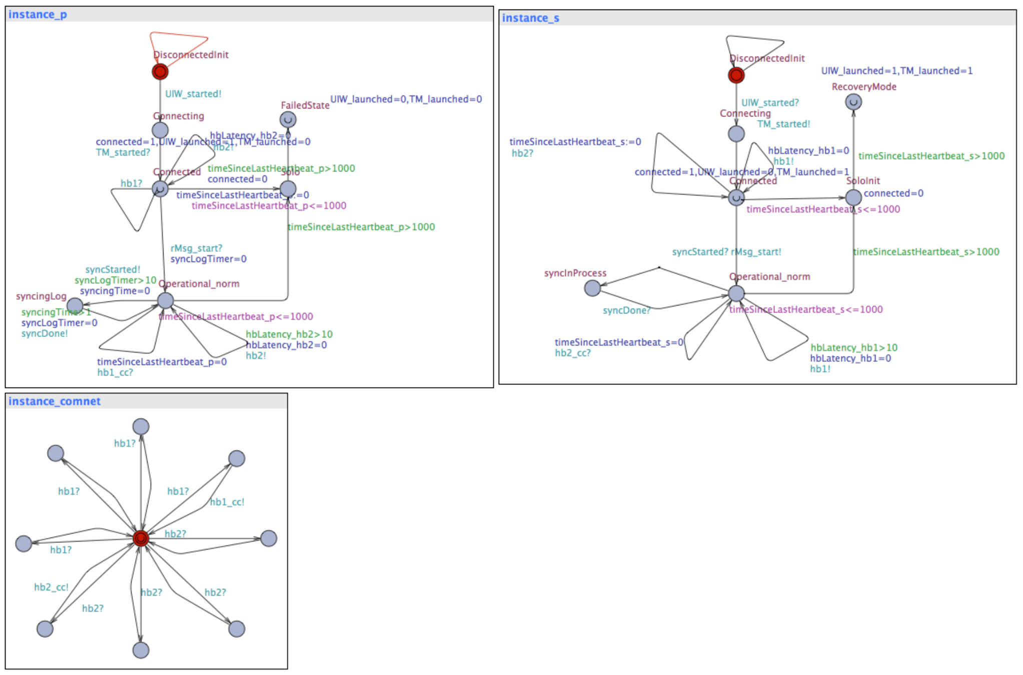
3.4.2 On Distributed Systems

Distributed systems have proven to be hard to understand, design, and reason about due to their complexity and non-deterministic nature. They usually involve subtle interactions of a number of components that have high level of parallelism. This is why the correctness of these systems is difficult to ensure. Several systems and protocols have been proven not to succeed in satisfying their intended goals after they have been published[23]. One of the promising solutions to this problem is the use of formal verification techniques such as the model checking technique.

3.4.3 Verification nuances of the Fault Tolerance Manager

We use the model checking tool UPPAAL to generate timed automaton model of our system. UPPAAL enables us to do a myriad of verification profiles.

Simplified Model:



In the above model we have three components whose behavior is mirrored based on the source code of the Fault Tolerance Manager. The communication network model can be tweaked to give us various failure testing scenarios and simulate recovery.

The evaluation environment is shown in 3.1

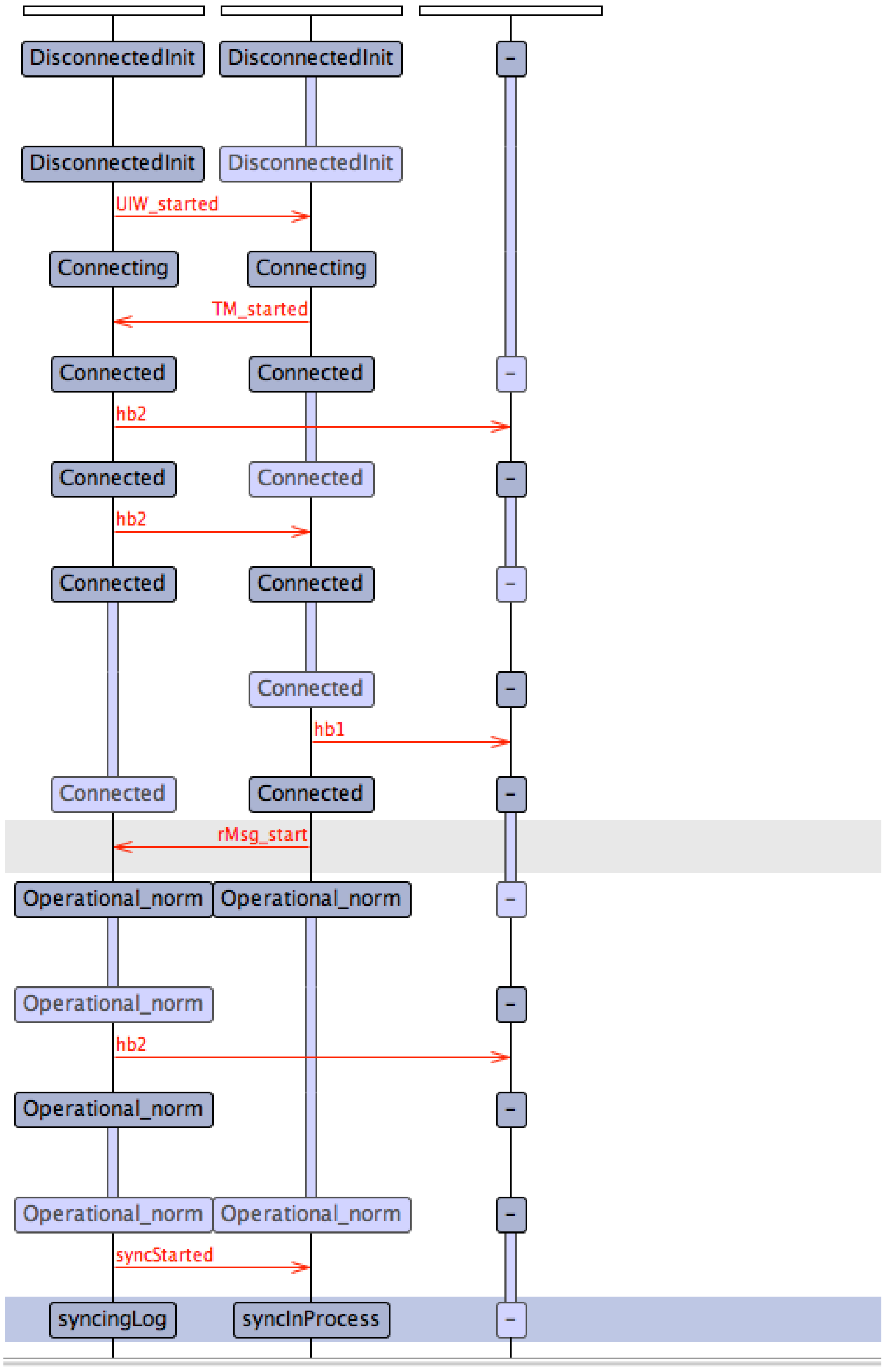
Model Checker	UPPAAL 4.1.20
Processor	Intel I7
RAM	16 GB
Metrics	Deadlocks,State space consumed and verification time

Table 3.1: Evaluation environment

Components Modelled:

- Instance of Primary(Active)
- Instance of Secondary(Standby)
- Communication Network

Figure 3.1: Time series operational snapshot of Model



As depicted in 3.1, under normal operation, the states remain in (**Operational norm**, **Operational norm**) and periodically enter the state transfer phase (**syncingLog**, **syncingProcess**). **rMsq start** is one of the control messages that allows the system to launch as per a certain required configuration ensuring synchronization. When heart beating fails, each of the instance can enter 'SoloInit' state and recover its state from the logs and enter 'RecoveryMode'.
We have verified the system to be deadlock free and correctly operational under simulated message packet drops in the communication model.
To simulate packet drops, we use probabilistic chance (state transitions) of messages being dropped by controlling weights in the communication channel model. This essentially acts as a black box and we can completely take down the channel at run time by parsing an array of inputs processed at different time slices. We can also verify out of order messages to simulate effect of latencies and our model (and in turn the source code) is capable of handling such inputs. Complete device failure is simulated as subset of the communication model failure since we do not deal with partial system failure under our fast fail design.
More information on various testing profiles are tabulated in Appendix A.
More information on the verified UPPAAL Properties are tabulated in Appendix B.

3.5 Iterative Design: Combining Testing & Model Checking

Our initial few testing suites and UPPAAL models revealed several flaws in the source code implementations. The following list is not exhaustive but is exemplary of some of bugs that we were able to catch at early testing stages and fix.

1. **Race Conditions**
   It is interesting to note that the Fault Tolerance manager handles interaction between three distinct independent components of the system that we described in section 1.1. It also provides a process control and monitoring framework. When such loosely coupled systems interact,you end up encountering race conditions. Our initial testing environment helped us detect these race conditions and enabled us to add suitable synchronization routine to help avoid race conditions. Here is a specific scenario:

   (a) Medical Order Managers attempts to communicate with the Main UI with either the Main UI/Workflow not already running or running with misconfiguration. Upon detecting this design flaw, the Fault Tolerance Manager was tweaked to incorporate a Control Messaging Channel that would enable message passing between the distributed FM[9] entities and add a acknowledgement based synchronization design into the system. This also makes each instance of our system more stateful, helping the model checking phase of the project[10],[11] .

   (b) Our synchronization routine ensures that the Main UI and Workflow are connected before the Medical Order Manager connects.

2. **Deadlocks**
   The initial few UPPAAL model based on the first few implementations revealed several deadlocks in our system which we first fixed in the UPPAAL model and then extended the same design principle to the Fault

[9]FM short for Fault Tolerance Manager

[10]The revised architecture helps us create the UPPAAL model more intuitively

[11]Again the modelling semantics of UPPAAL alone would not help us catch this since UPPAAL models by their very nature highly synchronous

Tolerance Manager source code. These deadlocks were rarely encountered during the testing phase. In safety critical systems it is essential to be able to say ”NO” with high confidence. Thus an exhaustive state space exploration via UPPAAL is one good way to do so which is what we strive to achieve.

3. **Towards real world communication networks**
   Our initial UPPAAL model did not model communication layer unreliability as in the case with real-world wireless networks. We hence added another component to probabilistically be able to simulate packet losses. This component also enabled us to externally control input to each of the instances.This is very powerful as it enabled us to manually insert erroneous inputs and check the behavior of the system.

Hence this iterative design strategy helped both ways i.e. the UPPAAL model helped in improvement of the source of implementation and understanding the nature of the source code helped us improve the UPPAAL model. This approach essentially serves like a feedback control system improving the systems overall design and implementation qualitatively. All of the above mentioned design principles are not confined to our system alone. Our team is already working on a general purpose system capable of handling other medical scenarios like Sepsis or critical care during patient transport and hope to follow similar guidelines for their Fault Tolerance Strategy. In this futuristic world that has started to move towards autonomous robotic systems such as self driving cars, advanced avionics and even simple devices such as autonomous vacuum cleaners, there will always be scope for a Fault Tolerance Manager. We strive to create some best practice design principles that is generic enough to apply to each of these field and our iterative testing/modelling approach would also be advantageous to ensure error free design.

3.6 Future Work

1. In the current work, the implementation is specific to the Resuscitation System. There scope to improve on this and make it generic enough to be able to extend it and make it reusable towards other allied situation awareness systems like the Sepsis System.

2. In the current system, the period of operation is in no way a limiting factor as per design[12]. However, the reliability testing and the verification model are time sensitive entities and might require additional changes and modifications.

3. From a Software Engineering Effort aspect, a good update to have is to be able to pickup testing scenarios via configuration files. In Software Engineering there is a practice known as Continuous Integration Testing. In this practice, every time a software modification is done, the testing and verification suites are automatically triggered and re-run. To enable such features later in the project encoding test cases into configuration files would be very helpful.

4. We mentioned the Reliability Growth Modeling and Markov modelling techniques for reliability testing in 3.1.2. There is potential to incorporate such testing strategies into our current implementation thereby enabling us to make stronger claims of robustness and fault tolerance.

5. There is scope to improve the User Interface Design of the Fault Tolerance Manager. The current design is not tested against usability and is purely functional in nature.

[12]Clinical procedures like resuscitation span minutes while treatment for severe sepsis can span weeks

CHAPTER 4

CONCLUSION

In closing,this document gives a brief summary of my contribution towards two essential components of the resuscitating project. A lot of the implementation details are beyond the scope of this document and the respective documentation of each module can be found on the subversion host repository of the project[1]. Both the Workflow Manager and Fault Tolerance Manager strive to be low complexity entities that are verifiable as well as adhere to design principles necessary in building safety critical systems.I am grateful and delighted to have been presented with the opportunity to work on them by Prof. Lui Sha who has guided me along every step.

[1]http://mdpnp.cs.illinois.edu/svn/ResuscitationProject/

REFERENCES

[1] D. J. Cullen, B. J. Sweitzer, D. W. Bates, E. Burdick, A. Edmondson, and L. L. Leape, "Preventable adverse drug events in hospitalized patients: a comparative study of intensive care and general care units," *Critical care medicine*, vol. 25, no. 8, pp. 1289–1297, 1997.

[2] A. Latif, N. Rawat, A. Pustavoitau, P. J. Pronovost, and J. C. Pham, "National study on the distribution, causes, and consequences of voluntarily reported medication errors between the icu and non-icu settings*," *Critical care medicine*, vol. 41, no. 2, pp. 389–398, 2013.

[3] L. T. Kohn, J. M. Corrigan, M. S. Donaldson et al., *To err is human: building a safer health system.* National Academies Press, 2000, vol. 627.

[4] W. Kang, P. Wu, M. Rahmaniheris, L. Sha, R. B. B. Jr., and J. M. Goldman, "Towards organ-centric compositional development of safe networked supervisory medical systems," in *Proceedings of the 26th IEEE International Symposium on Computer-Based Medical Systems, Porto, Portugal, June 20-22, 2013*, 2013, pp. 143–148.

[5] M. Rahmaniheris, W. Kang, L. Lee, L. Sha, R. B. B. Jr., and J. M. Goldman, "Modeling and architecture design of an mdpnp acute care monitoring system," in *Proceedings of the 26th IEEE International Symposium on Computer-Based Medical Systems, Porto, Portugal, June 20-22, 2013*, 2013, pp. 514–515.

[6] P. Wu, D. Raguraman, L. Sha, R. B. B. Jr., and J. M. Goldman, "A treatment validation protocol for cyber-physical-human medical systems," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, Verona, Italy, August 27-29, 2014*, 2014, pp. 183–190.

[7] P. Deutsch, "The eight fallacies of distributed computing," https://blogs.oracle.com/jag/resource/Fallacies.html, 1991.

[8] P.-L. Wu, W. Kang, A. Al-Nayeem, L. Sha, R. B. Berlin Jr, and J. M. Goldman, "A low complexity coordination architecture for networked supervisory medical systems," in *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*. ACM, 2013, pp. 89–98.

[9] A. L. King, L. Feng, O. Sokolsky, and I. Lee, "A modal specification approach for on-demand medical systems," in *Proceedings of the 3rd International Symposium on Foundations of Health Information Engineering and Systems*, 2013.

[10] W. Kang, P. Wu, L. Sha, R. B. Berlin, and J. M. Goldman, "Towards safe and effective integration of networked medical devices using organ-based semi-autonomous hierarchical control," University of Illinois at Urbana Champaign, Tech. Rep., 2012. [Online]. Available: http://hdl.handle.net/2142/34774

[11] M. Pajic, R. Mangharam, O. Sokolsky, D. Arney, J. Goldman, and I. Lee, "Model-driven safety analysis of closed-loop medical systems," *IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS*, 2012.

[12] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam, "From verification to implementation: A model translation tool and a pacemaker case study," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*. IEEE, 2012, pp. 173–184.

[13] R. Alur, D. Arney, E. Gunter, I. Lee, J. Lee, W. Nam, F. Pearce, S. Van Albert, and J. Zhou, "Formal specifications and analysis of the computer-assisted resuscitation algorithm (cara) infusion pump control system," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 5, no. 4, pp. 308–319, 2004.

[14] D. Arney, M. Pajic, J. Goldman, I. Lee, R. Mangharam, and O. Sokolsky, "Toward patient safety in closed-loop medical device systems," in *Proceedings of the 1st ACM/IEEE ICCPS*. ACM, 2010.

[15] J. Goldman, S. Whitehead, S. Weininger, and M. Rockville, "Eliciting clinical requirements for the medical device plug-and-play (MD PnP) interoperability program," *Anesthesia & Analgesia*, vol. 102, pp. S1–54, 2006.

[16] J. M. Field, M. F. Hazinski, M. R. Sayre, L. Chameides, S. M. Schexnayder, R. Hemphill, R. A. Samson, J. Kattwinkel, R. A. Berg, F. Bhanji et al., "Part 1: executive summary 2010 american heart association guidelines for cardiopulmonary resuscitation and emergency cardiovascular care," *Circulation*, vol. 122, no. 18 suppl 3, pp. S640–S656, 2010.

[17] G. Behrmann, A. David, and K. Larsen, "A tutorial on uppaal," *Lecture Notes in Computer Science*, pp. 200–236, 2004.

[18] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, ser. LNCS, M. Bernardo and F. Corradini, Eds., no. 3185. Springer–Verlag, September 2004, pp. 200–236.

[19] Michael R. Lyu, *Handbook of Software Reliability Engineering*. IEEE Computer Society Press, 1996.

[20] Michel Raynal, *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems*. Morgan and Claypool Publishers, 2010-06-06.

[21] "MS Windows NT windows api dot net framework 4.5," https://msdn.microsoft.com/en-us/library/system.diagnostics.process.responding.aspx, accessed: 2015-03-15.

[22] "MS Windows NT windows api dot net framework 4.5," https://msdn.microsoft.com/en-us/library/system.diagnostics.process.workingset64.aspx, accessed: 2015-03-15.

[23] O. Al-Bataineh, T. French, and T. Woodings, "Formal modeling and analysis of a distributed transaction protocol in uppaal," in *Temporal Representation and Reasoning (TIME), 2012 19th International Symposium on*, Sept 2012, pp. 65–72.

APPENDIX A TEST SUITES

Testing ID	**Description**	**Expected Result**
SystemInit	1. Start the Fault Tolerance Manager 2. Press "Start System" button	1. Main UI, Medical Order Manager, Workflow Manager starts 2. These three programs connects successfully 3. Heartbeat is available on both the devices
ManStartProg	1. Start the Fault Tolerance Manager 2. • Press the Start MainUI Only button • Press the Start Workflow Only • Press the Start MOM Only	1. Main UI starts 2. Workflow Manager starts 3. Medical Order Manager starts[2]

Table A.1

Testing ID	**Description**	**Expected Result**
ManKillProg	1. Start the Fault Tolerance Manager 2. Press "Start System" button 3. • Press the Kill MainUI Only button • Press the Kill Workflow Only • Press the Kill MOM Only	1. Main UI Terminates 2. Workflow Manager terminates 3. Medical Order Manager Terminates[3]
AutoProgFail	1. Start the Fault Tolerance Manager 2. Press "Start System" button 3. Kill one of three programs manually	1. Main UI, Medical Order Manager, Workflow Manager all start successfully on standby 2. These three programs connects successfully 3. State is recovered from stable storage 4. Resumption flow is initiated upon restart 5. Physicians can continue to operate on stand by device

Table A.2

Testing ID	**Description**	**Expected Result**
FTHbFail	1. Start the Fault Tolerance Manager 2. Press "Start System" button 3. Kill network	1. Main UI, Medical Order Manager, Workflow Manager all start successfully on standby 2. These three programs connects successfully 3. State is recovered from stable storage 4. Resumption flow is initiated upon restart 5. Physicians can continue to operate on stand by device

Table A.3

APPENDIX B VERIFIED PROPERTIES

Type	Description
Safety Properties	P1: Medical Order Manager is active utmost on one instance.
	P2: WorkFlow Manager is active utmost on one instance
	P3: Main UI is active utmost on one device
	P4: Fault Tolerance Manager is active on both devices
Protocol properties	P5: There is no deadlock in the system
	P6: Control Messages and Heartbeat are on different communication channels
	P7: When Active instance is in OperationalNorm state(state of normal dual device operation), Standby instance is also in OperationalNorm state
	P8: When Active instance is syncingLog State, Standby instance is in syncinProgress case and vice versa
	P9: If Hearbeat Timeout $>$ 10 units while in any dual operation states[4], either instance would be in SoloInit state[5]
	P10: Instance would go into RecoveryMode only after successfully launching all three components
	P11: Standby Instance would go into RecoveryMode state implies previously active instance is in Failed state and vice versa[6]

Table B.1