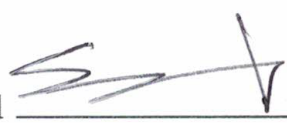# A FRAMEWORK FOR MANAGING UNSPECIFIED ASSUMPTIONS IN SAFETY-CRITICAL CYBER-PHYSICAL SYSTEMS

BY

ZHICHENG FU

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved _____
Advisor

Chicago, Illinois
May 2020

ACKNOWLEDGMENT

I am grateful to my advisors, teammates, friends, and family members for their help on my PhD journey. My adviser, Prof. Shangping Ren, has been a constant source of inspiration. I am particularly grateful to her for introducing me to the assumption problems in safety-critical cyber-physical systems. She always takes so much time to discuss research with her students. This thesis would not have taken shape without her suggestions and guidance, based on her vast experience, brilliant insight, and her innumerable rounds of feedback. I am also grateful to Dr. Lui Sha from the University of Illinois Urbana-Champaign. His personal stories and remarkable anecdotes about every possible situation in life are etched permanently in my memory.

My friends and teammates ensured that the life of a Ph.D. student was not all work and without fun. I thank Dr. Chunhui Guo, Dr. Hao Wu, Dr. Xiayu Hua, Dr. Andrew-Yizhong Ou, Dr. Zhenyu Zhang for their help and suggestions throughout my Ph.D. program. They have made my path towards my Ph.D. a joyful one.

I was fortunate to have a very helpful committee. I want to thank Prof. Eunice Santos, Prof. Kevin Jin, and Pro. Jialing Xiang for their helpful suggestions and guidance on this thesis.

My family members have played the most important part in this seemingly daunting journey. I am blessed to have a beautifully supportive family around me. They share my happiness and sadness and witness the fulfillment of me. When I countered difficulties, they often help me to figure the solutions and encourage me to face them. They make me know that a happy, loving and warm family is the core and the base of our lives. Thank you, my parents, who bring me to the world, and raise me in love. I wish you are healthy and happy forever. Thank you, my elder brother and sister in law. Because of you two, I could leave home without much

concern about our parents. Thank you, my fiancee, we will spend the rest of our lives together, raising our family together, being successful in our careers together, and sharing every wonderful moment with our loved ones.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

For a cyber-physical system, its execution behaviors are often impacted by its operating environment. However, the assumptions about a cyber-physical system's expected environment are often informally documented, or even left unspecified during the system development process. Unfortunately, such unspecified assumptions made in cyber-physical systems, such as medical cyber-physical systems, can result in patients' injures and loss of lives. Based on the U.S. Food and Drug Administration (FDA) data, from 2006 to 2011, there were 5,294 recalls and 1,154,451 adverse events resulting in 92,600 patient injuries and 25,800 deaths. One of the most critical reasons for these medical device recalls is the violations of unspecified assumptions. These compelling data motivated us to research unspecified assumptions issues in safety-critical cyber-physical systems, and develop approaches to reduce the failures caused by unspecified assumptions.

In particular, this thesis is to study the issues of unspecified assumptions in cyber-physical system design process, and to develop an unspecified assumption management framework to (1) identify unspecified assumptions in system design models; (2) facilitate domain experts to perform impact analysis on the failures caused by violating unspecified assumptions; and (3) explicitly model unspecified assumptions in system design models for system safety validation and verification.

Before starting to develop the unspecified assumption management framework, we first need to study how unspecified assumptions may be introduced into cyber-physical systems. We took cases from the FDA medical device recall database to analyze the root causes of medical device failures. By analyzing these cases, we found two important facts: (1) one of the major reasons that causes medical device recalls is violation of some unspecified assumptions; and (2) unspecified assumptions are often introduced into the system design models through *syntactic carriers*. Based on the

two findings, we propose a framework for managing unspecified assumption in cyber-physical system development process. The framework has three components. The first component is called the Unspecified Assumption Carrier Finder (*UACFinder*), which is to identify unspecified assumptions in system design models through automatically extracting *syntactic carriers* associated with unspecified assumptions. However, as the number of unspecified assumptions identified from system design models can be large, and it may not be always feasible for domain experts to validate and address the most safety-critical assumptions at different system development phases. Therefore, the second component of the framework is a methodology that uses the Failure Mode and Effects Analysis (FMEA) based prioritization approach to facilitate domain experts to perform impact analysis on unspecified assumptions identified by the *UACFinder* and asses their safety-critical level. The third component of the framework describes a model architecture and corresponding algorithms to model and integrate assumptions into system design models, so that system safety associated with these unspecified assumptions can be validated and formally verified by existing tools.

We also have conducted case-studies on representative system models to demonstrate how *UACFinder* can identify unspecified assumptions from system design models, and how the FMEA based prioritizing approach can facilitate domain experts to verify the appropriateness of identified assumptions. In addition, case studies are also conducted to demonstrate how system safety properties can be improved by modeling and integrating unspecified assumptions into system models. The results of case-studies indicate that the unspecified assumption management framework can identify unspecified assumptions, facilitate domain experts to validate and verify the appropriateness of identified assumptions, and explicitly specify assumptions that would cause defects in these systems.

xi

CHAPTER 1

INTRODUCTION

An assumption is defined as "a thing that is accepted as true or as certain to happen, without proof" [1] or as "a fact or statement taken for granted" [2]. Making assumptions during the system development process is at all inevitable. During the system development life-cycle, every time a decision is made about how to design an interface, how to implement an algorithm, if and how to encapsulate an external dependency, assumptions are made concerning how the system will be used, how it will evolve, and what environments it will operate in. However, the unfortunate aspect of system development is that these assumptions are seldom, if ever, recorded, communicated, or reviewed. We call assumptions that are not explicitly documented or specified but are understood by system designers and developers **unspecified assumptions**. When unspecified assumptions are violated, failures can be introduced. There have been many catastrophic incidents caused by violations of unspecified assumptions that resulted in loss of revenue in tune of hundreds of millions of dollars and loss of lives [3]. Based on the U.S. Food and Drug Administration (FDA) data, from 2006 to 2011, there were 5,294 recalls and 1,154,451 adverse events resulting in 92,600 patient injuries and 25,800 deaths [4].

1.1 Incidents of System Failure Due to Unspecified Assumptions

1.1.1 Ariane 5 Disaster. One of the most well-known failures due to an unspecified assumption made by a software component is that of the Ariane 5 rocket. The summary of the expert analysis in [3] is as follows - "In about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 meters, the launcher veered off its flight path, broke up and exploded". Ariane 5 had reused the same

software which was used by Ariane 4. There was an operand overflow in the Inertial Reference System module, due to a conversion of a 64-bit floating point value to a 16-bit signed integer value. The error occurred because the Ariane 5 was a much bigger rocket than the Ariane 4 and had a higher value for the horizontal velocity component, which overflowed a 16-bit variable.

Though a simplification, this is a classic case of reuse of a software component with unspecified assumptions made on the environment in the system. Though this fact "The horizontal velocity value of Ariane 5 is higher than Ariane 4" was known beforehand, the assumption was not documented. Thus, the assumption made by the old software that the horizontal velocity variable will never overflow 16-bits was left unchecked, it was violated and caused the accident.

1.1.2 HAMILTON-T1 Ventilators Recall. According to the U.S. Food and Drug Administration (FDA) medical device recall database, medical device recalls caused by software failures are at an all-time high [5]. One of the main reasons for the recalls is that the environment can not satisfy medical devices' assumed operation conditions. One recent medical device recall case is HAMILTON-T1 ventilator recall in 2013 [6]. The FDA has labeled this recall as a Class I recall, the most serious type of recall issued by the FDA. According to the manufacturer, there is an unexpected high internal oxygen consumption of HAMILTON-T1 ventilator when they are used on small pediatric patients. The current labeling does not include sufficient information about the internal oxygen consumption of the ventilators. Depending on the ventilator setting and the patient's lung impedance, the internal gas consumption of the HAMILTON-T1 may be higher than expected. The internal oxygen consumption has to be taken into account in addition to the oxygen requirements for the patient's ventilation, especially when oxygen is limited. If the available oxygen supply during transport is depleted, the life of the patient may be endangered.

In this recall, the internal oxygen consumption is calculated by $(MinVol + FlowTrigger) \times C \times (FiO_2 - 20.9)/79.1$, where constant $C = 1.5$ is initially set for adult patients – an unspecified assumption that the formula of internal oxygen consumption is used for adult patients. For pediatric patients, the value of C should be 4.0, which is different from the value used for adult patients. When $C = 1.5$ is used for small pediatric patients, it causes unexpected high internal oxygen consumption, which could cause patient in danger or even death.

1.2 Challenges of Managing Unspecified Assumptions

These incidents of system failures in safety-critical cyber-physical systems indicate an inarguable fact that unspecified assumptions are dangerous and can lead to catastrophes. However, system developers constantly make assumptions about the interpretation of requirements, design decisions, operational environment, characteristics of input data, and other factors during system development and deployment phases. But these assumptions are seldom documented and less frequently validated by the domain experts who have the knowledge to verify their appropriateness. Once the operating environment of cyber-physical systems violate the unspecified assumptions, failures may occur and catastrophic accidents may happen resulting in loss of revenue in tune of hundreds of millions of dollars and loss of lives.

Given the importance of assumptions in system development stage, much research has been done in explicitly specifying assumptions. For instance, Lehman and Ramil proposed a few guidelines to correctly specify assumptions [7]. The authors believed that it is necessary to train all stakeholders to identify and record assumptions at all stages of the system development with a standard form or structure. Lewis et al. [8] also developed an assumption management framework for improving the quality of software development. The framework can extract assumptions from source code and record them into a repository for management. Besides the code level assump-

tion management, Tirumala [9] also developed an assumption management framework (AMF) at system component levels. The goal of AMF is to have a well-defined vocabulary to encode assumptions in a machine-checkable format [9]. AMF introduces a systematic process that performs automatic validations for machine-checkable assumptions. This framework addresses the issues caused by unspecified assumptions at component interface levels. These tools and approaches mentioned above are to support assumption description at the requirement gathering phase. However, how to identify unspecified assumptions at an early stage of system development process, and how to facilitate domain experts to perform impact analysis on failures caused by violating unspecified assumptions, and how to help to validate and verify the assumption-based safety-properties through the system development phases has yet to be addressed.

To address these problems, a framework for managing unspecified assumptions needs to solve the following challenges:

1.2.1 Understand How Unspecified Assumptions Exist in Safety-Critical Cyber-Physical Systems. To develop an effective approach for managing unspecified assumptions, we need to understand how unspecified assumptions are unintentionally introduced into the system during the system development cycle. To understand the problems, we have to find and analyze system failure cases caused by unspecified assumptions.

1.2.2 Identify Unspecified Assumptions in System Design Models. Over the last decades, many modeling languages have been proposed and developed to support the design and development of complex software systems [10], [11], [12], [13], [14]. These languages are designed to provide a coherent and unambiguous vision of the system under design, to ease the communication among stakeholders. However, developers often make assumptions throughout the system development life-cycle. Hence,

identifying unspecified assumptions made throughout the system development activities and validating them are essential to ensure the safety of systems that have high safety requirements. To specify assumptions during the system development cycle, many approaches and tools have been developed to facilitate assumptions management [7], [9], [8], [15]. However, even with these tools and approaches, it is still unavoidable that there are unspecified assumptions in systems, just as there are always some bugs left in the code even with good compilers. Similar to we need debugging tools to help to identify bugs at the code level, we also need tools to help uncover potential unspecified assumptions at the model level.

1.2.3 Facilitate Domain Experts to Analyze the Impact on System Failures Caused by Unspecified Assumptions.

The number of unspecified assumptions in system design models can be large and it is not always feasible for domain experts to validate all of them at different development phases. Therefore, ensuring the most safety-critical unspecified assumptions will be validated in time is extremely important. To determine the safety levels of each unspecified assumptions, impact analysis need to be performed on the possible failures caused by unspecified assumptions. Hence, providing an effective way to facilitate domain experts to perform impact analyze becomes essential to ensure system safety.

1.2.4 Model Assumptions and Integrate Assumption Models into System Design Models.

For safety-critical cyber-physical systems, simulation and validation are essential but not adequate to provide safety assurance. We need to ensure that safety properties are always satisfied under both specified and unspecified assumptions. Formal model-based approaches are appealing because they provide a unified basis for formal analysis to achieve the expected level of correctness and safety guarantees. To have the formal system design models, it is necessary to model assumptions and integrate assumption models into system design models first. Then

the integrated models can be transformed into the formal models that existing formal verification tools can be applied to verify system safety properties depended on different assumptions.

In this thesis, we propose a framework to address these challenges in managing unspecified assumptions during system development process. The following section provides a high-level overview of the design of the framework and explain the approach of how these challenges are addressed.

1.3 The Architecture of The Framework

To address the technical challenges presented in Section 1.2, this thesis designs a framework to manage the unspecified assumption in safety-critical cyber-physical systems. The architecture of the framework is depicted in Fig. 1.1.
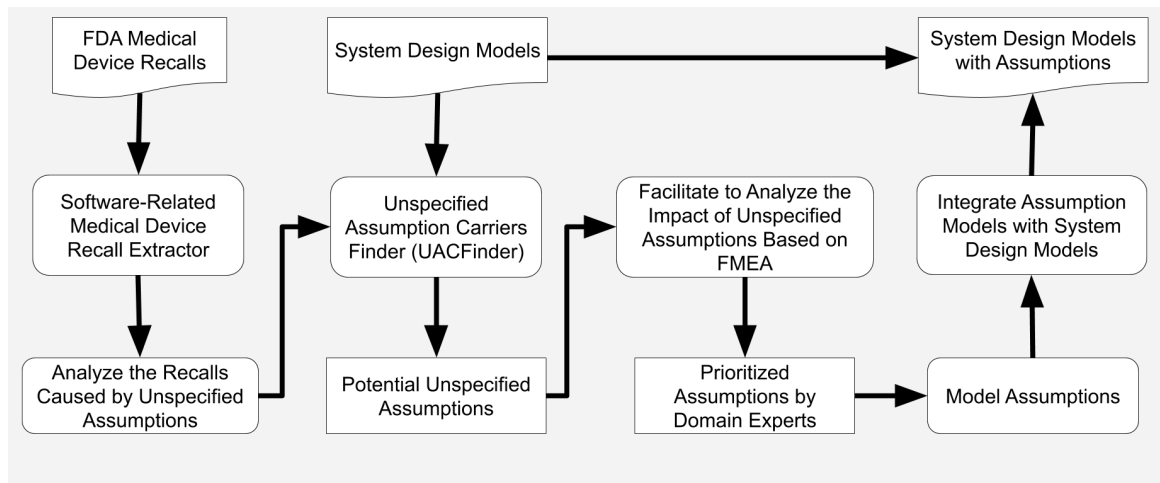
Figure 1.1. The Architecture of the Framework to Manage Unspecified Assumptions in Safety-Critical Cyber-Physical Systems

The implementation of the framework contains four components. A brief introduction of each component is as follows.

1.3.1 Component 1: Understand How Unspecified Assumptions Are Injected to System Design Models. This component takes the FDA medical

device recall database as a dataset. It contains an automatic process that uses data mining algorithms to filter out software-related medical device recalls from the FDA database. We then analyze if and how unspecified assumptions nay have caused medical device recalls. Through the analysis, we find out that unspecified assumptions are often embedded in system design models through syntactical carriers. We have identified three types of such carriers: 1) *constant variables/values*, 2) *frequently read/updated variables*, and 3) *frequently executed actions.*

1.3.2 Component 2: Identify Unspecified Assumptions in System Design Models.

The component is to develop an Unspecified Assumption Carrier Finder (*UACFinder*). The *UACFinder* uses data mining techniques to identify potential syntactic carriers of unspecified assumptions from system design models. The *UACFinder* is currently focusing on mining three types of syntactic carriers: constant variables, frequently read/updated variables, and frequently executed action sequences. Whether unspecified assumptions indeed exist in these carriers are yet to be validated by domain experts or model designers. The techniques used by the *UACFinder* can automatically extract these carriers without requiring any prior knowledge about annotations, templates, or weight assignments from end-users. The component also involves medical doctors in the loop to validate the contents in these carriers found by the *UACFinder.*

1.3.3 Component 3: Facilitate Domain Experts to Perform Impact Analysis on Unspecified Assumptions.

The component develops an approach to facilitates domain experts using Failure Mode and Effects Analysis (FMEA) to determine failure modes when unspecified assumptions are violated. By analyzing the effects of potential device failure modes, the most safety-critical assumptions will be identified so that appropriate action plans can be made to improve system safety in

a timely manner.

1.3.4 Component 4: Model and Integrate Identified Assumptions into System Design Models.

For this component, our strategy is to define an assumption model and composition rules for engineers to explicitly and accurately specify assumptions during system development process. The assumption model is then automatically transformed into a statechart model and integrated with system design models. The integrated system design models are then transferred to formal models with existing tools, such as our proposed tool Y2U [16]. Domain experts can validate the integrated models, and system safety properties associated with assumptions can be formally verified with existing model verification tools.

1.4 Organization of the Thesis

This thesis is organized as follows. Chapter 2 describes the procedures of understanding the problem of how unspecified assumptions are embedded in safety-critical cyber-physical systems. In particular, we develop an automatic approach to extract software-related medical device recalls from the FDA medical device recall database. We then analyze these recalls to understand how the unspecified assumptions cause medical device recalls. Chapter 3 describes the design of unspecified assumption carriers finder (*UACFinder*). The *UACFinder* is designed to take system design models as input, and automatically mine unspecified assumptions carried by *constant variables/values*, *frequently read/updated variables* and *frequently executed actions*. The number of assumptions discovered from system design models can be large, and it is not always feasible for domain experts to validate all of them at different system development phases. Therefore, Chapter 4 presents an approach, i.e., the Failure Modes and Effects Analysis (FMEA) approach, to determines possible failure modes when unspecified assumptions are violated. By analyzing the effects of failure modes, we can highlight the most safety-critical unspecified assumptions. However,

for safety-critical cyber-physical systems, validation by domain experts alone is not adequate for guaranteeing safety, and formal verification such as model checking is required. The formal model-based approach is appealing because it provides a unified basis for formal analysis to achieve the expected level of correctness and safety guarantees. In order to apply formal methods to verify the assumptions-based properties, we need to model the validated assumptions and integrate the assumption models into the system design model. Hence, in the Chapter 5, we present a mathematical model and composition rules for domain experts to explicitly and accurately specify assumptions. The mathematical assumption models can be automatically integrated into system design models. The integrated system design models are then transferred to formal models with existing tools, such as our proposed tool Y2U [16]. So that system safety properties associated with unspecified assumptions can be formally verified with existing model verification tools.

Every chapter in Chapters 3-5, starts with a discussion of background and related work and ends with case studies. In the end, the conclusions and future applications and extensions of the framework are presented in Chapter 6.

CHAPTER 2

ANALYZE UNSPECIFIED ASSUMPTIONS IN SAFETY CRITICAL CYBER
PHYSICAL SYSTEMS

2.1 Background and Related Work

The medical device recalls mentioned in Chapter 1 and many other examples that can be found in the FDA recall database [17] show an inarguable fact that unspecified assumptions in safety-critical cyber-physical systems are dangerous and can lead to loss of human lives. Hence, being able to explicitly and accurately specify assumptions is important to ensure the safety of safety-critical cyber-physical systems. However, before we can step into the actions to address the problem, we must understand how unspecified assumptions are unintentionally introduced into the system during system development processes. To answer the question, we have to analyze an amount of safety-critical cyber-physical system failures caused by unspecified assumptions.

While looking for appropriate dataset of system failures, we noticed that medical devices are often subject to failures that potentially cause catastrophic impacts on patients. The number of such failed cases are significant. Based on the U.S. Food and Drug Administration (FDA) data, from 2006 to 2011, there were 5,294 recalls and 1,154,451 adverse events resulting in 92,600 patient injuries and 25,800 deaths. In addition, there is a total of 7,771 device recalls that were announced by the FDA on its public recall database from September 1, 2012, to August 31, 2015. The large number of failures in medical cyber-physical systems indicates that the FDA medical device recalls is an appropriate dataset for analysis.

To analyze the failures and preventing future recalls, the FDA classifies recalls

into three classes based on the relative degree of health hazards a medical device presents, i.e., Class I, Class II and Class III with Class I for the most severe hazards. In addition, the FDA has released an analysis about the distributions of these three medical device recall classes [5]. However, the analysis of the FDA does not describe the information about the root causes of failures in the recalls, let alone if the causes are due to unspecified assumptions.

In this Chapter, we focus on analyzing the unspecified assumptions that causes software failures in the medical device recalls. In Section 2.3, we present a process that filters out software-related medical device recalls from the FDA database. Collecting all software-related medical device recalls is a joint effort that needs the support and contributions from a large research, industrial, and medical communities, To facilitate such effort, we have developed a web-based platform for different users to contribute and share new software-related medical device recalls. In Section 2.4, we analyze one hundred software-related recalls that have been collected from the FDA database. Our analysis reveals that there are four major categories of software failures in medical device recalls and unspecified assumptions made by medical device manufacturers are among one of the leading causes in medical device recalls. In Section 2.5, we conduct analysis of different types of assumptions in software-related recalls. The analysis on how assumptions contributed to the four major categories of software failures leads to an important finding that unspecified assumptions are often introduced into system design models by syntactical carriers, such as *constant variables/values*, *frequently read/write variables*, and *frequently executed actions*. This finding provides us the key to uncover the unspecified assumptions in the system development processes.

2.2 Introduction of the FDA Medical Device Recall Database

The FDA regulates medical devices sold in the U.S. by requiring manufacturers to follow a set of pre and post market regulatory controls. The FDA has classified

and described over 1,700 distinct types of devices and organized them with the Code of Federal Regulations into 16 medical specialties "panels" such as Cardiovascular devices or Ear, Nose, and Throat devices [18]. After a medical device is distributed in the market, the FDA monitors reports of adverse events and other problems with the device and, when necessary, alerts health professionals and the public to ensure proper use of the device and safety of patients.

The FDA Medical Device Recall database contains medical device recalls since November 1, 2002 [5]. A recall is a voluntary action that a manufacturer, distributor, or other responsible party takes to correct or remove from the market any medical device that violates the laws administered by the FDA. Recalls are initiated to protect the public health and well-being from devices that are defective or that present health risks such as disease, injury, or death. In rare cases, if a company fails to voluntarily recall a device that presents a health risk, the FDA can issue a recall order to the manufacturer.

The FDA classifies recalls into three classes based on the relative severity of health hazard a device presents. Class I recalls indicate that there is a reasonable chance that the use of the device will cause serious adverse health problems or death. Class II indicates devices that might cause temporary or medically reversible adverse health consequences or pose a remote chance of serious health problems. Class III indicates that devices violate the laws administered by the FDA but are not likely to cause adverse health consequences.

In the FDA database, a medical device recall entry, as shown Fig. 2.1, has the following information: 1) recall title, 2) recall class type, 3) recall posted date, 4) recall number, 5) device name, 6) recalling firm, 7) reasons of the recall, 8) action of the recall, 9) instructions for recovery, and 10) device distribution. The *reasons of the recall* contains human-written, unstructured text explaining the main causes of

the recall.

Date Initiated by Firm	February 20, 2019
Create Date	May 06, 2019
Recall Status[1]	Open[3], Classified
Recall Number	Z-1288-2019
Recall Event ID	82311
510(K)Number	K081543
Product Classification	Analyzer, chemistry (photometric, discrete), for clinical use - **Product Code** JJE
Product	VITROS¿ 5600 Integrated System Refurbished-Software V3.3.2 & below Product Code: 6802915 For use in the in vitro quantitative, semi-quantitative, and qualitative measurement of a variety of analytes of clinical interest, using VITROS Immunodiagnostic Products Reagents
Code Information	Serial Numbers: Affects systems that had the Luminometer replaced during a service-repair UDI: 10758750007110
Recalling Firm/ Manufacturer	Ortho-Clinical Diagnostics 100 Indigo Creek Dr Rochester NY 14626-5101
Manufacturer Reason for Recall	Luminometer Malfunction May Cause Inability to Process MicroWell Assays on VITROS¿ Systems
FDA Determined Cause [2]	Software design

Figure 2.1. An Example of FDA Medical Device Recall

2.3 Mining Software-Related Medical Device Recalls

To identify the recalls that are possibly due to software issues from the FDA database, we first need to identify whether the reasons for a recall contains semantic-related software keywords, then manually review the reason and communicate with its recalling firms to confirm whether it is a software-related recall. Fig. 2.2 shows the workflow that how we mine software-related medical device recalls from the FDA medical device recall database.

At the beginning, we extract all the medical device recalls reported to the FDA between 1 January 2014 and 31 December 2016, and store the result as **Recall Record V1**. For each recall in **Recall Record V1**, we use a Part-Of-Speech Tagger (POS Tagger) [19] to marking up each words in the reason field of the recall as nouns, verbs, adjectives, adverbs, etc. For example, the reason of Simens's Picture Archiving and Communication System recall [20] is "RGB images will show?,
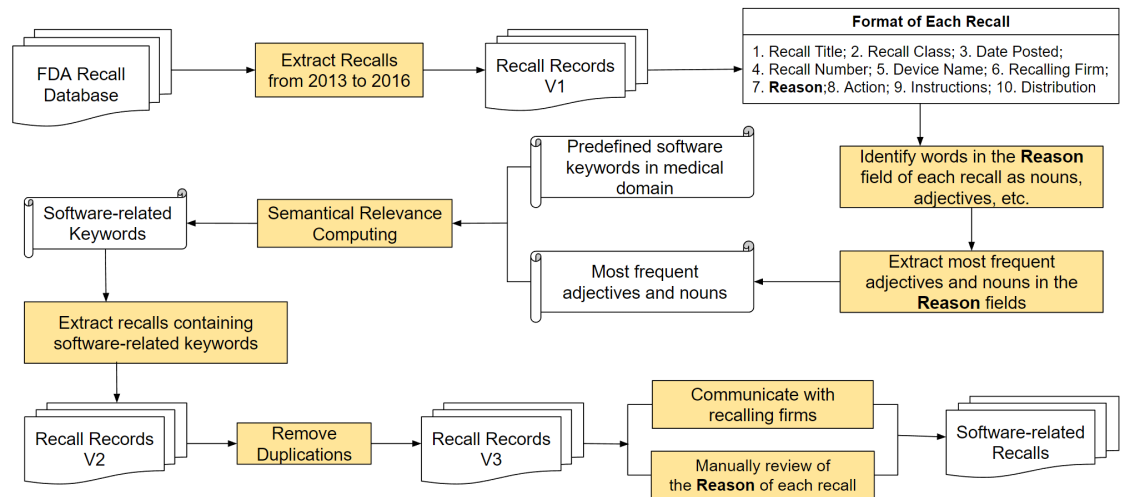
Figure 2.2. Flow for Mining Software-Related Medical Device Recalls

since the calculation of HU is not possible". After tagging process, the sentence is presented as "RGB/**NN** images/**NN** will/**RB** show/**VB** ?/**.** ,/**,** since/**CC** the/**DT** calculation/**NN** of/**IN** HU/**NN** is/**VB** not/**RB** possible/**JJ**", where the meanings of tags are shown in Table 2.1.

Table 2.1. Notation of Tags

Tag	Description	Example
CC	Coordin. Conjunction	since
DT	Determiner	the
IN	Preposition	of
JJ	Adjective	possible
NN	Noun	images
RB	Adverb	not
VB	Verb	show
.	Sentence-final punc	(. ? !)
,	Comma	,

Based on the tagging results, we calculate the number of times a word occurs in the recalls, which is called as *term frequency*. However, in the previous example, the term "the" is so common that term frequency will tend to incorrectly emphasize

recalls which happen to use the word "the", and fail to give enough weight to the more meaningful term "calculation". In other words, the term "the" is not a good keyword to distinguish relevant or non-relevant software-related recalls, while the less common word "calculation" does.

The term frequency-inverse document frequency (TF_IDF) [21] is one of the most popular term-weighting schemes intending to reflect how important a word is in a document. Hence, rather than using simple *term frequency* to identify important terms, we apply the TF_IDF method to extract the most relevant nouns and adjectives from the recalls, and reduce the weight of terms (such as "the") that occurs frequently.

To determine which noun or adjective is semantically related to software in the medical domain, we need to define a set of software-related keywords in the medical domain as a comparison objective. For mining and analyzing software-related medical device recalls, we have developed a web-based application [22] that allows users to upload software-related medical device recalls, and provide the ability that allows users to use a few keywords to tag the recalls. By extracting the tagging words from our website, we have collected a set of software-keywords, as shown in Table 2.2.

Table 2.2. Pre-defined Software-related Keywords

Pre-defined Software-related Keywords
system, software, application, database, program, integration, display, function, code, bug, error, fail, verification, validation, self-test, reboot, web, robotic, calculation, document, performance, workstation, expected, senor, alarm, message, screen, signal, interface, monitor, button, key, network, terminal, model, mode, communication, interaction, battery, power, supply, outlet, plug, power-up, discharge, charger, pause, terminate, dosage, environment

We can then calculate the semantic relevance of words from most relevant

nouns and adjectives by comparing with the pre-defined software-related keywords. Algorithm 1 computes the semantic relevance and retrieves software-related nouns and adjectives.

Algorithm 1 S-RKEYWORDS(W, W_{pre})

Input: A set of words W, and the pre-defined set of software-related keywords W_{pre}.
Output: The set of software-related words W_{out}.
1: Use word2vec model [23] to map each word $w_i \in W$ and $w_j \in W_{pre}$ to vector representation as $vec(w_i)$ and $vec(w_j)$
2: **for** each work w_j in W_{pre} **do**
3:    **if** there exists a word $w_i \in W$, $\cos(vec(w_j), vec(w_i)) \geq$ `threshold` **then**
4:       Put w_i to W_{out}
5:    **end if**
6: **end for**
7: **return** W_{out}

Algorithm 1 generates a set of most frequently used software-related words. We use keywords matching to extract the medical device recalls whose reason fields contain software-related words identified by the algorithm. The extracted records are denoted as **Recall Record V2**. As many of the recall records may have the same reasons because the same components or parts are used in different devices or models manufactured by the same company, we use recall title and recalling firm as a basis to remove duplicated entries and get another set of medical device recalls as **Recall Record V3**.

By manually reviewing each recall in **Recall Record V3**, we exclude the records whose reason for the recalls do not indicate they are software-related recalls. In addition, we also communicate the recalling firms about unclear reasons of the recalls and request more details about the recalls, such as requesting detailed recall letters. At the end, we finalize a list of software-related recalls as **Software-related Recalls**, which can be accessed through `http://gauss.cs.iit.edu/~code/recalls.html`.

To utilize **Software-related Recalls** and provide a platform for users freely participating in adding new recalls, tagging reasons of the recalls, querying and analyzing the recalls, we have developed a social networking service based web application `http://gauss.cs.iit.edu/~code/recalls.html`. The application front-end interface is organized around three main interconnected visualization panels: 1) the recalls display, 2) the modal panel for uploading new recalls, and 3) the querying panel. All three panels are interactive and allow users to navigate intuitively among them.

The recall display panel, shown in Fig. 2.3 allows the user to see at a glance the latest recalls posted by others. Each recall contains: 1) recall title, 2) recalling
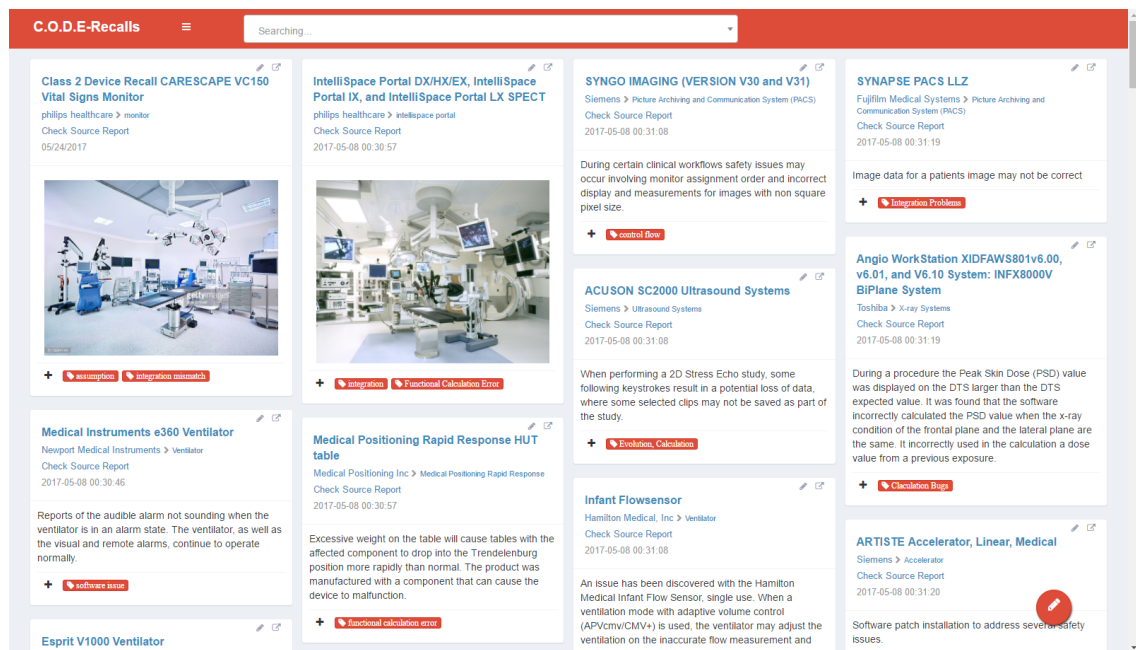
Figure 2.3. Home Page Display of the Application

firm, 3) device type, 4) original URL, 5) date posted and 6) keywords to represent the recall. All these information is required to post a new recall to the application as shown in Fig. 2.4.

As shown in Fig. 2.3, we use red icons to represent the tagging keywords of
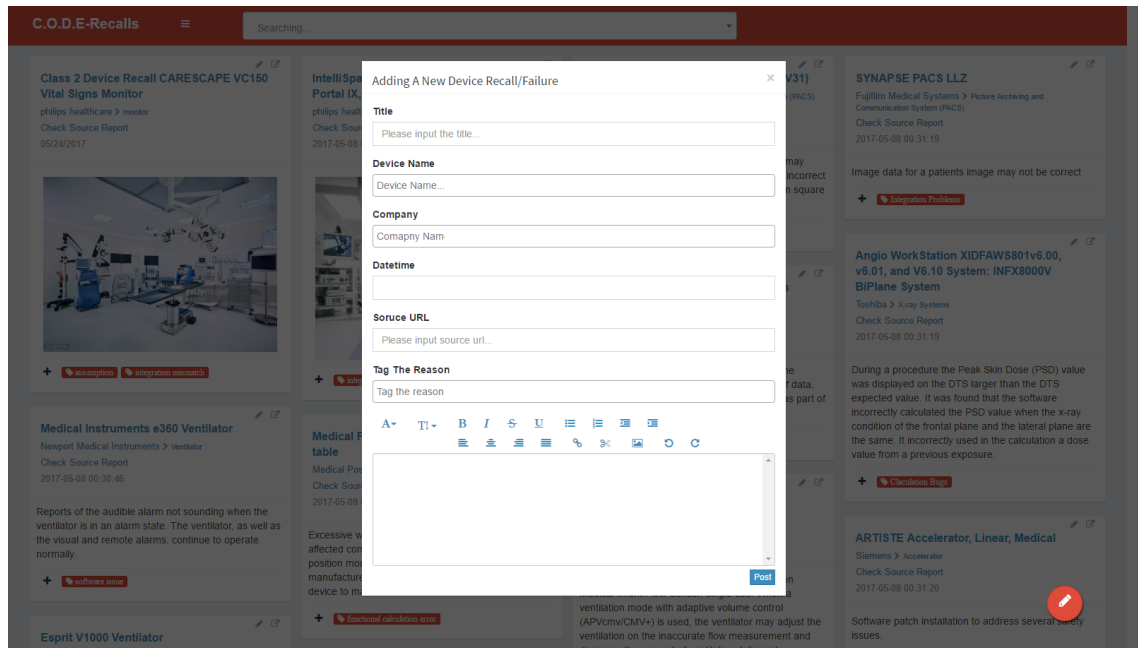
Figure 2.4. Uploading New Recall

each recall. The modification of the tagged words can be processed through the red plus icon. In addition, the application allows users to filter recalls by tagging words, as well as providing different querying methods, such as querying by tagging words, device names, reasons of recalls or recalling firm names, as shown in Fig. 2.5. Through this free online platform, users not only can capture, share and analyze the medical device recalls, but also can find interesting motivating examples to drive their research directions.

2.4 Analyzing Software-Related Medical Device Recalls

We use 100 identified software-related recalls as the basis for deriving statistics on fault categories of software-related failures. Before analyzing the recalls based on software-related fault categories, we illustrate the distribution of the recalls across different risk classes defined by the FDA. As shown in Fig. 2.6, 82% of software-related recalls are classified as class II with a medium risk of health consequences. However, the FDA's risk level itself can not reflect the root causes of software-related
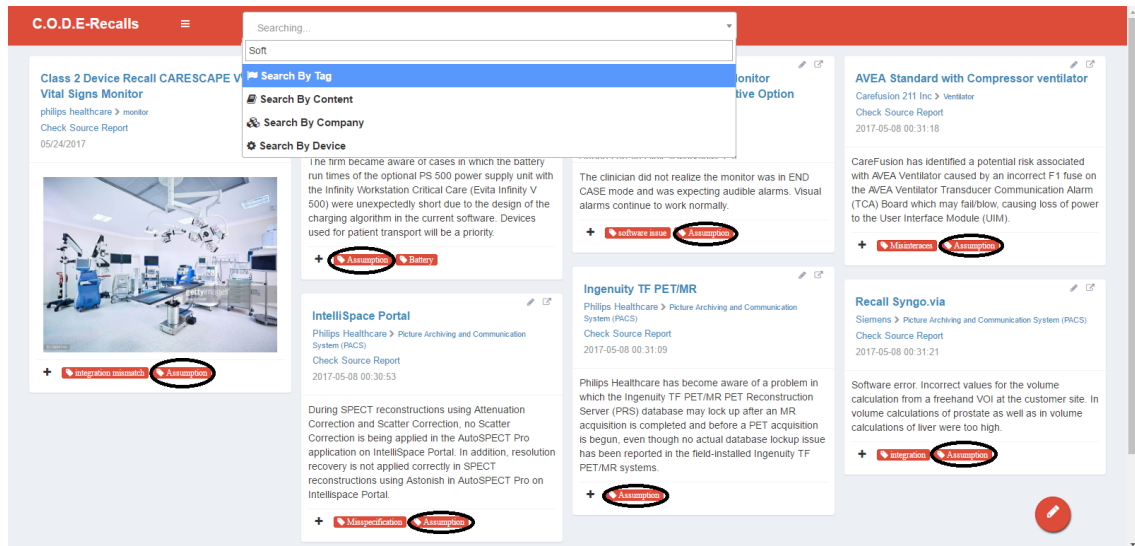
Figure 2.5. Query Interfaces and Results

medical device recalls. To give more detailed information about the failures that might impact the safe functioning of a software-based medical device, we further group the failures under four categories:**(1) Control Flow Fault; (2) Calculation Fault; (3) System Integration Fault; (4) Human-Machine Interaction Fault**. Fig. 2.7 illustrates the distribution of recalls across different fault categories. The majority (93%) of software-related recalls were classified into these four fault categories, while the rest of the recalls, whose descriptions indicate they are software-related failures but do not clearly belong to any of the four categories, are classified as *Other* category.

2.4.1 Control Flow Fault. Medical treatment scenarios are often complicated resulting in complicated control flow in medical systems, which increase the probability of control flow fault. For example, the medical treatment guideline [24] for stroke contains more than 30 main steps, and the complicated execution orders of the steps make control flow more error-prone. We group the failures of control flow into five fields: 1) inconsistent logic from requirements; 2) exception handling fault; 3) block or unblock interrupts fault; 4) execution orders fault; and 5) conditional statement
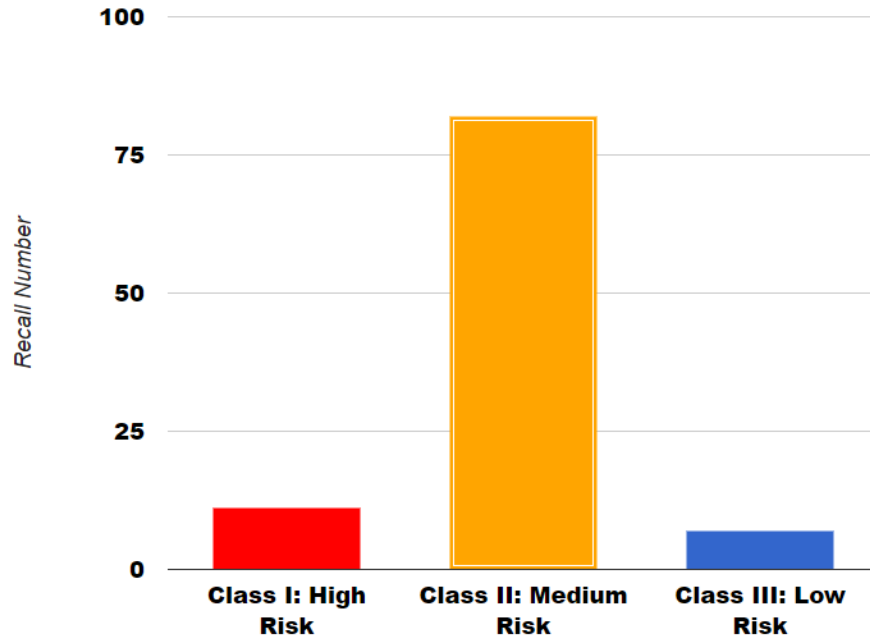
Figure 2.6. Distribution of software-related recalls in the FDA's risk levels

fault. Fig. 2.8 illustrates the distribution of recalls across different fault categories. The following shows examples of these control flow faults in each field.

- **Inconsistent logic from requirements example. Device Name:** Ventilator, **Date Posted:** 05/09/2016, **Recall Class:** 2, **Recalling Firm:** Hamilton Medical, Inc, **Recall URL:** `http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=145040`, and **Recall Reason:** After performing the suctioning maneuver, including disconnecting the patient, suctioning , and reconnecting the patient, the preset pattern of ventilation many not continue as expected.

- **Exception handing fault example. Device Name:** Picture Archiving and Communication System, **Date Posted:** 07/08/2014, **Recall Class:** 2, **Recalling Firm:** Philips Healthcare, **Recall URL:** `http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=128568`, and **Recall Reason:** Faulty Automatic Motion Controller (AMC), a problem
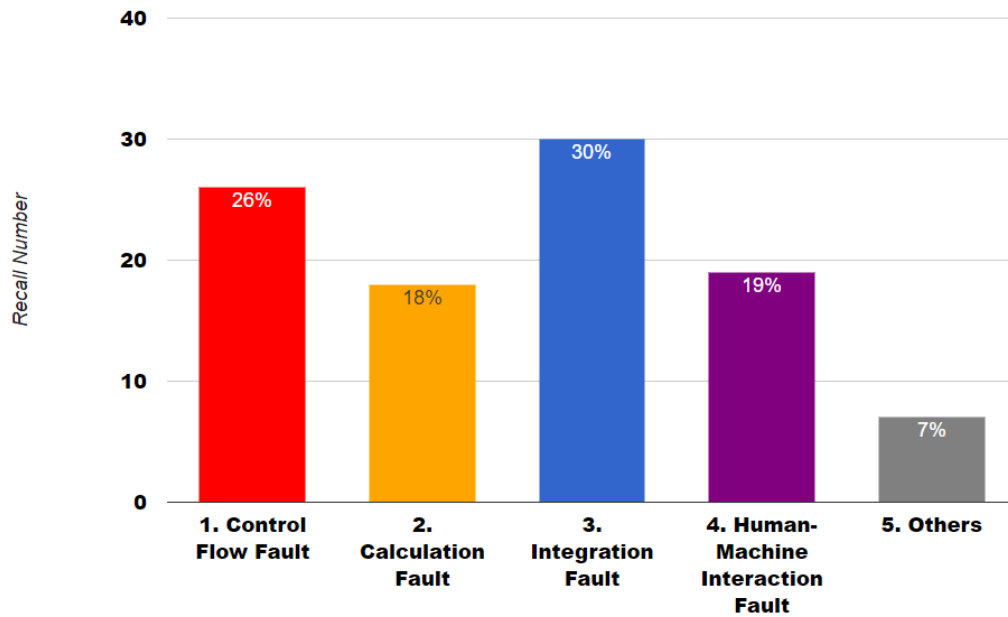
Figure 2.7. Distribution of 100 Recalls across Fault Categories

in the Power On Self Test (POST) error handling was detected, can result in a hazardous movement of the C-arc. system.

- **Block or unblock interrupts example. Device Name:** Picture Archiving and Communication System, **Date Posted:** 07/08/2014, **Recall Class:** 2, **Recalling Firm:** Siemens, **Recall URL:** `http://www.accessdata.f da.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=133015`, and **Recall Reason:** In case of a system crash, images may not be written to the hard disk and this may result in inconsistencies in the database. In case of a system crash (e.g. blue screen, power outage) images may not be written from cache to the hard disk and might get lost.

- **Execution orders fault example. Device Name:** Picture Archiving and Communication System, **Date Posted:** 03/30/2015, **Recall Class:** 2, **Recalling Firm:** Siemens, **Recall URL:** `http://www.accessdata.fda .gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=134102`, and **Re-**
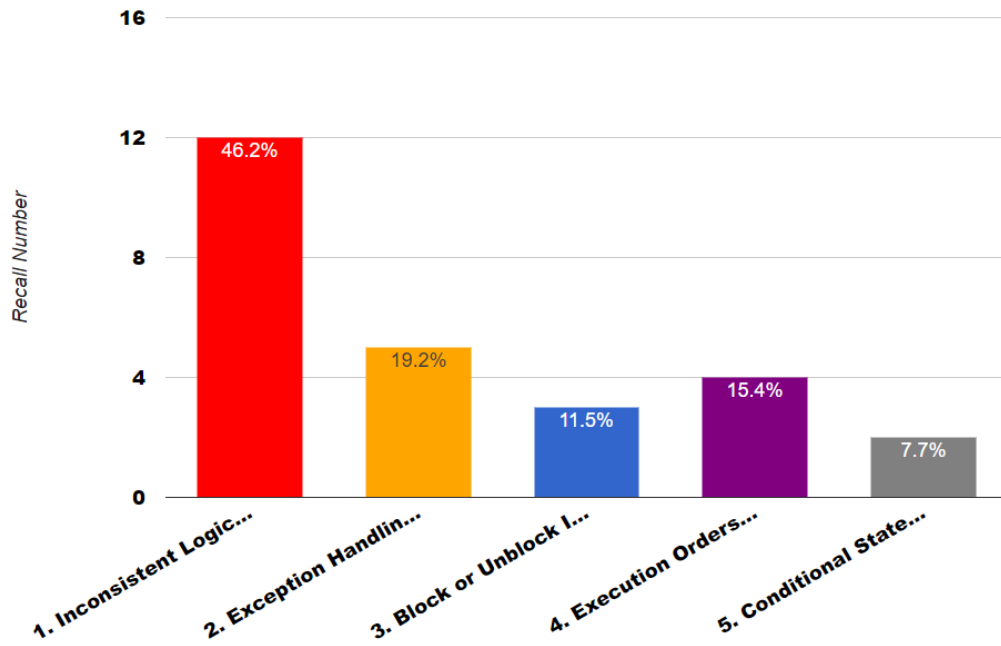
Figure 2.8. Distribution of 26 Recalls across Control Flow Fault

call Reason: Possibly incomplete archived studies during pre-fetch. In a server farm setup, when pre-fetch/retrieve operation is performed for partially archived studies, the series that have not yet been archived, will remain unarchived.

- **Conditional statement fault example. Device Name:** Radiation Therapy System, **Date Posted:** 06/21/2014, **Recall Class:** 2, **Recalling Firm:** VARIAN MEDICAL SYSTEMS PARTICLE THERAPY GMBH, **Recall URL:** http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRes /res.cfm?ID=127498, and **Recall Reason:**Anomaly with the ProBeam System where under certain conditions, the Treatment Control and Monitoring application could fail to send treatment history records to the ARIA database.

2.4.2 Calculation Fault. A medical treatment scenario often involves many medications, which are prescribed with different precision of dosages, different units of medicines, different data types of dosages. These facts increase the probability of causing calculation faults. We group calculation failures into four fields: 1) incorrect

arithmetic/formula; 2) incorrect/outdated constants; 3) incorrect conversion; and 4) incorrect approximation/precision. Fig. 2.9 illustrates the distribution of recalls across different calculation fields. The following indicates these calculation faults, along with a related recall in each field.

Figure 2.9. Distribution of 18 Recalls across Calculation Fault

- **Incorrect arithmetic/formula example: Device Name:** Ventilator, **Date Posted:** 05/09/2014, **Recall Class:** 2, **Recalling Firm.** Spacelabs Healthcare, **Recall URL:**`http://www.accessdata.fda.gov/scripts/cdr h/cfdocs/cfRes/res.cfm?ID=127487`, and **Recall Reason:** Spacelabs Healthcare is voluntarily recalling the Hamilton Galileo Ventilator Flexport, Model 90436A-07, where the monitored Minute Volumes (Vmin) has been reported at one time to reach ten times the actual value on the bedside monitor.

- **Incorrect/outdated constants example. Device Name:** Neurological Stereo Instrument, **Date Posted:** 12/23/2015, **Recall Class:** 2, **Recalling**

Firm: Synaptive Medical, Inc, **Recall URL:** `http://www.accessdata` `.fda.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=142094,` and **Recall Reason:** Out of tolerance for radio frequency emissions. At the 150-1000MHz frequency, the testing indicated the BrightMatter Navigation system was up to 20dB uV/meter higher than the applicable IEC 60601-1-2:2007 (Ed3.0) standard specification.

- **Incorrect conversion example. Device Name:** Ultrasound Systems, **Date Posted:** 09/17/2015, **Recall Class:** 2, **Recalling Firm:** Siemens, **Recall URL:** `http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/c` `fRes/res.cfm?ID=127515,` and **Recall Reason:** The ACUSON SC2000 ultrasound system considers uppercase/lowercase differences in the same patient name as unique patient instances when registered on the same ultrasound system. If these differences are not corrected at the time of registration, the system does not capture images or clips.

- **Incorrect approximation/precision example. Device Name:** Picture Archiving and Communication System, **Date Posted:** 02/23/2016, **Recall Class:** 2, **Recalling Firm:** Siemens, **Recall URL:** `http://www.acce` `ssdata.fda.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=14` `3694,` and **Recall Reason:** Siemens is releasing an updated software version to address several software issues including RGB images will show "?" since calculation of HU is not possible; save as option enabled; changes in access for loading studies; breast region is now properly fitted to segment boundary when clicking fit breast to screen.

2.4.3 Integration Fault. A medical treatment scenario often involves many different medical devices working together, which increases the complexity of system integration, and increases the probability of integration failures. We group the fail-

ures of integration into four fields: 1) mismatch of reused component; 2) mismatch of component interfaces; 3) inconsistent system evolution; and 4) mismatch of components configurations. Fig. 2.10 illustrates the distribution of recalls across different integration fields and the followings represent these integration faults, along with a related recall in each field.
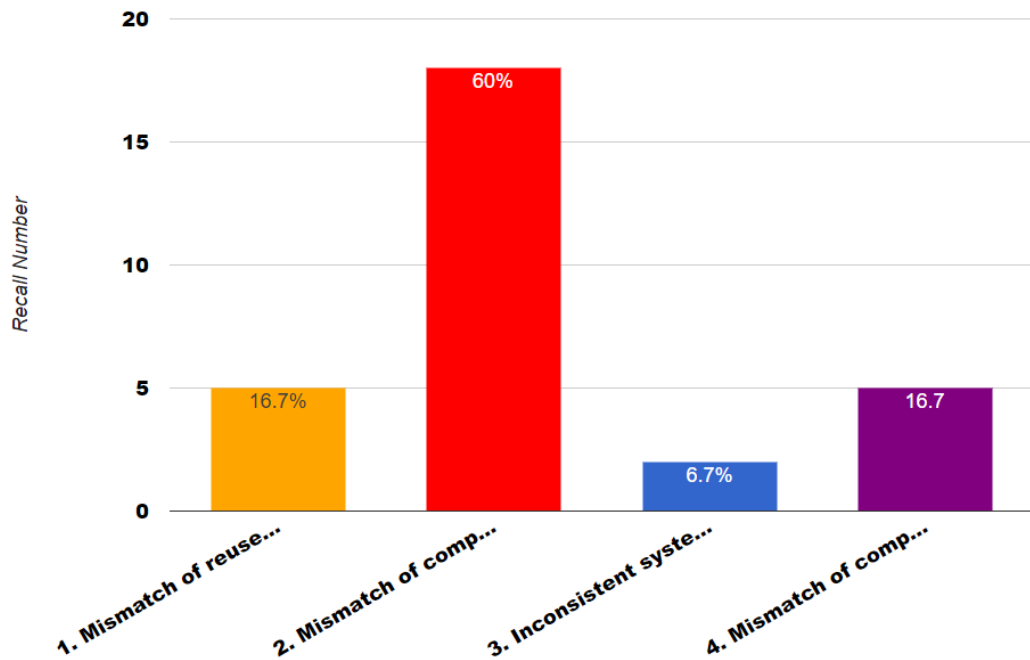
Figure 2.10. Distribution of 30 Recalls across Integration Fault

- **Mismatch of reused components example. Device Name:** Picture Archiving and Communication System, **Date Posted:** 03/17/2016, **Recall Class:** 2, **Recalling Firm:** Siemens, **Recall URL:** `http://www.accessdata.f da.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=144270`, and **Recall Reason:** Siemens' conducting a recall due to a potential issue when using the measurement package of the VA10 version of syngo Dynamics.

- **Mismatch of components interfaces example. Device Name:** Picture Archiving and Communication System, **Date Posted:** 07/02/2015, **Recall**

**Class:** 2, **Recalling Firm:** Synaptive Medical, Inc, **Recall URL:** `http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=138013`, and **Recall Reason:** When the E-NMT-01 module is used in conjunction with the ElectroSensor, the Neuromuscular Transmission (NMT) values may indicate a deeper level of muscle relaxation than the actual level of muscle relaxation. In the clinical situation visual movements of the hand are seen after TOF (Train of Four) stimulation, but the patient monitor shows no counts, or counts are not corresponding to the actual value.

- **Inconsistent system evolution example. Device Name:** Intranasal Splint, **Date Posted:** 08/07/2014, **Recall Class:** 2, **Recalling Firm:** Enhancement Medical, **Recall URL:** `http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=128288`, and **Recall Reason:** Manufacturer made a change in the production process that resulted in a change in final gel weight. RECALL EXPANDED 7/8/2014 Firm expanded their recall to include all lots of product.

- **Mismatch of components configurations example. Device Name:** Monitor, **Date Posted:** 08/05/2014, **Recall Class:** 2, **Recalling Firm:** Curbell Medical, Inc, **Recall URL:** `http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=129118`, and **Recall Reason:** The firm became aware of a potential problem that was initiated by a customer complaint. After consultation with the manufacturer, it was discovered that a resistor was incorrectly placed within the circuit board on the monitor. This change to the resistor was a planned change to address a product improvement (improve battery drain).

2.4.4 Human-Machine Interaction Fault. For medical systems, human-machine interactions are often performed. Some unexpected interaction patterns can cause in-

tegration faults. We group the failures of human-machine interaction into three fields: 1) missing/wrong functions of human-machine interactions; 2) missing/misleading/-confusing/error information; and 3) inappropriate use of keyboard/button. Fig. 2.11 illustrates the distribution of recalls across different integration fields and the followings show these interaction faults, along with a related recall in each field.
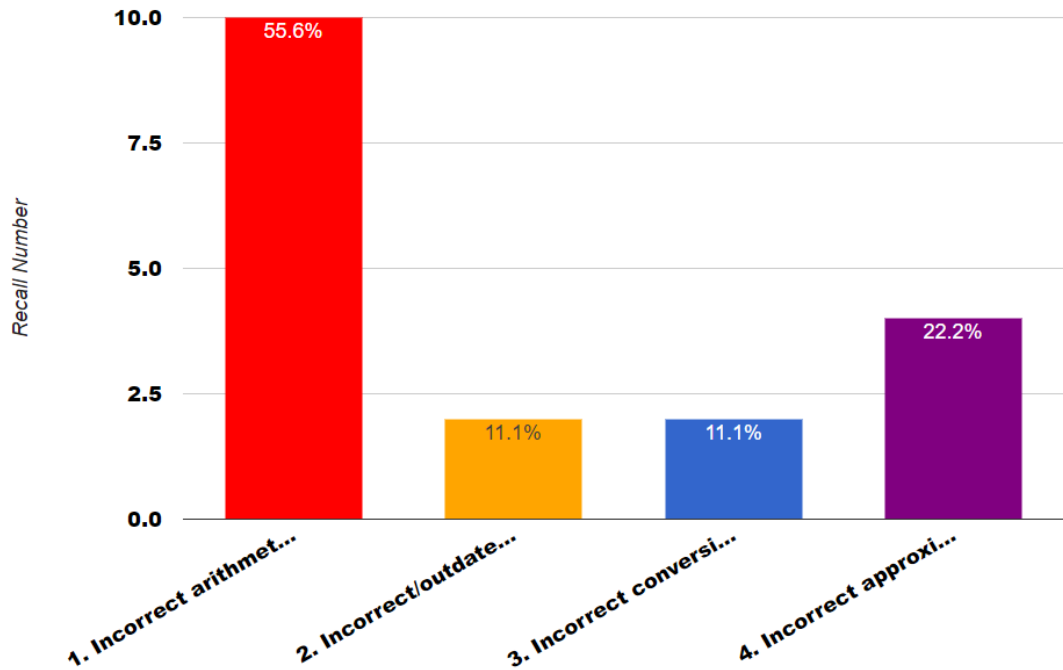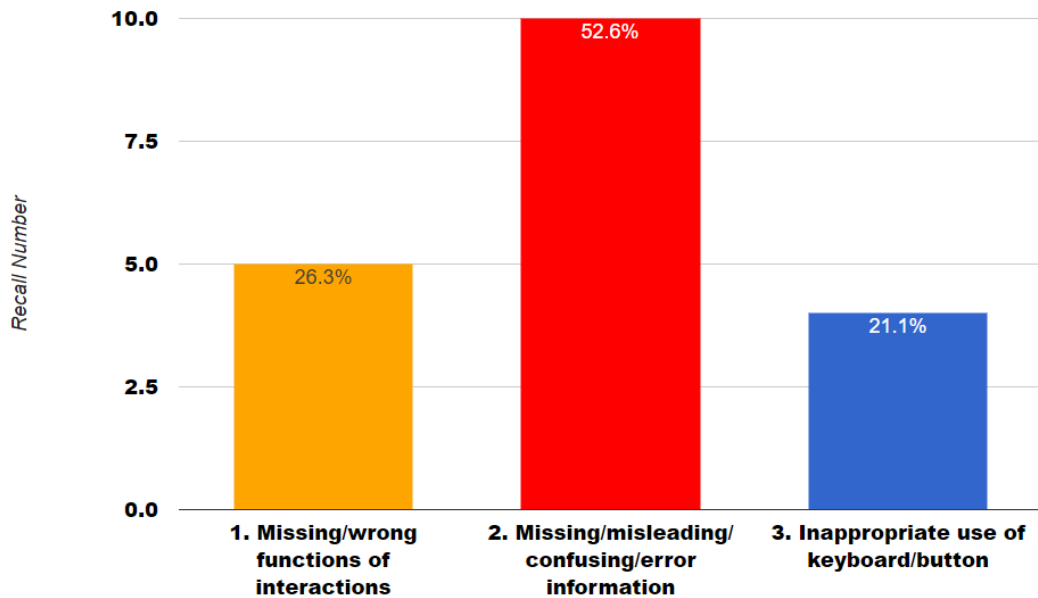

Figure 2.11. Distribution of 19 Recalls across Human-Machine Integration Fault

- **Missing/wrong functions of human-machine interactions. Device Name:** MAMMOMAT Inspiration, **Date Posted:** 04/25/2014, **Recall Class:** 2, **Recalling Firm:** Siemens, **Recall URL:** `http://www.accessdata.fda.g ov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=127169`, and **Recall Reason:** There is a potential and possible hazard to the user when using the MAMMOMAT Inspiration PC monitor at the control desk, in that the holder of the PC monitor can break causing an unstable monitor to fall causing possible serious injury.

- **Missing/misleading/confusing/error information. Device Name:** Ven-

tilator, **Date Posted:** 03/17/2016, **Recall Class:** 2, **Recalling Firm:** Covidien, **Recall URL:** `http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=143047`, and **Recall Reason:** In the case of a loss of GUI display due to a Backlight Inverter PCBA failure, the ventilator continues to provide uninterrupted ventilatory support at the programmed settings for the patient. However, there is a loss of display and thus there is a necessity to move the patient to another ventilator.

- **Inappropriate use of keyboard/button. Device Name:** Ventilator, **Date Posted:** 07/16/2015, **Recall Class:** 1, **Recalling Firm:** Breas Medical, **Recall URL:** `http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=131484`, and **Recall Reason:** Unintended treatment termination could result from a keypad malfunction in some situations. The device erroneously interprets this as a Stop Treatment Instruction. An alarm will not sound, or be registered. Accessories and monitoring equipment connected to the Vivo 50 will stop functioning as the device enters a stand-by mode.

Fig. 2.12 depicts the types of medical devices accounted for the different percentage of device recall events. The proportions of medical device types in the recalls is helpful for us to address the issues and challenges that may impact device quality, safety, and effectiveness from industry-wide perspective.

Through manually review the software-related recalls and their fault categories, we observe some major challenges related to medical devices safety have not been addressed in existing works [25, 26, 27]. The following are our insights on some of the future challenges in safety-critical cyber-physical system design:

- Develop model-driven design procedures that consider unspecified assumptions,
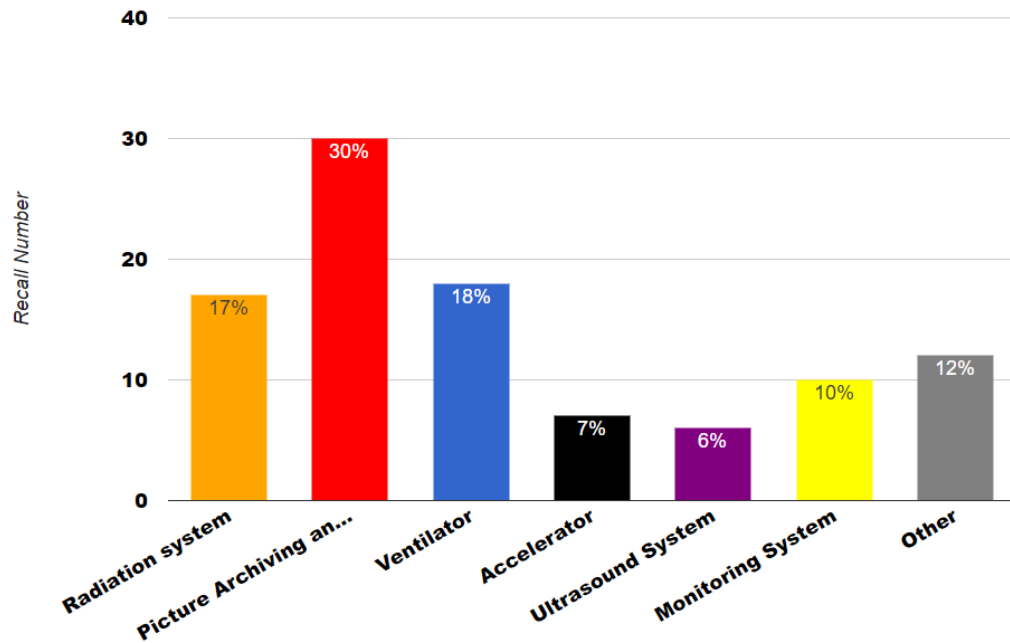
Figure 2.12. Distribution of 100 Recalls across Device types

flawed requirements, complex software errors, accidents due to incorrect functional interactions among components.

- Integrating patient/doctor/nurse modeling, medical equipment's modeling and simulation into M-CPS design.

- Propose safe and efficient strategies that applying advanced techniques such as model checking, comprehensive validation of the system, and run-time monitoring in M-CPS design.

2.5 Analyzing Unspecified Assumptions in Software-Related Medical Device Recalls

Among the 100 software-related medical device recalls, there are 46 recalls that are caused by violation of unspecified assumptions. In particular, 8, 10, and 18 recalls that are related to unspecified assumptions about constant variables, unspecified assumptions about correlated variables, and unspecified assumptions about

execution pattern, respectively.

2.5.1 Unspecified Assumptions about Constant Variables. As we know, system behaviors are often restricted by environment conditions. For instance, the discharge rate of a ventilator's battery can be changed when the temperature decreases. During the system development process, environment conditions are often represented by constant variables, and the declaration and initialization of constant variables are often associated with the assumptions made for the system's operating environment. However, the assumptions about constant variable declarations often exist in the domain experts' mind, which cause the assumptions to be unspecified. And system failures can be caused by violating these unspecified assumptions.

Taking the ventilator recall [28] as an example. The battery in the ventilator did not last as long as expected. The battery installed in the ventilator depleted much earlier than expected although the battery indicator showed a sufficiently charged battery. Even when the battery was totally depleted, the power fail alarm was not generated. If the ventilator shuts down without alarm, a patient may not receive necessary oxygen. The root cause of this failure is that ventilators are assumed to be installed in temperature controlled areas, in which the battery discharge rate is a constant variable. However if the temperature is not in the assumed range, the fixed value of discharge rate becomes invalid and cause the ventilator system to miscalculate the remaining time and fail to send an alarm event on time.

2.5.2 Unspecified Assumptions about Correlated Variables. In the system development process, some variables are read/updated together consistently. We call these variables correlated variables. For example, when monitoring patient's activities who is using medical ventilators, heart rate, oxygen level, blood flow and respiratory rate related to the patient must be obtained and updated together [29]. However, the assumptions of these correlated variables that need to be read and updated together

consistently often are unspecified in the system. But it is necessary to record these data and verify whether they are valid and consistent.

2.5.3 Unspecified Assumptions about Execution Actions. In the medical devices recall database, some recalls are due to that medical professionals were not aware of the sequences of operations for performing a task. A medical device is expected to execute human tasks followed by sequence as specified. However the assumptions of such executed actions are often not documented. For instance, with a medical ventilator, users should set the mode of mechanical ventilation before setting the value of tidal volume.

In medical domain, performing a procedure involve different devices. The assumptions of the sequences of tasks should be explicitly specified. For a stroke patient in ICU room, in order to get vital information of the patient from the monitoring system, multiple physicians need to perform different interactions with various devices such as infusion pump and ventilator. Assumptions of the interactions are required to be explicitly specified to navigate users' interactions correctly.

The analysis of unspecified assumptions in the medical device failures reveals that unspecified assumptions are often introduced into systems by *constant variables/values*, *frequently read/updated variables*, and *frequently executed actions*.

2.6 Summary

In this Chapter, we presented a procedure to collect software-related medical device recalls from the FDA database and developed a web-based platform that enables users to add new and share about software-related medical device recalls. In addition, we classified major categories of software failures most frequently occurred in medical domain and conducted an analysis on these recalls to determine the leading causes of these recalls. The analysis reveals that unspecified assumptions is one

of the root causes of medical device recall. We analyzed how unspecified assumptions may lead to the medical device recalls. The analysis inferred to a key finding that unspecified assumptions are often introduced into systems by 1) *constant variables/- values*, 2)*frequently read/updated variables*, and 3) *frequently executed actions*. Next Chapter, we will present an approach to identify unspecified assumptions at an early stage of system development process based on the key finding.

CHAPTER 3

IDENTIFY UNSPECIFIED ASSUMPTIONS IN SYSTEM DESIGN MODELS

3.1 Background and Related Work

The analysis in the Chapter 2 indicates that unspecified assumptions often exist through *syntactic carriers*, such as *constant variables*, *frequently read/updated variables*, and *frequently executed actions*. We use the following FDA medical device recall example to illustrate how unspecified assumptions are injected to systems through *syntactic carriers* and have caused an M-CPS failure.

Recall Case 1 (HAMILTON-T1 ventilators recall with software versions 1.1.2 and lower. 01/23/2013 [6])**.** The FDA has identified this recall as a Class I recall–the most serious type of recall issued by the FDA. According to the recall report [6], there is unexpected high internal oxygen consumption of HAMILTON-T1 ventilators during the ventilation of small pediatric patients. The current labeling does not include sufficient information about the internal oxygen consumption of the ventilators. Depending on the ventilator setting and the patient's lung impedance, the internal gas consumption of the HAMILTON-T1 may be higher than expected. The internal oxygen consumption has to be taken into account in addition to the oxygen requirements for the patient's ventilation, especially when oxygen is limited. If the available oxygen supply d uring transport is depleted, the life of the patient may be endangered.

In this recall, the internal oxygen consumption is calculated by $(MinVol + FlowTrigger) \times C \times (FiO_2 - 20.9)/79.1$, where constant $C = 1.5$ is initially set for adult patients – an unspecified assumption that the formula of internal oxygen consumption is used for adult patients. For pediatric patients, the value of C should

be 4.0, which is different from the value used for adult patients. When $C = 1.5$ is using for children, it causes unexpected high internal oxygen consumption. This recall and many other examples in FDA recall database [17] indicate that unspecified assumptions are dangerous and can lead to catastrophes. Therefore, being able to identify such assumptions at an early stage of M-CPS development process is critical to ensure systems' safety.

Many researchers have pointed out the importance of assumption management in software development. For example, Corbato [30] mentioned in his ACM Turing Award lecture that "design bugs are often subtle and occur by evolution with early assumptions being forgotten as new features or uses are added to systems". Many problems in software development can be traced to unspecified assumptions [8]. Steingruebl and Peterson [31] argued that unspecified software assumptions could lead to software failures, and they proposed methods and techniques to manage software assumptions. In addition, Bazaz et al. [32] pointed out that the violation of unspecified assumptions about system resources might cause the system to be vulnerable and even fail. Unspecified assumptions often cause failures in safety-critical cyber-physical systems [33]. For instance, the Ariane 5 [3] and Child-seat Airbag Incident [34] were caused by unspecified assumptions made in the system development process. The unspecified assumption issue is also magnified in M-CPS development such as many medical devices recalls documented in FDA recall database [17].

Given the importance of assumptions in the system development process, many efforts have been made in explicitly specifying assumptions. Lehman and Ramil proposed a few guidelines to specify assumptions correctly [7]. The authors believed that it is necessary to train all stakeholders to identify and record assumptions at all stages of the system development with a standard form or structure. Lewis et al. [8] developed an assumption management framework for improving the quality of

software development. The framework can extract assumptions from source code and record them into a repository for management. Besides the code level assumption management, Tirumala [9] developed an assumption management framework (AMF) at system component levels. The goal of AMF is to have a well-defined vocabulary to encode assumptions in a machine-checkable format [9]. AMF introduces a systematic process that performs automatic validations for machine-checkable assumptions. This framework addresses the issues caused by unspecified assumptions at component interface levels. These tools and approaches mentioned above are to facilitate domain experts actively to specify assumptions during the system development cycle.

However, even with these tools and approaches, it is still unavoidable that there are unspecified assumptions made in systems, just as there are always some bugs left in the code even with good compilers. Similar to we need debugging tools to help to identify bugs at the code level, we also need tools and approaches to help uncover unspecified assumptions at different levels in the system development process. Based on the fact that programmers often use assertions and comments in their code to specify assumptions which make it possible to mine unspecified assumptions directly, Li and Zhou proposed an approach to efficiently extract unspecified programming rules from large software code by integrating data mining technologies and static analysis at the code level [35]. In addition, as large datasets of programming codes become available and as the machine learning advance, combining static analysis and machine learning to improve the correctness of hardware and software has become a new hot research topic. For example, based on the datasets provided by the National Institute of Standards and Technology (NIST): the NVD [36] and the Software Assurance Reference Dataset (SARD) project [37], a deep learning-based vulnerability detection system, called Vulnerability Deep Pecker (VulDeePecker), has been developed to detect vulnerabilities [38]. In addition, by using 38383 open-source C files as the dataset, Cong Wang, et al. have developed a weakness-oriented assertion

recommendation toolkit based on machine learning technologies, called Weak-Assert, uses well-designed patterns to match the abstract syntax trees of source code automatically, and inserts assertions into proper locations of programs for improving the quality of software testing and verification [39]. Another example is Daikon, which uses machine learning-based dynamic invariant detection to run a program, observe the values that the program computes, and then report properties that were true over the observed executions [40].

Although assumptions can be detected from source code, it can still cost huge human efforts to trace the assumptions back to an earlier stage of the development cycle, let alone the cost of fixing the failures caused by the unspecified assumptions. Therefore, it is highly desirable to have tools to identify such unspecified assumptions at system design level automatically. However, how to efficiently applying the approaches that are successful at the code level on the system design level for improving the quality of validation and verification has not been addressed well yet. One reason is lacking large open-source datasets of system design models. Besides, different from the programming code, the assertions and comments are rarely used in system design models. The differences make mining invariant or mis-specifications from assertions and comments at model levels be much more difficult. Based on these two reasons, existing solutions for unspecified assumptions detection at model level mainly rely on human experts. However, even for domain experts and developers, this is a tedious, subjective, and sometimes error-prone task because of the massive complexity.

After analyzing over one hundred software-related medical device recalls in the FDA database [41], we observed that unspecified assumptions are often introduced into the system design models through syntactical structures of modeling languages, such as *constant variables*, *frequently read/updated variables*, and *frequently executed*

actions, and we call them as *syntactic carriers*. For example, in the FDA Medical Device Recall 1, the unspecified assumption (the formula of internal oxygen consumption can be only used for adult patients) is injected into the system development through the initialization of the constant variable $C = 1.5$.

Based on the observation, in this Chapter, we present a new approach to identify unspecified assumptions through finding potential unspecified assumption *syntactic carriers* rather than unspecified assumptions themselves. Once the *syntactic carriers* are identified, domain experts and developers can validate unspecified assumptions associated with the carriers. And based on the approach, we present an unspecified assumption *syntactic carriers* finder called *UACFinder*, which uses data mining techniques to identify potential *syntactic carriers* of unspecified assumptions from system design models. The *UACFinder* currently focuses on mining three types of *syntactic carriers*: *constant variables*, *frequently read/updated variables*, and *frequently executed actions*. Whether unspecified assumptions exist in these carriers are yet to be validated by domain experts or model developers. The techniques used by the *UACFinder* can automatically extract these carriers without requiring any prior knowledge about annotations, templates, or weight assignments from end-users. We use a simplified cardiac-arrest statechart model as a case study to evaluate the *UACFinder* in mining potential *syntactic carriers* of unspecified assumptions. We also invite a medical doctor[1] to validate the possible unspecified assumptions about these carriers found by the *UACFinder*. The validation results indicate that the *UACFinder* is of practical use in identifying potential unspecified assumptions from system design models.

---

[1]Doctor Li Meng is an associate chief physician from the Department of Neurology in the First Hospital of Hebei Medical University, China.

3.2 Overview of the *UACFinder*

Developing the *UACFinder* is inspired by our extensive study of the FDA recall database [41], from which we have two critical findings: (1) unspecified assumptions can cause adversary events on patients, and (2) many unspecified assumptions are implemented through syntactic carriers provided by modeling languages. Hence, the high-level idea of the *UACFinder* is to automatically find *syntactic carriers* of potential unspecified assumptions from system design models. In this paper, we use statechart models as a design modeling language, and we focus on mining *syntactic carriers* from system statechart models. However, the fundamentals behind the *UACFinder* can be applied in other system modeling languages.

A statechart model is an extended form of the exemplary state diagram representing a finite-state machine (FSM) [42]. Statecharts have high similarities with many medical diseases and treatment models. In addition, statecharts have been successfully used to represent the behavior of different safety-critical systems including avionics [43], air traffic control systems [44]. In particular, it is worth highlighting that [45] has proven that statecharts can successfully capture the behavioral aspects of surgical care delivery. These distinguishing features of statecharts enable many groups to model medical cyber-physical systems with statecharts [46], [25], [47], [48], [49]. Therefore, mining *syntactic carriers* of potential unspecified assumptions from system statechart models serves as a good starting point.

A statechart model has the following five primary elements:

1. *Variable.* They are four types of variables: *integer*, *real*, *boolean*, and *string*.

2. *Event.* An event represents a point in time when something of importance

happens in the context of a state machine, such as a user pushes a button; a value is delivered, or a period of time is passed.

3. *Action.* There are four types of actions: assignment, condition statement, event raising, and function call.

4. *State.* A state represents the states of a state machine. A state can have a behavior, and the behavior specifies which actions are executed and on what conditions. Actions can be triggered by entering a state, leaving a state, an occurrence of an event such as conditions becoming true, or time passes.

5. *Transition.* a transition is the transfer of one state to another. A transition specifies boolean expressions for when this transition is active and actions that are to execute when the transition takes place.

In this paper, we focus on extract *syntactic carriers* in the form of *constant variables*, *frequently read/updated variables*, and *frequently executed actions* from system statechart model. We discuss each of the forms in the following subsections. The architecture of *UACFinder* is depicted in Fig. 3.1.
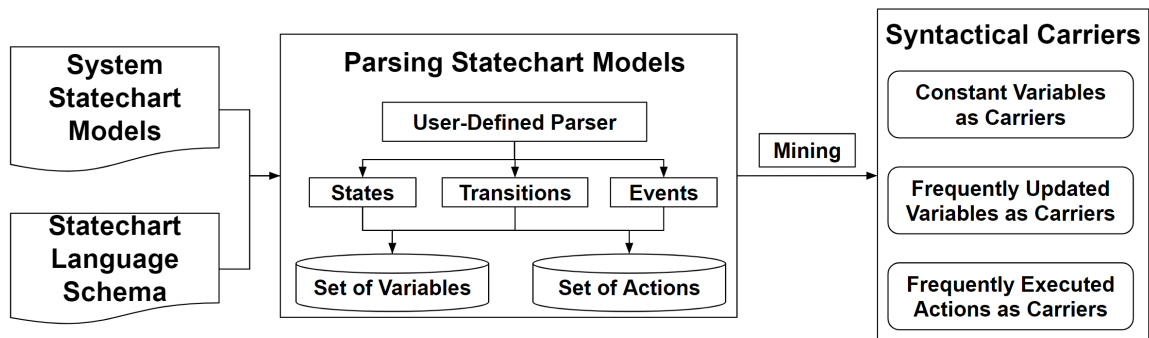
Figure 3.1. The Architecture of *UACFinder*

In particular, to efficiently mine out these *syntactic carriers*, the *UACFinder* converts the problem into a frequent itemset mining problem by first parsing the

statechart models to proposed data structures. Second, we treat a state and one of its incoming transitions as a combined state, and each action in the combined state is hashed into a number, forming an itemset (a set of numbers). In the end, the itemset is written as a row into the itemset database. As a result, all the actions in a statechart model are converted into a database that contains many itemsets of actions. By mining this database using a frequent itemset mining algorithm such as FPclose [50], we can find the frequent sub-itemsets that appear for many times. These frequent sub-itemsets can then be used to infer *syntactic carriers* as *frequently read/updated variables* and *frequently executed actions*. Before introducing the detailed mining procedures, we first explain how unspecified assumptions exist in the statechart models through the three types of *syntactic carriers*.

3.2.1 Constant Variables as Syntactic Carriers of Potential Unspecified Assumptions.

In system design models, a constant variable is a variable whose value never changes during the execution of the system. During the development of system design models, the declaration and initialization of a constant often associate with assumptions. However, the assumptions about constant variable declarations often exist in the developers' mind, which cause the assumptions to be unspecified.

Taking the renal insufficiency statechart model [51] as an example (Renal insufficiency represents the poor function of the kidneys that may be due to a reduction in blood flow to kidneys caused by renal artery disease [52]). In the renal insufficiency statechart model, the variable $Kidney.Potassium\_High\_Threshold$ is declared as a constant and represents the high threshold of potassium in patients' body, and the initial value of $Kidney.Potassium\_High\_Threshold$ is set to be 5.5 since the safety range for potassium levels is usually between 3.5 and 5.5 mEq/L [53] for adults. The consequence of the initialization leads an unspecified assumption to be introduced into the renal insufficiency statechart model: **the renal insufficiency model is**

only designed for adult patients since the normal range of potassium is **3.4-4.7 mEq/L for children** [**53**]. However, if the assumption is unspecified and the model is used to examine pediatric patients, it would output incorrect test results and may lead doctors to make wrong decisions, which can cause patients harm and even death. Therefore, we take constant variables as one of the *syntactic carriers* of unspecified assumptions. They need to be identified from the system design models to help uncover potential unspecified assumptions in system design models.

3.2.2 Frequently Read/Updated Variables as Syntactic Carriers of Potential Unspecified Assumptions. In the system design models, some variables are read/updated together consistently. For example, when monitoring a patient's activities who is using medical ventilators, two variables related to the patient's ventilator dependence must be read and updated together are oxygenation and CO_2 elimination [29]. We use a simple statechart model depicted in Fig. 3.2 to briefly illustrate how *frequently read/updated variables* exist a statechart model.
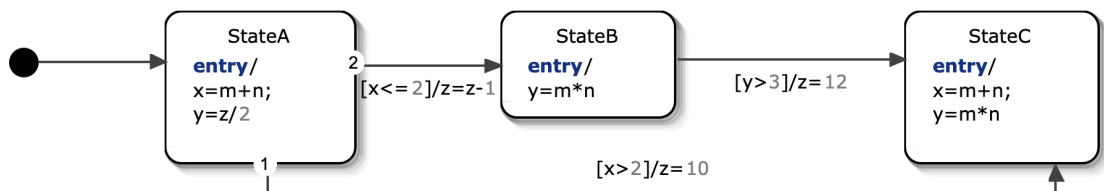
Figure 3.2. An Example for Illustrating *frequently read/updated variables* in A Simple Statechart Model

In this statechart model, there are three states and tree transitions. The actions in these states and transitions form itemsets of actions as $\mathcal{A} = \{\{x = m + n; y = z/2\}, \{z = z - 1; y = m * n\}, \{z = 12, x = m + n, y = m * n\}, \{z = 10, x = m + n, y = m * n\}\}$. From the itemsets of actions, we can obeserve that variable x and y are often updated together and the supprot/appereance of this pair of updated

variables is 3. In addition, we find out that variables m and n are often read together. Therefore we can infer that x and y are possiblely frequently updated together, and m and n are possible frenquently read together. For simplicity of description, we refer to those variables that share such read/updated correlation as *frequently read/updated variables*. However, these types of variables that need to be read or updated together consistently often are implicitly represented in system design models, which leads the assumptions of why these variables need to read/updated together to be unspecified. Therefore, in this paper, we take *frequently read/updated variables* as the second *syntactic carriers* of unspecified assumption that *UACFinder* need to mine out from the system design models.

3.2.3 Frequently Executed Action Sequences as Syntactic Carriers of Potential Unspecified Assumptions.

A simple example of a *frequently executed actions* is the function call pair of *lock* and *unlock*: a call to *lock* should be followed by a call to *unlock* later. In addition to the well-known *frequently executed actions* there are also many *frequently executed actions* in the development of medical cyber-physical system models. For instance, laser surgery is a surgical procedure that uses a laser to remove problematic tissues and is widely used in airway surgery, thoracic surgery, eye surgery, etc. For airway laser surgery, to avoid the potential risk of fire, we have to ensure two actions to be always executed by the sequence: **turn-off-ventilator**, and then **turn-on-laser**.

We use a simple statechart model in Fig. 3.3 to illustrate how *frequently executed actions* may exist in a system statechart model.
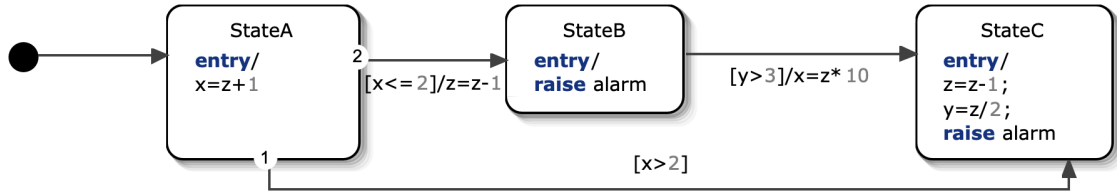
Figure 3.3. A Statechart Example for Illustrating *frequently executed actions*

In the statechart model depicted in the Fig. 3.3, there are four itemsets of actions: 1) itemset of actions in $stateA$: $\{x = z + 1\}$; 2) itemset of actions in $transition1$ and $stateB$: $\{z = z - 1, raise\, alarm\}$; 3) itemset of actions in $transition3$ and $stateC$: $\{x = z * 10, z = z - 1, y = z/2, raise\, alarm\}$; and 4) itemset of actions in $transition2$ and $stateC$: $\{z = z - 1, y = z/2, raise\, alarm\}$. From the four itemsets of actions, we observe that the sub-itemset $\{z = z - 1, raise\, alarm\}$ occur three times. Comparing with occurrences of the rest action itemsets, we can infer that $\{z = z - 1, raise\, alarm\}$ may be frequently executed actions (an execution pattern). However, the assumptions of why some actions are treated as *frequently executed actions* are often too tedious to be documented by model developers and hence left unspecified. When these unspecified assumptions are violated by other model developers who are unaware of or forget about the *frequently executed action*, defects can be easily introduced into systems. Therefore, the *UACFinder* takes *frequently executed actions* as a targeted *syntactic carrier* for potential unspecified assumptions.

3.3 Obtain Itemset Database of Actions from Statechart Model

As this thesis focuses on extracting *syntactic carriers* in the form of *constant variables*, *frequently read/updated variables*, and *frequently executed actions* from statechart models, we develop a parsing component in the *UACFinder* that parses a statechart model, and then builds an itemset database of actions and variables. At the end, we convert the extract *frequently read/updated variables* and *frequently executed*

actions into a frequent itemset mining problem. Although the current version of the *UACFinder* is targeting statechart models, it can be easily applied to other modeling languages by replacing the parsing component.

To parse the source code of a statechart model (the source file of a statechart is an XML file), the *UACFinder* first makes use of DOM parser [54] to obtain the intermediate representations of the statechart model. The intermediate representation is stored in our proposed data structure, with each instance of the data structure representing different types of elements in statechart models, such as variables, actions, events, transitions, and states. The proposed data structures for representing the statechart model are listed as follows:

- *Variable.* A *variable* is represented as (*type*, *name*, *value*, *scope*), where *type* $\in$ {*bool*, *string*, *int*, *real*}, *name* is a string value, and *scope* $\in$ {*const*, ϕ}.

- *Event.* As an *event* in a statechart is implemented as a boolean variable, we treat an event as a specific variable with fixed type *bool*.

- *Action.* An *action* is represented with (*statement*, V_c, V_u), where *statement* is string to represent the action itself. $V_c = \{v_1, \ldots, v_i, \ldots, v_n\}$, where for each variable $v_i \in V_c$, the value of v_i will be updated after execution of this action. $V_u = \{v_1, \ldots, v_j, \ldots, v_m\}$ and for each variable $v_j \in V_u$, v_j is involved in this action but the value of v_j is not been updated after execution of this action.

- *Transition.* A *transition* is represented as (*id*, *guard*, s_f, s_e, A), where *id* is a string indicating the identity of this transition. s_f and s_e are the ids of *from* and *to* states of the transition, respectively, and $A = \{a_1, \ldots, a_i, \ldots, a_n\}$ is a set of *actions* which are to be executed when the transition is active. *gurad* is a string of boolean expression for determining whether this transitions can be active.

- *State.* A *state* is defined as $(id, name, T_{in}, T_{out}, A)$, where id is a string indicating the identity of this state. *name* is also a string indicating the name of this state. $T_{in} = \{t_1, \ldots, t_i, \ldots, t_n\}$ and $T_{out} = \{t_1, \ldots, t_j, \ldots, t_k\}$ are sets of ids of transitions *into* and *outof* the state, respectively, and $A = \{a_1, \ldots, a_i, \ldots, a_n\}$ is a set of *actions* which are to be executed when the state is active.

- *Statechart*: A *statechart* is represented as $(name, V, T, S)$, where *name* is also a string indicating the name of the statechart. V is a set containing all the variables in this statechart. T is a set containing all the transitions in this statechart. S is a set containing all the states in this statechart.


```
<sgraph:Statechart xmi:id="st1"
  specification="internal:&#xA;var x : integer&#xA;var y : integer"
  name="example">
    <vertices xsi:type="sgraph:Entry" xmi:id="s0">
      <outgoingTransitions xmi:id="t0" target="s1"/>
    </vertices>
    <vertices xsi:type="sgraph:State" xmi:id="s1"
      specification="entry/ x=1" name="A" incomingTransitions="t0">
      <outgoingTransitions xmi:id="t1"
        specification="[x>0] /&#xD;&#xA;y=0" target="s2"/>
    </vertices>
    <vertices xsi:type="sgraph:State" xmi:id="s2"
      specification="entry/ x=5" name="A2" incomingTransitions="t1"/>
</sgraph:Statechart>
```
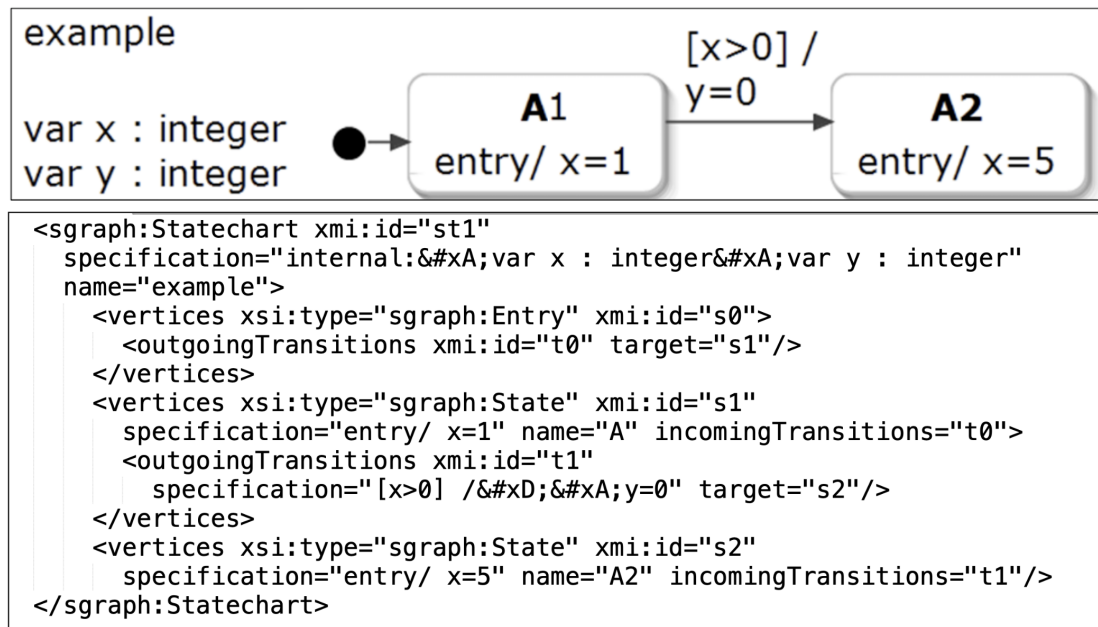
Figure 3.4. A Simple Yakindu Statechart Model and Its XML File

We use a Yakindu statechart example shown in Fig. 3.4 to illustrate how to use the proposed data structures to represent a statechart model. Since a Yakindu statechart is constructed by using specified XML schemes, we first use the DOM parser to obtain the tree structure of the statechart. In the Yakindu statechart, `sgraph : Statechart` is the identifier representing a statechart; `specification` is

the attributes to represent variables, actions, guards. `incomingTransitions` and `outgoingTransitions` are two identifiers indicating transitions, states are determined with the identifier `vertices`. As shown the Fig. 3.4 , there are three `vertices` representing states: `initial`/`entry`, `A1`, and `A2`, respectively. We have implemented an extended DOM parser to parse Yakindu statechart XML document to the proposed structures described above. Algorithm 2 depicts the procedure for constructing the proposed structures from a statechart model, and the time complexity of Algorithm 2 is $\mathcal{O}(S * T)$, where S is the total number of states and T is the total number of incoming transitions. The Java code for the procedure is at `https://github.com/fuzhicheng-java/UAFinder`. Because it is not sufficient to only access independent elements and extract their information, the parser also records relationship information among the elements, such as in which state an action takes place, and what variables are involved in an action or changed by the action. Such information is critical in mining *constant variables*, *frequently read/updated variables* and *frequently executed actions*.

3.4 Mining Syntactic Carriers

The *UACFinder* aims to extract the *syntactic carriers* in the form of *constant variables*, *frequently read/updated variables*, and *frequently executed actions*. Mining variables that are frequently read/updated together and actions that are frequently executed together are similar to the problem of finding a frequent *itemset* problem [50]. Therefore, we can apply data mining techniques to mine *frequently read/updated variables* and *frequently executed actions*. In particular, the *UACFinder* uses an FP-tree-based mining algorithm called FPclose [50] to identify the *syntactic carriers* that we are interested in. By combining all actions in a combination of a state and one of its incoming transition, we can form one itemset of actions. Then the itemsets of all the combinations form an itemset database, and we can apply data mining techniques to

---

Algorithm 2 Parsing Yakindu Statechart Model

Input: A XML Source File X of a Yakindu Statechart
Output: An Instantiated Yakindu Statechart $\mathtt{st} = (name = \text{""}, \mathtt{V}, \mathtt{T}, \mathtt{S})$

1: Parse X with DOM parser to be a tree structure $\mathtt{T_s}$
2: Declare $\mathtt{st} = (id = \text{""}, name = \text{""}, \mathtt{V} = \emptyset, \mathtt{T} = \emptyset, \mathtt{S} = \emptyset)$
3: For the root node $\mathtt{n_0}$ of T_s, set $\mathtt{Y}.name = \mathtt{n_0}.name$ and partition $\mathtt{n_0}.specification$ with Yakindu statechart keywords to instantiate V
4: **for** each *vertice* node $\mathtt{n_i} \in \mathtt{T_s}$ **do**
5:     Declare a state $\mathtt{s_i} = (id = \text{""}, name = \text{""}, \mathtt{T_{in}} = \emptyset, \mathtt{T_{out}} = \emptyset, \mathtt{A} = \emptyset)$.
6:     Set $\mathtt{s_i}.id = n_i.id$, $\mathtt{s_i}.name = \mathtt{n_i}.name$. Partition $\mathtt{n_i}.specification$ to instantiate $\mathtt{A}$. Partition $\mathtt{n_i}.incomingTransitions$ to instantiate $\mathtt{T_{in}}$. $\mathtt{S} = \{\mathtt{s_i}\} \cup \mathtt{S}$.
7: **end for**
8: **for** each *outgoingTransition* node $\mathtt{n_j} \in \mathtt{T_s}$ **do**
9:     Declare a transition $\mathtt{t_j} = (id = \text{""}, \mathtt{s_f} = \text{""}, \mathtt{s_t} = \text{""}, \mathtt{A} = \emptyset)$.
10:     Set $\mathtt{t_j}.id = n_j.id$ and $\mathtt{t_j}.\mathtt{s_t} = \mathtt{n_j}.target$. Partition $\mathtt{n_j}.specification$ to instantiate $\mathtt{A}$.
11:     Get the parent node $\mathtt{n_j^p}$ of $\mathtt{n_j}$, set $\mathtt{t_j}.\mathtt{s_f} = \mathtt{n_j^p}.id$ and $\mathtt{T} = \{\mathtt{t_j}\} \cup \mathtt{T}$.
12:     Find the state $\mathtt{s} \in \mathtt{S} \wedge \mathtt{s}.id = \mathtt{n_j^p}.id$. $\mathtt{s}.\mathtt{T_{out}} = \{\mathtt{t_j}.id\} \cup \mathtt{s}.\mathtt{T_{out}}$.
13: **end for**
14: **return** $\mathtt{st}$

---

mine the itemset of the database. The reason to convert a combination of a state and one of its incoming transition as an itemset is that in statechart the actions in a state and its incoming transitions must be executed together. It is worth pointing out that some actions can span across multiple states. However, mining these actions is much more complicated, and it requires more in-depth inter-procedural analysis of the statechart models, which is beyond the scope of the paper. In this section, we first introduce the background of frequent itemset mining. We then explain the mining procedures for the three *syntactic carriers*.

3.4.1 Frequent Itemset Mining. The initial application of the frequent itemset mining was in the analysis of straight market baskets [55]. It now has broad applications, including mining motifs in DNA sequences, analysis of customer shopping behavior, etc [56], [57]. The goal of frequent itemset mining is to efficiently find frequent itemsets (an itemset is a set of items) in a large database. In a database composed of a large number of itemsets, if a sub-itemset (the subset of an itemset)

is contained in more than a specified threshold (called min support) of itemsets, it is considered frequent. The number of occurrences of a sub-itemset A is denoted as A's support. The itemset that contains A is called its supporting itemset. For example, in an itemset database D: $D = \{\{a, b, c, d, e\}, \{a, b, d, e, f\}, \{a, b, d, g\}, \{a, c, h, i\}\}$. The support of sub-itemset $\{a, b, d\}$ is 3, and its supporting itemsets are $\{a, b, c, d, e\}$, $\{a, b, d, e, f\}$, and $\{a, b, d, g\}$. If the threshold of support is set as 3, the frequent sub-itemsets for D are $\{a\}$:4, $\{b\}$:3, $\{d\}$:3, $\{a, b\}$:3, $\{a, d\}$:3, $\{b, d\}$:3, and $\{a, b, d\}$:3, where the numbers are the supports of the corresponding sub-itemsets.

To solve the frequent itemset mining problem, a few algorithms have been proposed [55], [58], [59]. The *UACFinder* chooses to use the FPclose algorithm [50], one of the most efficient frequent itemset mining algorithms. Instead of generating the complete set of frequent sub-itemsets, FPclose mines only the closed sub-itemsets. A closed sub-itemset is the sub-itemset whose support is different from that of its super-itemsets. In the example above, the frequent sub-itemsets $\{b\}, \{d\}, \{a, b\}, \{a, d\}$ and $\{b, d\}$ are not closed since their supports are the same as their super-itemset$\{a, b, d\}$. FPclose only generates the closed sub-itemsets $\{a\}$:4 and $\{a, b, d\}$:3 as result. This can significantly improve time and space performance since it can avoid generating an exponential number of frequent sub-itemsets.

After the *UACFinder* parses a statechart model and generates an itemset database of all actions, it applies the closed frequent itemset mining algorithm, FPclose, on the database to find closed frequent sub-itemsets. If a set of numbers appear together in any itemsets for more than a specified threshold number (min support) of times, this sub-itemset is considered frequent. However, it is not sufficient to only know the patterns and their support values (i.e., how many times the pattern occurs). It would be more useful for domain experts and developers if the location in which one pattern occurs. Such information is also needed later for domain experts'

validation. Unfortunately, the original data mining algorithms, FPclose in particular, are not designed precisely for our purpose. They only output the support values for each discovered pattern but not their supporting itemsets. We enhance the mining algorithm to address the problem by also maintaining the supporting itemsets during the mining process.

3.4.2 Mining Constant Variables. In computer programming languages, constant variables are variables whose value cannot change after the initial assignment, and a constant variable is declared with a keyword. For instance, in C/C++, the keyword *const* is used to declare these constant variables. Similar to programming languages, many modeling languages also define a constant variable with one keyword. However, during system models development, model developers often declare global or local variables and use them as constant variables. This practice makes extracting all constant variables much more difficult. We not only need to extract the variables already declared with the keyword, such as *const*, but also have to examine the system design models to extract the variables that are not declared as constant variables but used as constants. We use the following two rules to determine if a variable is a constant:

- `R1`. A variable is declared with the keyword for constant.

- `R2`. A variable is not declared with the keyword for constant, but the value of the variable is never changed after its initialization during the execution of the system.

After constant variables being explicitly identified based on the two rules, domain experts and model developers can work together to determine whether there exist unspecified assumptions about the declaration of each constant variable. However, only highlight constant variables is not enough to validate the safety of system models

because domain experts also need to understand which parts of system models may be affected when the initialized values of constant variables are incorrect. Therefore, we not only extract all the constant variables but also record locations where the variables are used. Based on the locations of variables in the system models, domain experts can perform impact analysis about the condition when the unspecified assumptions of constant variables are violated.

We use the Yakindu statechart model as an example to illustrate how to extract constant variables and record their traces in the system design models. To record constant variables and states/transitions where the variables are used, we define a structure $C = \{c|c = (v, S, T)\}$, where v, S, and T is a constant variable, a set of states in which v is used; and a set of transitions in which v is used, respectively. The brief explanation of the procedure for extracting constant variables is described as follows.

Given an instantiated Yakindu statechart $st = (\text{name}, V, T, S)$, we declare $C = \emptyset$. For each variable $v \in V$, we traverse all actions in T and S. If v satisfies $R.1$ or $R.2$, we declare $c = < v, S_v = \emptyset, T_v = \emptyset >$ and put the states and transitions in which v is used to S_v and T_v, respectively. At the end, we set $C = C \cup \{c\}$. Algorithm 3 depicts the procedure extracting constant variables from an instantiated Yakindu statechart model, and the time complexity of Algorithm 3 is $\mathcal{O}(V * T)$, where V is the total number of variables and T is the total number of incoming transitions in the statechart model. In Yakindu statechart models, there exist two keywords: *always* and *oncycle* to enable an action or a state for updating variables to be executed in every run-to-completion step. At this specific situation, we treat the occurrences of updated variables to be infinity. Although the Algorithm 3 currently works for statechart models, it can be easily extended to other modeling languages by replacing the user-defined data structure described in Section 3.3.

Algorithm 3 EXTRACTING CARRIERS AS CONSTANT VARIABLES

Input: An Instantiated Yakindu Statechart $\mathtt{st} = (name, V, T, S)$

Output: A Set $\mathtt{C} = \{c | c = < v, S_v, T_v >\}$, where v is a constant variable, S_v and T_v are the sets of states and transitions in which v is used, respectively.

1: Declare $\mathtt{C} = \emptyset$
2: **for** each variable $\mathtt{v_i} \in V$ **do**
3:     Declare a boolean variable $isConst = true$ and $c_i = (v_i, S_u = \emptyset, T_u = \emptyset)$.
4:     **for** each state $\mathtt{s_i} \in S$ **do**
5:         For each action in $\mathtt{s_i}$, if $\mathtt{v_i}$ is used, $S_u = S_u \cup \{\mathtt{s_i}\}$; if $\mathtt{v_i}$ is changed, set $isConst = false$.
6:     **end for**
7:     **for** each transition $\mathtt{t_i} \in T$ **do**
8:         For each action $\mathtt{t_i}$, if $\mathtt{v_i}$ is used, $T_u^i = T_u^i \cup \{\mathtt{t_i}\}$; if $\mathtt{v_i}$ is changed, set $isConst = false$.
9:     **end for**
10:     **if** $isConst$ **then**
11:         $\mathtt{C} = \mathtt{C} \cup \{\mathtt{c_i}\}$
12:     **end if**
13: **end for**
14: **return** $\mathtt{C}$

3.4.3 Mining Frequently Read/Updated Variables and Executed Action Sequences. In this section, we explain how to extend the closed frequent itemset mining algorithm-FPclose algorithm [50] to mine *frequently read/updated variables* and *frequently executed actions*. Note that actions in a state and one of its incoming transitions must execute together. For example, given the statechart model in Fig. 3.5, the itemset of actions in the StateC and the transition from StateB to StateC is $\{x = z * 10, z = z - 1, y = z/2, raise\ alarm\}$.
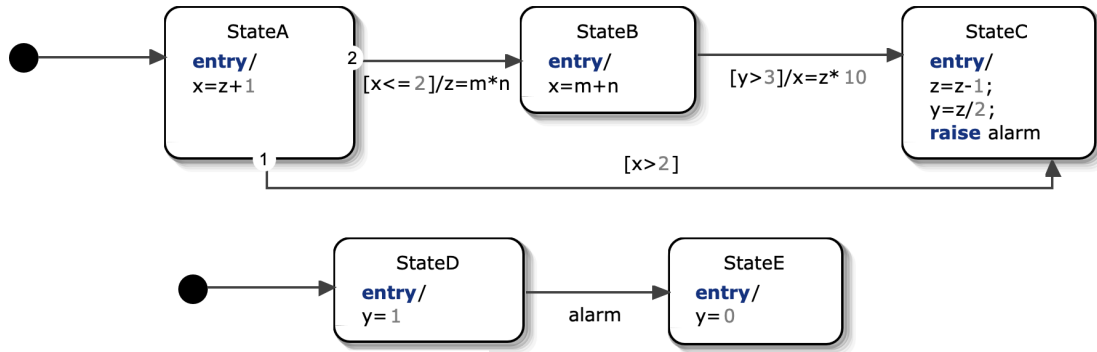
Figure 3.5. Simple Statechart Model

As we mentioned above, an action is represented in the form of assignments, event raises, and function calls. If `raise alarm` is executed in `StateC`, any state whose incoming transition only needs `alarm` event as the guard will also be executed. Furthermore, `StateE` will be active after the `alarm` is raised. Based on the semantics of event execution, when `StateC` and its incoming transition are activated, only using the itemset of actions $\{x = z * 10, z = z - 1, v = v/2, raise\ alarm\}$ to represent all executed actions is not correct. The right itemset of actions should include all actions in `StateC`, the transition from `StateD` to `StateE`, except event raise actions, i.e., $\{x = z * 10, z = z - 1, y = z/2, y = 0\}$.

The Algorithm 4 is designed to extract the itemsets of actions based on the execution feature of events. The time complexity of Algorithm 4 is $\mathcal{O}((S*T)^2)$, where S is the total number of states and T is the total number of incoming transitions in the statechart model.

After collecting the itemset database of actions, we can start to mine *frequently read/updated variables* and *frequently executed actions*.

3.4.4 Mining Frequently Read/Updated Variables . The *UACFinder* conducts static analysis to collect *frequently read/updated variables* from all actions in

---

Algorithm 4 EXTRACT ITEMSET OF ACTIONS BASED EVENT FEATURE

Input: An instantiated Yakindu statechart $\mathtt{st} = (\mathtt{name}, \mathtt{V}, \mathtt{T}, \mathtt{S})$

Output: A set $\mathcal{A} = \{\mathtt{A_1}, \mathtt{A_2}, ... \mathtt{A_n}\}$. For each $\mathtt{A_i} \in \mathcal{A}$, $\mathtt{A_i} = (\mathtt{AS}, \mathtt{T_i}, \mathtt{S_i})$ where $\mathtt{AS}$ is the itemset of actions of the transitions in $\mathtt{T_i}$ and the states in $\mathtt{S_i}$.

 1: Declare $\mathcal{A} = \emptyset$.

 2: **for** each state $\mathtt{s_i} \in \mathtt{S}$ **do**

 3:    **for** each incoming transition $\mathtt{t_i}$ of $\mathtt{s_i}$ **do**

 4:       Declare $\mathtt{AS} = \emptyset$, $\mathtt{T_i} = \{\mathtt{t_i}\}$, $\mathtt{S_i} = \{\mathtt{t_i}\}$.

 5:       Put all the actions in $\mathtt{s_i}$ and $\mathtt{t_i}$ into $\mathtt{AS}$.

 6:       **if** $\mathtt{AS}$ contains event raise actions **then**

 7:          **for** each event raise action $\mathtt{e_k} \in \mathtt{AS}$ **do**

 8:             For each transition $\mathtt{t_k} \in \mathtt{T}$ whose gurad is $\mathtt{e_k}$, find all states $\mathtt{S_k}$ whose incoming tranistion is $\mathtt{t_k}$, and $\mathtt{T_i} = \mathtt{T_i} \cup \{\mathtt{t_k}\}$.

 9:             For each $\mathtt{s} \in \mathtt{S_k}$, put all the actions in $\mathtt{s}$ into $\mathtt{AS}$, and $\mathtt{S_i} = \mathtt{S_i} \cup \{\mathtt{s}\}$.

10:             $\mathtt{AS} = \mathtt{AS} \backslash \mathtt{e_k}$.

11:          **end for**

12:       **end if**

13:       Declare $\mathtt{A} = (\mathtt{AS}, \mathtt{T_i}, \mathtt{S_i})$, and $\mathcal{A} = \mathcal{A} \cup \mathtt{A}$.

14:    **end for**

15: **end for**

16: **return** $\mathcal{A}$

---

statechart models. Similar to constant variables mining, the *UACFinder* not only mines the *frequently read/updated variables* but also records the locations of variables. The goal of this step is to identify *frequently read/updated variables* in statechart models whose occurrences exceed a given threshold. For each set of read/updated variables that satisfies this property, we refer it as a pattern of *frequently read/updated variables*. Given the itemset of read/updated variables, different approaches can be used to extract such patterns. For example, one solution is to count the number of itemsets containing both x and y for every pair of variables $\{x, y\}$. Although this solution is relatively simple, it cannot scale to large system design models with a large number of states and transitions. Moreover, it would be difficult to extend this algorithm to consider *frequently read/updated variables* that involve more than two variables such as $\{x, y, z\}$. Our approach is to use a well-studied data mining technique, i.e., frequent itemset mining [50]. Frequent itemset mining

examines a database where each entry is an itemset. For example, in an itemset database D $= \{\{w, y, z\}, \{v, w, y, z\}, \{w, x, y\}\}$, support indicates how many times a set of items occurs in the database D, and min_support is the threshold for filtering frequent itemsets out. In this example, if min_support $= 3$, the mining result will show that itemsets $\{w\}, \{y\}, \{w, y\}$ are frequent. If min_support $= 2$, itemsets $\{w\}, \{y\}, \{z\}, \{w, y\}, \{w, z\}, \{y, z\}, \{w, y, z\}$ are frequent. We separate the procedure of mining *frequently read/updated variables* into two sub-procedures: 1) mining *frequently updated variables*, and 2) mining *frequently read variables*.

To mine *frequently updated variables*, we only need to collect the updated variables in the actions. For instance, the itemset of actions $\{z = m * n, x = m + n\}$ of StateB and its incoming transition in the statechart models in Fig. 3.5, we only need to extract the variables on the left side of assign statements, which is $\{z, x\}$. After collecting all the itemsets of actions $\mathcal{A}$ from a given statechart model by applying Algorithm 4, we first extract the itemsets of updated variables $\mathcal{V}$ from $\mathcal{A}$. We then apply FPclose algorithm on $\mathcal{V}$ to mine *frequently updated variables*. The whole procedure for mining *frequently updated variables* is described in Algorithm 5. The time complexity of Algorithm 5 is $\mathcal{O}((S * T)^2)$, where S is the total number of states and T is the total number of incoming transitions in the statechart model.

To mine *frequently read variables*, we need to collect itemsets of variables in the actions excluding updated variables. For instance, the itemset of actions $\{z = m * n, x = m + n\}$ of StateB and its incoming transition in the statechart models in Fig. 3.5, we only want to extract the variables on the right side of assign statements, which is $\{m, n\}$. The procedure of mining *frequently read variables* is similar to the procedure for mining *frequently updated variables*. The Algorithm 6 depicts the procedures for mining *frequently read variables*. The time complexity of Algorithm 6 is $\mathcal{O}((S * T)^2)$, where S is the total number of states and T is the total number of

---

Algorithm 5 MINING FREQUENTLY UPDATED VARIABLES

Input: An instantiated Yakindu statechart $\mathtt{st} = (\mathtt{name}, \mathtt{V}, \mathtt{T}, \mathtt{S})$, and the threshold number $\mathtt{min\_support}$ for filtering *frequently updated variables* out.

Output: A set of patterns of *frequently updated variables* $\mathtt{P} = \{\mathtt{p}|\mathtt{p} = (\mathtt{V_i}, \mathtt{S_i}, \mathtt{T_i}, \mathtt{support})\}$, where $\mathtt{V_i}$ is an itemset of *frequently updated variables*, $\mathtt{S_i}$ is a set of states in which the itemset of variables occurs, $\mathtt{T_i}$ is a set of transitions in which the itemset of variables occurs. $\mathtt{support}$ indicates how many times the itemset of variables $\mathtt{V_i}$ occurs in $\mathtt{st}$.

1: Applying Algorithm 4 to get the itemset database of actions $\mathcal{A}$ from $\mathtt{st}$. For each $\mathtt{A_i} \in \mathcal{A}, \mathtt{A_i} = (\mathtt{AS}, \mathtt{T_i}, \mathtt{S_i})$ where $\mathtt{AS}$ is the itemset of actions of transitions in $\mathtt{T_i}$ and states in $\mathtt{S_i}$

2: Declare $\mathtt{V} = \emptyset$.

3: **for** each element $\mathtt{A_i} \in \mathcal{A}$ **do**

4:    Extract the itemset of updated variables $\mathtt{V_i}$ from $\mathtt{A_i}.\mathtt{AS}$. Then $\mathtt{V} = \mathtt{V} \cup \{\mathtt{V_i}\}$

5: **end for**

6: Use FPclose to mine itemsets of *frequently updated variables* from $\mathtt{V}$: $\mathtt{V_{out}} \leftarrow FPCLOSE(\mathtt{V}, \mathtt{min\_support})$

7: Declare $\mathtt{P} = \emptyset$.

8: **for** each itemset of *frequently updated variables* $\mathtt{V_i} \in \mathtt{V_{out}}$ **do**

9:    Find the set of states $\mathtt{S_i}$ and the set of transitions $\mathtt{T_i}$ where the itemset of updated variables $\mathtt{V_i}$ occurs. Let $\mathtt{p_i} = (\mathtt{V_i}, \mathtt{S_i}, \mathtt{T_i}, \mathtt{support})$, where $\mathtt{support}$ is the time of occurrences of $\mathtt{V_i}$. $\mathtt{P} = \mathtt{P} \cup \{\mathtt{p_i}\}$.

10: **end for**

11: **return** $\mathtt{P}$

---

incoming transitions in the statechart model.

3.4.5 Mining Frequently Executed Actions. The purpose of this step is to extract *frequently executed actions* in statechart models whose occurrences exceed a given threshold. For each set of executed action sequences that satisfies this property, we treat it as a pattern of *frequently executed actions*. Through the Algorithm 4, we can extract the itemsets of all actions in a statechart model. As mentioned in Section 3.3, an action in statechart is represented in the form of strings. However, sometimes a string can be too large to ensure the mining procedures to be efficient enough. Therefore, to improve the efficiency of the *UACFinder* for extract *frequently executed actions*, we first use BKDR Hash Function [60] to hash each action string to a number. For example, for the itemset of actions $\{z = z - 1, x = x * 2\}$

Algorithm 6 MINING FREQUENTLY READ VARAIBLES

Input: An instantiated Yakindu statechart $st = (name, V, T, S)$, and the threshold number $min\_support$ for filtering *frequently read variables* out.

Output: A set of patterns of *frequently read variables* $P = \{p|p = (V_i, S_i, T_i, support)\}$, where A_i is an itemset of *frequently read variables*, S_i is a set of states in which the itemset of variables occurs, T_i is a set of transitions in which the itemset of variables occurs. $support$ indicates how many times the itemset of variables V_i occurs in st.

1: Applying Algorithm 4 to get the itemset database of actions $\mathcal{A}$ from st. For each $A_i \in \mathcal{A}, A_i = (AS, T_i, S_i)$ where AS is the itemset of actions of transitions in T_i and states in S_i

2: Declare $V = \emptyset$.

3: **for** each element $A_i \in \mathcal{A}$ **do**

4:     Extract the itemset of read variables V_i from $A_i.AS$. Then $V = V \cup \{V_i\}$

5: **end for**

6: Use FPclose to mine *frequently read variables* from V: $V_{out} \leftarrow FPCLOSE(V, min\_support)$

7: Declare $P = \emptyset$.

8: **for** each itemset of *frequently read variables* $V_i \in V_{out}$ **do**

9:     Find the set of states S_i and the set of transitions T_i where the itemset of updated variables V_i occurs. Let $p_i = (A_i, S_i, T_i, support)$, where $support$ is the time of occurrences of V_i. $P = P \cup \{p_i\}$.

10: **end for**

11: **return** G

---

in *stateB* and its incoming transition shown in the Fig. 3.5, after applying the hash procedure, the converted itemset of actions is $\{z = z - 1, x = x * 2\}$ is $\{8683372948524200635, 8541304859824854267\}$.

Before we hash all the actions into numbers, we have to solve the duplication problem. We use the following two examples to illustrate the problem.

- Taking the actions $a_1 = \{z = z - 1\}$ and $a_2 = \{z - 1\}$ as an example, since there exists white spaces in the action a_2, if we just hash to these two actions into numbers, the *UACFinder* will treat a_1 and a_2 as two different actions.

- For two actions $a_3 = \{z = z - 1\}$ and $a_3 = \{-1 + z\}$, after hashing to the two actions into numbers, the *UACFinder* will treat a_3 and a_4 as two different

actions. Similar cases often exist in actions, such as $a_5 = \{a = a - b - (c - d)\}$ and $a_6 = \{a = a - b - c + d\}$.

To address the duplication problems, we first remove all the white spaces in the actions, we then develop an algorithm to re-organized the sequences of one action. A simple idea behind the algorithm is to keep a record of the Global and Local Sign (operators such as +, / * -) in each action. The Global Sign here means the multiplicative sign at each operand. The resultant sign for an operand is the local sign multiplied by the global sign at that operand. For example, the action $a = a + b - (c - d)$ is evaluated as $a = (+) + a(+) + b(-) + c(-) - d => a = a + b - c + d$. The global sign (represented inside bracket) is multiplied to the local sign for each operand. In addition, a stack and a vector are used to keep the record of the global signs, and the counts of the operands, respectively. The entire code for this algorithm can found at `https://github.com/fuzhicheng-java/UAFinder`.

After removing duplication of actions, the *UACFinder* applies FPclose algorithm to mine *frequently executed actions*. Similarly, it is not sufficient to only know the possible *frequently executed actions*. It is also important to know in which parts of the system models the extracted *frequently executed actions* occur. Unfortunately, the original FPclose algorithm only outputs the support values for each discovered group. We extend FPclose algorithm to not only extract *frequently executed actions*, but also identify the traces where these *frequently executed actions* occur. We use the Yakindu statechart as an illustrative example and describe the procedure to mine *frequently executed actions*in Algorithm 7. The time complexity of Algorithm 7 is $\mathcal{O}((S * T)^2)$, where S is the total number of states and T is the total number of incoming transitions in the statechart model.

Algorithm 7 MINING FREQUENTLY EXECUTED ACTIONS

Input: An instantiated Yakindu statechart $st = (name, V, T, S)$, and the threshold number `min_support` for filtering *frequently executed actions* out.

Output: A set of patterns of *frequently executed actions* $P = \{p | p = (E_i, S_i, T_i, support)\}$, where E_i is an itemset of *frequently executed actions*, S_i is a set of states in which the itemset of variables occurs, T_i is a set of transitions in which the itemset of actions occurs. `support` indicates how many times the itemset of actions E_i occurs in `st`.

1: Applying Algorithm 4 to get the itemset database of actions $\mathcal{A}$ from `st`. For each $A_i \in \mathcal{A}$, $A_i = (AS, T_i, S_i)$ where AS is the itemset of actions of transitions in T_i and states in S_i.

2: Declare $E = \emptyset$.

3: **for** each element $A_i \in \mathcal{A}$ **do**

4:     Filter action duplications in $A_i.AS$ to generate a new itemset of actions A_k, and let $E = D \cup \{A_k\}$.

5: **end for**

6: Use FPclose algorithm to mine *frequently executed actions* from E: $E_{out} \leftarrow FPCLOSE(E, \texttt{min\_support})$

7: Declare $P = \emptyset$.

8: **for** each itemset of *frequently executed actions* $E_i \in E_{out}$ **do**

9:     Find the set of states S_i and the set of transitions T_i where the itemset of actions E_i occurs. Let $p_i = (E_i, S_i, T_i, support)$, where `support` is the time of occurrences of E_i, and $P = P \cup \{p_i\}$.

10: **end for**

11: **return** P

3.5 A Case Study – Using the *UACFinder* to Find Unspecified Assumptions in Cardiac Arrest Statechart Model

In this section, we use a cardiac-arrest treatment statechart model [51] to evaluate the *UACFinder*. Cardiac arrest is initially the devastating cessation of cardiac activity, which is caused by the loss of the heart's electrical and muscular pumping function. The completed statechart model can be found in [51]. In the cardiac-arrest statechart model [51], there are a total of 55 states, 155 transitions, and 45 declared variables. We apply the *UACFinder* to look for potential unspecified assumption carriers. We then invite a medical doctor footnote 1 to validate whether these carriers carry unspecified assumptions.

3.5.1 Mining Constant Variables as Syntactic Carriers.   By applying the

UACFinder on the cardiac-arrest treatment statechart model, we extracted 13 constant variables as syntactic carriers from the total of 45 variables in the cardiac-arrest statechart model. The extracted constant variables, their initialized values, and assumptions of the variable initialization are described in Table 3.1. The assumptions for initialization of these 13 variables are not specified in system design models. Therefore, the initial values set to these variables need to be validated by medical professionals to decides whether there exist unspecified assumptions about the initialization. For example, one unspecified assumption made in the initialization of the variable Kidney.Potassium_High_Threshold to be 5.5 is that the patient must be an adult. However, if the model with this unspecified assumption is used to examine pediatric patients, it would output inaccurate test results and may lead doctors to make incorrect decisions. Hence, such assumptions need to be specified for all stakeholders.

In addition, the *UACFinder* extracts all constant variables and output the traces where a constant variable is used. Fig. 3.6 shows the traces where the variable BGI.paCO2_High_Threshold is used in the Blood Gas Examination statechart model, which is a part of the cardiac-arrest statechart model. Blood gases are measurements of how much oxygen and carbon dioxide are in patients bloodstream, which is for detecting metabolic acidosis, respiratory acidosis, and hypercapnia [51].

3.5.2 Mining Frequently Read/Updated Variables as Syntactic Carriers.
To ensure mining as many frequently read/updated variables as possible, we set the threshold to be 2 for filtering *frequently read variables* and *frequently updated variables* out from the statechart model. With the *UACFinder*, we extract 14 possible *frequently executed actions* from the cardiac-arrest statechart model as shown in Table 3.2.

To facilitate validation, the *UACFinder* also highlights states and transitions where *frequently read variables* and *frequently updated variables* occurs in statechart

Blood Gas Imbalance
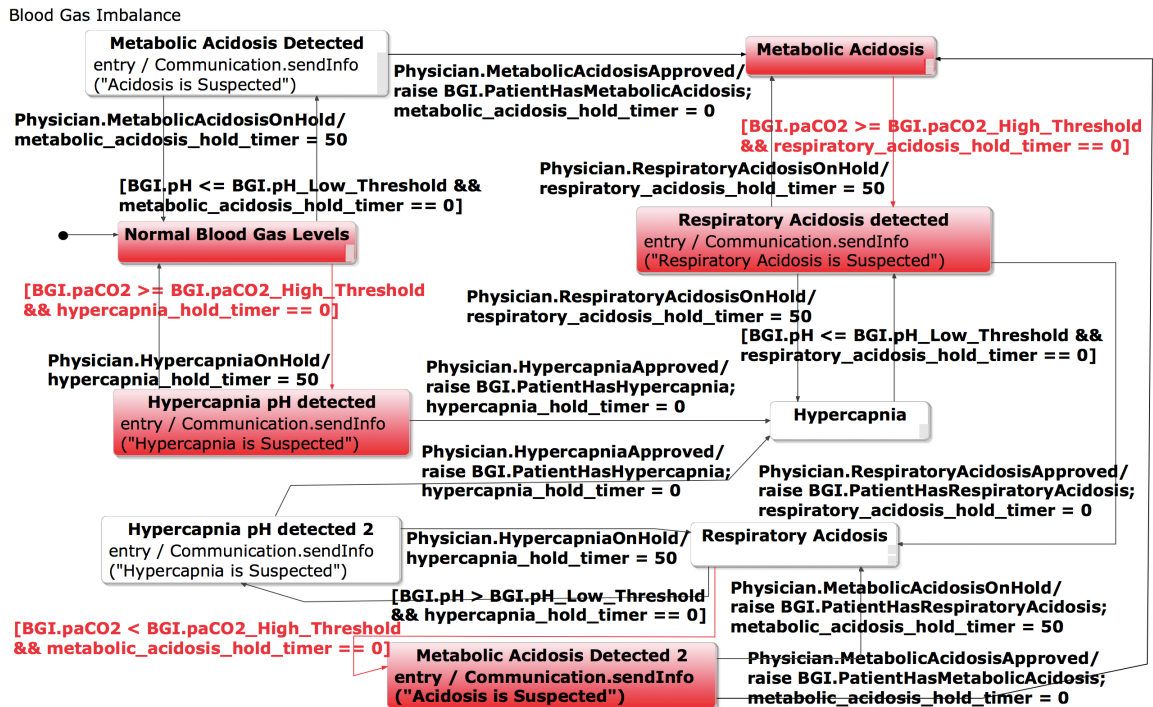
Figure 3.6. Traces Where $BGI.paCO2\_High\_Threshold$ Is Used

modes, such as recording the locations where the itemset of frequently read variables: $\{Kidney.Creatinine, Kidney.BUN\}$, which occurs in the Renal Insufficiency statechart model shown in Fig. 3.7.

3.5.3 Mining Frequently Executed Actions as Syntactic Carriers. Similar to mining *frequently read/updated variables*, we also set the threshold value of *support* to be 2 for finding out all possible *frequently executed actions*. However, the threshold value is not hard coded in the program, and end-users can declare a threshold value based on their domain knowledge. With the *UACFinder*, we extract 14 possible *frequently executed actions* from the cardiac-arrest statechart model as shown in Table 3.3.

To facilitate medical professionals' validation, the *UACFinder* also highlights states and transitions where the *frequently executed actions* occur in statechart modes, such as recording the locations where the *frequently executed action* $\{raise$ BGI.

Renal Insufficiency
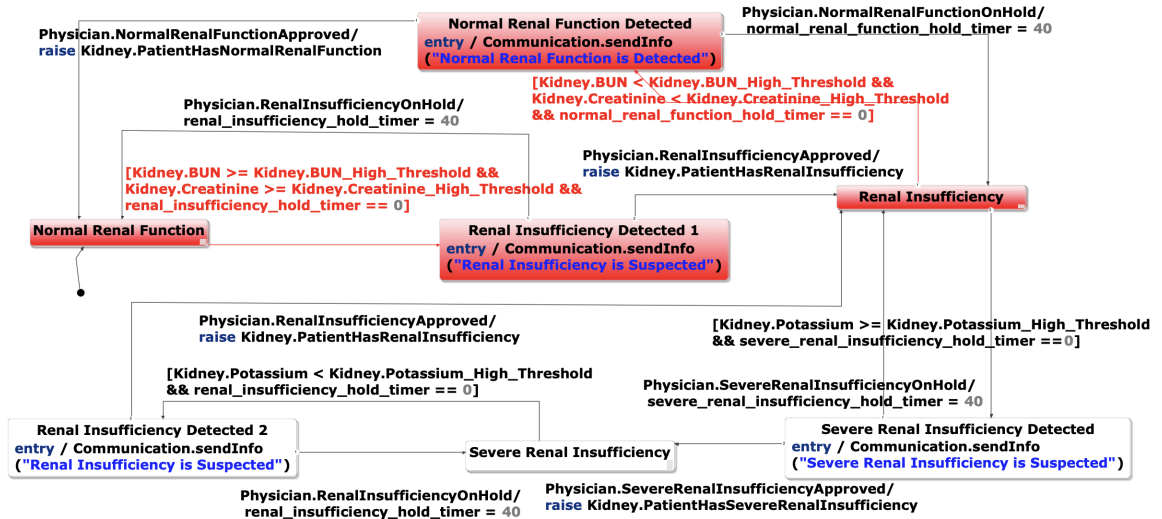
Figure 3.7. Traces Where Frequently Read Variables *Kidney.Creatinine, Kidney.BUN* Occurs in the Renal Insufficiency Satechart Model

PatientHasMetabolicAcidosis, $metabolic\_acidosis\_hold\_timer = 0$} occurs in the statechart model in Fig. 3.8.

3.5.4 Validating Unspecified Assumptions Associated with Mined Syntactic Carriers. Through the *UACFinder* we have mined syntactic carriers in the form of *constant variables*, *frequently read/updated variables*, and *frequently executed actions* from the simplified cardiac-arrest treatment statechart model [51]. The mined syntactic carriers are shown in the Table 3.1, Table 3.2, and Table 3.3, respectively. In addition, we invite a medical doctorfootnote 1 who is an expert in cardiac-arrest treatment to validate the unspecified assumptions associated with these carriers. The result of this validation is represented as identified assumptions that are listed in the rightest column in Table 3.1, Table 3.2, and Table 3.3.

For constant variables as syntactic carriers shown in Table 3.1, the medical doctor and developers have identified 13 unspecified assumptions associated with the constant variables. Taking `Kidney.BUN_High_Threshold` = 20 as an example, the initialization is based on one unspecified assumption that the normal range of
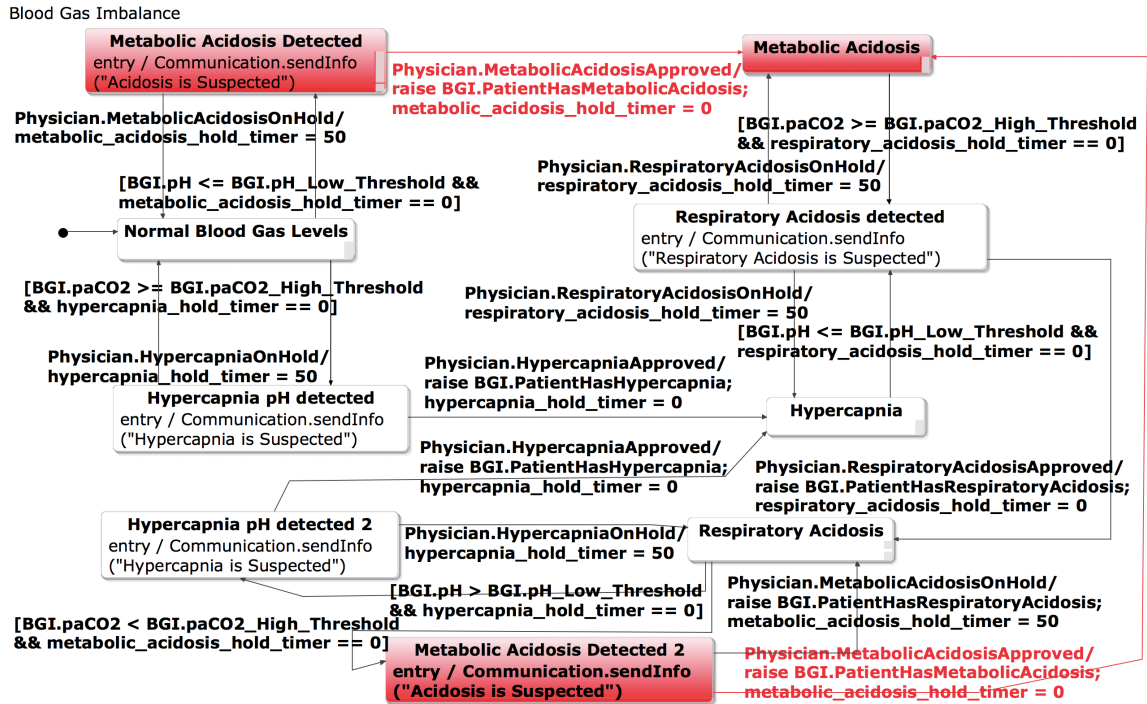
Figure 3.8.   Traces Where Frequently Executed Actions *raise BGI.PatientHas MetabolicAcidosis, metabolic_acidosis_hold_timer = 0* Occurs

BUN level for adults up to 60 years old is $6-20\text{mg/dL}$, and for adults over 60 years old the range is $8-23\text{mg/dL}$ [61].   However, assumptions associated with some constant variable initialization are not easy to be identified.   For instance of `Kidney.Creatinine_High_Threshold` $= 5$, the assumption for this initialization can not be identified. Because the normal range for creatinine in the blood could be 0.84 to 1.21 milligrams per deciliter (74.3 to 107 micromoles per liter), but the range can vary from lab to lab, between men and women, and by age [62], [63].

We has identified five unspecified assumptions that are associated with *frequently read/updated variables*, as shown in Table 3.2. For example, *rhythm (Arrhythmia.rhythm), blood pressure (Arrhythmia.BP), heart rate (Arrhythmia.HR)* must be read together to diagnose heart rhythm problems   [65]. Through validation of *frequently read/updated variables* as syntactic carriers, we observe that for specific disease treatment, some vital variables are always assumed to be binded together in

reading and updating for diagnosing purposes.

In addition, 10 unspecified assumptions that are associated with mined *frequently executed actions* have been identified and confirmed by the medical doctor, as shown in Table 3.3. We have sent the rest *frequently executed actions* to the original model developers for future validation. We take the *frequently executed actions*: *raise BGI.PatientHasMetabolicAcidosis; metabolic_acidosis_hold_timer = 0; Communication.sendInfo()* as an example, the unspecified assumptions behind this itemset of *frequently executed action* is that when a patient is admitted to the procedure of blood examination, a timer is triggered to record the diagnosis time, and when the diagnosis is confirmed, the timer should reset to zero, and the system should inform doctors about the vital symptom.

Through the analysis of these identified *syntactic carriers*, we find out that based on the mined *frequently executed actions*, the *UACFinder* can be also used to find potential bugs by detecting violations to these *frequently executed actions*. The main idea is that the *frequently executed actions* usually holds for most cases, and violations happen only occasionally. For example, from the table 3.3 we observe that in most cases each *event raise* is followed by the action *Communication.sendInfo()*. However, int itemset of executed actions: *raise BGI.PatientHasHypercapnia; hypercapnia_hold_timer = 0*, the *event raise* is not followed by *Communication.sendInfo()*, which indicates that the action *Communication.sendInfo()* may be missing. Whether the low-frequency *frequently executed action*is designed on purpose or just a copy-pasting error raises another interesting problem that may be addressed by detecting violations based on high-frequency *frequently executed actions* in our future work. In addition, the results of the validation indicate that it is necessary to involve domain experts during the system development process for identifying all possible unspecified assumptions.

3.6 Summary

In this Chapter, we present a tool called *UACFinder* which uses data mining techniques to automatically and efficiently extract *syntactic carriers* of unspecified assumptions in the form of *constant variables*, *frequently read/updated variables*, and *frequently executed actions* from system statechart models. The *UACFinder* also provides traces where these carriers occur in statechart models. We evaluate the *UACFinder* with a cardiac arrest-assist system statechart model. The results of the evaluation indicate that the *UACFinder* can identify unspecified assumptions and facilitate validation for domain experts at the system design stage of developing M-CPS systems. Although the current version of the *UACFinder* is based on statechart models, it can be easily applied to other modeling languages by replacing the parsing component. In addition, the fundamental methodology to identify potential assumptions through syntactical carriers can also be employed in other domain areas. Currently, the only focuses on potential unspecified assumption carriers in the forms of *constant variables*, *frequently read/updated variables*, and *frequently executed actions*. However, there may be other forms of carriers that associate with unspecified assumptions, which will be our future work.

As the *UACFinder* also provides traces where *constant variables*, *frequently read/updated variables*, and *frequently executed actions* occur, we can potentially use the information to perform impact analyze on the violation of unspecified assumptions. However, since the number of identified unspecified assumptions may be large, and how to determine the severity levels of unspecified assumptions may cause to systems is another challenge. In the next Chapter, we will focus on how to facilitate domain experts to perform impact analysis on unspecified assumptions.

Table 3.1. Constant Variables As Syntactic Carriers Mined From the Cardiac-Arrest Statechart Model. The right column represents the unspecified assumptions identified by domain experts and model developers.

Constant Variables	Identified Assumptions
Kidney.BUN_High_Threshold = 20	The initialization is based on the assumption that targeted patients are adults who is under 60 years. General reference ranges for a normal BUN level are: (1) Adults up to 60 years of age: 6-20 mg/dL; and (2) Adults over 60 years of age: 8-23 mg/dL [61].
Kidney.Creatinine_High_Threshold = 5	The assumption for this initialization can not be specified. Because the normal range for creatinine in the blood may be 0.84 to 1.21 milligrams per deciliter (74.3 to 107 micromoles per liter), although this can vary from lab to lab, between men and women, and by age [62], [63].
Kidney.Potassium_High_Threshold = 5.5	The initialization is based on the assumption that targeted patients are adults. Normal levels of potassium in the blood are generally between 3.7 and 5.2 mEq/L for adults and 3.4-4.7 mEq/L for children [53]. For adults, the safety range for potassium levels is usually between 3.5 and 5.5 mEq/L [53].
BGI.pH_Low_Threshold = 7.2	The initialization is based on the assumption that the normal blood pH range is 7.35-7.45 for adults [64].
BGI.paCO2_High_Threshold = 45	The initialization is based on the assumption that the normal blood paCO2 range is 35-45 for adults [64].
Physician.Deviation_Short_Timer = 20	The holding time for doctors to make decision at emergency condition is 20 seconds.
Physician.Deviation_Long_Timer = 100	The holding time for doctors to make decision at normal condition is 100 seconds.
hypercapnia_jump_timer = 0	The counter or timer starts from 0, and the unit is second.
metabolic_acidosis_deviation_counter = 0	Same as above.
hypercapnia_deviation_counter = 0	Same as above.

Table 3.2. Frequently Read/Updated Variables as Syntactic Carriers in the Cardiac Arrest Statechart Model. The value of support is $+\infty$ indicates that its corresponding itemset of frequently updated variables will be updated after each execution cycle of the statechart model.

Frequently Read/Update Variables	Support	Type	Assumptions
Arrhythmia.rhythm Arrhythmia.BP Arrhythmia.HR	12	Read	Rhythm, Heart Rate and Blood Pressure are three binded vital variables to diagnose heart rhythm problems(heart arrhythmias) [65].
Arrhythmia.rhythm Arrhythmia.BP	4	Read	Rhythm and Blood Pressure are two vital variables for diagnosing ventricular tachycardia [66].
Kidney.Creatinine Kidney.BUN	2	Read	Creatinine and Blood Urea Nitrogen are two vital variables to diagnose renal insufficiency [67].
Arrhythmia.rhythm Arrhythmia.BP Arrhythmia.HR	$+\infty$	Updated	Rhythm, heart rate and blood pressure are three binded vital variables that should be updated together for diagnosing heart rhythm problems(heart arrhythmias) [65].
Kidney.Creatinine Kidney.BUN	$+\infty$	Updated	Creatinine, Potassium, and Blood Urea Nitrogen are three vital variables that should be updated for monitoring renal insufficiency symptoms [67]

Table 3.3. Frequently Executed Actions as Syntactic Carriers Mined From the Cardiac-Arrest Statechart Model. The most right column presents the identified assumptions about the syntactic carriers by domain experts.

Frequently Executed Actions	Support	Assumptions
raise Kidney.PatientHasRenalInsufficiency; Communication.sendInfo()	10	When there is vital symptoms of patents, the system should inform doctors.
raise BGI.PatientHasNormalpH; Communication.sendInfo()	7	Same as above
raise BGI.PatientHasMetabolicAcidosis; metabolic_acidosis_hold_timer = 0; Communication.sendInfo()	7	Same as above
raise Arrhythmia.PatientIsPEA; PEA_deviation_counter=0; Communication.sendInfo()	6	Same as above
raise Arrhythmia.PatientIsVtach; vtach_deviation_counter=0; Communication.sendInfo()	6	Same as above
raise Arrhythmia.PatientIsSinus; sinus_deviation_counter=0; Communication.sendInfo()	6	Same as above
raise PatientHasNormalRenalFunction; Communication.sendInfo()	5	Same as above
sinus_deviation_counter+=1; Communication.addToLog	4	After sinus_deviation_counter being increased by one second, adding "Deviation from Sinus" into the log database.
PEA_deviation_counter+=1; Communication.addToLog()	4	Every one second, the counter increase, adding "Deviation from PEA" into the log database.
vtach_deviation_counter +=1; Communication.addToLog()	4	After vtach_deviation_counter being increased by one second, adding "Deviation from VTach" into the log database.

CHAPTER 4

FACILITATE DOMAIN EXPERTS TO ANALYZE THE IMPACT OF IDENTIFIED ASSUMPTIONS IN SYSTEM DESIGN MODELS

4.1 Background and Related Work

As technology advances, safety-critical systems are playing increasingly more important roles in our everyday lives, such as medical systems, aircrafts, and nuclear power plants. Because of their critical significance, they have high dependability requirements for assuring the safety and correctness of the systems. Any failure of such systems could lead to death or serious injury to people and cause severe damage to equipment/property. Considering the fact that violating unspecified assumptions can cause system failures,it is essential to determine the impact of failure caused by violating unspecified assumptions on overall system to take preventive and corrective actions. In Chapter 3, we have presented a tool, the *UACFinder*, which applies data mining techniques to identify unspecified assumptions from system design models. However, the number of unspecified assumptions in a system can be large, and it is not always feasible, neither necessary, for domain experts to validate all of them at different development phases. Furthermore, for the unspecified assumptions, to prioritize which assumptions are the most safety-critical is also needed for effectively addressing potential safety issues caused by unspecified assumptions.

Among many tools and approaches of performing impact analysis on failures, Failure Mode and Effects Analysis(FMEA) is one of the best management tools to analyze the potential failure modes within a system under conditions of uncertainties. FMEA is a structured qualitative analysis of systems, subsystems, components, or functions that highlights potential failure modes, their causes, and the effects of failures on system operation. There have been several published studies demonstrating

the benefits of employing FMEA in other domains [68], [69], [70], [71], [72]. For example, Snooke N. et al. described how model-based simulation can be employed to automatically generate the system-level effects of all possible failures on systems within the aircraft systems [68]. The application of functional modeling to the automatically produce FMEA information for mechanical systems is described in [69]. In addition, Faida Mhenni1 et al. have proposed an approach to generate fault tree from SysML System Models and extract the needed information on the dysfunctional behaviors from functional SysML models to generate FMEA reports for improving safety [70]. Beyond applying FMEA in validation, Papadopoulos et al. have proposed a model-based automated synthesis of fault trees from Matlab-Simulink models [71], [72] to improve system safety.

Though the FMEA was originally developed outside of the healthcare domain, it is now being used in health care to assess the risk of failures and harms in medical processes and to identify the most important areas for process improvements [73], [45], [74], [75]. A variety of Institute for Healthcare Improvement programs in many hospitals, including Idealized Design of Medication Systems (IDMS), Patient Safety Collaborative, and Patient Safety Summit, have also started to use FMEA to analyze safety problems in their medical processes to improve patient care safety [76], [77], [78]. The implementation of the FMEA is now required by healthcare organizations accredited by the Joint Commission on Accreditation of Healthcare Organizations as part of their patient safety standards [79].

Though the benefits of employing FMEA in medical domains are enormous, how to identify all possible failure modes has not been addressed well in the literature. There are two main challenges to conduct the FMEA: 1) how to find all possible failure modes and their corresponding root causes. 2)how to assess the relative impact of different failure modes. One most common approach to finding possible failure modes

is: analysis group members perform brainstorming to identify possible failure modes based on their experiences and understanding of the medical processes at hand [80]. However, the method is time-consuming and the identified failure modes may not be exhaustive and may be too abstract to support detailed effects analysis of these failure modes. In this thesis, we present an approach, making use of the concept of FMEA, to systematically identifying possible failure modes related to assumptions in system design models and determine the priority of assumptions to facilitate impact analysis by domain experts. In addition, based on our literature review [81], [82], [83], [84] and discussion with medical professionals, we have concluded a few necessary steps in conducting the Failure Mode and Effects Analysis for medical cyber-physical systems, but specific details may vary based on standards of individual organizations:

1. Assemble a cross-functional team with knowledge in both medical and computer domains.

2. For a medical process, identify all possible failures that may happen. These are potential failure modes.

3. For each failure mode, identify all possible effects on end users or related systems.

4. Determine the severity of each effect ($\mathcal{S}$), with 1 most severe effect and 10 the least one.

5. For each failure mode, determine all the potential root causes.

6. For each cause, determine its potential occurrence rate ($\mathcal{O}$), with 1 extremely unlikely and 10 inevitable.

7. For each cause, determine its detection rate ($\mathcal{D}$), with 1 for easy detection and 10 almost impossible to detect.

8. Determine the risk priority number for each failure ($\mathcal{RPN}$).

9. Recommend actions to improve system safety.

In summary, we present an approach that makes use of the concept of Failure Mode and Effects Analysis (FMEA) to determine failure modes when unspecified assumptions are violated. By analyzing the effects of potential failure modes, we prioritize the assumptions to enable domain experts to analyze assumptions accordingly and develop appropriate action plans to ensure system safety in a timely manner. We assemble an analysis team including two medical doctors and take a blood gas examination statechart model as an example to illustrate how the proposed approach can facilitate domain experts to analyze the impact of failures caused by violating unspecified assumptions in practice.

4.2 The Approach of Applying FMEA for Facilitating the Analysis of Unspecified Assumptions

In the procedure of the FMEA described in Section 4.1, there are two important sequential steps, i.e., find possible failure modes and then identify their corresponding root causes. Given a system design model, we can apply to mine as many unspecified assumptions as possible. However, the number of mined unspecified assumptions may be too large to be validated efficiently. For example, how to make sure the most safety-critical unspecified assumptions can be validated first by domain experts. Therefore, prioritizing the mined unspecified assumptions from the perspective of system safety is highly demanded. In this section, we apply the FMEA to prioritize mined unspecified assumptions through the *UACFinder*, we revise the two steps as 1) for each assumption, we assume the assumption is violated and treat the violation as the root cause of a failure; and 2) based on the identified root cause, we find all possible failure modes. With the modification, the approach of using the FMEA to prioritize assumptions is depicted in Fig. 4.1.
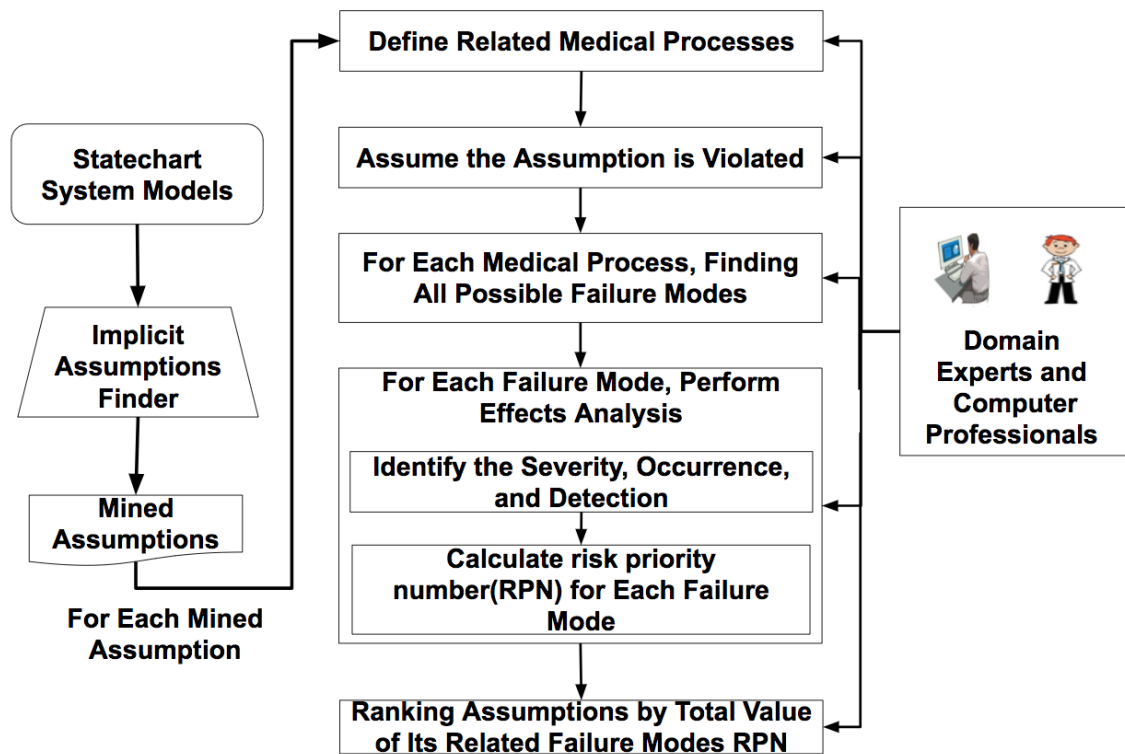
Figure 4.1. The Approach of Using Failure Mode and Effects Analysis to Analyze the Impacts of Assumptions

To illustrate how the proposed approach can help domain experts to analyze the impacts of violations of unspecified assumptions in system design models, we use a real medical system design model as an example to explain each procedure of the proposed approach. We take the blood gas to examine statechart model [51], shown in the Fig. 4.2, as a target medical system design mode. The following subsections give the details of each step in the approach.

4.2.1 Assemble Analysis Team. To make sure applying the FMEA for analyzing system design models, there must be a large engineering team that makes sure that the system is designed correctly, as well as a vital group with domain experts to validate the design and implementation [76], [77], [78]. To ensure analysis procedures are valid both from the perspectives of computer science and medical domain, we assemble an
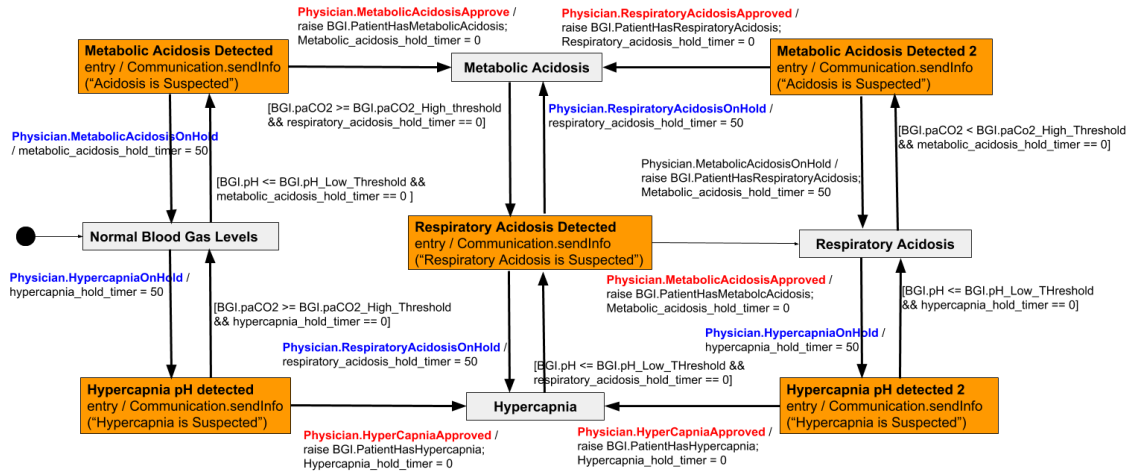
Figure 4.2. Blood Gas Examine Yakindu Statechart Model

analysis team, which includes two medical doctors and five computer professionals.

4.2.2 Find Possible Failure Modes Caused by Violations of Unspecified Assumptions.

To conduct effects analysis for a medical process, we need to identify the maximal number of failure modes, if not all the possible failure modes, in the medical process. A failure mode is anything that could go wrong during the completion of the medical process. However, how to identify all possible failure modes has not been addressed well in the literature. One most common approach is: analysis group members perform brainstorming to identify possible failure modes based on their experiences and understanding of the medical processes at hand [80]. However, the method is time-consuming, the identified failure modes may not be exhaustive and may be too abstract to support detailed effects analysis of these failure modes. Rather than using brainstorming approach, we propose a systematic way to identify possible failure modes from statechart models. The approach has two main steps:

1. For a unspecified assumption, search the given statechart models and find states and transitions that are related to the assumption. Failures are most likely to happen when the unspecified assumption is violated; and

2. Identify possible failure modes that may occur in these states and transitions by assuming the unspecified assumption is violated.

We have applied the *UACFinder* on the Blood Gas Examination statechart model shown in Fig. 4.2 and are able to find three types of unspecified assumptions in the statechart model:

- Assumptions that describe the expected environment of how to use constant variables. For example, the constant variable `BGI.paCO2_High_Threshold` is initialized to be 45 which is based on an assumption that the intended patients are adults, i.e., `A1` in Table 4.1;

- Assumptions that describe which set of actions should be executed together. For example, an approval event's trigger and its corresponding holder timer's reset are assumed to be executed together, i.e., `A4` in Table 4.1;

- Assumptions that describe how user interactions with the system should take place. For example, if a transition's guard contains an approval event from users, the transition's *from* states are assumed to send a notification to inform user for the following actions, i.e., `A3` in Table 4.1.

Given a statechart model and a set of unspecified assumptions found through the *UACFinder*, based on features of each assumption types, we define the following rules to find which parts of statechart models most likely have failure modes if some assumptions are violated. The rules are list as follows:

- **Rule 1.** If an assumption describes an expected environment about how to use specific constant variables, search all states and transitions that involve the constant variables;

Table 4.1. Unspecified Assumptions Identified in the Blood Gas Examination State-chart Model

ID	Assumption
A1	The constant variable $BGI.paCO2\_High\_Threshold$ is hard-coded to be 45 implicitly only for adult patients.
A2	The constant variable $BGI.pH\_Low\_Threshold$ is hard-coded to be 7.2 implicitly only for adult patients.
A3	When there is a need for medical physicians to perform actions to the system, such as medical physician approval actions which are highlighted with red color in the Fig. 4.2, the system model must send a notification to inform medical physicians which are highlighted with orange.
A4	When an approval event is triggered by medical physicians (approval events are highlighted as red in the Fig. 4.2), its corresponding holder timer should be reset to be 0.
A5	When an on hold event is triggered by medical physicians (hold events are highlighted as blue in the Fig. 4.2), its corresponding holder timer should be reset to be 50.

- **Rule 2.** If the assumption describes which set of actions should be executed together, search all states and transitions which execute all actions in a group;

- **Rule 3.** If the assumption describes how user interaction should be performed, search all pairs of states and transitions (S, T) where the guard of transition T involves user *events* and the S is transition T's *from* state and sends user notifications.

We use Example 1 to illustrate how to apply these three rules to find states and transitions that are related to a given assumption.

Example 1. Given the statechart model shown in Fig. 4.2 and assumptions A1, A3, and A4 in Table 4.1. As the assumption A1 describes that the constant variable BGI.paCO2_High_Threshold is assigned to be 45, we apply **Rule 1** and find three related transitions using the constant variable: transition T_1 from state "Metabolic Acidosis" to state "Respiratory Acidosis Detected", transition T_2 from state "Hyper-

capnia pH detected" to state "Normal Blood Gas Levels", and transition T_3 from state "Respiratory Acidosis" to state "Metabolic Acidosis Detected 2". Similarly, we apply **Rule 3** and **Rule 2** to find related states and transitions of assumption A3 and A4, respectively. For instance, one transition related to assumption A4 is transition T_4 from state "Metabolic Acidosis Detected" to state "Metabolic Acidosis". For the assumption A3, one pair of state and transition is (S_1, T_4), where S_1 is state "Metabolic Acidosis Detected".

With the defined three rules, a limited set of states and transitions where failure modes are most likely to happen are located. By narrowing down the scope where failure modes could happen, domain experts in the analysis group can quickly and accurately determine possible failure modes with the following steps.

1. Find execution paths which contain the identified states and transitions.

2. Assume the target assumption is violated, analyze the semantic meaning of the identified paths to determine the possible failure modes.

3. Provide a concrete scenario as an evidence for the failure modes.

To ensure the failure modes are correctly identified, domain experts must be involved in the analysis group. We use the assumption A1 given in the Table 4.1 as an example to illustrate how to perform the steps to identify failure modes.

1. States and transitions that are related to the assumption A1 are extracted. For instance, 1) the transition from state "Metabolic Acidosis" to state "Respiratory Acidosis Detected", 2) the transition from state "Hypercapnia pH detected" to state "Normal Blood Gas Levels", and 3) the transition from state "Respiratory Acidosis" to state "Metabolic Acidosis Detected 2" are extracted.

2. Extract the execution paths which contain the identified transitions, such as the execution path `P1`: "Normal Blood Gas Levels" → "Metabolic Acidosis" → "Respiratory Acidosis Detected" → "Respiratory Acidosis".

3. Assume `A1` is violated and identify the possible failure mode in the execution of `P1`. One failure mode is that the transition from state "Metabolic Acidosis" to state "Respiratory Acidosis Detected" should take place but actually does not.

4. Give concrete examples to show when the failure mode will happen. For instance, an evident example is that when a patient is an infant, BGI.paCO2_High _Threshold=45 is no longer accurate for calculating the patient's blood gas level since the value of BGI.paCO2_High_Threshold for infant patients should be 42.

Based on the procedure described above, we can systematically identify possible failure modes from a given statechart model more rapidly and accurately comparing to only relying on domain experts' brainstorming. Table 4.2 lists the possible failure modes identified from the blood gas examination statechart model with violations of assumptions in Table 4.1.

4.2.3 Conduct Effects Analysis On Identified Failure Modes. For each failure mode, analysis group members work together to review its impacts on end users or systems. Each identified failure mode is scored according to its potential severity and impact [75], [73]. The main three criteria to quantify the effects of a failure mode [75], [73]: 1) the severity of the effect on end users or systems, 2) how frequently the failure mode is likely to occur and 3) how easily the failure mode can be detected. Participants must set and agree on a ranking between 1 and 10 (1 = low, 10 = high) for the severity, occurrence, and difficulty level of detecting each of the failure modes, respectively.

However for some specific applications, not every failure mode will result a catastrophe, but the impacts of a failure may manifest as procedural delays, system breakdowns, reductions in throughput, and other factors affecting the system's quality. Therefore, for different applications, analysis group members should develop customized scoring rules for severity. For instance, severity scoring rules for use in the medical domain have been introduced and developed over the last 30 years [85]. They allow an assessment of the severity of the disease and provide an estimate of in-hospital mortality. A weighting factor is applied to each variable, and the sum of the weighted individual scores produces the severity score [86].

To evaluate the effects of the identified failure modes in the Blood Gas Examination Statechart Model. The medical physicians in the analysis group first identify the specific patient data and corresponding weight factors to determine severity scoring rules. Table 4.3 lists the available patient's data for use in scoring rules. Based on the guideline that how to interpret a blood gas [87] and the importance of patient data in the Table 4.3, a simple scoring table of severity is constructed by medical physicians in the analysis group, which is shown in Table 4.4

In determining the probability of a failure mode occurring and detection, reference should be made to the data from previous adverse events and to the personal experience of domain experts. The higher the ranking (scale of 1-10), the more likely it is for the failure mode to occur and detection. Based on the experience of the analysis group, we provide a simple occurrence scoring table and a detection scoring table, as shown in Table 4.5 and Table 4.6, respectively.

The next step is to define a proper formula to achieve risk priority number based on the value of severity, occurrence, and detection. The risk priority number ($\mathcal{RPN}$), also referred to as the critical index, is a quantitative measure used to evaluate and assess a failure mode [88]. In our proposed approach, the $\mathcal{RPN}$ is

derived from the product of severity ($\mathcal{S}$), the probability of occurrence ($\mathcal{O}$), detectability ($\mathcal{D}$), and the weight factor of severity ($\mathcal{W}$), as $\mathcal{RPN} = \mathcal{S} \times \mathcal{O} \times \mathcal{D} \times \mathcal{W}$. The $\mathcal{RPN}$ are then ranked to allow prioritization of the failure modes and to highlight the failure modes that exceed acceptable limits and should, therefore, be targeted for change. The highest $\mathcal{RPN}$ should be prioritized for corrective action. Regardless of the $\mathcal{RPN}$ attention should always focus on any domain where the severity ranking is high. Those steps with low $\mathcal{RPN}$ (and therefore of low impact in the spectrum of failure) are unlikely to affect the process and should therefore not be prioritized as part of this process. For an assumption, after we have finished the calculation of $\mathcal{RPN}$ for each of failure mode caused by violations of the assumptions, we just simply add $\mathcal{RPN}$ of each failure mode to get final $\mathcal{RPN}$ of each assumption. Table 4.7 indicates the result of effects analysis of failure modes caused by violation of the unspecified assumptions in Table 4.1.

Once the major failure modes and the corresponding effect analysis have been identified, strategies should be developed and implemented to prevent the subsequent occurrence.

4.2.4 Discussion of Using the FMEA. Although the application of the FMEA in safety-critical systems is promising, the FMEA has only been introduced into this area relatively recently, and many improvements are still needed. For example, if we use numeric product to calculate $\mathcal{RPN}$, a failure mode with a high severity but low probability of detection can have the same score as a failure mode with a low severity but high probability of detection. However, failure modes with high severity should be targeted for analysis with high priority, and an action plan should promptly be put in place to address the risk. Although the default $\mathcal{RPN}$ calculation may work in other industries that use the FMEA, a failure mode with a high severity but the low probability of detection is unacceptable in medical systems, because it may lead

to patient morbidity or even mortality. Therefore, for healthcare failure modes with a high severity, scores should probably be analyzed regardless of the probability of occurrence and detection.

In addition, the overall reliability of the FMEA also has been questioned [89]. In a comparative study performed by two groups working on the same topic but in different hospitals, investigators found only a 17% overlap of failure modes that were identified [90]. The two groups also had considerably different $\mathcal{RPN}$s for their common failure modes, resulting in the different prioritization of the failure modes. These data suggest that developing generic standards to perform the FMEA with different expertise and experience, is most likely to help the FMEA process succeed.

4.3 Summary

In this Chapter, we present an approach that makes use of the concept of Failure Mode and Effects Analysis (FMEA) to facilitate domain experts to perform impact analysis on possible failures caused by violating unspecified assumptions. By analyzing the effects of potential failure modes, we prioritize the assumptions to enable domain experts to analyze assumptions accordingly and develop appropriate action plans to ensure system safety in a timely manner. Currently, our approach is semi-automatic since some steps in the approach to are still required manual work by domain experts, such as identifying failure modes. Therefore, how to integrate appropriate domain knowledge into our approach to minimize manually efforts will be investigate in our future work.

Table 4.2. The Identified Failure Modes from The Blood Gas Examine Model with Violation of Assumptions in Table 4.1

FM ID	A ID	Failure Mode Description
FM1	A1	When a patient is an infant, $BGI.paCO2\_High\_Threshold = 45$ is no longer accurate for calculating the patent's blood gas level, consequently, some transitions in the blood gas examine statechart should not but will take place. These transitions include: 1) the transition from state "Metabolic Acidosis" to state "Respiratory Acidosis Detected", 2) the transition from state "Hypercapnia pH detected" to state "Normal Blood Gas Levels", and 3) the transition from state "Respiratory Acidosis" to state "Metabolic Acidosis Detected 2".
FM2	A2	When a patient is an infant, $BGI.pH\_Low\_Threshold = 7.2$ is no longer accurate for calculating the patient's blood pH, and consequently some transitions in the blood gas examine statechart should not but will take place. These transitions include: 1) the transition from state "Normal Blood Gas Levels" to state "Metabolic Acidosis Detected", 2) the transition from state "Hypercapnia" to state "Respiratory Acidosis Detected", and 3) the transition from state "Hypercapnia pH detected 2" to state "Respiratory Acidosis".
FM3	A3	The system fails to send notifications to inform users to take the corresponding user-interactions on time. Alternatively, we can say that in the blood gas examine statechart, transitions containing user-interaction events: $Physician.MetabolicAcidosisApproved$, $Physician.HypercapniaApproved$, and $Physician.RespiratoryAcidosisApproved$, should take place but not.
FM4	A4	The system fails to reset $respiratory\_acidosis\_hold_timer = 0$, $hypercapnia\_hold\_timer = 0$, and $metabolic\_acidosis\_hold_timer = 0$. Alternatively, we can say that in the blood gas examine statechart, transitions containing $respiratory\_acidosis\_hold_timer == 0$, $hypercapnia\_hold\_timer == 0$, and $metabolic\_acidosis\_hold_timer == 0$, should take place but not.
FM5	A5	The system fails to reset $respiratory\_acidosis_hold\_timer = 50$, $hypercapnia\_hold\_timer = 50$, and $metabolic\_acidosis\_hold\_timer = 50$, which may case the corresponding notice can be send out every 50 seconds.

Table 4.3. Patient Data for Determining Scoring Rules of Severity within Blood Gas Examine Process

Weight(1-10)($\mathcal{W}$)	Patient's Variables
10	$paCO_2$
10	$paCO_2$ Threshold
10	pH
10	pH Threshold
8	Age
8	$Temperature$

Table 4.4. Severity Scoring Table for Blood Gas Examine Process

Severity($\mathcal{S}$)	
10	Terminal injury or death
7	Permanent lessening of body function, surgical intervention required, disfigurement
5	Temporary patient harm
3	May affect patients
1	No effect

Table 4.5. Occurrence Scoring Table for Blood Gas Examine Process

Occurrence($\mathcal{O}$)	
10	Documented, almost certain; or happens 90-100% of the time
7	Documented and frequent; or happens 70-80% of the time
5	Documented but less frequent; or happens 40 - 60% of the time
3	Possible, but no known data; or happens 10 - 30 % of the time
1	No known occurrence; or happens < 10 % of the time

Table 4.6. Detection Scoring Table for Blood Gas Examine Process

Detection($\mathcal{D}$)	
10	Detection not possible at any point; or we can never catch it!
7	Low likelihood of detection; or we can catch it 3 out of 10 times
5	Moderate likelihood of detection; or we can catch it 5 out of 10 times
3	Error almost always detected, or we can catch it 9 out of 10 times
1	Error almost always detected, or we can catch it 9 out of 10 times

Table 4.7. The Effects Analysis of Unspecified Assumptions Including Failure Modes, Effect Description, Severity, Occurrence, Detection, Weight, and Risk Priority Number

A ID	FM ID	Effects	$\mathcal{S}$	$\mathcal{O}$	$\mathcal{D}$	$\mathcal{W}$	$\mathcal{RPN}$
A1	FM1	Medical physicians can not find out the $paCO2$ value of an infant patient is greater than the threshold in time, which could cause the infant patient can not get correct treatment on time and make the patient in danger.	10	10	10	10	10000
A2	FM2	Medical physicians may find out the pH value of an infant patient is lower than the threshold by mistake, which could cause the infant patient can not get correct treatment on time and make the patient in danger.	10	10	10	10	1000
A3	FM3	Medical physicians can not get notification about patient status in time, which could lead medical physicians give incorrect treatment to patients or delay patient treatment. The misleading or delay could make patients in danger.	10	10	3	1	300
A4	FM4	Medical physicians can not get accurate patient status in time, which could lead medical physicians give incorrect treatment to patients, which makes patients in danger.	7	7	1	1	49
A5	FM5	Medical physicians can not get the accurate value of patient holding time, which could lead medical physicians give incorrect treatment to patients or delay patient treatment. The misleading or delay could make patients in danger.	7	7	1	1	49

CHAPTER 5

MODEL AND INTEGRATE ASSUMPTIONS INTO SYSTEM DESIGN MODELS

5.1 Background and Related Work

In the previous chapters, we have presented techniques and tools to identify unspecified assumptions in system design models. In addition, we have also presented the approaches for facilitating domain experts to perform impact analysis on possible failures caused by violating identified unspecified assumptions. All these approaches described are bottom-up solutions in which we are working on the recovery from given system design models. However, if assumptions are given before the design phase, a challenging question is how we can ensure the safety of system design models under these assumptions. An intuitive approach is to explicitly specify assumptions in human language, so that the system's safety associated with different assumptions can be validated by domain experts.

Based on the intuitive approach, An amount of significant work has been done to specify assumptions in system design models in different domains [15], [8], [9]. For example, in the guidance "Applying Human Factors and Usability Engineering to Medical Devices" released by U.S. Food and Drug Administration [91], it recommends developers to evaluate and understand relevant assumptions of all intended use environments and describe them for the purpose of safety evaluation and design [91]. In addition, the Medical Device Plug-and-Play Interoperability (MDPnP)program [92], this program has been leading the development of the Integrated Clinical Environment (ICE) standard and gap analysis on the ability of the IEEE 11073 family of standards [93] to meet the clinical use cases described in the ICE standard. Currently, much work about assumptions for MDPnP focuses on establishing dynamic

connectivity of devices with different data format assumptions [94], synchronization among devices with diverse clock assumptions [95], and ensuring fair access to a communication medium [96]. All these work have shown the promising approaches to specify assumptions with natural languages, but how to formally model assumptions and integrate assumption models into system models from the system development perspective has not been addressed yet.

In the previous chapters, we have developed the *UACFinder* to identify assumptions about *constant variables*, *frequently read/updated variables*, and *frequently executed actions*. Based on the features of these assumptions, we proposed the following methodology to model assumptions. For the assumptions related to *constant variables* and *frequently read/updated variables* we model them with numeric parameters, and mathematical functions to describe the constraints between such parametric values and possibly specifications of dynamic behavior are given. For the assumptions related to *frequently executed actions* the resulted behaviors can be modeled by means of formulas in temporal logic.

Based on the proposed methodology, we present an approach to model and integrate assumptions about *constant variables* into system design models. As we mentioned in Chapter 2, many assumptions about the physical environment can be treated as assumptions about *constant variables*. In this section, we first show an FDA medical device recall example, and then an analysis shows how assumptions associated with *constant variables* of environment can cause fatal M-CPS failures.

Recall Case 2 (Dräger Medical, Evita V500 and Babylog VN500 Ventilators – Faulty Batteries, July 13, 2015 [28])**.** FDA has identified this recall as a Class I recall, the most serious type of recall. According to the recall report [28], the battery capacity of optional PS500 Power Supply Unit of the Infinity ACS Workstation Critical Care (Evita Infinity V500) did not last as long as expected. The batteries installed in the

PS500 depleted much earlier than expected although the battery indicator showed a sufficiently charged battery. Even when the battery depleted totally, the power fail alarm was not generated. If the ventilator shuts down without alarm, a patient may not receive necessary oxygen. This could cause patient injury or death.

In the FDA recalled ventilator, there are three major components in the system: *Controller*, *Alarm* and *Battery* [97]. The *Controller* calculates remaining time that the *Battery* can supply. If the remaining time is within the range of 30 to 35 minutes, the *Controller* should send an event to the *Alarm* component to trigger an alarm for medical staffs. One unspecified assumption is that ventilators are installed in temperature controlled areas, such as hospital rooms, where room temperature are maintained at normal room temperature. In this environment, the capacity the battery can supply is a constant. However, if a hospital room's temperature is kept to be the assumed operating temperature, the capacity of the battery will be not reduced. This unanticipated change of battery capacity will cause *Controller* to miscalculate the remaining time and hence fail to send the alarm event before the ventilator is out of power. In this recall, the assumption is associated with the room temperature, under which the ventilator operates. We can use a parameter to represent such assumption. There are many conceptually similar assumptions can be represented by parameters, such as clock skews rate, network delays, and task execution times. The challenging question is how to ensure these assumptions are correctly reflected in the system design models and they do not cause violations of system safety.

For safety-critical cyber-physical systems, validations by domain experts alone are not adequate for ensuring systems safety. Formal verification of the systems are essential. There are many formal verification techniques exist in the literature, such as model checking [98], theorem proving [99], and knowledge based verification [100]. Formal model based approaches have successfully provided a unified basis for formal

analysis to achieve the expected level of safety guarantees. It has been applied in many safety-critical areas such as automotive systems [101] and avionics systems [102]. Hence, to ensure the safety of safety-critical system under different assumptions, the desired approach not only can model and integrate assumptions into system design models, but also the integrated system design models can be easily transferred to formal models for formal verification to ensure the safety.

The design and development of safety-critical cyber-physical systems requires knowledge from different domain experts. For instance, the design and development of a medical cyber-physical system will need the joint efforts from engineers, computer scientists, and medical professionals. Engineers are more familiar with mathematical structures and operations, whereas medical professionals are more used to statecharts as disease and treatment models since they have high resemblance to statecharts. Hence, to explicitly integrate assumptions into system design models, our strategy is to define a mathematical model and composition rules for engineers to explicitly and accurately specify design assumptions for their physical devices. The mathematical assumption model is then automatically transformed into a specific system design model and integrated with system models so that the integrated models can be validated by domain experts and system safety properties can be formally verified with existing model verification tools.

Fig. 5.1 depicts the high level view of our proposed approach for modeling and integrating assumptions into system design models for validation and formal verification. In this approach, we use statechart model as the specific system design model to illustrate the procedures of the proposed approach. Currently, we focus on applying the proposed approach for modeling and integrating assumptions associated with *constant variables* into the system design models. In this chapter, we call the assumptions associated with *constant variables* as *environment assumptions*.
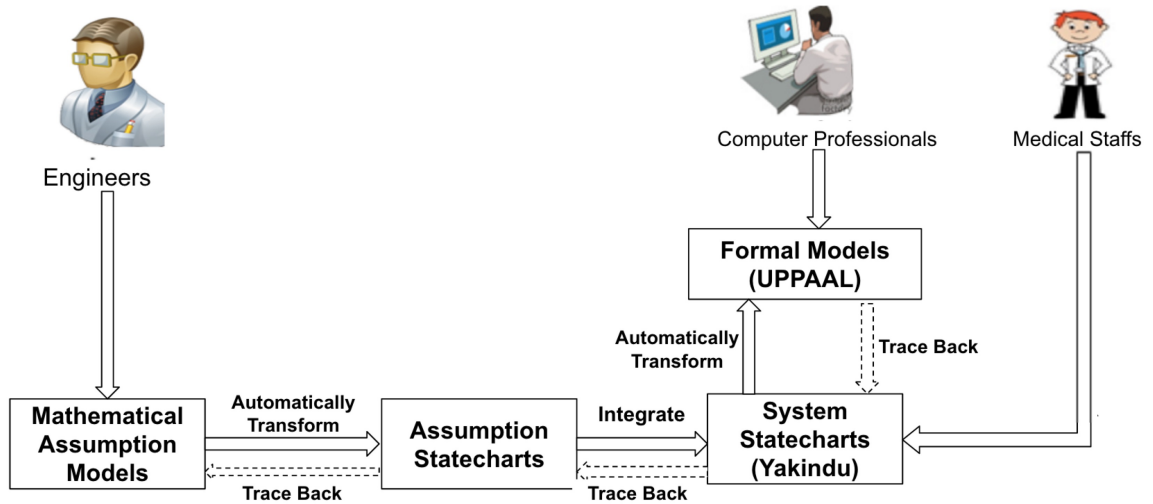
Figure 5.1. The Architecture of Modeling and Integrating Assumptions into System Design Models for Validation and Formal Verification

5.2 Modeling Environment Assumptions

The impact of environment change on M-CPS' behaviors is often due to the fact that environment change can cause M-CPS system parameters to change, as in the FDA recall presented above. An environment assumption describes how a relation between a system parameter and a set of environment parameters. Hence, to bring to light environment assumptions in systems, we need to explicitly specify the relationships between system parameters and environment parameters. In this section, we define a mathematical structure to specify environment assumptions.

5.2.1 Mathematical Structure of Assumptions. An environment assumption is represented as $A(s, E, \mathcal{F})$, where s is a system parameter, $E = \{e_1, \cdots, e_n\}$ indicates an set of environment parameters. $\mathcal{F}$ is a function that defines a relation that associates environment parameters in E to the system parameter s, $\mathcal{F} : E \to s$. We use the following simple example to illustrate how to use this mathematical structure to represent environment assumptions.

Example 2. Consider the following two scenarios of battery behaviors in the FDA

Recall 2 example. The system parameter C indicates the percentage of capacity that battery can indeed provide in different environment [103]. And the following two scenarios describe the relationships between the system parameter and environment parameters.

- **S1**: *when the temperature T is within the range $[-10, 15)$, the percentage is $C = (1 - 2 \times (25 - T)/100)$.*

- **S2**: *when the temperature T is within the range $[15, 35]$, the percentage $C = 1$.*

The two scenarios **S1** and **S2** in the Example 2 describes an environment assumption between the system parameter C and environment parameter-temperature T. The assumption can be represented as $A_1(C, \{T\}, \mathcal{F}_1)$, where

$$C = \mathcal{F}_1(T) = \begin{cases} 1, & \text{if } 15 \leq T \leq 35 \\ 1 - 2 \times (25 - T)/100, & \text{if } -10 \leq T < 15 \end{cases}$$

Through the example above, we illustrate how to use the proposed mathematical structure to model environment assumptions without ambiguity. In the two scenarios **S1** and **S2**, we notice the system parameter C is impacted by only one environment parameter-temperature T. However, in a cyber-physical system, a system parameter is often impacted by multiple environment parameters. For instance, both temperature and humidity can impact battery capacity in the the FDA Recall 2. The following two scenarios **S3** and **S4** indicate the percentage of capacity C that battery can indeed provide can also be affected by humidity [104].

- **S3**: when the relative humidity H is in range $[10\% RH, 30\% RH]$, the battery capacity reduces by 10%, i.e., $C = 0.9$.

- **S4**: when the relative humidity H is in range $[40\%RH, 60\%RH]$, the battery capacity does not change, i.e., $C = 1$.

These two scenarios **S3** and **S4** describe another environment assumption about the impact of humidity on battery capacity. We present the mathematical structure of this assumption as $A_2(C, \{H\}, \mathcal{F}_2)$, where

$$C = \mathcal{F}_2(H) = \begin{cases} 0.9, & \text{if } 0.1 \leq H \leq 0.3 \\ 1, & \text{if } 0.4 \leq H \leq 0.6 \end{cases}$$

These two environment assumptions A_1 and A_2 indicate that a system parameter can be impacted by multiple environment parameters at the same time, and we define that A_1 and A_2 are interfered by each other.

5.2.2 Assumption Composition. Given two assumptions $A_i(s_i, E_i, \mathcal{F}_i)$ and $A_j(s_j, E_j, \mathcal{F}_j)$, if $s_i = s_j$, we call these two assumptions *interfering* assumptions. For *interfering* assumptions, their original relation functions that define the relationships between a system parameter and environment parameters may need modification. Domain experts might need to compose the *interfering* assumptions to be a new assumption with re-defining the relation function $\mathcal{F}$ based on their domain knowledge. We define the composition operation ($\oplus$) to compose two *interfering* assumptions: $A(s, E, \mathcal{F}) = A_i(s, E_i, \mathcal{F}_i) \oplus A_j(s, E_j, \mathcal{F}_j)$, where $E = E_i \cup E_j$, and $Dom(\mathcal{F}) = E$

Noting that defining a new relation function $\mathcal{F}$ to replace $\mathcal{F}_1$ and $\mathcal{F}_2$ is performed by domain experts. For instance, based on the domain knowledge described in [28], [104], [103], we use a simple piecewise function $\mathcal{F}_3(T, H) = min(\mathcal{F}_1(T), \mathcal{F}_2(H))$ to indicate the relationship between the system parameter C and both environment parameters T and H in the Example 2. With the new relation function $\mathcal{F}_3$, we can composite A_1 and A_2 to be a new assumption $A_3(C, E_3, \mathcal{F}_3) = A_1(C, \{T\}, \mathcal{F}_1) \oplus$

$A_2(C, \{H\}, \mathcal{F}_2)$, where $E_3 = E_1 \cup E_2 = \{T, H\}$, and

$$C = \mathcal{F}_3(T, H) = \begin{cases} 0.9, & \text{if } 5 \leq T \leq 35 \wedge 0.1 \leq H \leq 0.3 \\[2mm] 1, & \text{if } 5 \leq T \leq 35 \wedge 0.4 \leq H \leq 0.6 \\[2mm] 1 - \frac{2(25-T)}{100}, & \text{if } -10 \leq T < 15 \wedge 0.1 \leq H \leq 0.3 \\[2mm] 1 - \frac{2(25-T)}{100}), & \text{if } -10 \leq T < 15 \wedge 0.4 \leq H \leq 0.6 \end{cases}$$

During the development of a cyber-physical system, there will be many environment assumptions made by domain experts. With the proposed mathematical structure to mode assumptions, we can easily filter the *interfering* assumptions out. Then by preforming the composition operation iterative, we can avoid interference among different assumptions on the same system parameter, which will help to reduce the workload for validation and formal verification.

5.3 Integrating Assumption Models with System Model

To validate and verify assumptions, we can not just validate and verify the assumptions themselves. We need to validate and verify the system with these assumptions to ensure the safety. Therefore, we need to integrate assumption models with system models to enable the validation and verification. In M-CPS domain, it is important that engineers and medical staffs can understand M-CPS models easily and validate them through user-friendly simulation. Noting that statechart has high remembrance to disease and treatment models, can be easily understood by field professionals, is executable, and can be indirectly verified, we hence transform the mathematical model of *environment assumptions* to statecharts. We choose Yakindu statechart tool. Yakindu is an open-source tool kit based on the concept of statecharts. It has a well-designed user interface, provides simulation and code generation functionalities, and hence enables rapid prototyping and validation with field profes-

sionals.

5.3.1 Transforming Assumption Models to Statecharts.

The interference among multiple assumptions on the same system parameter increase the difficulty of transforming mathematical assumption models into statecharts. We first perform composition operation to compose assumptions and remove their interference. For the transformation purpose, we can then assume all assumptions $A \in \mathcal{A}$ are non-*interfering*. For each non-*interfering* assumption $A(s, E, \mathcal{F}) \in \mathcal{A}$, we use an independent sub-statechart in an orthogonal state to represent the assumption. If $\mathcal{F}$ is piecewise function, for each piece $\mathcal{F}_i$ in the $\mathcal{F}$, we create a state $\mathtt{S_d}$ with entry action $s = \mathcal{F}_i$ and add a transition from the initial state to state $\mathtt{S_d}$ with guard $Dom(\mathcal{F}_i)$ to represent the system parameter's corresponding change under the environment condition. If $\mathcal{F}$ is not a piecewise function, treating $\mathcal{F}$ as a whole piece and follow the same steps described above. For all states in the sub-statechart except the initial state, we add transitions back to the initial state with guard $\mathtt{true}$ to enable the statechart capturing environment conditions. Algorithm 8 depicts the transform procedure. Example 3 illustrates how we apply Algorithm 8 to transform A_3 to a statechart.

Algorithm 8 TRANSFORM ASSUMPTIONS TO STATECHARTS

Input: A system's assumption set $\mathcal{A}$.

1: Create a statechart
2: Add a orthogonal state named $\mathtt{Assumptions}$ to the statechart
3: **for** each $A_i(s_i, E_i, \mathcal{F}_i) \in \mathcal{A}$ **do**
4:     Add a sub-statechart $\mathtt{st_i}$ to the orthogonal state–$\mathtt{Assumptions}$
5:     Add the initial state $\mathtt{InitState}$ to $\mathtt{st_i}$
6:     **for** each piece $\mathcal{F}_{i_j}$ of $\mathcal{F}_i$ **do**
7:         Add a state $\mathtt{sd_j}$ to $\mathtt{st_i}$ with entry action $s_i = \mathcal{F}_{i_j}$
8:         Add a transition $\mathtt{T_j}$ from $\mathtt{InitState}$ to $\mathtt{sd_j}$ with guard $Dom(\mathcal{F}_{i_j})$
9:         Add a transition from $\mathtt{sd_j}$ to $\mathtt{InitState}$ with guard $\neg Dom(\mathcal{F}_{i_j})$
10:     **end for**
11: **end for**

Example 3. *To transform A_3 in Example 2, we create the statechart* $\mathtt{Environment}$

that contains an orthogonal state `Assumptions`. *To represent* A_3, *we add a sub-statechart* `batteryassumptions` *containing an initial state* `Init_State` *to the orthogonal state* `Assumptions`. *For* $r_{13} \in R_3$, *we create state* $\mathbf{r_{13}}$ *with entry action* $c = 0.9$, *add a transition from state* `Init_State` *to* $\mathbf{r_{13}}$ *with guard* $T >= 15 \;\&\&\; T <= 35 \;\&\&\; H >= 0.1 \;\&\&\; H <= 0.3$, *and set the priority of the added transition as* 1. *According to the Line 9 in Algorithm 8, we also add a transition from state* $\mathbf{r_{13}}$ *back to* `Init_State` *with guard* `true`. *Similar procedures can be taken for the rest dependencies in* A_3.
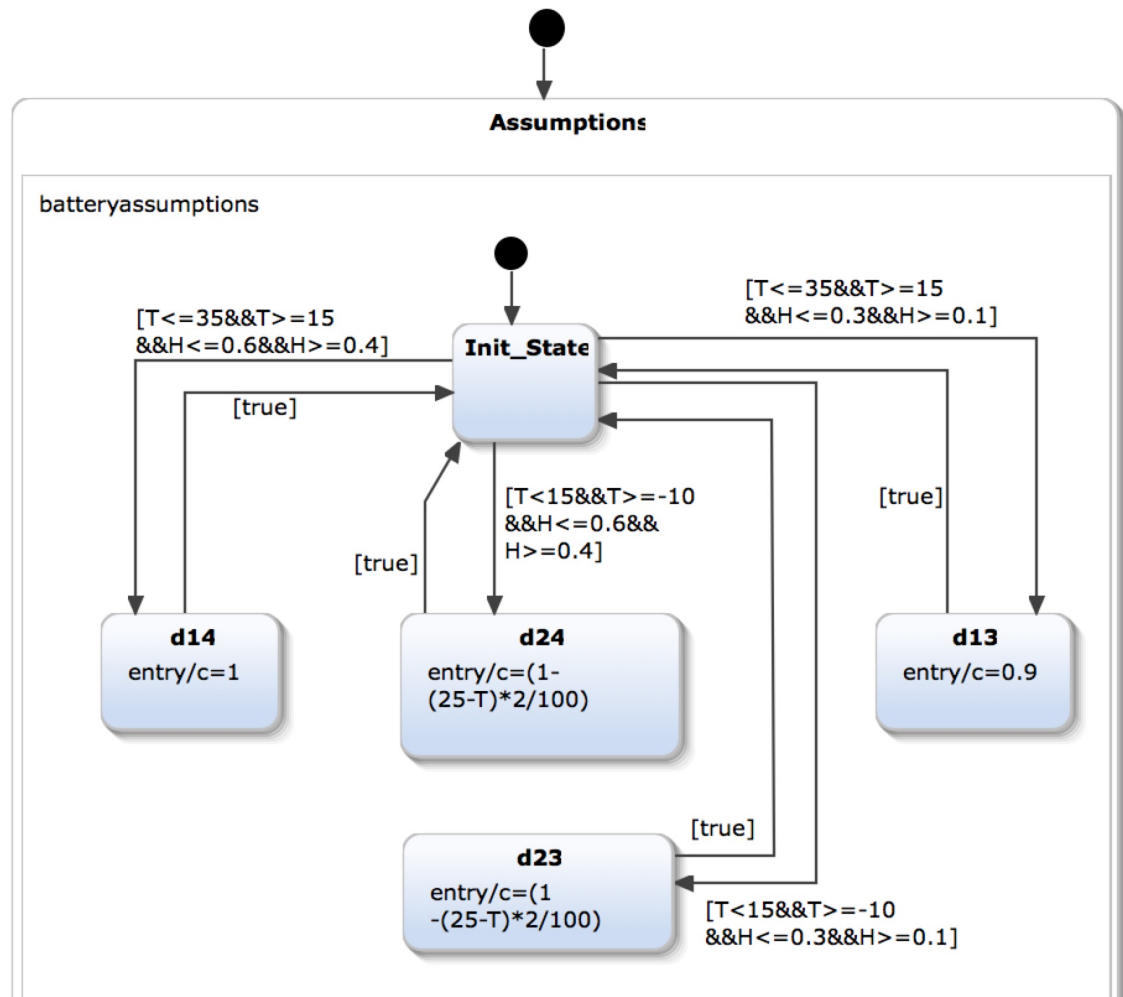

Figure 5.2. Assumption Statechart

5.3.2 Integrating Assumption Statecharts with System Model. To integrate assumption statecharts with system model, we model the interactions between assumption statechart models and system statechart models with following rules:

- **Integration Rule 1**. For each system parameter s, declare an event e_s to implement the interaction.

- **Integration Rule 2**. For each state in the assumption statechart model, if it changes the value of a system parameter s, raise the event e_s in the state's entry action.

- **Integration Rule 3**. For the system statecharts, modify it by the followed rules.

    - **Integration Rule 3.1**. For each transition $T(G, A)$, if its guard G or action A involves a system parameter s, $G = G$ && e_s.

    - **Integration Rule 3.2**. For each state, if its action involves a system parameter s, replace the guard of all its incoming transitions $\{T_i(G_i, A_i)\}$ by $G_i = G_i$ && e_s.

We integrate the assumption statechart shown in Fig. 5.2 with the medical ventilator statechart model in Fig. 5.3. and Fig. 5.4 shows the integrated system statechart model. In particular, based on **Integration Rule 1**, we declare an event upC for the battery capacity c. According to **Integration Rule 2**, we raise the event upC in the entry action of all states in the sub-statechart batteryassumptions except state Init_State. In the ventilator model, there are two transitions involving the battery capacity c: transition $T_1(G_1, A_1)$ from state Monitor to state OutPower and the self-loop transition $T_2(G_2, A_2)$ of state Monitor, where $G_1 = [c <= 0]$ and $G_2 = [c > 0]$. Based on **Integration Rule 3.1**, the two transitions' guards are set as

$G_1 = [c <= 0 \,\&\&\, \texttt{upC}]$ and $G_2 = [c > 0 \,\&\&\, \texttt{upC}]$. The only one state involving battery capacity c in the ventilator model is $\texttt{Monitor}$, which has three incoming transitions: transition $T_2(G_2, A_2)$, transition $T_3(G_3, A_3)$ from state $\texttt{Battery\_Control}$, and transition $T_4(G_4, A_4)$ from state $\texttt{Power\_Low}$, where $G_3 = [\texttt{true}]$ and $G_4 = [\texttt{true}]$. The guard of transition T_2 has been updated, hence we just change guards of transition T_3 and T_4 as $G_3 = [\texttt{upC}]$ and $G_4 = [\texttt{upC}]$ by applying **Integration Rule 3.2**.

5.4 Medical Ventilator Case Study

In this section, we perform a case study on the recalled medical ventilator scenario given in the beginning of the Chapter. We demonstrate the differences of two models shown in Fig. 5.3 and Fig. 5.4.

5.4.1 Validation of System Models. We define a safety criteria as: the ventilator must raise a low power alarm before shutdown. In the system statechart models shown in Fig. 5.3 and Fig. 5.4, the safety criteria is expressed as: the state $\texttt{B\_Low\_Alarm}$ must be activated before state $\texttt{ShutDown}$'s activation.

We take the scenario **S2** and **S4** in Example 2, i.e., $-10 \le T < 15 \wedge 0.4 \le H \le 0.6$, as an example to show the validation of medical ventilator models with/without physical environment assumptions. The simulation results show that: (1) the safety criteria fails in the model without environment assumptions as shown in Fig. 5.5; and (2) the criteria is satisfied in the model with assumptions as shown in Fig. 5.6.

5.4.2 Formal Verification of System Models.

For safety critical Medical Cyber-Physical Systems, validation is not adequate for guaranteeing their correctness and safety, and formal verification is required. In this paper, we use the Y2U[1] tool [46] to transform system models represented by

---

[1]The Y2U tool is available at www.cs.iit.edu/~code/software/Y2U/index.html.

Yakindu statecharts to UPPAAL timed automata for formal verification.

The UPPAAL models transformed from Yakindu statechart models by the Y2U tool are shown in Fig. 5.7 and Fig. 5.8. The safety criteria can be checked by the following formula in UPPAAL: $A[\,]$ *Battery.ShutDown imply PLF == true*. The verification results also show that: (1) the criteria fails in the model without considering environment assumptions (Fig. 5.7); and (2) the criteria is satisfied in the model with assumptions (Fig. 5.8).

5.5 Summary

In this Chapter, we have presented an approach to model and integrate assumptions associated with *constant variables* (unspecified assumptions) into system design models for validation and formal verification. We develop a mathematical assumption model and composition rules that allow domain experts to explicitly and precisely model environment assumptions. Algorithms are then developed to integrate mathematical assumption models with the system model so that the safety of the system can be not only validated by both medical and engineering professionals but also formally verified by existing formal verification tools. We also use a FDA recalled medical ventilator scenario as a case study to show how the mathematical assumption model and its integration in system design may improve the safety of the ventilator. We believe this approach can be used in the design of other safety-critical cyber-physical systems.
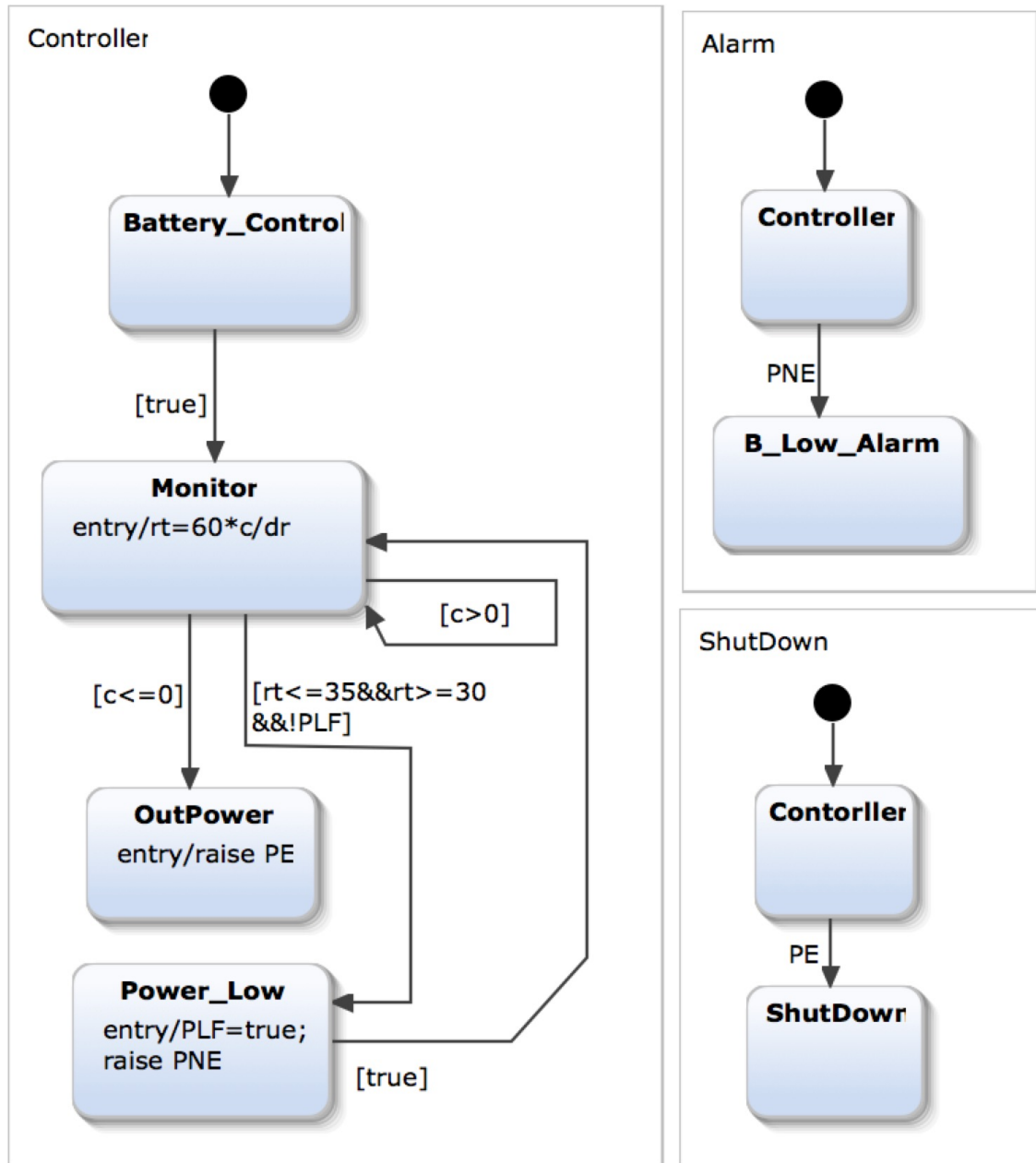
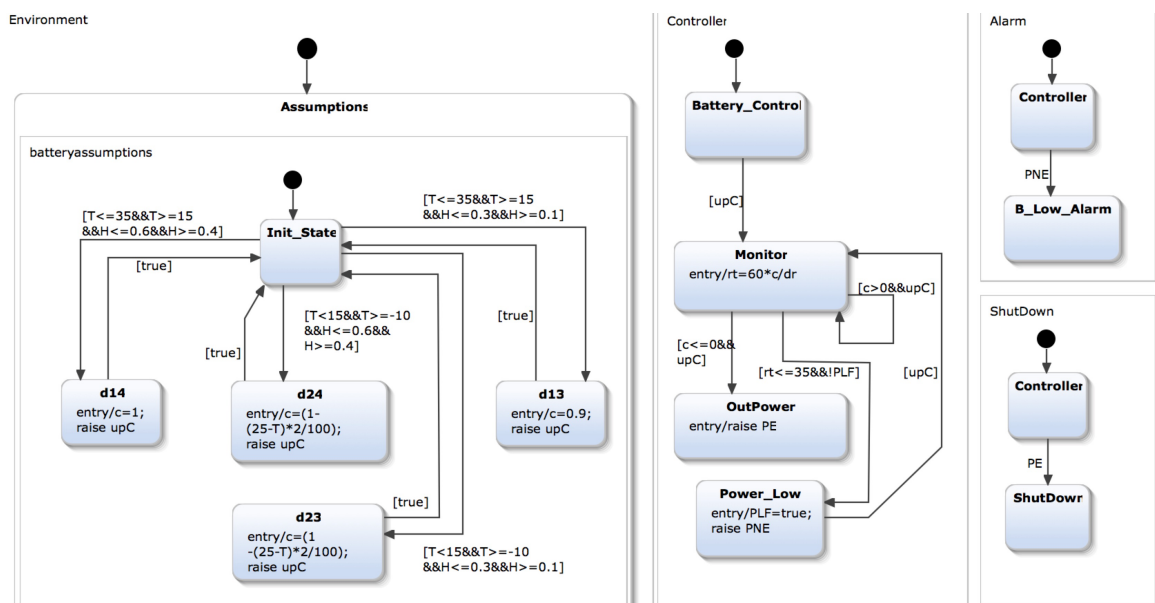Figure 5.3. Medical Ventilator Model without Environment Assumptions

Figure 5.4. Medical Ventilator Model with Environment Assumptions
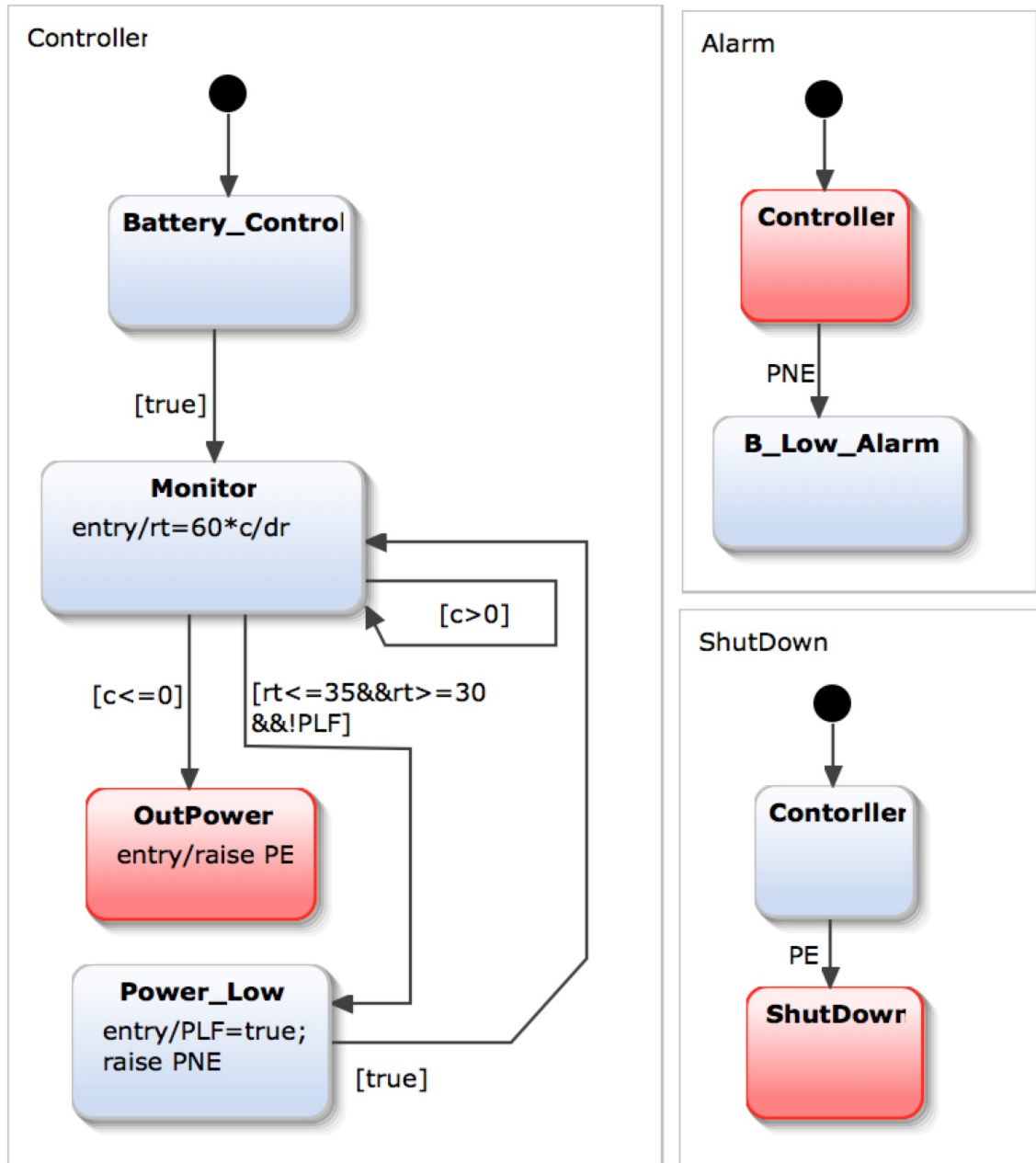
Figure 5.5. Simulation Result of Ventilator Model without Assumptions
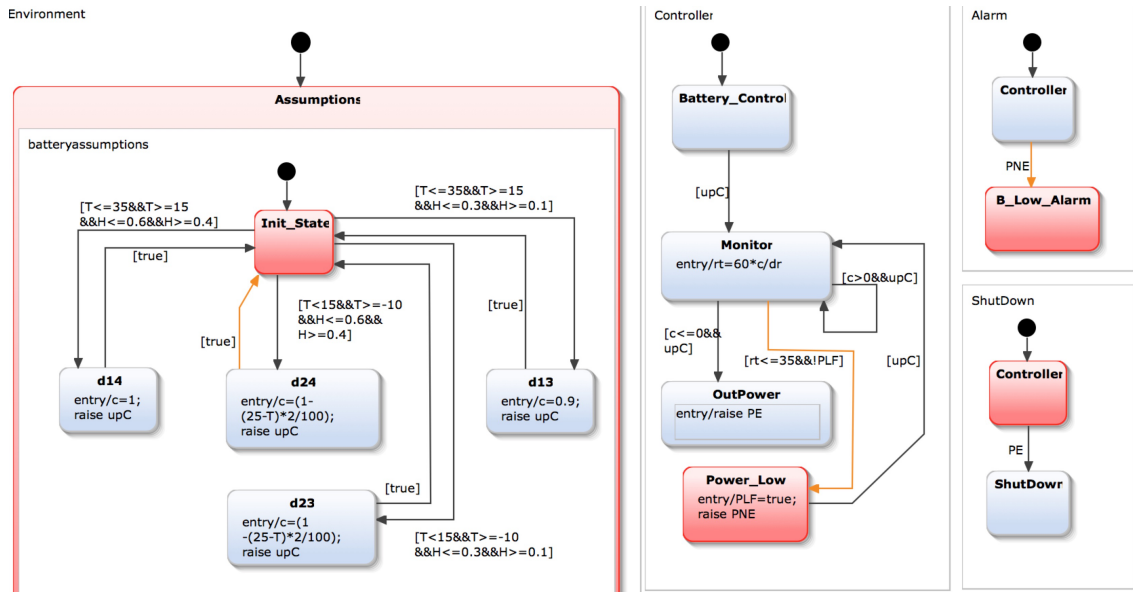
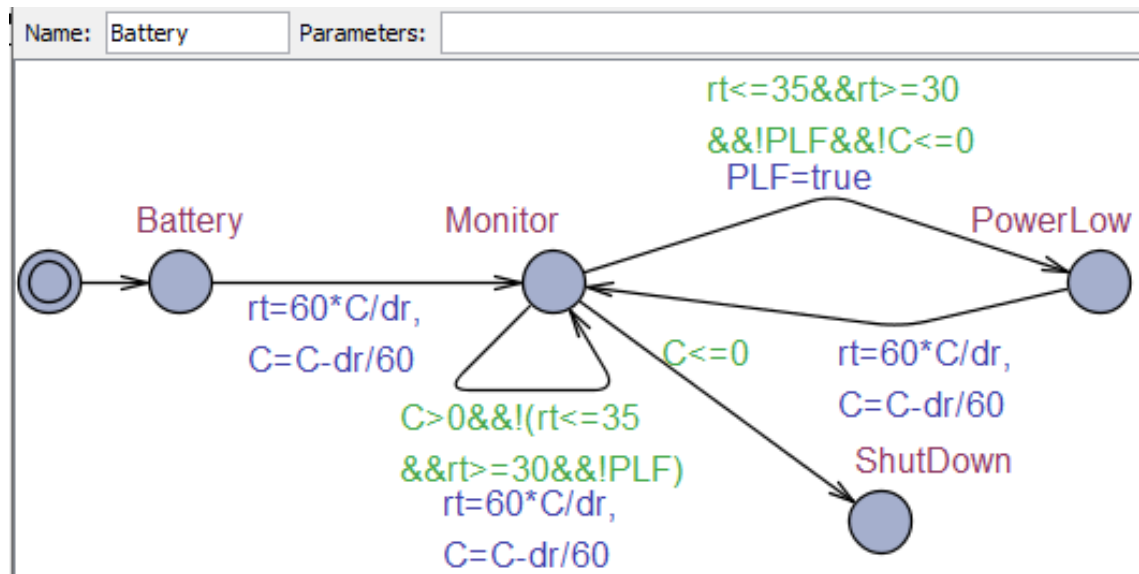Figure 5.6. Simulation Result of Ventilator Model with Assumptions
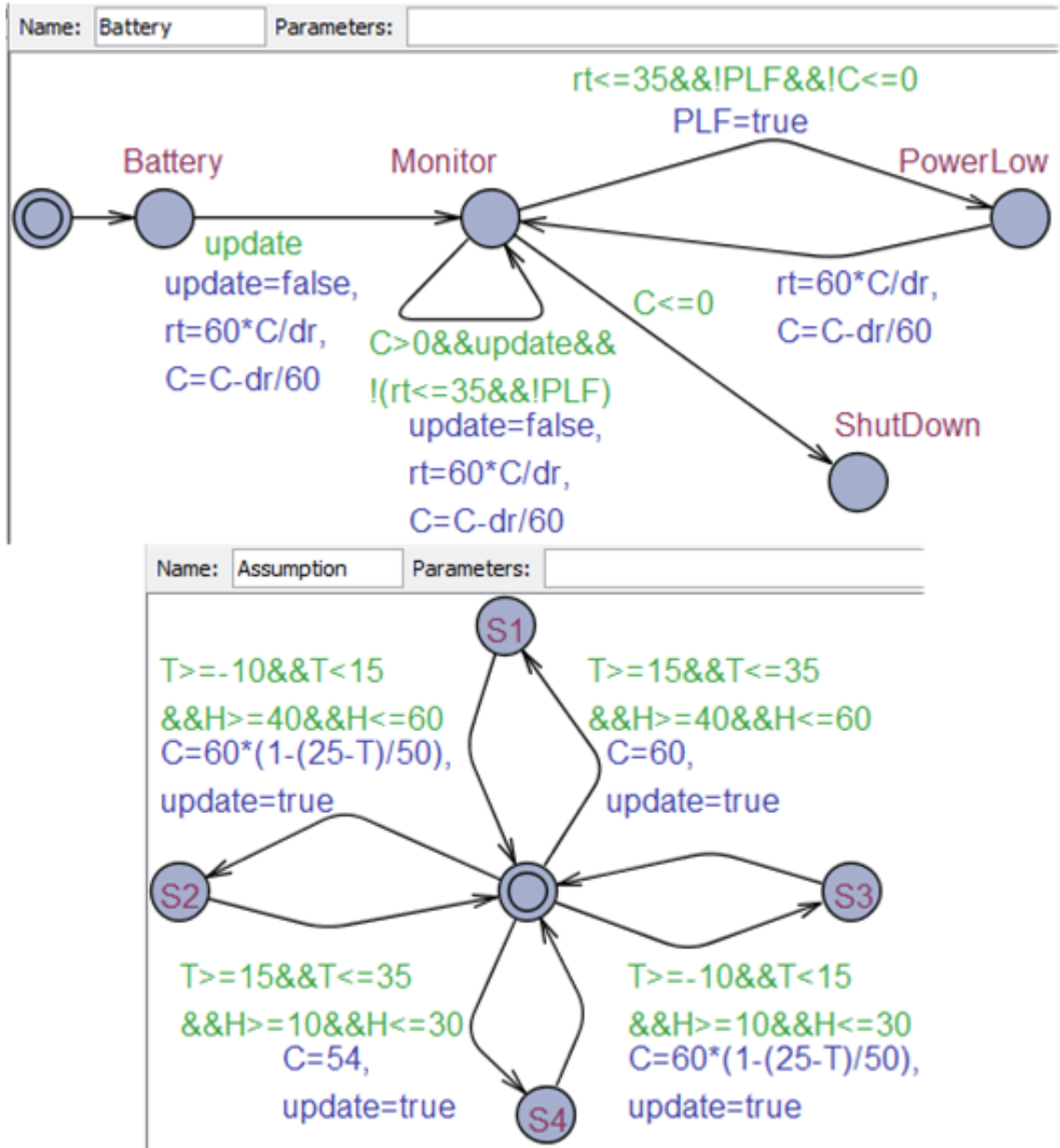

Figure 5.7. Formal Model without Assumptions

Figure 5.8. Formal Model with Assumptions

CHAPTER 6

CONCLUSION

For a cyber-physical system, its execution behaviors are often impacted by its operating environment. However, the assumptions about a cyber-physical system's expected environment are often informally documented, or even left unspecified in the system design. Unfortunately, such unspecified environment assumptions made in safety-critical cyber-physical systems, such as medical cyber-physical systems, can lead to catastrophes. This thesis studies the issues of unspecified assumptions and develops an unspecified assumptions management framework to identify unspecified assumptions, facilitate domain experts to perform impact analysis on the failures caused by unspecified assumptions, and explicitly model and integrate unspecified assumptions into system design models for validation and formal verification. We first present the design and implementation of a tool called *UACFinder* that can automatically extract unspecified assumptions associated with constant variables, frequently read/write variables, and execution patterns from system design models. However, the number of unspecified assumptions discovered from system design models can be enormous. It may not always be feasible for domain experts to identify the most safety-critical ones at different system development phases. Neither it is always necessary to validate all of them. Therefore, we present a Failure Mode and Effects Analysis (FMEA) based approach to facilitate domain experts to conduct impact analysis on unspecified assumptions. In addition, the framework also provides a model for encoding assumptions in a machine-checkable format and composing different unspecified assumptions. Algorithms are also developed to integrate the assumption models with system design models so that system safety properties associated with assumptions can be not only validated by domain experts but also formally verified by existing

formal verification tools.

Case-studies are also conducted on representative system models to demonstrate how *UACFinder* extracts unspecified assumptions from system design models, and how the prioritizing approach based on FMEA facilitates domain experts to verify the appropriateness of identified assumptions. In addition, case studies are also conducted to demonstrate system safety properties may be improved by modeling and integrating unspecified assumptions into system models. The results of case-studies indicate that the proposed unspecified assumptions management framework can identify unspecified assumptions, facilitate domain experts to verify the appropriateness of identified assumptions, and explicitly specify the many unspecified assumptions that caused defects in these systems.

Beyond the scope of this thesis, there are several open challenges.

- In the future work, we need to collect and study more software-related medical device recalls to find out or confirm the detailed root causes of software failures in medical device recalls. By conducting analysis of the root causes, we hope to identify the potential hazards, safety requirements and assumptions, and risk mitigation techniques and strategies to design the next generation of devices and prevent re-occurrence of similar adverse events in the future.

- As the *UACFinder* also provides traces where *constant variables*, *frequently read/updated variables*, and *frequently executed actions* occur, we can potentially use the information to perform impact analyze on element changes, and identify what needs to be modified to realize an element change. In addition, this thesis only focuses on potential unspecified assumption carriers in the forms of *constant variables*, *frequently read/updated variables*, and *frequently executed actions*, there may be other forms of carriers that associate with unspecified

assumptions, which need more investigations in future.

- It is possible that certain properties fail to hold during model verification. In this case, being able to trace back to the unspecified assumptions that causes the failed properties is important and needs to be further studied.

BIBLIOGRAPHY

[1] Lexico.com, "The definition of assumption." `https://www.lexico.com/en/definition/assumption`, 2019.

[2] Merriam-Webster, "The definition of assumption." `https://www.merriam-webster.com/dictionary/assumption`, 2019.

[3] ESA-CNES, "Ariane 5 - flight 501 failure report." `http://www.di.unito.it/~damiani/ariane5rep.html`, 1996.

[4] H. Alemzadeh, R. K. Iyer, Z. Kalbarczyk, and J. Raman, "Analysis of safety-critical computer failures in medical devices," *IEEE Security and Privacy*, July 2013.

[5] Food and Drug Administration Center for Devices and Radiological Health and Office of Compliance Division of Analysis and Program Operations, "Medical device recall reportvfy2003 to fy2012." `https://www.fda.gov/downloads/aboutfda/centersoffices/officeofmedicalproductsandtobacco/cdrh/cdrhtransparency/ucm388442.pdf`, 2013.

[6] U.S. Food and Drug Administration, "Class 1 device recall the hamilton t1." `https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=115383`, 2013.

[7] M. M. Lehman and J. F. Ramil, "Rules and tools for software evolution planning and management," *Annals of Software Engineering*, vol. 11, pp. 15–44, Nov 2001.

[8] G. Lewis, T. Mahatham, and L. Wrage, "Assumptions management in software development," Tech. Rep. CMU/SEI-2004-TN-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2004.

[9] A. Tirumala, *An Assumptions Management Framework for Systems Software*. PhD thesis, University of Illinois at Urbana-Champaign, 2006.

[10] C. B. Moler, "Introduction to MATLAB / SIMULINK," in *Numerical Computing with MATLAB, Revised Reprint*, ch. 1, pp. 1–51, Philadelphia: SIAM, 2008.

[11] C. A. Petri and W. Reisig, "Petri net," *Scholarpedia*, vol. 3, no. 4, p. 6477, 2008.

[12] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183 – 235, 1994.

[13] W3C, "State chart xml (scxml): State machine notation for control abstraction," February 2007.

[14] Itemis, "Yakindu statechart tools (sct)." `https://www.itemis.com/en/yakindu/statechart-tools/`, 2016.

[15] P. Lago and H. van Vliet, "Explicit assumptions enrich architectural models," in *In Proceedings of the 27th international Conference on Software Engineering*, 2005.

[16] C. Guo, S. Ren, Y. Jiang, P. Wu, L. Sha, and R. B. Berlin, "Transforming medical best practice guidelines to executable and verifiable statechart models," in *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 1–10, April 2016.

[17] U.S. Food and Drug Administration, "Medical device databases." `http://www.fda.gov/medicaldevices/deviceregulationandguidance/databases/`, 2017.

[18] U.S. Food and Drug Administration, "Device classification panels." `https://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/Overview/ClassifyYourDevice/ucm051530.htm`, 2017.

[19] K. Toutanova, D. Klein, C. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *HLT-NAACL*, 2003.

[20] U.S. Food and Drug Administration, "Syngo plaza picture archiving and communication system recall." `https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=142876`, 2015.

[21] H. Wu, R. Luk, K. Wong, and K. Kwok, "Interpreting tf-idf term weights as making relevance decisions," *ACM Transactions on Information Systems*, vol. 26, 2008.

[22] Z. Fu, C. Guo, S. Ren, Y. Jiang, and L. Sha, "C.o.d.e recalls-a communication platform for software-related medical device recall." `http://gauss.cs.iit.edu/~code/recalls.html`, 2017.

[23] T. Mikolov, "Word2vec." `https://deeplearning4j.org/word2vec`, 2017.

[24] H. J. Audebert, C. Kukla, S. Clarmann von Claranau, J. Kuhn, B. Vatankhah, J. Schenkel, G. W. Ickenstein, R. L. Haberl, and M. Horn, "Telemedicine for safe and extended use of thrombolysis in stroke: the Telemedic Pilot Project for Integrative Stroke Care (TEMPiS) in Bavaria," *Stroke*, vol. 36, pp. 287–291, Feb 2005.

[25] Y. Jiang, H. Liu, H. Kong, R. Wang, M. Hosseini, J. Sun, and L. Sha, "Use runtime verification to improve the quality of medical care practice," in *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pp. 112–121, IEEE, 2016.

[26] Y. Jiang and etl, "Data-centered runtime verification of wireless medical cyber-physical system," *IEEE Transactions on Industrial Informatics*, 2016.

[27] A. Y.-Z. Ou, Y. Jiang, P.-L. Wu, L. Sha, and R. B. Berlin, "Using human intellectual tasks as guidelines to systematically model medical cyber-physical systems," in *Systems, Man, and Cybernetics (SMC), 2016 IEEE International Conference on*, pp. 004394–004399, IEEE, 2016.

[28] U.S. Food and Drug Administration, "Medical ventilators - faulty batteries." `http://www.fda.gov/MedicalDevices/Safety/ucm460951.htm`.

[29] C. D. Smallwood, B. K. Walsh, L. J. Bechard, and N. M. Mehta, "Carbon dioxide elimination and oxygen consumption in mechanically ventilated children," *Respiratory Care*, vol. 60, no. 5, pp. 718–723, 2015.

[30] F. Corbato, "On building systems that will fail," *ACM Turing Award lectures*, vol. 34, no. 9, pp. 72–81, 1991.

[31] A. Steingruebl and G. Peterson, "Software assumptions lead to preventable errors," *IEEE Security Privacy*, vol. 7, pp. 84–87, July 2009.

[32] A. Bazaz, J. D. Arthur, and J. G. Tront, "Modeling security vulnerabilities: A constraints and assumptions perspective," in *2006 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pp. 95–102, Sep. 2006.

[33] L. Sha and J. Meseguer, "Analytical system composition," in *The First Analytic Virtual Integration of Cyber-Physical Systems Workshop*, 2010.

[34] S. Leue, "Baby death due to software-controlled air bag deactivation?," in *ACM Risks Digest*, 1999.

[35] Z. Li and Y. Zhou, "Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 306–315, 2005.

[36] NIST Information Quality Standards, "National Vulnerability Database." `https://nvd.nist.gov/`, 2019.

[37] NIST Information Quality Standards, "NIST Software Assurance Reference Dataset." `https://samate.nist.gov/SARD/`, 2019.

[38] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *CoRR*, vol. abs/1801.01681, 2018.

[39] C. Wang, Y. Jiang, X. Zhao, X. Song, M. Gu, and J. Sun, "Weak-assert: A weakness-oriented assertion recommendation toolkit for program analysis," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, (New York, NY, USA), pp. 69–72, ACM, 2018.

[40] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, "Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms," *IEEE Transactions on Software Engineering*, vol. 41, pp. 408–428, April 2015.

[41] Z. Fu, C. Guo, S. Ren, Y. Jiang, and L. Sha, "Study of software-related causes in the FDA medical device recalls," in *22nd International Conference on Engineering of Complex Computer Systems, ICECCS 2017, Fukuoka, Japan, November 5-8, 2017*, pp. 60–69, 2017.

[42] D. Harel and A. Naamad, "The statemate semantics of statecharts," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, pp. 293–333, Oct. 1996.

[43] M. Romdhani, A. Jeffroy, P. de Chazelles, A. E. K. Sahraoui, and A. A. Jerraya, "Modeling and rapid prototyping of avionics using statemate," in *Rapid System Prototyping, 1995. Proceedings., Sixth IEEE International Workshop on*, pp. 62–67, Jun 1995.

[44] S. J. Landry, "Human centered design in the air traffic control system," *Journal of Intelligent Manufacturing*, vol. 22, pp. 65–72, Feb 2011.

[45] M. Apkon, "Design of a safer approach to intravenous drug infusions: failure mode effects analysis," *Quality and Safety in Health Care*, vol. 13, pp. 265–271, 08 2004.

[46] C. Guo, S. Ren, Y. Jiang, P.-L. Wu, L. Sha, and R. Berlin, "Transforming medical best practice guidelines to executable and verifiable statechart models," in *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 1–10, April 2016.

[47] C. Guo, Z. Fu, S. Ren, Y. Jiang, M. Rahmaniheris, and L. Sha, "Pattern-based statechart modeling approach for medical best practice guidelines - a case study," in *2017 IEEE 30th International Symposium on Computer-Based Medical Systems (CBMS)*, June 2017.

[48] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam, "Safety-critical medical device development using the upp2sf model translation tool," *ACM Trans. Embed. Comput. Syst.*, vol. 13, pp. 127:1–127:26, Apr. 2014.

[49] 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 *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pp. 173–184, April 2012.

[50] G. Gosta and Z. Jianfei, "Efficiently using prefix-trees in mining frequent itemsets," in *the 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.

[51] M. Rahmaniheris, P. Wu, L. Sha, and R. R. Berlin, "An organ-centric best practice assist system for acute care," in *29th IEEE International Symposium on Computer-Based Medical Systems, CBMS 2016, Belfast, UK and Dublin, Ireland, June 20-24, 2016*, pp. 100–105, 2016.

[52] M. Rahman and M. C. Smith, "Chronic Renal Insufficiency: A Diagnostic and Therapeutic Approach," *Archives of Internal Medicine*, vol. 158, pp. 1743–1752, 09 1998.

[53] Cleveland Clinic, "Hyperkalemia (high blood potassium)." `https://my.cle velandclinic.org/health/diseases/15184-hyperkalemia-hi gh-blood-potassium`, 2018.

[54] B. Shah, P. R. Rao, B. Moon, and M. Rajagopalan, "A data parallel algorithm for xml dom parsing," in *Database and XML Technologies* (Z. Bellahsène, E. Hunt, M. Rys, and R. Unland, eds.), (Berlin, Heidelberg), pp. 75–90, Springer Berlin Heidelberg, 2009.

[55] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Rec.*, vol. 22, pp. 207–216, June 1993.

[56] S. Naulaerts, P. Meysman, W. Bittremieux, T. N. Vu, W. Vanden Berghe, B. Goethals, and K. Laukens, "A primer to frequent itemset mining for bioinformatics," vol. 16, pp. 216–231, 03 2015.

[57] C.-Y. Tu, T.-J. Chen, and L.-F. Chou, "Application of frequent itemsets mining to analyze patterns of one-stop visits in taiwan," *PLOS ONE*, vol. 6, pp. 1–6, 07 2011.

[58] J. S. Park, M.-S. Chen, and P. S. Yu, "An effective hash-based algorithm for mining association rules," *SIGMOD Rec.*, vol. 24, pp. 175–186, May 1995.

[59] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, (San Francisco, CA, USA), pp. 487–499, Morgan Kaufmann Publishers Inc., 1994.

[60] B. W. Kernighan, *The C Programming Language.* Prentice Hall Professional Technical Reference, 2nd ed., 1988.

[61] A. O. Hosten, "Clinical methods: The history, physical, and laboratory examinations. 3rd edition." `https://www.ncbi.nlm.nih.gov/books/NBK305/`, 1993.

[62] American Association for Clinical Chemistry, "Creatinine." `https://labtestsonline.org/understanding/analytes/creatinine/tab/glance/`, 2015.

[63] Mayo Clinic, "Creatinine test." `https://www.mayoclinic.org/tests-procedures/creatinine-test/about/pac-20384646`, 2015.

[64] V. Singh, S. Khatana, and P. Gupta, "Blood gas analysis for bedside diagnosis," vol. 4, pp. 136–141, Jul-Dec 2013.

[65] American Heart Association, "About arrhythmia.." `http://www.heart.org/HEARTORG/Conditions/Arrhythmia/AboutArrhythmia/About-Arrhythmia_UCM_002010_Article.jsp`, 2015.

[66] Mayo Clinic, "Tachycardia: Fast heart rate." `https://www.mayoclinic.org/diseases-conditions/tachycardia/symptoms-causes/syc-20355127`, 2016.

[67] National Kidney Disease Education Program, "Chronic kidney disease (ckd) and diet: Assessment, management and treatment." `http://www.niddk.nih.gov/health-information/health-communication-programs/nkdep/a-z/Documents/ckd-diet-assess-manage-treat-508.pdf`, 2016.

[68] N. A. Snooke, *Automated Failure Effect Analysis for PHM of UAV*, pp. 1027–1051. Springer Netherlands, 2015.

[69] N. Hughes, E. Chou, C. Price, and M. Lee, "Automating mechanical fmea using functional models," 05 1999.

[70] F. Mhenni, N. Nguyen, and J.-Y. Choley, "Automatic fault tree generation from sysml system models," *2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pp. 715–720, 2014.

[71] Y. Papadopoulos, D. Parker, and C. Grante, "Automating the failure modes and effects analysis of safety critical systems," vol. 8, pp. 310 – 311, 04 2004.

[72] Y. Papadopoulos and M. Maruhn, "Model-based synthesis of fault trees from matlab-simulink models," in *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS)*, DSN '01, (Washington, DC, USA), pp. 77–82, IEEE Computer Society, 2001.

[73] B. Duwe, B. D. Fuchs, and J. Hansen-Flaschen, "Failure mode and effects analysis application to critical care medicine," *Crit Care Clin.*, pp. 21–30, 2005.

[74] W. Adachi and A. E. Lodolce, "Use of failure mode and effects analysis in improving the safety of i.v. drug administration," *American Journal of Health-System Pharmacy*, vol. 62, pp. 917–920, 05 2005.

[75] D. Wang, J. Pan, G. S. Avrunin, L. A. Clarke, and B. Chen, "An optimization to automatic fault tree analysis and failure mode and effect analysis approaches for processes," in *International Conference on Computer Design and Applications (ICCDA)*, 2010.

[76] E. Stalhandske, J. DeRosier, R. Wilson, and J. Murphy, "Healthcare fmea in the veterans health administration," *Patient Safety and Quality Healthcare*, vol. 6, pp. 30–33, 2009.

[77] P. Lago, G. Bizzarri, F. Scalzotto, A. Parpaiola, A. Amigoni, G. Putoto, and G. Perilongo, "Use of fmea analysis to reduce risk of errors in prescribing and administering drugs in paediatric wards: A quality improvement report," *BMJ open*, vol. 2, 10 2012.

[78] Y. Lu, F. Teng, J. Zhou, A. Wen, and Y. Bi, "Failure mode and effect analysis in blood transfusion: A proactive tool to reduce risks," *Transfusion*, vol. 53, 04 2013.

[79] The Joint Commission on Accreditation of Healthcare Organizations, "Joint commission." `https://www.jointcommission.org/`, 2018.

[80] M. C. Cantone, M. Ciocca, F. Dionis, P. Fossat, S. Lorentin, M. Krengl, S. Molinel, R. Orecchia, M. Schwarz, I. Veronese, and V. Vitolo, "Application of failure mode and effects analysis to treatment planning in scanned proton beam radiotherapy," in *Radiation Oncology*, 2013.

[81] Wikipedia, "Failure mode and effects analysis." `https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis`, 2017.

[82] SIEMENS, "How to conduct a failure modes and effects analysis." `https://polarion.plm.automation.siemens.com/hubfs/Docs/Guides_and_Manuals`, 2017.

[83] NDP Solutions, "Failure modes and effects analysis." `http://www.npd-solutions.com/fmea.html`, 2017.

[84] The Centers for Medicare & Medicaid Services, "Guidance for performing failure mode and effects analysis with performance improvement projects." `https://www.cms.gov/Medicare/Provider-Enrollment-and-Certification/QAPI/downloads/GuidanceForFMEA.pdf`, 2017.

[85] Vincent, Jean-Louis, and R. Moreno, "Clinical review: Scoring systems in the critically ill," *Critical Care*, 2014.

[86] D. C. Bouch and J. P. Thompson, "Severity scoring systems in the critically ill," *Continuing Education in Anaesthesia Critical Care & Pain*, vol. 8, no. 5, pp. 181–185, 2008.

[87] D. A. Kaufman, "Interpretation of arterial blood gas," 2017.

[88] C. Marwick, E. Watts, J. Evans, and P. Davey, "Quality of care in sepsis management: Development and testing of measures for improvement," *The Journal of antimicrobial chemotherapy*, vol. 60, pp. 694–7, 10 2007.

[89] E. Thornton, O. R. Brook, M. Mendiratta-Lala, D. T. Hallett, and J. B. Kruskal, "Application of failure mode and effect analysis in a radiology department," *Radiographics*, vol. 31, pp. 281 – 293, Jan 2011.

[90] S. NA, F. BD, and B. N., "Is failure mode and effect analysis reliable?," *Journal of Patient Safety*, vol. 5, pp. 86 – 94, 2009.

[91] U.S. Food and Drug Administration, "Applying human factors and usability engineering to medical devices." `http://www.fda.gov/downloads/Medi calDevices/.../UCM259760.pdf`.

[92] J. Goldmann., "Medical device interoperability to enable system solutions at the sharp edge of healthcare delivery," in *White House Homeland Security Council Biodefense Directorate Conference*, Apr 2010.

[93] L. Schmitt, T. Falck, F. Wartena, and D. Simons., "Novel iso/ieee 11073 standards for personal telehealth systems interoperability," in *In Proc. of Joint Workshop on HCMDSS-MDPnP*, 2007.

[94] Medical Device "Plug-and-Play" Interoperability Program, "Ice standard: Integrated clinical environment." `http://www.mdpnp.org/mdice.html`.

[95] Medical Device "Plug-and-Play" Interoperability Program, "Device clock synchronization." `http://www.mdpnp.org/devicesynchronization.ht ml`.

[96] C. Kim, M. Sun, S. Mohan, H. Yun, L. Sha, and T. F. Abdelzaher, "A framework for the safe interoperability of medical devices in the presence of network failures," in *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, pp. 149–158, 2010.

[97] C. Kerechanin, P. Cutchis, J. Vincent, D. Smith, and D. Wenstrand, "Development of field portable ventilator systems for domestic and military emergency medical response," *Johns Hopkins Apl Technical Digest*, vol. 25, pp. 214–222, 07 2004.

[98] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking.* The MIT Press, 1999.

[99] D. W. Loveland, *Automated Theorem Proving: A Logical Basis (Fundamental Studies in Computer Science).* sole distributor for the U.S.A. and Canada, Elsevier North-Holland, 1978.

[100] G. Duftschmid and S. Miksch, "Knowledge-based verification of clinical guidelines by detection of anomalies," *Artif. Intell. Med.*, vol. 22, pp. 23–41, Apr. 2001.

[101] M. Jersak, K. Richter, R. Racu, J. Staschulat, R. Ernst, J.-C. Braam, and F. Wolf, "Formal methods for integration of automotive software," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pp. 45–50, 2003.

[102] O. Laurent, "Using formal methods and testability concepts in the avionics systems validation and verification (v&v) process," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pp. 1–10, April 2010.

[103] Cadex Electronics, "Battery university." `http://batteryuniversity.com/`.

[104] European Hearing Instrument Manufacturers Association, "Ehima recommendations for zinc-air hearing aid batteries." `http://www.ehima.com/wp-content/uploads/2014/03/EHIMA-Battery-Recommendations_V2.0.pdf`.