2-2011

Improving Processes Using Static Analysis Techniques

Bin Chen
University of Massachusetts Amherst

IMPROVING PROCESSES USING STATIC ANALYSIS TECHNIQUES

A Dissertation Presented

by

BIN CHEN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2011

Computer Science

IMPROVING PROCESSES USING STATIC ANALYSIS TECHNIQUES

A Dissertation Presented

by

BIN CHEN

Approved as to style and content by:

Lori A. Clarke, Co-chair

George S. Avrunin, Co-chair

Leon J. Osterweil, Member

Elizabeth A. Henneman, Member

Jenna L. Marquard, Member

Andrew G. Barto, Department Chair
Computer Science

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere thanks and appreciation to Lori Clarke and George Avrunin, my dissertation advisors. For their encouragement and patient, even during tough times in my Ph.D. pursuit. For their guidance, advice and an unwavering interest in my research. And for their contributions of time, ideas, and funding to make this work possible.

I also want to thank the other members of my thesis committee, Leon Osterweil, Elizabeth Henneman and Jenna Marquard for their valuable suggestions and support. Altough not my advisor, Leon Osterweil provided firm support and constructive feedback along the way to enable and shape this work.

This work has benefitted enormously from my interactions with fellow colleagues in the LASER group. The group has been a source of friendships as well as good advice and collaboration. I would like to thank Sandy Wise for clarifying the Little-JIL semantics and taking care of releasing the process analysis framework. Thanks to Heather Conboy for helping me with the FLAVERS system. I also want to thank Bobby Simidchieva, Stefan Christov, and Huong Phan for using the process analysis framework and discovering all kinds of bugs in the implementation. For their friendship and the opportunity to discuss ideas of all kinds, I also want to thank other past and present group members, especially M.S. Raunak, Jianbin Tan, Shangzhu Wang, Zongfang Lin, Rachel Cobleigh, Jamieson Cobleigh, Yan Zeng, and Aaron Cass.

My time at UMASS was made enjoyable in large part due to the many friends and that became a part of my life. I am grateful to Xiaotao Liu, Weifeng Chen, Yong Yuan, Ting Yang, Yuandong Yang, Shichao Ou, Ao Feng, among many others, for all kinds of fun that we had together.

Lastly, I'm utterly grateful to my family for all their love and encouragement. For my parents who raised me with unselfish love and supported me in all my pursuits. For my brother Chao Chen for his encouragement. And most of all for my loving, supportive, encouraging, and patient wife Wenjie Liu whose faithful support during the final stages of this Ph.D. is so appreciated.

ABSTRACT

IMPROVING PROCESSES USING STATIC ANALYSIS TECHNIQUES

FEBRUARY 2011

BIN CHEN

B.Sc., PEKING UNIVERSITY, CHINA

M.Sc., PEKING UNIVERSITY, CHINA

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Lori A. Clarke and Professor George S. Avrunin

Real-world processes often undergo improvements to meet certain goals, such as coping with changed requirements, eliminating defects, improving the quality of the products, and reducing costs. Identifying and evaluating the defects or errors in the process, identifying the causes of such defects, and validating proposed improvements all require careful analysis of the process. Human-intensive processes, where human contributions require considerable domain expertise and have a significant impact on the success or failure of the overall mission, are of particular concern because they can be extremely complex and may be used in critical, including life-critical, situations. To date, the analysis support for such processes is very limited. If done at all, it is usually performed manually and can be extremely time-consuming, costly and error-prone. There has been considerable success lately in using static analysis techniques to analyze hardware systems, software systems, and manufacturing processes. This thesis explores how such analysis techniques can be automated and employed to effectively analyze life-critical, human-intensive processes.

In this thesis, we investigated two static analysis techniques: Finite-State Verification (FSV) and Fault Tree Analysis (FTA). We proposed a process analysis framework that is capable of performing both FSV and FTA on rigorously defined processes. Although evaluated for processes specified in the Little-JIL process definition language, this is a general framework independent of the process definition language. For FSV, we developed a translation-based approach that is able to take advantage of existing FSV tools. The process definition and property to be evaluated are translated into the input model and property representation accepted by the selected FSV tool. Then the FSV tool is executed to verify the model against the property representation. For FTA, we developed a template-based approach to automatically derive fault trees from the process definition. In addition to showing the feasibility of applying these two techniques to processes, much effort has been put on improving the scalability and the usability of the framework so that it can be easily used to analyze complex real-world processes. To scale the analysis, we investigated several optimizations that are able to dramatically reduce the translated models for FSV tools and speed up the verification. We also developed several optimizations for the fault tree derivation to make the generated fault tree much more compact and easier to understand and analyze. To improve the usability, we provided several approaches that make analysis results easier to understand.

We evaluated this framework based on the Little-JIL process definition language and employed it to analyze two real-world, human-intensive processes: an in-patient blood transfusion process and a chemotherapy process. The results show that the framework can be used effectively to detect defects in such real-world, human-intensive processes.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS . . . . **iv**

ABSTRACT . . . . **vi**

LIST OF TABLES . . . . **xiii**

LIST OF FIGURES . . . . **xiv**

CHAPTER

1. INTRODUCTION **1**

2. BACKGROUND AND RELATED WORK **7**

2.1 Process Improvement 7

2.1.1 Industrial Process Improvement 7
2.1.2 Software Process Improvement 10

2.2 Process Modeling 13

2.2.1 Little-JIL Process Definition Language 13
2.2.2 Other Process Modeling Languages 18

2.2.2.1 BPEL 18
2.2.2.2 UML 20
2.2.2.3 Petri Nets 21
2.2.2.4 Rule-Based 22
2.2.2.5 Medical Guideline Modeling Languages 24

2.3 Finite-State Verification 27

2.3.1 Introduction to Finite-State Verification 27
2.3.2 Finite-State Verification Tools 29

2.3.2.1 SPIN 29

2.3.2.2 SMV 30
2.3.2.3 FLAVERS 30
2.3.2.4 INCA 31
2.3.2.5 LTSA 31
2.3.2.6 Bandera 32

2.3.3 Verification of Business Process 33
2.3.4 Verification of Medical Processes 35

2.4 Fault-Tree Analysis 36

2.4.1 Introduction to Fault-Tree Analysis 36
2.4.2 Related Work of Fault-Tree Analysis 39

3. FINITE-STATE VERIFICATION 41

3.1 Process Verification Framework 42
3.2 Process Translation 46

3.2.1 Process Well-Formedness Check 48
3.2.2 Process Unrolling 48
3.2.3 Little-JIL-to-BIR Translation 50

3.2.3.1 Bandera Intermediate Representation 51
3.2.3.2 Translation Overview 52

3.2.3.2.1 Threads 52
3.2.3.2.2 Variables 53
3.2.3.2.3 Predicates 55

3.2.3.3 Translation Templates 56

3.2.3.3.1 Leaf Step Started Template 58
3.2.3.3.2 Sequential Step Started Template 59
3.2.3.3.3 Parallel Step Started Template 59
3.2.3.3.4 Exception Handling Template 61

3.2.3.4 Translation Algorithm 63

3.2.4 Summary 63

3.3 Property Specification and Translation 65

3.3.1 Property Specification 65
3.3.2 Property Event Binding Specification 66
3.3.3 Property Specification and Property Event Binding
Translation 72

3.3.4 Summary 74

3.4 Optimizations 75

3.4.1 Step Abstraction 77
3.4.2 Step Removal 79
3.4.3 Thread Inlining 81
3.4.4 Exception Elimination 83
3.4.5 Variable Reuse 85
3.4.6 Summary 90

4. FAULT-TREE ANALYSIS 91

4.1 Overview of Process Fault-Tree Analysis 92
4.2 Automatic Fault Tree Derivation 94

4.2.1 Events 97

4.2.1.1 Category 1 98
4.2.1.2 Category 2 98
4.2.1.3 Category 3 99

4.2.2 Templates 101

4.2.2.1 Reverse Control Flow Graph and Artifact Flow Graph 103
4.2.2.2 Template for *Artifact o is Wrong When S is Started* 106
4.2.2.3 Template for *Artifact o is Wrong When S is Posted* 107

4.2.3 Derivation algorithm 112

4.3 Evaluating Fault Trees 114
4.4 Process FTA Issues 118

4.4.1 Scalability 118

4.4.1.1 Step Abstraction 118
4.4.1.2 Step Removal 120
4.4.1.3 Equivalent Event Removal 120

4.4.2 Looping Constructs 123
4.4.3 NOT Gate 126
4.4.4 Usability 128

4.4.4.1 Partial Fault Tree 129

4.4.4.2 Process Trace 129

4.5 Limitations of Our Approach 130
4.6 Summary 130

5. TRACE GENERATION AND VISUALIZATION 132

5.1 High-Level Trace Generation 133

5.1.1 Introduction to A* Algorithm 134
5.1.2 High-Level Trace Generation Tool Architecture 135

5.1.2.1 Customization for FSV 138

5.1.2.1.1 Goal Constructor 138
5.1.2.1.2 Little-JIL Event Manager 140

5.1.2.2 Customization for FTA 140

5.1.2.2.1 Goal Constructor 141
5.1.2.2.2 Little-JIL Event Manager 143

5.1.3 Scaling Problem 143

5.2 Trace Visualization 144

5.2.1 Structured Textual View 145
5.2.2 Timeline View 147

5.3 Summary 148

6. EVALUATION 149

6.1 Introduction to Selected Cases 149

6.1.1 In-Patient Blood Transfusion Process 151
6.1.2 Chemotherapy Process 153

6.2 Methodology 155

6.2.1 Process Specification 156
6.2.2 Property Specification 156
6.2.3 Finite-State Verification 157
6.2.4 Fault-Tree Analysis 158

6.3 Results and Discussion 158

6.3.1 Finite-State Verification 158

6.3.1.1 Errors in the process definitions 159
6.3.1.2 Errors in the property specification 160
6.3.1.3 Errors in the real processes 161

6.3.2 Fault-Tree Analysis 164

6.4 Summary 165

7. CONCLUSION AND FUTURE WORK 167

7.1 Finite-State Verification 168
7.2 Fault Tree Analysis 169
7.3 Trace Generation and Visualization 170
7.4 Failure Mode and Effects Analysis 171
7.5 Interplay of Different Analysis Techniques 173

APPENDICES

A. LITTLE-JIL PROCESS TRANSLATION TEMPLATES 175
B. FAULT TREE DERIVATION TEMPLATES 193
C. UNOPTIMIZED BIR PROGRAM FOR SIMPLIFIED BLOOD TRANSFUSION PROCESS 208
D. IN-PATIENT BLOOD TRANSFUSION PROCESS 213
E. BLOOD TRANSFUSION PROCESS VERIFICATION REPORT 225
F. BLOOD TRANSFUSION PROCESS FAULT TREE 293
G. BLOOD TRANSFUSION PROCESS MCSS 301
H. CHEMOTHERAPY PROCESS 338
I. CHEMOTHERAPY PROCESS VERIFICATION REPORT 356
J. CHEMOTHERAPY PROCESS FAULT TREE 388
K. CHEMOTHERAPY PROCESS MCSS 404

BIBLIOGRAPHY 441

LIST OF TABLES

Table **Page**

4.1 Fault Tree Derivation Templates 103

B.1 Fault Tree Derivation Templates 193

LIST OF FIGURES

Figure **Page**

2.1 DMAIC Model 9

2.2 Little-JIL Step 14

2.3 Simplified Blood Transfusion Process 16

2.4 Fault Tree Elements 37

2.5 Fault Tree Example 37

3.1 Process Verification Framework 43

3.2 Simplified Blood Transfusion Process 46

3.3 Implementation of Process Translation 47

3.4 Unbounded Threads Examples 49

3.5 BIR Representation for Leaf Step Started Template 58

3.6 Sequential Step Example 60

3.7 BIR Representation for the Sequential Started Template 60

3.8 BIR Representation for the Parallel Started Template 62

3.9 BIR Representation for the Exception Handling Template 62

3.10 Process Example for Discussing Step Identifier 67

3.11 Step Instance Identifier Syntax 68

3.12 Property Event Binding Syntax 70

3.13 Parameter Def/Use Event Example 71

3.14 Simplified Blood Transfusion Process after Step Abstraction 78

3.15 Simplified Blood Transfusion Process after Step Removal 80

3.16 BIR Code After Thread Inlining 82

3.17 Simplified Blood Transfusion Process after Exception Elimination 84

3.18 Check Whether Two Steps May in Parallel 87

4.1 Simplified Blood Process 96

4.2 Reverse Control Flow Graph for Simplified Blood Transfusion Process 104

4.3 Artifact Flow Graph for Simplified Blood Transfusion Process 105

4.4 Partial Fault Tree for Template 3 106

4.5 Instantiated Partial Fault Tree for Template 3 107

4.6 Partial Fault Tree for Template 1 108

4.7 Partial Fault Tree for Template 2 110

4.8 Instantiated Partial Fault Tree for Template 2 112

4.9 Fault Tree for Simplified Blood Transfusion Process 115

4.10 Step Abstraction for Fault Tree Derivation 119

4.11 Step Removal for Fault Tree Derivation 120

4.12 Groups in Fault Tree for Simplified Blood Transfusion Process 121

4.13 Example of Process Containing Loop 123

4.14 Fault Tree Containing Loop 124

4.15 Fault Tree without Loop 125

4.16 Simple Process for Discussing NOT gates 127

4.17 Fault Tree without NOT Gates 128

4.18 Partial Fault Tree for MCS {E6} 129

5.1 Little-JIL Trace Generation Tool Architecture 136

5.2 Goal FSA from Counter-Example Trace 139

5.3 Goal FSA from MCS 143

5.4 Simple Trace Example 145

5.5 Structured Textual View 146

5.6 Timeline View 147

5.7 Timeline View - Collapsed 148

6.1 In-Patient Blood Transfusion Process 152

6.2 Chemotherapy Process 154

A.1 BIR for Posting Template - Channel Blocking Take 176

A.2 BIR for Posting Template - Channel Non-Blocking Take 176

A.3 BIR for Posting Template - Channel Blocking Read 176

A.4 BIR for Posting Template - Channel Non-Blocking Read 176

A.5 BIR Representation for Choice Sub-Step Retracted Template 178

A.6 BIR Representation for Parallel Sub-Step Retracted Template 179

A.7 BIR Representation for Opted-Out Template 180

A.8 Sequential Step Example 180

A.9 BIR Representation for Sequential Started Template 181

A.10 Try Step Example 181

A.11 BIR Representation for Try Started Template 182

A.12 Parallel Step Example 183

A.13 BIR Representation for the Parallel Started Template 184

A.14 Choice Step Example 186

A.15 BIR Representation for the Choice Started Template 187

A.16 BIR Representation for Sub-step of Choice Step 188

A.17 BIR Representation for Leaf Step Started Template 189

A.18 BIR Representation for Completing Template Example 190

A.19 Exception Handling Example 190

A.20 BIR Representation for the Exception Handling Template 191

B.1 Template 1 194

B.2 Template 2 196

B.3 Template 3 197

B.4 Template 4 198

B.5 Template 5 199

B.6 Template 6 200

B.7 Template 7 201

B.8 Template 8 202

B.9 Template 9 203

B.10 Template 10 204

B.11 Template 11 205

B.12 Template 12 206

B.13 Template 13 206

B.14 Template 14 207

D.1 Diagram “perform in-patient blood transfusion” 214

D.2 Diagram “check for existence of type and screen” 214

D.3 Diagram “pick up blood from blood bank” 214

D.4 Diagram “perform pre-release checks” 215

D.5 Diagram “perform transfusion” 215

D.6 Diagram “administer a single unit of blood product” 215

D.7 Diagram “perform pre-infusion work” 216

D.8 Diagram “assess patient” 216

D.9 Diagram “perform bedside checks” 216

D.10 Diagram “verify blood product information” 217

D.11 Diagram “confirm product tag matches patient ID band” 217

D.12 Diagram “confirm product label matches product tag” 217

D.13 Diagram “handle failed product verification by replacement” 218

D.14 Diagram “handle previously unrecognized patient problem” 218

D.15 Diagram “handle suspected transfusion reaction” 218

D.16 Diagram “infuse unit of blood product” 219

D.17 Diagram “perform post-infusion work” 219

D.18 Diagram “perform Blood Specimen Obtaining process” 219

D.19 Diagram “order test(s)” 220

D.20 Diagram “order test(s) on computer” 220

D.21 Diagram “order test(s) on patient chart” 220

D.22 Diagram “collect labels” 221

D.23 Diagram “walk to patient” 221

D.24 Diagram “obtain and label specimen” 221

D.25 Diagram “apply label” 222

D.26 Diagram “obtain blood specimen” 222

D.27 Diagram “send blood specimen to lab” 222

D.28 Diagram “log specimen transport action” 223

D.29 Diagram “verify the correct patient to get specimen” 223

D.30 Diagram “verify patient ID to ID band” 223

D.31 Diagram “verify patient first and last names” 224

D.32 Diagram “verify patient-DOB” 224

D.33 Diagram “verify patient ID on artifact to artifact” 224

F.1 Blood Transfusion Process Fault Tree Part 1 294

F.2 Blood Transfusion Process Fault Tree Part 2 295

F.3 Blood Transfusion Process Fault Tree Part 3 296

F.4 Blood Transfusion Process Fault Tree Part 4 297

F.5 Blood Transfusion Process Fault Tree Part 5 298

F.6 Blood Transfusion Process Fault Tree Part 6 299

F.7 Blood Transfusion Process Fault Tree Part 7 300

H.1 Diagram “chemotherapy process” 339

H.2 Diagram “perform consultation and assessment” 339

H.3 Diagram “confirm all necessary information is present” 340

H.4 Diagram “create treatment plan and orders” 340

H.5 Diagram “perform initial review of patient records” 341

H.6 Diagram “perform Triage MA tasks before giving treatment plan to Practice RN” 341

H.7 Diagram “perform Practice RN verifications” 342

H.8 Diagram “confirm pretesting has been done” 342

H.9 Diagram “confirm existence and not staleness of height/weight data in CIS” 343

H.10 Exception Handlers Used in Diagram “perform Practice RN verifications” 343

H.11 Diagram “verify doses (practice RN)” 344

H.12 Exception Handlers Used in Diagram “verify doses (practice RN)” 344

H.13 Diagram “perform Triage MA tasks after receiving treatment plan from Practice RN” 345

H.14 Diagram “perform pharmacy tasks after receiving treatment plan from Triage MA” 345

H.15 Exception Handlers Used in Diagram “perform pharmacy tasks after receiving treatment plan from Triage MA” 346

H.16 Diagram “obtain patient informed consent and install portacath” 346

H.17 Diagram “perform final tasks on the day before the administration of chemotherapy” 346

H.18 Diagram “perform pharmacy tasks on the day before administration of chemotherapy” 347

H.19 Diagram “process individual patient (anonymous)” 347

H.20 Exception Handlers Used in Diagram “process individual patient (anonymous)” 348

H.21 Diagram “perform verification tasks in MedManager” 348

H.22 Diagram “first day of chemo” 349

H.23 Diagram “prepare for patient arrival” 349

H.24 Diagram “review patient paperwork(anonymous)” 350

H.25 Diagram “verify treatment plan and orders” 350

H.26 Diagram “verify dosages” 351

H.27 Diagram “handle MissingTreatmentPlanFromPatientChart” 351

H.28 Diagram “perform checks on patient before calling pharmacy” 352

H.29 Diagram “handle low blood counts” 352

H.30 Diagram “perform pharmacy tasks and give pre-medications to patient” 353

H.31 Diagram “perform first day of chemo pharmacy tasks” 353

H.32 Definitions of Steps Used in Diagram “perform first day of chemo pharmacy tasks” 354

H.33 Diagram “transcribe and place consult note in patient’s record” 354

H.34 Diagram “perform nurse tasks after picking drugs from pharmacy” 355

J.1 Chemotherapy Process Fault Tree Part 1 389

J.2 Chemotherapy Process Fault Tree Part 2 390

J.3 Chemotherapy Process Fault Tree Part 3 391

J.4 Chemotherapy Process Fault Tree Part 4 392

J.5 Chemotherapy Process Fault Tree Part 5 393

J.6 Chemotherapy Process Fault Tree Part 6 394

J.7 Chemotherapy Process Fault Tree Part 7 395

J.8 Chemotherapy Process Fault Tree Part 8 396

J.9 Chemotherapy Process Fault Tree Part 9 397

J.10 Chemotherapy Process Fault Tree Part 10 398

J.11 Chemotherapy Process Fault Tree Part 11 399

J.12 Chemotherapy Process Fault Tree Part 12 400

J.13 Chemotherapy Process Fault Tree Part 13 401

J.14 Chemotherapy Process Fault Tree Part 14 402

J.15 Chemotherapy Process Fault Tree Part 15 403

CHAPTER 1

INTRODUCTION

Processes pervade our society. Products are produced by various manufacturing processes; software is developed following software development processes; government or business services are provided via established processes; and healthcare is delivered through different medical processes. Likewise, processes in many other domains can be identified quite readily. Although different terminology might be used in different application domains, a process usually consists of a set of tasks that coordinate and communicate with each other to achieve certain goals and objectives, where a task defines a specific unit of work performed by an agent that can be either a human (e.g., a software developer or a medical practitioner) or automated (e.g., a piece of software or a electronic device).

Real-world processes often undergo continuous improvements to meet certain goals, such as coping with changed requirements, eliminating defects, increasing the quality of the products, and reducing costs. As suggested by industrial process improvement practices (e.g., [106]), effective process improvements need to be carried out incrementally through continuous improvement iterations, as opposed to radical re-engineering. A process improvement iteration usually involves five major steps: 1) define the concrete problem to be solved; 2) measure the process performance if necessary; 3) identify and evaluate the root causes of the problem; 4) change the process to address the root causes; 5) validate that no errors are introduced and the changes indeed solve the problem.

Analysis plays a critical role in this incremental process improvement. Identifying and evaluating the root causes of the problem require careful analysis of the process. In addition, since errors may be introduced by changes to the process, analysis should be carried out to detect such errors before the modified process is deployed. Analyses are usually performed based on the specification of the process. In many domains, processes are defined in natural language, perhaps supplemented by diagrams or other informal notations. Processes defined with such informality are often ambiguous, incomplete or inconsistent, preventing many formal analyses from being applied on these processes. Even the analyses that can be applied have to be performed manually and can be highly time-consuming and costly. Such informal analyses may also produce inaccurate or even incorrect results, leading to ill-advised changes that fail to achieve the improvement goal. To date many process definition languages have been developed to formally define processes, but their support for analysis is still very limited.

Human-intensive processes, where "the human contributions require considerable domain expertise and have a significant impact on the success or failure of the overall mission" [34], are of particular concern because they can be extremely complex and prone to errors. Medical processes are excellent examples of human-intensive processes that involve many different types of medical professionals, such as doctors and nurses with different specializations and roles, pharmacists, lab technicians, and support staff. Such processes usually contain complex interactions and constraints between human agents and their collaborating hardware and software agents. These agents often perform their activities in parallel, introducing substantial amounts of concurrency and communication into the process. During the execution of the process, exceptional conditions may arise requiring specialized exception handling actions that may vary considerably depending upon the circumstances. Dealing with such concerns can make human-intensive processes extremely complex, presenting a great challenge for analysis.

Many human-intensive processes, like medical processes, are used in critical, including life-critical, situations. Even a small defect in such a process may lead to hazards that threaten human life. A hazard is “a state or set of conditions of the system that, together with certain other conditions in the environment, will lead inevitably to an accident” [80]. One fundamental requirement of developing or improving such a process, therefore, is to prevent or control the potential hazards. A variety of analysis techniques have been developed to identify potential hazards, assess their effect, and identify and evaluate the causal factors that could lead to the hazards [80]. Some of these have achieved considerable success lately in analyzing complex hardware systems, software systems, or processes in certain manufacturing industries. We believe that these analysis techniques can also be employed effectively to analyze life-critical human-intensive processes. We also believe that some of these analysis techniques may be automated if the process is specified in a language that has precisely defined semantics.

In this thesis, we investigate how two selected analysis techniques, *Finite-State Verification (FSV)* and *Fault Tree Analysis (FTA)*, can be automated and applied to analyze human-intensive processes. Both of these are static analysis techniques, in that they are performed without actually executing the system. Such static analysis techniques can be very costly for large systems. *Finite-State Verification (FSV)* has been successfully applied to analyze complex hardware and software systems. It is able to systematically detect errors by exhaustively checking all possible executions of a system against a property that formally defines a specific aspect of the system behavers. When the property is violated, a violation trace will usually be provided to show how the violation occurs. *Fault Tree Analysis (FTA)* is widely used in many industries, including the nuclear industry and the aerospace industry. It is used to produce fault trees that systematically identify and evaluate all possible events in a system that could lead to given hazards. Once a fault tree has been derived, qualitative and quantitative analysis can be applied to provide information that can be used to identify weakness in the process. For instance, *Minimal Cut Sets (MCSs)* can be computed to show sets of events that are sufficient to cause the hazard to occur.

We developed an analysis framework that is capable of performing both FSV and FTA on processes specified in a process definition language with precisely defined semantics. For FSV, we developed a translation-based approach that is able to take advantage of existing FSV tools. The process definition is first translated into the input model of the selected FSV tool. The property is also translated into the property representation accepted by the FSV tool. Then the FSV tool is executed to verify the model against the property representation. For FTA, we proposed a template-based approach to derive the fault tree from the process definition. We also provided support for computing MCSs and generating representations that help process developers to easily understand MCSs. This design of this process analysis framework is independent of the process definition language. We implemented and evaluated this framework using the Little-JIL process definition language. Little-JIL is a general process definition language well suited to capture the full complexity inherent in human-intensive processes. It provides support for abstraction, composition, and restricted, well-formed control flow constructs that are able to capture the rich control models needed for human behaviors. It also has extensive support for exception handling, a major aspect of many processes, particularly human-intensive processes. Furthermore, Little-JIL has well-defined semantics that facilitate the two kinds of analysis that we want to apply. Our contributions can be summarized as follows.

- To make human-intensive process analysis techniques easier to apply and less prone to human error, we provide automated support for two of them. For FSV, translators are implemented to automatically translate process definitions as well as the associated property specifications to input representations of various existing FSV tools. For FTA, an automatic fault tree derivation algorithm has been developedd to construct fault trees from process definitions. Furthermore, *Minimal Cut Sets (MCSs)*, which are critical for evaluating fault trees, are also computed automatically.

- We proposed several optimizations to scale the analysis. For FSV, optimizations are performed during the translation, dramatically reducing the translated models and speeding up the verification. For FTA, optimizations performed during the fault tree derivation make the derived fault tree much smaller.

- To improve the usability, we provided several approaches that make analysis results easier to understand. For FSV, a counter-example trace produced by the underlying FSV tool is represented as an execution of the input model of the FSV tool. Due to the translation and optimizations used to create this model, such counter-example traces can be hard to understand. To address this issue, we developed a high-level process trace generator that generates a high-level process trace from a counter-example trace. In addition, we proposed several graphic representations to show the high-level process traces to the analysts. For FTA, the high-level process trace generator is also used in FTA to generate high-level process traces from a MCS. The high-level process trace shows how the events in the MCS can cause a hazard to happen. In addition to the high-level process trace, we also developed an approach to create a sub fault tree for a MCS to help the analyst to understand the MCS.

- We evaluated the proposed process analysis framework by applying our implementation of the framework to two real-world medical processes: an in-patient blood transfusion process and a chemotherapy process. The results showed that the framework can be used to effectively detect the defects in such real-world human-intensive processes. The FSV detected a large number of errors in the process definitions and the property specifications. More interestingly, it identified a subtle error in the chemotherapy process itself. Through FTA, we discovered a single point of failure in the chemotherapy process.

It should be noted that although discussions in this thesis are based on our implementation for Little-JIL, most of the issues as well as the approaches that we proposed to handle those issues, are also applicable to other process definition languages.

The rest of this thesis is organized as follows. Chapter 2 presents the background and related work of process improvement, process modeling, finite-state verification, and fault tree analysis. Chapter 3 describes the architecture of our translation-based process verification framework. The focus of this chapter is put on the translation and the optimizations, as well as certain related issues. Chapter 4 discusses the automatic fault tree derivation algorithm for processes. It also briefly talks about the evaluation of fault trees. Chapter 5 presents the high-level process trace generator as well as two visual representations for high-level process traces. Chapter 6 introduces the two case studies used in the evaluation of our approach, and it discusses the results and some observations based on this experience. Finally, Chapter 7 presents our conclusions and discusses potential future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

The first part of this chapter presents a brief history as well as an overview of the state-of-the-art of process improvement research and practice in the manufacturing domain and the software development domain. The second part introduces the Little-JIL language - the process definition language that we used in our research. A survey of the most influential process modeling languages from various domains is then given. The last part discusses the backgrounds and related work of two analysis techniques: Finite-State Verification and Fault-Tree Analysis.

2.1 Process Improvement

Process improvement has long been studied and practiced in manufacturing industries. Various approaches have been proposed and adopted, resulting in significant improvements in the quality and the productivity associated with these processes. Encouraged by the success of industrial manufacturing process improvement, the software development community is trying to adapt those approaches to improve the software development process. This section focuses on the process improvement in these two domains.

2.1.1 Industrial Process Improvement

In the early 20th century, the practice to improve product quality in manufacturing industries focused on product inspection. Specifically, the product quality was improved by identifying and then fixing or removing finished products with low quality. The cost, however, was also increased because of the extra effort involved in product inspection. To improve product quality without increasing the cost, the manufacturing processes had to be changed to produce fewer low quality products. This required systematic measurement and analysis of the processes. Walter Andrew Shewhart, known as the father of statistical quality control, was the first to bring statistical control into product process improvement [117]. He pointed out that degraded quality products are the results of variation in the manufacturing process. Such variation has two kinds of distinct causes: *common causes* which he called "*chance causes*", and *special causes* which he called "*assignable causes*". Common causes are factors that are constantly present in the process, and special causes are unusual conditions or factors from outside the process. To improve the processes, the first step is to eliminate the special causes and bring the process under statistical control. Once the process is in statistical control, which means that it becomes stable and predictable, analysis can be taken to identify the common causes. Then changes to the process can be made in response to those common causes to further reduce the variation. Shewhart developed the control chart to distinguish common causes and special causes. A control chart is a graphic display of the values of a selected variable in the process over time. If the process is in statistical control, the values should distribute consistently within certain ranges in the control chart. Any abnormal pattern in the control chart, therefore, suggests the presence of the special cause. Note that the control chart only indicates the existence of special causes. It does not show what special causes actually cause the abnormal patterns. The manager or the process designer would need to consider all possible conditions or factors to identify the special cause with the help of control charts. Shewhart's idea was developed into Statistical Process Control (SPC) and the control chart is still the primary tool for SPC.

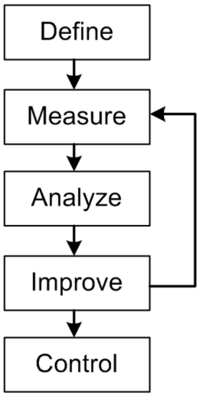


Figure 2.1. DMAIC Model

William Edwards Deming applied SPC to improve production in the United States during World War II. From 1950, he taught top management in Japan how to improve manufacturing and business processes using various statistical methods. His theory was described in his book "Out of the Crisis" [41]. The primary limitation of SPC is that it only provides a way to monitor and measure the performance of processes. It does not show how to improve the processes. Therefore, Deming proposed fourteen key principles for management to improve manufacturing and business processes. The basic idea is to apply incremental process improvement as opposed to radical re-engineering. The improvement should be based on analyzing the whole process as a system instead of focusing on optimization of some parts of the process.

In 1986, Bill Smith at Motorola introduced Six Sigma [106] to standardize the practices, including ones developed by Shewhart and Deming, to systematically improve processes. The core of Six Sigma is the DMAIC model for process improvement. DMAIC stands for the five steps performed by the process improvement team (see Figure 2.1):

- *Define* the concrete problem to be solved. Problems typically are inefficiency, high cost, or certain defects in the product.

- *Measure* process performance. Various techniques are performed to collect data/information about the existing process.

- *Analyze* the data/information to determine the root causes of the problem.

- *Improve* the process by changing the process to address the root causes. It is often necessary to iterate through the Measure-Analyze-Improve steps until the problem is solved.

- *Control* the improved process to sustain its performance.

2.1.2 Software Process Improvement

The success of the industrial manufacturing process improvement approaches encouraged people to apply them to software development processes. Those approaches, however, were not directly applicable because the software development process has some characteristics that distinguish it from the traditional manufacturing processes. The software development process involves many intellectual design activities as well as communication and coordination of humans. Therefore, many approaches have been proposed to improve software process development. Among these, the most widely known and used are the Capability Maturity Model (CMM) [97] [98], which has been superseded by the Capability Maturity Model Integration (CMMI) [125] [124], and the Software Process Improvement and Capability dEtermination (SPICE) [44].

Realizing that the quality of a software product is mainly determined by the software development process used to produce it, the Software Engineering Institute (SEI) at Carnegie-Mellon University developed CMM to evaluate the maturity of software development processes. The CMM defines five maturity levels for software development processes [97]. Processes with a higher maturity level are considered to have higher predictability of timely production of high quality software products.

1. *Initial* The software products are developed in an ad hoc, sometimes even chaotic, way. Success depends on individual effort and is unpredictable.

2. *Repeatable* For a project, basic project management processes are used to keep track of cost, schedule and functionality. This enables the repetition of earlier successes on projects with similar applications.

3. *Defined* A standard software process is defined. Management and engineering activities in this process are documented, standardized and integrated. All projects are developed and maintained by an approved and tailored version of this standard process.

4. *Managed* Both the software process and the product are quantitatively measured and controlled.

5. *Optimizing* Once the process is in control, continuous process improvement can be implement to remove defects or incorporate new ideas and technologies based on the quantitative feedback.

Except for Level 1, each maturity level is associated with several predefined Key Process Areas (KPA). KPAs identify the issues that must be addressed to achieve a maturity level. For example, to achieve level 4, two KPAs, Software Quality Management and Quantitative Process Management, must be satisfied. For each KPA, CMM provides several Key Practices that "describe the infrastructure and activities that contribute most to the effective implementation and institutionalization of the key process area." [97]. These activities reflect best practices that successful projects have demonstrated to achieve the KPA.

The CMM is also intended to be used as a framework to guide an organization to improve its processes. First, the current status of the software development process in an organization is determined by evaluating and assessing the organization's practices in terms of KPAs at each level. The assessment also help the organization to identify the highest priority KPAs that need to be focused on to achieve a higher level of maturity. The Key Practices then can be followed to implements those KPAs.

After the adoption and usage of the CMM to improve the software development processes in many organizations, the CMM was adapted for other disciplines such as system engineering, risk management, and personnel management etc. The use of multiple models within one organization is problematic because of the overlap and inconsistencies among those models. Therefore, the CMM Integration (CMMI) Project was formed to integrate the various CMMs into a single integrated framework. Three widely adopted models, the Capability Maturity Model for Software (SW-CMM) V2.0 (Draft C) [70], the Systems Engineering Capability Model (EIA/IS 731) [13], and the Integrated Product Development Capability Maturity Model (IPD-CMM) V0.98a (Draft) [69], are integrated into the first version of CMMI. The CMMI framework was designed to be flexible enough to support the integration of models for other disciplines in the future.

SPICE described in ISO/IEC 15504 was developed by the Joint Technical Subcommittee between ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission). It is a software process assessment and improvement framework similar to CMMI. Compared with CMMI, the SPICE framework not only defines five capability level for processes, but also explicitly distinguishes different kinds of processes. SPICE divides processes into five process categories: customer-supplier, engineering, supporting, management, and organization. Another difference is that SPICE focuses on software development processes while CMMI is more generic.

CMM and SPICE are basically best practices based approaches where the best practices only reflect common practices used in previous successful projects. Careful adoption of such best practices might indeed increase the chance that a project will be successful. Instead of being an honest self assessment method, however, maturity or capability levels are often misinterpreted as the rank of a company. Therefore, without careful investigation, some companies spend a lot of effort to adjust their processes and produce the required documents just to fit into a higher maturity level.

2.2 Process Modeling

Process modeling plays a critical role in process improvement. A well specified process model facilitates effective communication, training, and application of the process. It also provides a solid basis for further analysis to identify weaknesses and defects and to guide the improvement of the process. In our research, we choose the Little-JIL process definition language [26] to define processes because it provides supports for the complex behaviors that often arise in processes, such as concurrency and exception handling, and has well-defined semantics that facilitate the kinds of analysis that we want to apply. In addition, Little-JIL has a visual representations that can be relatively easily understood by domain experts, including those outside of computer science.

2.2.1 Little-JIL Process Definition Language

A Little-JIL process definition consists of three components, an *artifact specification*, a *resource specification,* and a *coordination specification.* The artifact specification contains the items that are the focus of the activities carried out by the process. The resource specification specifies the agents and capabilities that support performing the activities. The coordination specification ties these together by specifying which agents and supplementary capabilities perform which activities on which artifacts at which time(s). A Little-JIL coordination specification has a visual representation, but is precisely defined using finite-state automata, which makes it amenable to definitive analyses. Among the features of Little-JIL that distinguish

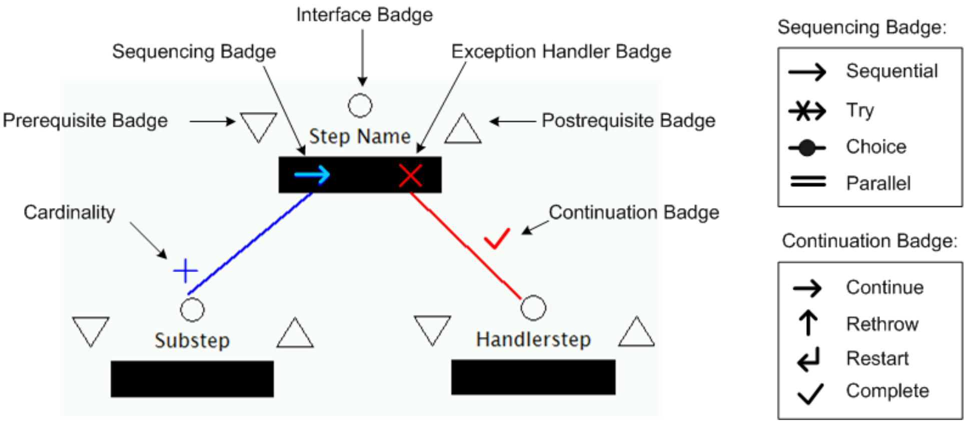


Figure 2.2. Little-JIL Step

it from most process languages are its 1) use of abstraction to support scalability and clarity, 2) use of scoping to make step parameterization clear, 3) facilities for specifying parallelism, 4) capabilities for dealing with exceptional conditions, and 5) clarity in specifying iteration. The analysis capabilities developed for this thesis use the information in the coordination specification, so only this aspect of Little-JIL is defined here.

A coordination specification consists of hierarchically decomposed steps (see Figure 2.2), where a step represents a task to be done by an assigned agent. Each step has an *interface* specifying its input/output artifacts, the resources it requires, the exceptions it throws, etc. A step with no sub-steps is called a *leaf step* and represents an activity to be performed by an agent, without any guidance from the process.

Non-leaf Little-JIL steps may be decomposed into two kinds of substeps, *ordinary substeps* and *exception handlers*. Ordinary substeps define how the step is to be executed and are connected to their parent by edges that may be annotated by specifications of the artifacts that flow between parent and substep and also by cardinality specifications. *Cardinality specifications* define the number of times the substep is instantiated, and may be a fixed number, a Kleene *, a Kleene +, or a Boolean expression (indicating whether the substep is to be instantiated). *Exception handlers* define how exceptions thrown by the step's descendants are handled.

A non-leaf step has a *sequencing badge* (an icon on the left of the step bar; e.g., the right arrow in Figure 2.2) that defines the order of substep execution. For example, a *sequential step* (right arrow) indicates that substeps execute from left to right. A *parallel step* (equal sign) indicates that substeps execute in any (possibly interleaved) order, although the order may be constrained by such factors as the lack of needed inputs. A *choice step* (circle slashed with a horizontal line) indicates a choice among alternative substeps. A *try step* (right arrow with an X on its tail) indicates the sequence in which substeps are to be tried as alternatives.

A Little-JIL step can be optionally preceded or succeeded by a *pre-requisite,* represented by a down arrowhead to the left of the step bar, or a *post-requisite,* represented by an up arrowhead to the right of the step bar. Pre-requisites check if the step execution context is appropriate before starting execution of the step, and post-requisites check if the completed step execution satisfied its goals. The failure of a requisite triggers an exception.

Channels are message passing buffers, directly connecting specified source step(s) with specified destination step(s). Channels are used to synchronize and pass artifacts among concurrently executing steps.

In a Little-JIL process, each step is defined exactly once. A step, however, may be used multiple times. These uses are represented by step references. A step reference is represented as a step bar without arrowheads that represent pre-requisite and post-requisite.

Although originally developed to define software development and maintenance processes, Little-JIL can also used to define process in other domains. It has been applied to define medical processes [65] [28], labor-management negotiation processes [93], election processes [109], and scientific data processing processes [21]. The work

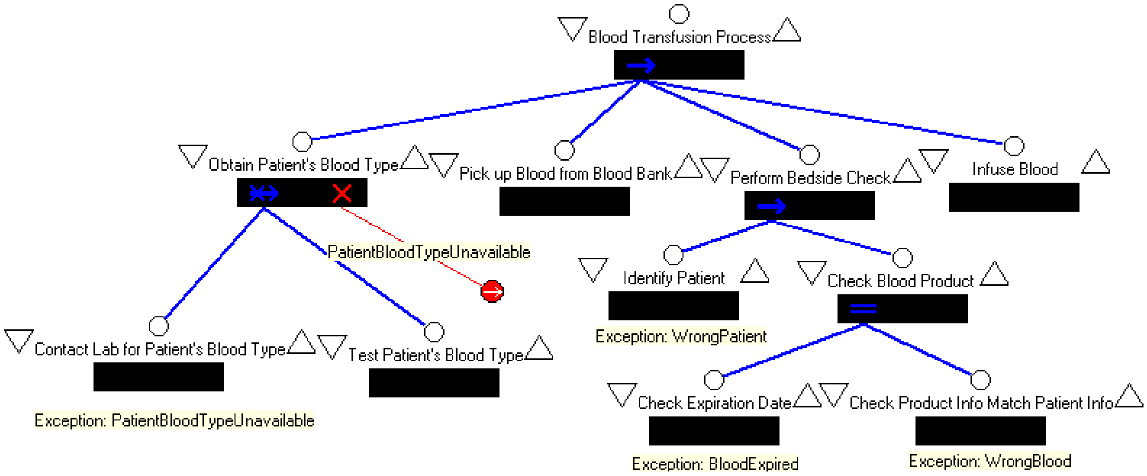

Blood Transfusion Process
Obtain Patient's Blood Type
Pick up Blood from Blood Bank
Perform Bedside Check
Infuse Blood
PatientBloodTypeUnavailable
Identify Patient
Check Blood Product
Exception: WrongPatient
Contact Lab for Patient's Blood Type
Test Patient's Blood Type
Check Expiration Date
Check Product Info Match Patient Info
Exception: PatientBloodTypeUnavailable
Exception: BloodExpired
Exception: WrongBlood

Figure 2.3. Simplified Blood Transfusion Process

in each domain has exposed several inadequacies of Little-JIL, such as the lack of good support for specifying timing constraints and transactions, but has confirmed the general applicability of this language.

Little-JIL is chosen as the process definition language in the implementation of our process analysis framework because it is well suited to capture the full complexity inherent in human-intensive processes that our analysis framework targets. It provides support for abstraction, composition, and restricted, well-formed control flow constructs that are able to capture the rich control models needed for human behaviors. It also has extensive support for exception handling, a major aspect of many processes, particularly human-intensive processes. Furthermore, Little-JIL has well-defined semantics that facilitate the two kinds of analysis that we want to apply.

Figure 2.3 shows the coordination specification of a simplified blood transfusion process defined in Little-JIL. The root step "*Blood Transfusion Process*" is a sequential step, meaning that its four sub-steps need to be carried out in order from left to right. The step "*Obtain Patient's Blood Type*" is a try step. When this step is started, its first sub-step "*Contact Lab for Patient's Blood Type*" will be executed. If the patient's blood type is already presented in the lab's database, "*Contact Lab* *for Patient's Blood Type*" is completed. This also completes the parent step "*Obtain Patient's Blood Type*". The nurse can then go ahead to pick up the blood product from the blood bank. On the other hand, if the patient's blood type is not available, an exception *PatientBloodTypeUnavailable* will be thrown. In the parent step "*Obtain Patient's Blood Type*", there is an exception handler that is able to handle exception *PatientBloodTypeUnavailable.* The exception handler has a continue control flow badge represented by the right arrow, which indicates that the next sub-step of "*Obtain Patient's Blood Type*" need to be executed to test patient's blood type. Once the blood product is picked up, step "*Perform Bedside Check*" is executed to first identify the patient and then check the blood product. If the patient is not the one who needs the blood, "*Identify Patient*" throws an exception *WrongPatient.* The parent step "*Perform Bedside Check*" does not have an exception handler to handle this exception, so it will be terminated and re-throw this exception. Since the root step also does not have an appropriate exception handler, exception *WrongPatient* eventually terminates the whole process. Since step "*Check Blood Product* " is a parallel step, its two sub-steps may run in parallel. If the blood product is expired, step "*Check Expiration Date*" throws an exception *BloodExpired.* For step "*Check Product Info Match Patient Info*", an exception *WrongBlood* will be thrown if the info (i.e. patient's name, patient' birthday, and patient' blood type) on the blood product tag does not match the patient's info. Similar to exception *WrongPatient*, *BloodExpired* and *WrongBlood* will also terminate the whole proces because they do not have appropriate exception handlers. After "*Perform Bedside Check*" completes, step "*Infuse Blood*" is executed to infuse the blood product to the patient. This process will be used as a running example in the discussion of the implementation of our process analysis framework.

2.2.2 Other Process Modeling Languages

A wide range of process modeling languages and formalisms have been proposed both in various domains. Here, we only provide an overview of several most widely used languages.

2.2.2.1 BPEL

Business Process Execution Language (BPEL) [11] is a business process modeling language developed by IBM, BEA, SAP, Siebel and Microsoft, and contains features from two previous language: IBM's Web Service Flow Language (WSFL) [81] and Microsoft's XLANG [129]. BPEL is designed to specify business processes built on applications encapsulated in web services. It provides various constructs borrowed from imperative programming languages, such as Java, to glue the distributed web services together to achieve certain business goals.

A BPEL process model is a composition of *activities*. An activity is either a *basic activity* or a *structured activity*. A set of predefined basic activities are used to define the atomic behaviors of a process. For instances, an *invoke* activity invokes an operation of a service (either synchronously or asynchronously), a *receive* activity simply waits for a matched message from a service, and a *reply* activity sends a message to a service. The other basic activities include *assign* (assign message data), *wait* (wait for a certain period of time), *empty* (just do nothing), *throw* (signal faults), *terminate* (terminate the whole process) etc.

A structured activity is composed of a set of activities and defines the control flow on those activities. BPEL provides several kinds of structured activities to capture the most commonly used control flow patterns, including *sequence* (sequential execution), *flow* (parallel execution), *switch* (conditional branching), *pick* (non-deterministic choice), and *while* (iterative execution).

In BPEL, a scope can be defined to wrap an activity. During the execution of the activity contained in the scope, faults might be triggered either by the *throw* activities or *invoke* activities. A fault handler can be attached to the scope to intercept and handle specific kind of faults occurring within the scope. In some situations, to keep the process in a consistent state, a fault hander may require certain completed activity in another scope to be reversed. To achieve this, a compensation handler can be attached to the scope wrapped around that activity. The compensation handler will be invoked to reverse the completed activity when a *compensate* signal from the fault handler is received.

BPEL processes are specified using XML and can be executed by an execution engine. XML is designed to facilitate data passing and processing between computers, but it is difficult for humans to write and understand directly. To solve this problem, the Business Process Modeling Notation (BPMN) [61] is proposed to provide a standardized graphic notation for BPEL. A process in BPMN is represented as a Business Process Diagram (BPD), which is basically a flow chart with notations tailored for elements in BPEL. Like many other flow chart representations, a BPD tries to display everything in the process, including message passing, event handling, and control-flow etc. For a large process, the BPD quickly becomes too complicated to specify and manage. Another problem with BPMN is that a BPD has to be translated into BPEL to be executed. Since formal semantics of the BPMN are not defined in the BPMN specification, many subtle details are actually decided by the translator. Therefore, the resulting BPEL process might not accurately reflect the intention of the process designer.

As mentioned above, BPEL is primarily designed to define business processes based on web services. It does not support processes that involve user interaction. An BPEL extension BPEL4People [77], therefore, is proposed to incorporate human activities into the process definition.

2.2.2.2 UML

The Unified Modeling Language (UML) [63] is a visual specification language for modeling the object-oriented(OO) software design. It is intended to unify the different OO design modeling approaches and serve as a standard modeling notation for software developers to document and communicate the design. The first version of UML was based on Booch's method [20], the Object Modeling Technique [113] and the Object-Oriented Software Engineering method [73]. After that, more individuals and organizations joined the development and many new features have been incorporated. It is currently an independent standard developed and maintained by the Object Management Group(OMG).

Instead of defining one single notation that can be used to model all aspects of the software system design, UML offers various types of diagrams to model different facets of the design. *Structural diagrams*, such as *class diagrams*, *component diagrams*, and *object diagrams*, can be used to model the static structures of the system (e.g. objects, operations, and relationships). *Behavior diagrams*, such as *statechart diagrams*, *activity diagrams*, and *sequence diagrams*, are able to specify the dynamic behavior of the elements in the system. Details about those diagrams can be found in [63]. Since each UML diagram only provides a specific view of the entire system, a complete model of a system in UML need to include a collection of diagrams. On the other hand, those UML diagrams are not designed to be orthogonal. The same elements or behavior may be modeled using different diagrams. Therefore, not all of the standard UML diagrams are required to be used in the model.

Because of its rich notations as well as flexibility and extensibility, UML is not restricted to modeling the design of OO software system. For instance, SysML [62] is a customized subset and interpretation of UML used for systems engineering modeling. People have also used UML as a process modeling language to model processes for different domains. In [68], Hruby explained how to use UML to define business processes by mapping various business workflow concepts to UML concepts. For example, business objects can be mapped to classes and objects in UML, and workflows can be modeled using the UML sequence diagram and the UML activity diagram. In [78] and [92], clinical practice guidelines or processes are modeled using UML activity diagrams complemented by other diagrams such as class diagrams. The UML activity diagram is a variation of a control flow graph that shows the overall flow of control in the system.

There are several problems that arise when using UML as a process modeling language. Since a process model will probably need to be represented using several different kinds of UML diagrams, it might be problematic to ensure the completeness and consistency of the overall model. In addition, the semantics of the UML activity diagram, which is used by many approaches to represent the process logic, is still described informally in the current version of UML. In fact, a few issues have been found during attempts to formalize the semantics of the activity diagram(e.g. [122] and [49]).

2.2.2.3 Petri Nets

Invented in 1962 by Carl Adam Petri in his dissertation [102] (English translation [103]), the Petri net is a well known formalism for the specification and analysis of concurrent, discrete-event systems. It provides graphical annotations that have precisely defined semantics. This facilitates the rigorous analysis of the Petri net models. Many Petri net based approaches have been proposed to model software development processes, such as SLANG [16], FUNSOFT Nets [47], and SoftPM [88].

SLANG [16] is a Petri net based software process modeling language developed at Politecnico di Milano and CEFRIEL. It is a part of a software process environment SPADE [15] that supports software process design, analysis, and enactment. SLANG is based on Environment/Relationship (ER) net [58], which is a high-level extension of the Petri net formalism.

A SLANG process specification consists of *Process Type* definitions and *Process Activity* definitions. Process types define the static structure of the process. The types of all artifacts used in the process are described in a similar way with the class description represented with an object-oriented type hierarchy. Process activities define the dynamic behavior of the process. Each process activity definition consists of an interface and an implementation. An activity interacts with the other activities in the process through its interface, which contains a set of starting events and a set of ending events. Those events are represented as *transitions*. An activity is invoked when one of the starting events occurs and produces an ending event when it finishes. The interface also includes a set of *places* that are used for parameter passing. The implementation of an activity then defines how input parameters are transformed into output parameters using ER nets.

One interesting feature of SLANG is that it supports modeling the dynamic changes of a process. Activities in SLANG are treated as artifacts and can be manipulated by other activities. Therefore, activities can be changed or created on the fly.

Although Petri net approaches like SLANG are highly expressive, they are awkward in specifying some common aspects of processes, such as exception handling and resource management. Furthermore, graphical annotations of Petri nets are very low-level. It is difficult to specify and understand complex processes using such notations, even for experts.

2.2.2.4 Rule-Based

Rule-based process modeling languages, such as the one used in MARVEL [76], are based on features from languages used for knowledge-based systems. MARVEL is a software development environment that provides automated execution support for the developers.

In MARVEL, objects used by a software development process (e.g. modules, procedures, and design documents) are stored in a database called the *object base*. The types of objects are defined as an object-oriented type hierarchy. A type definition of a object consists of a set of attributes indicating status of the object and activities that can be performed on the object.

The software development process is defined as a set of rules. Each rule consists of an *activity*, a *precondition*, and a *postcondition*. The activity represents a development task that may be performed by a tool (e.g. compile a module) or by a developer (e.g. edit a module). An activity will be executed only if the associated precondition is satisfied. The execution of the activity may have different results. The effects of these results are captured by the postcondition. The postcondition consists of several mutually exclusive assertions. Each assertion reflects the effect of one result. For example, compiling a module may succeed or fail. Therefore, the postcondition contains two assertions. One assertion indicates that the *compiled* attribute of the module must be true if the compilation succeeds. The other asserts that the *error* attribute of the module has to be true if the compilation fails. Both the precondition and the postcondition are first-order predicates in terms of the object attributes.

A process is executed through forward chaining and backward chaining. Forward chaining is applied when a rule finishes. Depending on the result, one assertion in the postcondition of the rule becomes true. This triggers the runtime environment to evaluate preconditions that might be affected by that assertion. Rules with preconditions being satisfied are enabled and will be executed. Backward chaining is triggered when a user attempts to invoke a rule whose precondition is not satisfied. The runtime environment identifies rules whose postconditions might satisfy this precondition and attempts to execute them. These rules may in turn cause further backward chaining. This procedure continues until the precondition of the rule invoked by the user is satisfied, or a notification is reported to the user indicating the rule cannot be executed.

In rule-based process modeling languages like MARVEL, the control flow of a process is implicit. Therefore, they do not have the problem of defining uncommon flow of control as in many other languages. This, however, is also a major weakness. It is difficult for users to specify and understand a process defined as a set of such rules.

2.2.2.5 Medical Guideline Modeling Languages

Medical guidelines (or medical protocols) are emerging as an important technology to increase the quality of heath-care in modern medical practice. Derived from up-to-date best practice, medical guidelines are essentially processes that provide medical practitioners with decision options and respective courses of actions about appropriate health-care for specific circumstances. In recent years, a large number of medical guidelines have been published (e.g. [5] [6]). The problem is that these guidelines are usually informally described and are often ambiguous, incomplete, and inconsistent. To address this problem, a large number of medical guideline specification languages as well as supporting systems have been developed to specify, manage, analysis, and execute medical guidelines (e.g. the Arden Syntax [121], GLARE [128], GLIF [22] [100], Asbru [116], PRO*forma* [123], EON [91], GUIDE [107], and PRODIGY [75]). The OpenClinical website [7] gives a relatively complete list of the most influential approaches, and the paper [99] compares several major approaches. Here, we focus on the medical guideline modeling languages used in a few selected approaches.

GLARE is a medical guideline acquisition, representation and execution system developed by Terenziani at University of Piemonte Orientale A. Avogadro, Italy [128]. A medical guideline specification in GLARE is defined in terms of *actions*. GLARE distinguishes between *atomic actions* and *composite actions*. *Atomic actions* are elementary steps in a guideline. There are four types of atomic actions: *work actions* (basic tasks performed by physicians), *query actions* (data acquisition), *decision actions* (decisions to choose alternatives), and *conclusion actions* (output of a decision action). *Composite actions* are composed of sub-actions (atomic or composite). The execution order of the sub-actions are specified using the pre-defined control relations: *sequence*, *controlled*, *alternative*, and *repetition*. The controlled relation allows the user to define flexible temporal constraints, such as "A during B", "start of A at least 1 hour after the beginning of B", etc. These flexible temporal constraints could introduce inconsistency into the medical guideline specification. To solve this problem, a temporal reasoning algorithm is used to check the consistency of the specification. GLARE also provides simple failure and exceptions handling mechanism, such as restart, abort, and suspend. To enhance the user-friendliness, a visual editor was developed for users to specify the guideline. In the editor, the structural decomposition of a guideline is displayed by a tree and the control flow is represented by a variation of a flowchart. Compared to Little-JIL, GLARE lacks explicit agent specifications and adequate artifact flow specifications. Its exception handling mechanism is also relatively weaker than the one in Little-JIL.

GLIF (GuideLine Interchange Format) is a structured medical guideline representation language developed by the InterMed Collaboratory at Stanford University [22] [100]. A GLIF guideline contains an *Algorithm*, represented as a flowchart that captures the temporal ordering of clinical steps. There are four kinds of steps in GLIF: *action steps* (representing tasks to be performed by medical professionals), *decision steps* (indicating decision points to choose alternatives), *branch and synchronization steps* (modeling multiple simultaneous paths), *patient-state step* (explicitly characterizing patient's clinical state). The action steps can be nested steps that contain subguidelines. An expression language is provided for representing decision criteria, triggering events, exceptional conditions, and states. In the most recent version, GLIF3, a guideline can be specified in three levels of abstraction: the *conceptual level*, the *computable level*, and the *implementable level*. The *conceptual level* focus on the control flows of the guideline, represented as a flowchart. Many details, such as decision criteria, relevant patient data, and iteration information are added to the specification at the *computable level*, which allows the validation and simulation of the guideline. At the *implementable level*, actions and data in the guideline are mapped to existing procedures and electronic medical records in a specific institute so that the guideline can be incorporated into the institutional information system. The main problem with GLIF is that the flowchart representation is cumbersome for capturing large guidelines with complex exceptional control flows.

Asbru is a medical guideline specification language used in the Asgaard project, which was collaboratively developed at Ben Gurion University and the Vienna University of Technology [116]. A guideline in Asbru is called a “plan”. Similar to many other languages, a plan can be further decomposed into sub-plans. There are three kinds of decomposable plans: *sequential*, *concurrent*, and *cyclical*. A plan that is not decomposed needs to be executed by the user or by an external call to a computer program. The semantics of a plan is rigorously defined by a state-transition system. One important feature of Asbru is that it provides rich real-time annotations that enables the expression of uncertainty in the starting time, ending time, and duration of a time interval, with the use of absolute, relative, and even cyclical reference annotations. This uncertainty allows a plan to capture the essence of a guideline but leaves room for execution-time flexibility. Since Asbru was designed to be a text-based, machine-readable language, it is difficult for humans to understand. Another limitation is that it lacks support for exception handling.

PRO*forma* [123] was developed at the Advanced Computation Laboratory of Cancer Research, UK. The basic building block of a guideline definition in PRO*forma* is the task. A task specification includes title, identifier (name and parameters), description, goal, trigger conditions, pre-conditions, post-conditions, and etc. PRO*forma* supports four types of tasks: *plan*, *decision*, *action*, and *enquiry*. A *plan* is a composition of sub-tasks. The tasks in a guideline are organized hierarchically into plans. A *decision* is a point at which certain decision needs to be made to determine an alternative to take. An *action* represents some procedure that needs to be carried out by a medical professional or a computer system. An *enquiry* represents a point at which data needs to be acquired. A flowchart-based graphic representation is also provided to enhance the uder-friendliness. Like other flowchart-based representation, this approach is not intuitive for modeling large guideline with complex exceptional control flows.

2.3 Finite-State Verification

In this section, we first give a brief introduction to finite-state verification. Then we discuss several existing finite-state verification tools. Finally, we present selected approaches that apply finite-state verification techniques to analyze business processes and medical processes.

2.3.1 Introduction to Finite-State Verification

Finite-state verification(FSV) is a static analysis technique that has been successfully applied to detect errors in complex software systems. Such systems often contain complex control flows and data flows among different components. For concurrent systems, the system behavior could be more complicated because even with the same input, the non-determinism of task scheduling may lead to different orders in which instructions from different tasks are executed, causing the concurrent system to produce different results. Subtle errors in such systems often involve intricate interactions between various components. In addition, some errors only manifest themselves under a few specific schedules. Therefore, it is extremely difficult to find such errors. Testing is the most widely used method for identify errors in software systems. For most systems, however, only a tiny fraction of the possible executions can be tested. FSV, on the other hand, is able to systematically detect certain kind of errors by exhaustively checking all possible executions against a given property.

FSV involves a finite-state model, a property, and a verification engine. The model, which is constructed from the system of interest, represents all possible relevant executions of the system. A property specifies an invariant that should always hold in the system. FSV is able to verify a large class of properties called temporal properties. A temporal property, which prescribes the temporal ordering of certain events in the system, can be rigorously defined using formalisms including Linear-time Temporal Logic(LTL) [105], Computation Tree Logic (CTL) [32], and automata [43]. Although slightly different from each other in terms of expressivity, these formalisms are all able to capture most of the *safety properties* and the *liveness properties*. Informally, a safety property specifies that something bad never happens during execution of a system. A liveness property, on the other hand, states that something good will eventually happen. It is important to note that one property only defines a partial specification of system's correctness. Thus, verification of a system usually requires a set of properties. Given a property and a model, the verification engine explores all executions through the model and checks whether the property holds on each execution. If the property does not hold, a counter-example trace through the model is produced to show a scenario where the property is violated. By studying the counter-example trace, the analyst can pinpoint the source of the error in the system. The analyst then can fix the error and try again. Note that finite-state verification is only able to identify errors with respect to the given properties. It cannot guarantee the complete correctness of the system. Errors may remain undetected if their corresponding property is not provided. The benefit of FSV is that by ensuring that the system satisfies the important and relevant properties, we increase our confidence in the correctness of the system.

The major issue faced by FSV is the state explosion problem. During verification, the explored states of the system need to be stored so that they will not be visited again. This prevents the FSV from being applied to large systems that have extremely huge state spaces. To attack this problem, many optimizations have been proposed, such as abstraction [40] [60], partial order reduction [33] [59], and symbolic methods [87]. It has been shown that FSV can often scale up to handle a lot of very large software systems with the help of these optimizations.

Similar to the software systems, real world processes often tend to be complex, concurrent, and exception-prone. This suggests that FSV might as well be effective in identifying errors in processes.

2.3.2 Finite-State Verification Tools

A variety of finite-state verification tools have been developed to verify software systems. Here, we discuss several selected approaches that exploit different algorithms for verification to give an idea of the range of approaches.

2.3.2.1 SPIN

SPIN is a widely used distributed system verification tool based on reachability analysis [67]. It accepts system models specified in a modeling language called Promela (Process Meta Language). A Promela model consists of a set of processes that communicate synchronously or asynchronously with each other using channels and global variables. The property to be verified can be written in standard Linear Temporal Logic (LTL) [105]. To verify the property, SPIN performs a Depth First Search (DFS) to visit each state that is reachable from the initial state. States explored during the search are stored so that they will not be visited again. To speed up the verification and save the memory, SPIN provides several optimizations such as partial order reduction, state compression, and bitstate hashing.

2.3.2.2 SMV

Instead of representing states explicitly, the Symbolic Model Verifier (SMV) [87] uses a data structure called Binary Decision Diagrams (BDD) to compactly encode a set of states. A BDD [25] is a directed acyclic graph that denotes a Boolean function, a compact representation of a set of the states that correspond to the assignments that make the function true. In addition, the transition relations can also be represented as Boolean functions and hence be encoded using BDDs. The verification, therefore, can be performed using efficient BDD operations. SMV is able to check properties expressed in a branching-time temporal logic called Computation Tree Logic [32]. The Cadence SMV [2], which is an extension of SMV, supports a more expressive modeling language and compositional verification. Another extension called NuSMV combines BDD-based model checking and SAT-based model checking [30].

2.3.2.3 FLAVERS

FLow Analysis for VERification of Systems (FLAVERS) is a finite-state verification tool developed to verify event-based, behavioral properties of concurrent systems [43]. In FLAVERS, the system is automatically abstracted into a concise event-based model that explicitly represents inter-task communication, synchronization, and event orderings. Then a polynomial-time, data-flow analysis algorithm is applied to determine whether the system satisfies a given property defined as a finite-state automaton. Unlike many other approaches that directly verify the complete system model, FLAVERS exploits an incremental procedure through which analysis of increasing precision can be constructed. First, a very small abstract model is constructed and verified. This abstract model *might* be adequate to verify the property. If the counter-example trace produced is an infeasible trace, the current model is not precise enough. In this case, the counter-example trace can be used to select a constraint to refine the model, and the refined model is verified again. This procedure continues until no violation of the property is found, or a feasible counter-example is reported. Usually, properties can be verified without using the complete system model. Therefore, this approach often seems to scale quite successfully.

2.3.2.4 INCA

The Inequality Necessary Conditions Analysis (INCA) applies integer linear programming techniques to the verification of concurrent programs [37]. In this approach, the system model is turned into a set of inequalities that describe the legal flow through the system. The complement of the property, which is supposed to hold for the system, is also represented as an inequality. Then integer linear programming techniques can be applied to determine whether there exists a solution for those inequalities or not. A computed solution indicates a violation of the property because it represents a legal flow through the system that also satisfies the complement of the property. Although the integer linear programming has an exponential complexity in the worst case, INCA is often able to verify large systems because the inequalities generated are usually extremely simple.

2.3.2.5 LTSA

The Labeled Transition System Analyzer (LTSA) was designed to modeling and analyzing the behavior of systems represented by labeled transition systems (LTSs) [84]. In LTSA, a concurrent system is modeled by a set of interacting processes, where a process is described textually as a finite-state process (FSP) that consist of sequences of events. The behavior of the concurrent system is captured by the composition, or cross-product, of all FSPs in the system. A property in LTSA is specified as an FSP in the same way as a process except for the additional ERROR state. Any sequence of events that reaches the ERROR state indicates a violation of the property. Checking for violation of a property is performed on-the-fly while computing the composite FSP of all processes FSPs and the property FSP.

2.3.2.6 Bandera

As discussed above, most verification tools only accept models specified in their own specification languages (e.g. state machines, label transition systems). They cannot be directly applied to verify software systems that are written in high-level programming languages such as Java. To address this issue, Bandera [38] is proposed to verify Java program using existing verification tools such as SPIN and SMV. It employs a two-stage translation approach to translate a Java program into the input models of various verification tools. The Java program is first translated into an intermediate representation called Bandera Intermediate Representation (BIR) [36]. The BIR modeling language is a guarded-command language for describing state-transition systems and was intended to support translation into the input languages of a variety of verification tools. The BIR is further translated into the input models of various verification tools. Then the resulting input models can be verified against given properties by the corresponding verification tools. If a property does not hold, a violation trace is provided to show how the property is violated.

To enhance the scalability, optimizations such as slicing and abstraction are performed during the Java to BIR translation to remove control points, variables, and data structures that are irrelevant for checking a given property. Since the input model is considerably different from the original Java program after the translation and optimizations, the violation trace provided by the underlying verification tool is usually very difficult to understand for the analyst. To solve this problem, Bandera maps violation traces produced by the underlying verification tools back to traces through the original Java program.

At a high-level, our implementation of the process verification functionality in our process analysis framework is very similar to Bandera. Our implementation first translates Little-JIL processes into BIR models and then to input models for various verification tools. In addition, we also apply optimizations in the translation to enhance the scalability and generate high-level Little-JIL traces from low-level violation traces. A main difference is that our implementation targets Little-JIL processes instead of Java program. Because of this, the optimizations that we propose are quite different from the ones used in Bandera. Further more, we use a search algorithm to generate the high-level trace from the violation traces, as oppose to the reverse-mapping approach in Bandera.

2.3.3 Verification of Business Process

Different formalisms, such as Petri nets, finite-state automata, and process algebra, etc., have been exploited to formally define the semantics of BPEL, allowing formal verification to be applied to BPEL processes.

In [119], Stahl provided a mapping of BPEL constructs to Petri net patterns. A tool called BPEL2PN has been implemented to automatically translate BPEL processes into Petri nets [66]. While the control flow of the BPEL processes is preserved by the translation, messages and data are abstracted as black tokens. The Petri nets then can be verified using the Low Level Petri Net Analyzer(LoLA) [115]. LoLA is able to analyze various Petri net general properties, such as reachability and boundedness [50]. It also supports the verification of process specific properties specified in the branching time temporal logic CTL. As pointed out by the authors, the direct translation does not scale because the models generated contain too many details that are not relevant to the property. Therefore, it is necessary to apply optimizations in the translation to reduce the size of the models.

Ouyang et. al developed a similar approach, called BPEL2PNML, [95] that translates BPEL into the Petri Net Markup Language (PNML) [133]. This approach also focuses on the translation of the control flow of BPEL processes. The resulting model can be verified by WofBPEL tool [94]. WofBPEL is able to perform three kinds of general analysis: detecting unreachable activities, detecting multiple activities that may consume the same message, and checking whether a message may eventually be consumed.

Colored Petri Nets (CP-nets) [74] have also been used for formalizing the semantics of BPEL. CP-nets extend the ordinary Petri nets by providing high-level constructs for data manipulation, which makes a CP-nets model of a business process more compact than the corresponding Petri nets model. Yang et. al proposed a set of rules that map most of the BPEL activity constructs to the CP-nets [137]. The verification of CP-net models can then be performed using existing CP-nets verification tools.

Fu, Bultan and Su developed the Web Service Analysis Tool (WSAT) to verify BPEL processes [56]. A BPEL process is first translated to a guarded automata model. The guarded automata model is further mapped to Promela and verified by SPIN. As opposed to most other work, which only focuses on translating the control flow of the process, WSAT is able to handle data manipulation using XPath expressions. This allows the verification of properties about message content [55]. Since SPIN is a finite-state verification tool, the size of the input queues has to be bound during the translation to achieve a finite model. This usually makes the verification unsound. They proved that for certain kind of processes with unbounded input queues, the verification of a bounded model is sound [54]. They called such processes "synchronizable" processes and gave several sufficient conditions that can be used to decide whether a process is "synchronizable" or not.

Nakajima also used SPIN to verify BPEL processes [118]. First, a BPEL activity is translated into an extended finite automaton. Then, the automaton is represented in Promela. During the translation, some abstraction techniques are used to reduce the resulting model.

Arias-Fisteus, Fernandez and Kloos develop a tool called VERBUS that translate BPEL processes into finite-state machines [51]. These finite-state machines are subsequently mapped onto Promela and the input language for SMV and NuSMV.

Foster, Uchitel, Magee, and Kramer have developed the tool LTSA-WS [52], an extension of LTSA, to verify BPEL processes. The activities of BPEL are translated into FSPs and verified by LTSA.

2.3.4 Verification of Medical Processes

A few approaches have been proposed to apply formal methods, mostly finite-state verification, to improve the quality of medical guidelines.

In [126], to analyze a medical guideline, the text description of the guideline is first turned into an Asbru model with the help of domain experts. Asbru [116] is specification language specific for modeling medical guidelines. Its semantics is formally defined using Structural Operational Semantics (SOS) [104]. This allows the Asbru model to be translated to a formal representation for the interactive theorem prover KIV [14]. KIV is able to verify properties specified using a variant of interval temporal logic (ITL) [90]. If a property does not hold, a counter example is provided to show how the property is violated. This approach was applied to two real-life medical protocols, a jaundice protocol and a diabetes mellitus protocol, and several flaws in the jaundice protocol were detected during the verification. The major problem with this approach is that it is not fully automatic. Some encodings have to be performed by the analyst for the translation. Moreover, in many cases, KIV might require the analyst to provide additional assumptions to complete a proof of the property.

In [17], the Asbru model is translated into the input language of the SMV model checker and verified by Cadence SMV. A simple data abstraction is proposed to abstract the data flow and time. This allows the translation to be performed automatically. The abstraction, however, might introduce infeasible counter-example paths. If this occurs, the analyst has to manually refine the model to eliminate the infeasible counter-example paths. This work also used the jaundice protocol as an example and found errors in it.

In [127], the clinical guideline defined in GLARE [89] is translated to a Promela [67] representation and verified by the SPIN model checker. As a proof of concept paper, no details of the evaluation were presented.

2.4 Fault-Tree Analysis

2.4.1 Introduction to Fault-Tree Analysis

During execution, faults might occur in components of a system due to hardware failures, human errors, etc. Those faults might propagate through the system and eventually cause hazards to occur. In the context of safety analysis, a hazard refers to an unsafe state of the system that will lead inevitably to a serious accident if certain conditions in the environment are presented. To prevent or control the potential hazards, one needs to understand what hazards could occur in the system and how they could happen. A variety of hazard analysis techniques have been developed to identify potential hazards, assess their effect, and identify and evaluate the causal factors related to the hazards [80]. Fault-Tree Analysis (FTA) [131] is a hazard analysis technique used to systematically identify and evaluate all possible causes of a given hazard.

Given a hazard, a fault tree is produced to show all the parallel and sequential combinations of events that could lead to the hazard. The basic elements of a fault tree are *events* and *gates*. Events are used to represent faults, such as component failures, human errors, or other pertinent conditions in the system or environment. Figure 2.4 shows symbols of several commonly used events and gates. *Basic events*

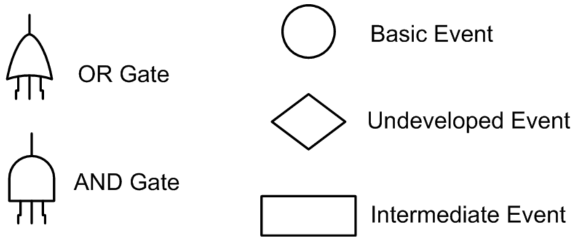


Figure 2.4. Fault Tree Elements

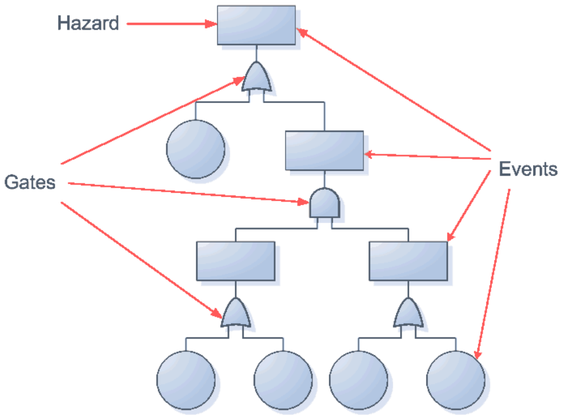


Figure 2.5. Fault Tree Example

are basic initiating faults or conditions. *Undeveloped events* are events that are not developed any further, either because necessary information for deriving the fault tree leading to these events is unavailable or because these events are considered to have insignificant consequence. Basic events and undeveloped events are also called *primary events* because they require no further development. As opposed to primary events, *intermediate events* are events that need to be developed. An intermediate event is developed by investigating the system to identify the immediate, necessary, and sufficient events that cause this event, and then connecting those events to it via a proper gate. In a fault tree, events are connected using *gates*. Each gate connects one or more input events to a single output event. The output event of an *AND gate* occurs if all of the input events occur. While the output event of an *OR gate* occurs if any of the input events occurs. Figure 2.5 shows an example fault tree.

Typically, FTA involves 3 steps:

1. *Modeling the system.* The system model is usually specified using a flow graph. It prescribes the boundary and granularity of the analysis.

2. *Constructing a fault tree.* Given the system model, the fault tree construction process starts with the TOP event, which is an intermediate event representing the given hazard. For each intermediate event, all immediate and necessary events that could lead to this event are identified and connected to this event using appropriate gates. These new events themselves may be intermediate events and need to be developed. This process continues until all leaf nodes in the fault tree are primary events.

3. *Evaluating the fault tree.* Once a fault tree has been derived, qualitative and quantitative analysis can be applied to provide information, such as specific sequences and sets of events that are sufficient to cause a hazard and overall system vulnerability to a hazardous outcome resulting from the occurrence of a particular event. This information can then be used as guidance for improvement of the design or implementation of the system.

Once a fault tree has been derived, *Minimal Cut Sets* (MCSs) for this fault tree can be computed. An MCS contains a minimal set of primary events that ensure that the hazard will occur. The MCSs may indicate certain weakness of the process and provide guidance for improvement of the design or implementation of the system. For instance, if a MCS only contains one event, the process is exposed to the single point of failure – the hazard will definitely occur if the event occurs. Therefore subsequent changes might need to be made to the process to remove these weaknesses. There exists many tools, such as *Espresso* [3], *OpenFTA* [8], and *Galileo* [4], that can be used to compute the MCSs.

2.4.2 Related Work of Fault-Tree Analysis

Fault-Tree Analysis was first developed by H.A.Watson of Bell Labs for the US. Air Force to study the Minutemen Launch Control System [48]. It then was adopted and extensively applied by the Boeing Company for designing and evaluation commercial aircrafts. Following its adoption by aerospace industry, FTA became widely used in many other industries, such as the nuclear power industry, the chemical industry, and the automotive industry.

To our knowledge, Leveson was the first to apply FTA to safety analysis of software systems. In [27], Leveson et al. proposed to use fault trees to guide the analyst to identify errors that cause an Ada program to produce incorrect outputs. The fault tree is constructed using templates, one for each kind of statements in Ada. The incorrect output of the program is represented as the top event. Then the set of causes that might occur in the previous statements are derived using the appropriate template. By inspecting the code, the analyst may eliminate some causes that are inconsistent with the corresponding statement. The remaining causes can then be further developed until the errors are identified.

Many other template-based approaches have been proposed for different languages thereafter. For instance, [53] presented a set of fault tree derivation templates for Pascal. Leveson et al. designed a fully automatic fault tree derivation tool that uses templates to generate fault trees from the Requirements State Machine Language (RSML) specifications [108]. The approach by Pai et al. is able to automatically derives fault trees from UML models [96]. This approach requires the dependency relationships to be explicitly specified in addition to the UML model. In [85], McKelvin et al. described an algorithm that produces fault trees from Fault Tolerant Data Flow (FTDF) models.

Instead of using templates, some approaches, such as [82] and [86], use model checking to generate fault trees. In these approaches, explicit state machine models need to be incorporated into the system model to represent all possible faults that may occur within components. Given a system model and a hazardous output, a model checking tool is applied to verify the system model against the property that states that the given hazardous output will never occur. The fault tree then can be constructed using the resulting violation paths. For each violation path, events in this path are connected to an intermediate event representing this path using an AND gate. Then events that represent all violation paths are connected to the TOP event using an OR gate. One problem with these approaches is that there may be an infinite number of paths. In their papers, however, the authors do not explain how their approaches deal with this problem.

CHAPTER 3

FINITE-STATE VERIFICATION

A human-intensive process often requires coordinating the efforts of many different parties who often perform their activities in parallel. This introduces substantial amounts of concurrency and communication into the process. Moreover, a human-intensive process also has to cope with various exceptional situations during the execution of the process. Handling of exceptions often leads to diversion of the normal control flow and causes the processes to be even more complicated. Such complexity increases the risk of errors in a process. Software engineers will readily note that the software development community already deals with the creation of complex procedures (e.g., complex software systems) that present a range of difficulties analogous to those found in processes. This suggests that the finite-state verification techniques used in software engineering to detect errors in complex, distributed systems might be effective in analyzing processes. In fact, many approaches have already been proposed to apply finite-state verification to processes, several of which have been discussed in the previous chapter. In these approaches, a process definition is first translated into the input model of an existing finite-state verification tool. Then the verification tool is executed to verify the model against given properties. The existing process verification approaches that we have investigated have the following issues.

- **Flexibility:** Most approaches translate process definitions to models of a single selected verification tool. Since each existing verification tool has its own strengths and weaknesses, being tied to a specific tool limits the verification capability of these approaches.

- **Scalability:** Many approaches only demonstrate the feasibility of applying finite-state verification to analyze processes. They do not address the state-explosion problem that is frequently encountered in verifying complex human-intensive processes. Such approaches will frequently fail to work as the processes become more complicated.

- **Usability:** Most approaches do not pay much attention to usability issues. Since the process definition is translated into the input model of the underlying verification tool, to verify the process, the properties need to be specified in terms of constructs in the translated model. This requires the users to have a fair understanding of the translation. Moreover, when a property is violated, the verification tool will produce a violation trace that shows how the violation occurs. Such violation traces are hard to understand due to the significant difference between the process definition and the translated input model.

To address these issues, we proposed a general process analysis framework that can be adapted to verify processes defined in different process definition languages using different finite-state verification tools.

3.1 Process Verification Framework

Figure 3.1 shows the architecture of the general process verification framework. At a high-level this framework works as follows. It first translates the given process definition into the model recognized by the underlying verification tool. During the translation, several optimizations are applied to reduce the model. Similarly, the property is also translated into the representation fitting with the verification tool. The underlying verification tool is then invoked to verify the translated model against the translated property. If the property does not hold for the process, the verification tool will produce a violation trace, showing how the property is violated. This low-

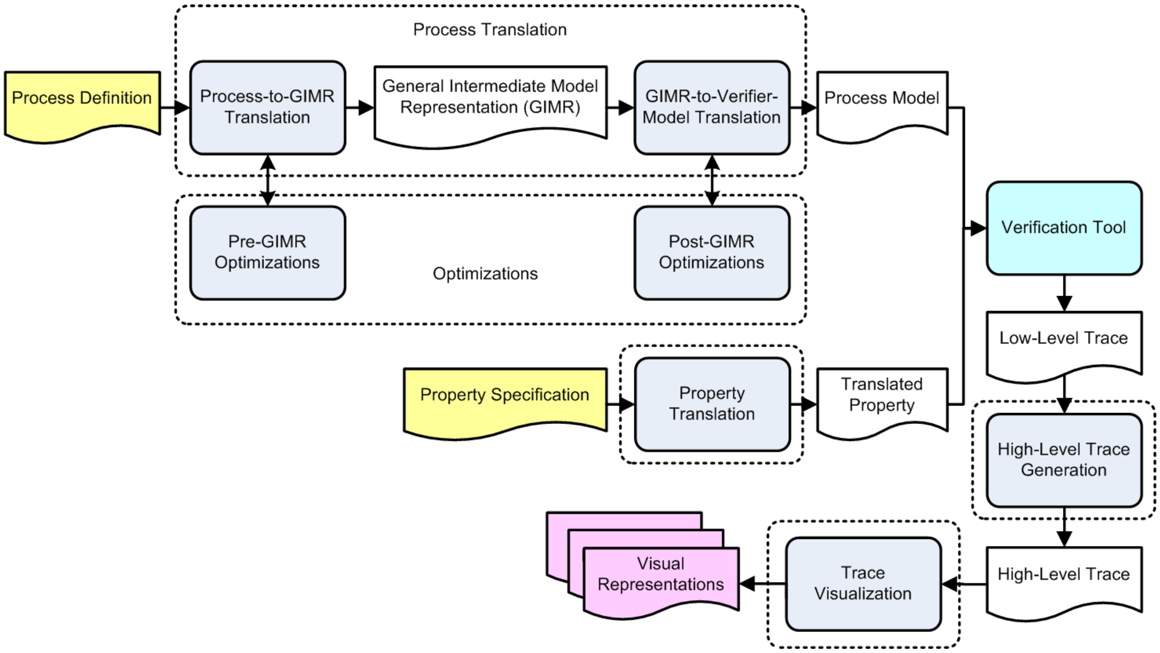


Figure 3.1. Process Verification Framework

level violation trace is automatically turned into the corresponding high-level trace through the original process, which is presented to the analyst via various graph representations.

At a high-level the framework can be viewed as consisting of five components, as indicated by dashed rounded rectangles in Figure 3.1, each intended to address one or more of the issues listed above. These components are:

- **Process Translation** A two-stage translation approach is employed to address the flexibility issue. The process definition is first translated into a *General Intermediate Model Representation (GIMR)*. The GIMR is then translated into the input model of the selected verification tool. The GIMR is a low-level representation that contains simple constructs that can be easily translated into input models of various verification tools.

- **Optimizations** To address the scalability issue, optimizations are applied to reduce the model during the translation. An optimization can either be performed during the process-to-GIMR translation (Pre-GIMR optimization) or be performed during the GIMR-to-verifier-model translation (Post-GIMR optimization). Pre-GIMR optimizations are able to achieve considerable reduction because they can exploit specific features of the process definition language. Being performed during the process-to-GIMR translation also allows Pre-GIMR optimizations to benefit all verification tools. Post-GIMR optimizations, on the other hand, are able to take advantage of features specific to the input modeling languages of the corresponding verification tools. Unlike Pre-GIMR optimizations, Post-GIMR optimizations are only applicable to specific verification tools.

- **Property Translation** Instead of requiring the analyst to encode a property with respect to the input model of the underlying tool, to better support usability we ask the analyst to specify the property in terms of constructs in the process definition. The property translation component automatically translates this high-level property specification into the equivalent property representation accepted by the selected verification tool.

- **High-Level Trace Generation** As noted, the violation trace is usually very difficult to understand for the analyst. To address this useability problem, the high-level trace generation component generates a trace over the original process based on the violation trace produced by the verification tool. Since the high-level trace represents an execution through the original process, it is much easier to understand.

- **Trace Visualization** Since human-intensive processes are often very complex, the high-level traces may be lengthy and verbose. To help the analyst to quickly identifying the error indicated by violation traces and thus help the useability, the trace visualization displays traces using graphic representations that allow the analyst to easily navigate the traces.

This framework is designed to support verifying processes defined in different process definition languages. To support a new process definition language, one needs to implement *Process-to-GIMR Translation* in the *Process Translation* component. Certain *Pre-GIMR Optimizations* can also be provided to reduce the translated GIMR models. In addition, one needs to implement a *High-Level Trace Generation* component and a *Trace Visualization* component to generate and display the high-level traces through the process defined in the target process definition language from violation traces produced by the verification tool. The rest of the framework, including *GIMR-to-Verifier-Model Translation*, *Post-GIMR Optimizations* and *Property Translation*, can be reused for different process definition languages.

The framework is also designed to support different verification tools. *Process-to-GIMR Translation*, *Pre-GIMR Optimizations*, and *Trace Visualization* can be reused for different underlying verification tools. To support a new verification tool, one needs to implement a *GIMR-to-Verifier-Model Translation* component, a *Property Translation* component, and a *High-Level Trace Generation* component for this verification tool. If necessary, certain *Post-GIMR Optimizations* can also be added.

To demonstrate the effectiveness of this framework, we provided an implementation of this framework and applied it to verify two real-world human-intensive processes. In this implementation, we selected Little-JIL as the process definition language. This is because this language provides rich facilities with well defined semantics that allow us to model the complex concurrency, coordination, and exception handling that often arise in human-intensive processes and we had Little-JIL definitions for some human-intensive processes to evaluate the framework. In addition, since we had access to the internal representation of the Little-JIL processes, the process translation would be easier to implement. The GIMR that we chose is the Bandera Intermediate Representation (BIR) [36] used by Bandera [38]. There are many existing finite-state verification tools as discussed in the background chap-

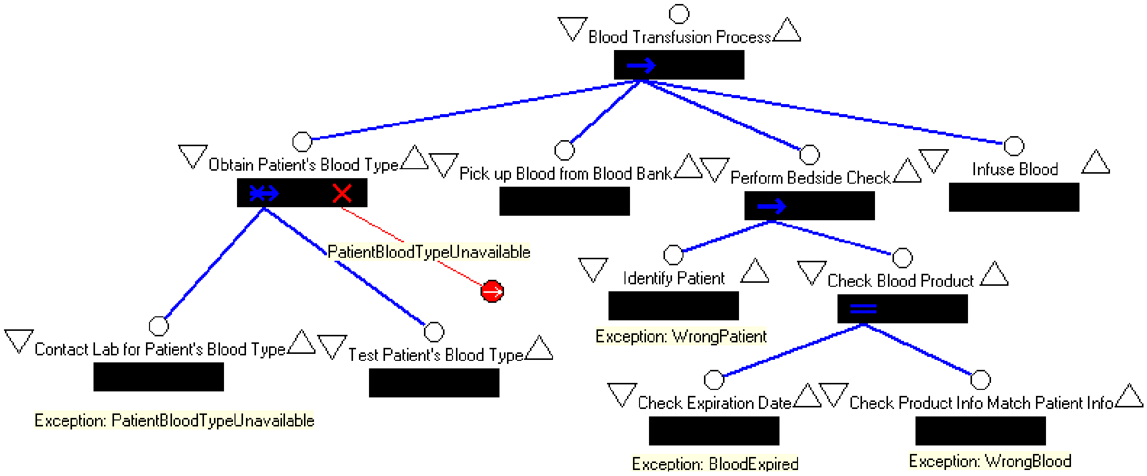

Blood Transfusion Process
Obtain Patient's Blood Type
Pick up Blood from Blood Bank
Perform Bedside Check
Infuse Blood
PatientBloodTypeUnavailable
Identify Patient
Check Blood Product
Exception: WrongPatient
Contact Lab for Patient's Blood Type
Test Patient's Blood Type
Exception: PatientBloodTypeUnavailable
Check Expiration Date
Check Product Info Match Patient Info
Exception: BloodExpired
Exception: WrongBlood

Figure 3.2. Simplified Blood Transfusion Process

ter. Each tool has its own strengths and weaknesses. In our implementation, we chose to use FLAVERS [43] and SPIN [67] simply because we are familiar with them (FLAVERS was developed in our laboratory), and they represent distinct modeling and checking approaches.

Since the *High-Level Trace Generation* and *Trace visualization* are also used by the Fault-Tree Analysis, they are presented later in Trace Generation and Visualization Chapter. In the remainder of this chapter, we focus on three components *Process Translation*, *Property Translation*, and *Optimizations.* Figure 3.2 shows the simplified blood transfusion process presented in the Background and Related Work Chapter. It will be used as the running example in our discussion. It should be noted that although the discussions here are based on our implementation of the framework, most of the issues that we discuss, as well as the approaches used to handle those issues, are also applicable to other implementations.

3.2 Process Translation

Figure 3.3 shows the details of our implementation of the *Process Translation* component for Little-JIL process definition language and two verifiers FLAVERS

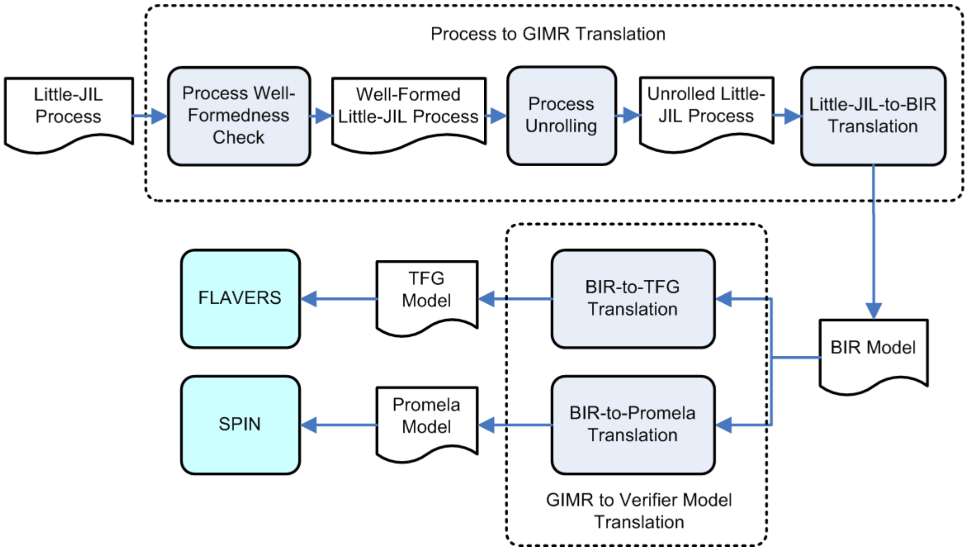

Process to GIMR Translation
Little-JIL Process
Process Well-Formedness Check
Well-Formed Little-JIL Process
Process Unrolling
Unrolled Little-JIL Process
Little-JIL-to-BIR Translation
BIR Model
BIR-to-TFG Translation
BIR-to-Promela Translation
TFG Model
Promela Model
FLAVERS
SPIN
GIMR to Verifier Model Translation

Figure 3.3. Implementation of Process Translation

and SPIN. As mentioned above, *Process Translation* contains two sub-components: *Process-to-GIMR Translation* and *GIMR-to-Verifier-Model Translation*. The implementation of *Process GIMR Translation* sub-component consists of three modules: *Process Well-Formedness Check*, *Process Unrolling* and *Little-JIL-to-BIR Translation*. *Process Well-Formedness Check* checks the well-formedness of the process. *Process Unrolling* turns the process into a finite representation. And *Little-JIL-to-BIR Translation* translates the finite representation of the process into the BIR model. As discussed in Background and Related Work chapter, this implementation is very similar to Bandera. The main difference is that our implementation targets Little-JIL processes instead of Java programs.

For *GIMR-to-Verifier-Model Translation*, a *BIR-to-TFG Translation* module and a *BIR-to-Promela Translation* module are provided for FLAVERS and SPIN respectively (Promela [67] is the input modeling language used by SPIN). Fortunately, the *BIR-to-TFG Translation* module for FLAVERS was already provided by the FLAVERS/Java front-end and the *BIR-to-Promela Translation* module has been constructed by the Bandera team [1]. Since BIR is a low-level representation, the translation from BIR to the input model of the verification tool should be easy to implemented. Therefore, in this section we focus on the tree modules in the implementation of *Process-to-GIMR Translation* sub-component.

3.2.1 Process Well-Formedness Check

The Little-JIL-to-BIR translator assumes that the Little-JIL process definition is well-formed. Errors such as a non-leaf step without any sub-step or a parameter without an associated parameter binding could cause the verification tool to produce confusing or even incorrect results. Such errors can be detected by checking certain simple *critics*. A *critic* is basically a rule that addresses a specific aspects of the well-formedness of Little-JIL processes. For example, critics to catch two kinds of errors mentioned above can be "*any non-leaf step must have at least one sub-step*" and "*any parameter must have at least one parameter binding*" respectively. Instead of hard-coding the checkers for these critics in the translation component, they are implemented as separate plug-ins to the Little-JIL process editor. This not only allows those errors to be caught as early as possible but also greatly simplifies the implmentation of Little-JIL-to-BIR translation. The implementation of critic checkers is straight-forward, and hence is not discussed here.

3.2.2 Process Unrolling

Little-JIL is a very expressive language, allowing users to precisely model complex real-world processes. Such expressiveness, however, also presents a challenge for the verification: Little-JIL process definitions allow an unbounded number of threads to be created during the runtime. Unbounded numbers of threads in a Little-JIL process can be introduced by two sources: *unbounded cardinality* and *recursion*. Figure 3.4(a) shows an example of *unbounded cardinality*. The cardinality of the step "C" is +, indicating that any number of instances of the step "C" can be created during an execution. Because the parent step "P" is a parallel step, those instances of "C" may

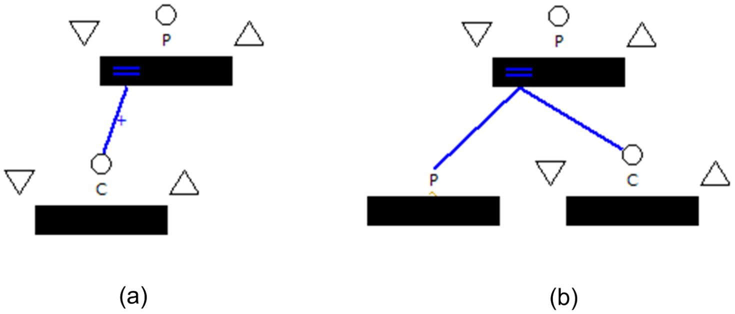


Figure 3.4. Unbounded Threads Examples

be executed in parallel. Figure 3.4(b) is an example of *recursion*. The parent step "*P*" is a parallel step, and the first child step is a step reference that refers to the parent step. Thus, during an execution, any number of instances of "*P*" and "*C*" could be created recursively and potentially run in parallel.

A Little-JIL process with an unbounded number of threads cannot be directly handled by most finite-state verification tools, including the ones that we target (FLAVERS and SPIN). As a matter of fact, this is a common problem in software model checking research. Many important software systems can also create unbounded numbers of threads, including web severs that might create service threads for unspecified number of requests or token ring algorithms involving any number of processes. This common model-checking problem is called the *parameterized model-checking problem*. It has been shown that to decide whether a temporal property holds for systems with unbounded numbers of threads is undecidable in general [12]. Approaches that attempt to tackle this problem fall into two categories [46]. One focuses on restricted subclasses of the systems and the properties, and applies various abstraction techniques to obtain finite models of those systems so that they can be verified [24] [31] [136]. The other kind of approaches simply impose a bound on the number of threads that can be created [71] [72] [19]. The underlying premise of these bounded approaches is called the "*small scope hypothesis*" [72], which suggests that most flaws in a system can be revealed by considering instances of the system with a small bound. Some researches [45] have indeed shown that a sound bound can be achieved for some particular subclasses of systems. No one, however, is able to provide the proof to valid this assertion in the general cases. Therefore, the analyst needs to bear in mind that this kind of bounded analysis may be unsound.

In our work, we take the bounded approach to solve this problem. The analyst is asked to designate upper bounds for recursive steps as well as steps with unbounded cardinalities. Given these bounds, a process unrolling preprocessor unrolls the input Little-JIL process definition into a unrolled version of the process that contains a finite number of step instances. In addition, a step may be used multiple times in a Little-JIL process via step references. The process unrolling preprocessor also turns each step reference into a step instance. As a result, the unrolled process becomes a tree of step instances. This unrolled process then can be further translated into a finite model that can be verified by FLAVERS or SPIN. Since the simplified blood transfusion process does not contain steps with cardinalities, recursive steps or step references, the unrolled process is the same as the original one.

In the remainder of this chapter, the discussions assume an unrolled Little-JIL processes. And unless stated otherwise, we use “step” and “step instance” interchangeably to refer to a step instance in an unrolled Little-JIL process.

3.2.3 Little-JIL-to-BIR Translation

The Little-JIL-to-BIR Translator translates a given unrolled Little-JIL process into a BIR program. In this subsection, we first briefly introduce the BIR specification language. Then we give a high-level overview of the mapping between the Little-JIL process and the BIR program. Next, we discuss the translation templates. Finally, we describe our template-based process translation algorithm.

3.2.3.1 Bandera Intermediate Representation

Bandera Intermediate Representation is a guarded command language used in the Bandera project, developed at KSU, for model checking concurrent Java programs [71]. BIR contains constructs close to Java, including threads, monitor locks and asynchronous communication primitives (e.g., `wait`, `notify`, `notifyAll`). Our translation is based on the BIR specification version 0.6 [36]. Here, we only briefly describe features that are enough to understand the translation templates. More details about BIR can be found in the BIR specification.

A BIR program is called a *process*. A process consists of a set of *threads*, and one of them is designated to be the main thread. The main thread is automatically executed when the process starts. BIR provides several thread actions that can be used to update the thread status. Threads other than the main thread need to be started by explicitly invoking the `start` action. Invoking the `exit` action changes the state of current thread to inactive. The `join` action cause the current thread to be blocked until another thread is inactive.

A thread is basically a directed graph represented as a non-empty list of *locations*. The first location in the list is the start location. Each location has a unique *label* that can be used to refer to the location. A location contains one or more *transformations*. Each transformation has a Boolean *guard expression*, a sequence of *actions*, and a *goto target*. When the execution reaches a location, guard expressions of all transformations in this location are evaluated. The transformation whose guard expression turns out to be true is enabled. The actions of the enabled transformation are then performed in the order of which they are defined. When all the actions complete, the control goes to the location identified by the goto target of the transformation. It is possible that more than one transformations are enabled. In this case, only one such transformation is non-deterministically selected to be executed.

BIR supports seven basic data types and their corresponding operations. Currently, we only used the `boolean` type, the integer `range` type, and the `lock` type in our translation. Locks are used to synchronize the threads. Operations similar to Java are provided on the locks, such as `lock`, `unlock`, `wait`, `unwait`, `notify`, and `notifyAll`. Threads can communicate with each other via global variables declared in the definition section of the process. A thread may also declare local variables that are only visible within the scope of the thread.

BIR allows predicates to be defined on the states of the process. A predicate consists a unique *identifier* and a *Boolean expression*. The Boolean expression is defined over two kinds of basic predicate expressions: *thread location tests* and *remote references*. A thread location test `threadName@locLabel` becomes true when the execution of the thread `threadName` reaches the location with the label `locLabel`. A remote reference `threadName:locLabel` refers to the local Boolean variable `locLabel` in the thread `threadName`.

3.2.3.2 Translation Overview

Given an unrolled process, the Little-JIL-to-BIR translator produces a BIR program that consists of *threads*, *variable declarations* and *predicates*.

3.2.3.2.1 Threads

The root step of the process is translated into the `main` thread in the BIR program. In addition, steps that may potentially run in parallel with other steps, including sub-steps of a parallel step, exception handling steps, and sub-steps of a choice step, are also translated into BIR threads. As indicated by the name of the sequencing badge, sub-steps of a parallel step may execute in any (possibly interleaved) order, and therefore are turned into different threads. At an exception handling point, e.g. after termination of a sub-step of a sequential step, multiple exceptions may have been thrown. Different exception handling steps could be invoked to handle these exceptions. According to the Little-JIL specification, these exception handling steps may run in parallel. Therefore, all exception handling steps should also be translated into threads. Translating sub-steps of choice steps into threads is not so intuitive because the choice is supposed to choose only one sub-step to execute each time. The problem is that the choice step does not restrict the order of its sub-steps. In other words, sub-steps of a choice step may execute in any order although not in parallel. One straight-forward translation option is to enumerate all possible such orders in the BIR code. This could easily blow up the BIR code if the choice step has many sub-steps. Instead, we translate sub-steps of a choice step into BIR thread and force them to run one after another. This ordering is achieved by adding a *choice variable* that indicates which sub-step is selected. In addition, each sub-step is assigned one *done variable* that represents whether the sub-step is done or not. The choice variable is randomly assigned to the index of a single sub-step at the beginning of each iteration. Then all sub-step threads are started. The thread of a sub-step first tests if it has been done. If so, the thread ends immediately. Otherwise the thread proceeds to check the sub-step index against the choice variable. Only the thread whose index equals the value of the choice variable will continue. And other threads simply retract their corresponding sub-steps. The major advantage of this approach is that it produces more compact BIR code, greatly relieving the burden of the model construction for verification tools. For the simplified blood transfusion process, the root step is translated to the `main` thread and two sub-steps of the parallel step "*Check Blood Product*" are translated to two threads respectively.

3.2.3.2.2 Variables

Variables in the BIR program are translated from Little-JIL artifacts, which in Little-JIL include exceptions and channels. In our approach, since the BIR program is intended to be used to verify the event-based temporal properties and to create the reversed control flow graph for fault tree derivation, the resulting BIR program needs to preserve the control flow of the original Little-JIL process. Details of artifacts that do not affect the control flow are ignored by the translation. In addition, temporary variables are introduced to enforce constraints imposed on certain steps, such as the constraint mentioned above to enforce the ordering of threads translated from substeps of a choice step.

- *Exception Variables* In Little-JIL, an exception can be defined as any type. In our translation, however, we only care about whether the exception is thrown or not, since that is the only way that an exception can affect the control flow of the process. Thus, each exception is translated into a Boolean type *exception variable*. An exception variable is initially set to `false`. It changes to `true` if the corresponding exception is thrown and reset to `false` after the exception is successfully handled by the exception handler. If the exception is re-thrown by the exception handler, the exception variable remains `true`. In the simplified blood transfusion process, there are four exceptions: *PatientBloodTypeUnavailable*, *WrongPatient*, *BloodExpired* and *WrongBlood*. Therefore, four exception variables are created for these exceptions during the translation.

- *Channel Variables* Channels in Little-JIL are used for passing artifacts as well as for synchronizing different steps. In our current implementation, a very simple model is used to translate channels. A channel can hold any number of artifacts. Similar to how unbounded numbers of threads are handled as discussed in process unrolling subsection, a bound needs to be imposed on channels to make the BIR program finite. In our model, the bound is set to 1. In addition, since the Little-JIL process definition does not define the behavior of a leaf step, in our translation the behaviors of leaf steps are abstracted using a conservative model (Please refer to the discussion of *Leaf Step Started Template* later for more details). Our implementation is based on Little-JIL 1.4. In this version, an artifact may only affect the flow of control in a leaf step. Since such effects are subsumed by the abstract model of the leaf step, the content of the artifact can be ignored. With only one artifact whose content can be ignored, a channel can simply be translated to a Boolean type *channel variable* that indicates whether the channel is empty. An channel variable is initially set to `false`, indicating the channel is empty. It changes to `true` when a `Write` operation puts an artifact into the channel. It is reset to `false` after a `Take` operation removes the artifact in the channel. This simplified model of channels captures most synchronization scenarios caused by the channels. This is enough for the verification of the processes in our case studies. If necessary, however, one can always replace this model with a more complicated model. This can be achieved by substituting the default channel translation module in our implementation with the one that creates a complicated channel model. Since the simplified blood transfusion process does not use channels, no channel variable is created during the translation.

- *Temporary Variables* Temporary variables are used to enforce constraints imposed on choice and parallel steps. As mentioned above, one *choice variable* is introduced for each choice step, and one *done variable* is assigned to each sub-step of a choice step. Similarly, one *done variable* is also assigned to each sub-step of a parallel to avoid re-entering the finished sub-step. In the simplified blood transfusion, step "*Check Blood Product*" is a parallel step. For each of its sub-steps, one *done variable* is created during the translation.

3.2.3.2.3 Predicates

Predicates in the BIR program are translated from *Property Event Bindings* that map abstract events in property specifications to concrete events in the Little-JIL process. Detailed discussion of property event bindings and their translation is given in Section 3.3.

3.2.3.3 Translation Templates

An unrolled Little-JIL process is a tree composed of steps. The translation of the Little-JIL process, therefore, is based on the translation of steps in the process. Depending on certain attributes (e.g. step kind, step kind of the parent step, etc.), steps may behave quite differently from each other. For example, a step may go from the *POSTED* state to the *RETRACTED* state if its parent is a parallel step or a choice step. But this will never happen if the parent step is a sequential step or a try step. When started, a step will invoke its sub-steps if it is a non-leaf step, but the order in which sub-steps are executed is decided by the sequencing badge of the step. Implementing the translation of a step as a monolithic step translator would result in a large chunk of code that is hard to maintain. Instead, we designed a few types of translation templates. Each type of template is responsible for translating a specific part of the behavior of a step. For example, a *Started Template* translates the behavior of a step after it is started, and the *Exception Handling Template* translates the exception handling actions of a step. One template type may have several templates that correspond to different values of a specific attribute. For instance, there are five *Started Templates* corresponding to five step kinds. A step translator simply consists of a collection of templates, one for each type. To translate a given step, a specific step translator can be constructed by selecting appropriate templates according to attributes of the step.

According to the Little-JIL semantics, we propose twelve templates that belong to seven template types.

- **Posting Template Type** This type of template translates the behaviors before a step is posted, including passing in parameters and obtaining artifacts from channels. Since such behaviors are the same for all steps, only one template is defined for this type.

- **Posted Template Type** This type of template translates the behaviors after a step is posted. After being posted, a step may be opted-out or retracted under some circumstances. If it is not opted-out or retracted, the step is started. This type has only one template that simply uses the appropriate *Retracted Template* and *Opted-Out Template* to translate behaviors that check whether the step should be opted-out or retracted respectively.

- **Retracted Template Type** This type of template translates the behavior that checks whether a step should be retracted. Two templates are defined: *Choice Sub-Step Retracted Template* handles the sub-steps of choice steps and *Parallel Sub-Step Retracted Template* handles the sub-steps of parallel steps.

- **Opted-Out Template Type** This type of template translates the behavior that checks whether an optional step should be opted-out. Only one template is defined for this type.

- **Started Template Type** This type of template translates the behaviors after a step is started. Corresponding to the five step kinds, five templates are defined for this type: *Sequential Step Started Template*, *Try Step Started Template*, *Parallel Step Started Template*, *Choice Step Started Template*, and *Leaf Step Started Template.*

- **Completing Template Type** This type of template translates the behaviors right before a step is completed, including passing out parameters, and putting artifacts in channels, etc. Such behaviors are the same for all steps, so only one template is defined for this type.

- **Exception Handling Template Type** This type of template translates the exception handling behavior for non-leaf steps. If exceptions are thrown from its sub-steps, a non-leaf step identifies and invokes the appropriate exception

```
1   loc Check_Expiration_Date_STARTED:
2       when true do { }
3         goto Check_Expiration_Date_COMPLETED;
4       when true do { b_BloodExpiration := true; }
5         goto Check_Expiration_Date_TERMINATED;
```

Figure 3.5. BIR Representation for Leaf Step Started Template

handlers to handle these exceptions. Such exception handling behaviors are the same for all non-leaf steps, so only one template is defined for this type.

We only discuss four representative templates here: *Leaf Step Started Template*, *Sequential Step Started Template*, the *Parallel Step Started Template*, and the *Exception Handling Template*. The description of all the templates can be found in Appendix A.

3.2.3.3.1 Leaf Step Started Template

In a Little-JIL process, leaf steps are performed by the assigned agents once they are started. The behaviors of agents, however, are not modeled in the Little-JIL process definition. The process definition only specifies the interface of a leaf step, which declares the parameters used by the step as well as exceptions that might be thrown by the step. Based on the interface, we translate the started behavior of a leaf step into a BIR model that conservatively captures such behavior.

In the simplified blood transfusion process, leaf step "*Check Expiration Date*" might throw exception *BloodExpired.* Figure A.17 shows the BIR code generated by *Leaf Step Started Template* for this step. There are two transformations in this location, corresponding to two behaviors that step "*Check Expiration Date*" might have. Both transformations have the condition `true`. This means that when the execution reaches the location, one transformation will be non-deterministically selected to be executed. In the first transformation (line 2-3), no exception is thrown and the step "*Check Expiration Date*" is completed. In the second transformation (line 4-5), the exception *Blood Expired* is thrown and the step "*Check Expiration Date*" is terminated. This is a conservative model because the conditions in which two behaviors may occur are ignored. It turns out that such conservative model for leaf steps is enough for verifying two processes in our case study.

3.2.3.3.2 Sequential Step Started Template

The *Sequential Step Started Template* is used to construct the behavior of a sequential step after it has been started. When a sequential step is started, its sub-steps are executed one by one from left to right. And the sequential step is only successfully completed after all of its sub-steps have successfully completed.

Figure A.8 shows the root step as well as its four sub-steps in the simplified blood transfusion. The corresponding BIR representation is given in the Figure A.9. When the sequential step "*Blood Transfusion Process*" is started, the first sub-step "*Obtain Patient's Blood Type*" is posted (line 1-3). When "*Obtain Patient's Blood Type*" is completed, the next sub-step "*Pick up Blood from Blood Bank*" is posted (line 8-10). Similarly, "*Perform Bedside Check*" is posted as soon as "*Pick up Blood from Blood Bank*" is completed (line 15-16) and "*Infuse Blood*" is posted as soon as "*Perform Bedside Check*" is completed (line 20-21). The sequential step "*Blood Transfusion Process*" is completed when its last sub-step is completed (line 27-29). Note that step "*Perform Bedside Check*" may throw exceptions. Therefore, BIR code must be inserted to check and handle those exceptions (line 22-23). This is achieved by invoking the *Exception Handling Template.*

3.2.3.3.3 Parallel Step Started Template

The *Parallel Step Started Template* is applied to construct the behavior of a parallel step after it has been started. It is quite different from the *Sequential Step Started Template* that we just discussed. When a parallel step is started, all its sub-steps are posted at the same time. Then the sub-steps can be executed in any (possibly

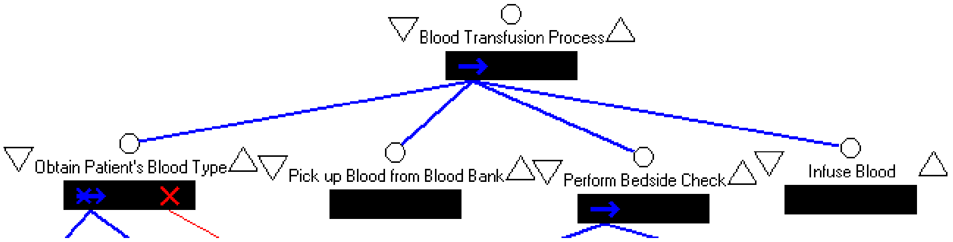

Blood Transfusion Process
Obtain Patient's Blood Type
Pick up Blood from Blood Bank
Perform Bedside Check
Infuse Blood

Figure 3.6. Sequential Step Example

```
loc Blood_Transfusion_Process_STARTED:
   when true do { }
     goto Obtain_Patients_Blood_Type_POSTED;
loc Obtain_Patients_Blood_Type_POSTED:
   when true do { }
     goto Obtain_Patients_Blood_Type_STARTED;
... // Obtain Patient's Blood Type
loc Obtain_Patients_Blood_Type_COMPLETED:
   when true do { }
     goto Pick_up_Blood_from_Blood_Bank_POSTED;
loc Pick_up_Blood_from_Blood_Bank_POSTED:
   when true do { }
     goto Pick_up_Blood_from_Blood_Bank_STARTED;
... // Pick up Blood from Blood Bank
loc Pick_up_Blood_from_Blood_Bank_COMPLETED:
   when true do { } goto Perform_Bedside_Check_POSTED;
loc Perform_Bedside_Check_POSTED:
   when true do { } goto Perform_Bedside_Check_STARTED;
... // Perform Bedside Check
loc Perform_Bedside_Check_COMPLETED:
   when true do { } goto Infuse_Blood_POSTED;
loc Perform_Bedside_Check_TERMINATED:
... // Exception handling
loc Infuse_Blood_POSTED:
   when true do { } goto Infuse_Blood_STARTED;
... // Infuse Blood
loc Infuse_Blood_COMPLETED:
   when true do { }
     goto Blood_Transfusion_Process_COMPLETED;
```

Figure 3.7. BIR Representation for the Sequential Started Template

interleaved) order. And the parallel step is only completed after all of its sub-steps has successfully completed.

Step "*Check Blood Product*" in the simplified blood transfusion is a parallel step. The BIR representation generated by the *Parallel Step Started Template* for this step is given in the Figure A.13. Before the *Parallel Step Started Template* is applied, the sub-steps of the parallel step have already been translated into different BIR threads. In this example, threads `T_Check_Expiration_Date` and `T_Check_Product_Info_Match_Patient_Info` are created for sub-step "*Check Expiration Date*" and "*Check Product Info Match Patient Info*" respectively. When the parallel step "*Check Blood Product*" is started, two threads of its sub-steps are started by the `start` action (line 3-7). Then the parallel step waits until these thread to finish (line 8-23). Note that each `join` action on a thread is followed by a `threadTerminated` test for that thread. This is a constraint imposed by the BIR language specification. Note that the target of the last location is empty. It will be set to the label of the first location in the exception handling BIR code generated by the *Exception Handling Template*.

3.2.3.3.4 Exception Handling Template

The *Exception Handling Template* is invoked by various *Started Templates* to generate the code to handle exceptions thrown by the sub-steps. When a sub-step throws an exception, the matching exception handler associated with the parent step is invoked to handle the exception. If the handler is associate with an exception handler step, the handler step is executed. The exception handler also has a control-flow badge indicating how the step catching the exception executes after the handler step finishes. Since this exception handling mechanism is the same for all kinds of steps, only one *Exception Handling Template* is defined.

Figure A.20 shows the exception handling code generated by the *Exception Handling Template* for step "*Check Blood Product*" to handle two exceptions, *BloodExpired* and *WrongBlood*, thrown by its sub-steps. There are two transformations in the only

```
loc Check_Blood_Product_STARTED:
   when true do { } goto Check_Blood_Product_fork_subs;
loc Check_Blood_Product_fork_subs:
   when true do {
      start(T_Check_Expiration_Date);
      start(T_Check_Product_Info_Match_Patient_Info);
   } goto Check_Blood_Product_join_Sub1;
loc Check_Blood_Product_join_Sub1:
   when true do { join(T_Check_Expiration_Date); }
     goto Check_Blood_Product_terminate_test_Sub1;
loc Check_Blood_Product_terminate_test_Sub1:
   when threadTerminated(T_Check_Expiration_Date) do { }
     goto Check_Blood_Product_join_Sub2;
loc Check_Blood_Product_join_Sub2:
   when true do {
       join(T_Check_Product_Info_Match_Patient_Info);
   } goto Check_Blood_Product_terminate_test_Sub2;
loc Check_Blood_Product_terminate_test_Sub2:
   when
     threadTerminated(
       T_Check_Product_Info_Match_Patient_Info
     )
   do { } goto ⊔;
... // Exception Handling
```

Figure 3.8. BIR Representation for the Parallel Started Template

```
loc Check_Blood_Product_control_rethrow:
   when (b_BloodExpired==true || b_WrongBlood==true)
     do {  }
     goto Check_Blood_Product_TERMINATED;
   when (b_BloodExpired==false && b_WrongBlood==false)
     do {  }
     goto Check_Blood_Product_COMPLETED;
```

Figure 3.9. BIR Representation for the Exception Handling Template

location in this code. The first transformation (line 2-4) checks whether those two exceptions are thrown. If any one is thrown, step "*Check Blood Product*" is terminated because no exception handlers are provided to handle these exceptions. In addition, the exception variables for two exceptions are not reset to `false` because they will be re-thrown to the parent step of "*Check Blood Product*". The second transformation (line 5-7) handles the case where no exception is thrown. In this case, step "*Check Blood Product*" is completed. Note that this simple example does not involve exception handlers. For more complicated exception handling scenarios, please refer to the discussion of the *Exception Handling Template* in Appendix A.

3.2.3.4 Translation Algorithm

The translation algorithm involves three phases. Phase 1 goes over the unrolled process, creating various BIR variables mentioned earlier. In Phase 2, the algorithm performs a post-order traverse over the unrolled process, which is basically a tree of step instances. Upon visiting a step instance, the children of this step instance have already been translated into the BIR representations. A step translator for this step instance is constructed by selecting appropriate templates according to attributes of the step instance. The step translator is then applied to assemble the BIR representations of its children and produce the BIR representation for this step instance. All the BIR threads are completed when the traverse is finished. Phase 3 identifies all locations in BIR threads where concrete events defined in property event bindings occur and generates predicates based on these locations. The BIR program translated from the simplified blood transfusion process is presented in Appendix C.

3.2.4 Summary

In this section, we discussed three issues that we encountered in the implementation of the process translation. The first issue is the well-formedness checks of the process definition. Performing the well-formed check on the process definition before the translation is very important because it not only catches errors earlier but also greatly simplifies the implementation of the translation templates. In our approach, the well-formedness check is achieved by checking the process against a set of predefined critics. The critics that we developed as well as their implementation are specific to the Little-JIL language and cannot be used for other process definition languages. Using critics to check the well-formedness, however, is a general idea that can be applied in the realizations of the framework for other languages.

The second issue is how to handle unbounded numbers of threads in the process. Currently, we simply impose upper bounds on the number of threads that can be created. The problem of this approach is that it might be unsound for some properties. In the future, we may consider implementing some abstraction approaches that are sound for certain classes of processes.

The last issue is how to actually translate the process. In our approach, we proposed a template-based translation algorithm that simplifies the implementation of the process translation. In the original design, only five templates were provided to translate five kinds of Little-JIL steps. It turned out that this design caused the implementation of templates to be very complex, containing a lot of duplicated code. Therefore, we revised the design and divided those five templates into templates with finer granularity. Each template translates a specific part of the behavior of a step. Our experience shows that the current design greatly simplifies the implementation of the templates. It is also very flexible, allowing new templates to be easily added into the translator to support new features. Although the templates and the translation algorithm are Little-JIL dependent, the idea of using templates may also be applied to simplify the translation of other languages.

3.3 Property Specification and Translation

A property describes a single aspect of the behavior of a process. A process often has different implementations that share certain common core components. Therefore, the property should be stated at a high-level and usually is independent of any specific implementation of the process. Typically, a property can is either *event-based* or *state-based.* An event-based property is defined over events in the process and a state-based property is defined over states in the process. In our implementation, we used an event-based model for the property specification because it maps nicely to Little-JIL process definitions, which tend to focus on events, represented as flow of control, coordination, or task invocation. To be verified, a property needs to be precisely specified using a formal notion. In addition, since the events in the property specification are not tied to any process implementation, they need to be defined in terms of events in the process implementation to be verified. Then the property specification together with the property events definitions are translated into the property representation accepted by the selected verifier.

In this section, we discuss our approach to specify and translate event-based properties. We first present the tool that we used to precisely specify properties. Then we discuss property event bindings that are used to define the events in property specifications. Finally, we describe the translation of property specifications and property event bindings. In the discussion, property "*the nurse must identify the patient before infusing the blood*" for the simplified blood transfusion process is used a running example. It should be noted that although this approach is based on the event-based model, a similar approach can be applied for the state-based model.

3.3.1 Property Specification

Our experience shows that correctly specifying properties is a surprisingly difficult task. Even experienced developers may overlook subtle, but important, details. In our approach, we use Propel to precisely specify properties. Propel provides users with a set of property templates, each of which can be viewed as an *extended Finite-State Automaton representation*, a *Disciplined Natural Language representation*, or a *Question Tree*. Each representation contains options (or questions) that explicitly indicate the variations that must be considered, thereby ensuring that users do not overlook important subtle details. In addition, the *Question Tree* can be used to guide the user in selecting the appropriate template. All three representations are views of a single underlying representation so that a change in any representation is reflected automatically in the others. More details about Propel can be found in [35].

3.3.2 Property Event Binding Specification

As noted, events in a property specification are not tied to a specific process implementation. To verify a process implementation against the property specification, the events in the property specification need to be clearly defined in terms of the events in the process implementation. To distinguish those two kinds of events, "*abstract events*" refer to the events in the property specification and "*concrete events*" refer to the events in the process implementation. In our approach, we propose to use *property event bindings* to define the abstract events in terms of concrete events.

A property event binding describes the mapping from an abstract event to one or more concrete event. The property "*the nurse must identify the patient before infusing the blood*" has two abstract events: "*IdentifyPatient*" and "*InfuseBlood*". For the simplified blood transfusion process example, it is obvious that the abstract event "*IdentifyPatient*" and "*InfuseBlood*" can be defined as the concrete event "*Identify Patient COMPLETED*" and "*Infuse Blood STARTED*" respectively. Therefore, the property event bindings can be defined as:

```
IdentifyPatient -> Identify Patient COMPLETED
InfuseBlood -> Infuse Blood STARTED
```

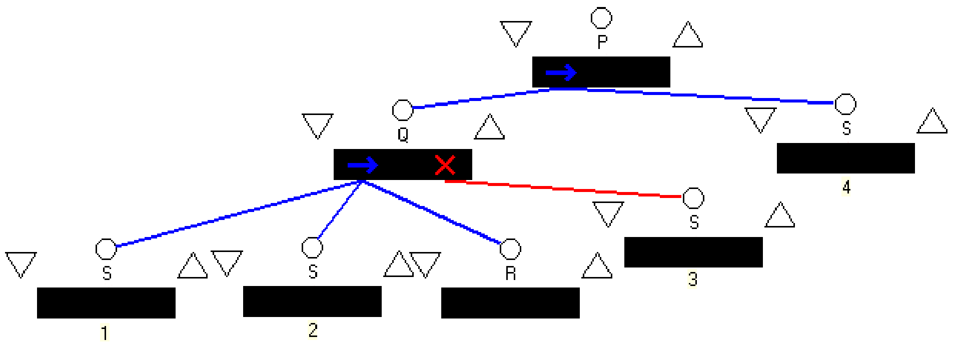


Figure 3.10. Process Example for Discussing Step Identifier

In this example, both concrete events only reference one step instance in the process implementation. Therefore, we can use the step name to identify the corresponding step instance. This simple way of identifying step instances, however, is not enough for complex real-world processes. During an execution of such processes, multiple instances of a step may be created. A property may be concerned with a concrete event occur only at some of those step instances. Therefore, one must be able to distinguish a particular set of step instances from the others. In our approach, the *step instance identifier* is introduced to identify a particular set of step instances. To facilitate the discussion of the step instance identifier, we contrived an example process, shown in Figure 3.10. There are four instances of step "S" in this unrolled process, with label 1, 2, 3, and 4.

Figure 3.11 gives the syntax of the step identifier. The basic form of a step instance identifier is `step-name`, which represents all instances of the step with the name `step-name`. For instance, the step name S represent all four instances of step "S" in the example process. To achieve finer granularity, however, certain constraints have to be posed on the step name. There are three kinds of constraints that can be used in step identifiers: *prefix*, *index*, and *type*. The *prefix* is used to distinguish step instances from the others with different parents. The *type* is used to distinguish step instances from their siblings according to their relationships with the parent (i.e.

```
⟨step-instance-identifier⟩ ::= ⟨prefix⟩?  ⟨step-name⟩
                               (‘[’⟨type⟩‘]’)?  (‘[’⟨index⟩‘]’)?
                   ⟨prefix⟩ ::= ⟨step-instance-identifier⟩  ‘.’
                               ⟨step-instance-identifier⟩  ‘.’  ⟨prefix⟩
                     ⟨type⟩ ::= SUB-STEP|PRE-REQUISITE|POST-REQUISITE|HANDLER
                    ⟨index⟩ ::= 1|2|3|...
```

Figure 3.11. Step Instance Identifier Syntax

sub-step, pre-requisite, post-requisite, and exception handler). And the *index* is used to distinguish different step instances with the same parent and the same relationship with the parent.

- **Prefix** A prefix is a step instance identifier itself that defines a set of step instances. Given a prefix `prefix` and a step instance identifier `step-identifier`, `prefix.step-identifier` represents the set of step instances in `step-identifier` whose parents appear in the set `prefix`. In the example process, among all four instances of "*S*", the instance labeled 4 is the only one that has the parent step "*P*". Therefore, we can use `P.S` to distinguish it from the other instances. On the other hand, `P.Q.S` can be used to represent the instances with label 1, 2 and 3. For this particular process, one may just use `Q.S` to represent the instances with label 1, 2 and 3 because the instance with label 4 has a different parent "*P*".

- **Type** In terms of the relationship with its parent, a step instance can be one of four types: `SUB-STEP`, `PRE-REQUISITE` step, `POST-REQUISITE` step, or (exception) `HANDLER` step. The type in the identifier is used to distinguish one type of instances from the others. For example, `S[HANDLER]` uniquely identifies the instance of "*S*" with label 3 because it is the only one that is an exception handling step. The other three are all sub-steps and therefore can be identified by `S[SUB-STEP]`.

- **Index** In a unrolled process, it is not unusual to see several instances of a step that are sub-steps of the same parent, like the two instances of "S" labeled 1 and 2 in the example process. This usually results from unrolling steps with cardinality `+` or *. In these cases, we may use an `index` (starting from 1) to distinguish one instance from the others. For instances of "S" labeled 1 and 2 in the example process, `S[SUB-STEP][1]` represents the instance labeled 1 and `S[SUB-STEP][2]` refers to the one labeled 2. It should be noted that the index does not imply the execution order of those instances. For example, suppose the step "*Q*" is a parallel step, `S[SUB-STEP][1]` might be executed after `S[SUB-STEP][2]`.

Step identifiers are used to identify the step instances for the concrete events in property event bindings. The syntax of the property event binding specification language is shown in Figure 3.12. As mentioned earlier, an *property event binding* maps an *abstract event* to one or more *concrete event* in the process. This means that if any concrete event in the property event binding occurs during execution of the process, the corresponding abstract event is considered to occur. Currently, we support three kinds of concrete events: *step state events*, *exception throwing events*, and *parameter def/use events*.

- **Step State Event** This kind of event occurs when a step instance identified by ⟨`step-identifier`⟩ goes into the particular state ⟨`step-state`⟩. The ⟨`step-state`⟩ contains all runtime step states defined in the Little-JIL language specification (e.g. `POSTED`, `STARTED`, `RETRACTED`, `OPTED-OUT`, `COMPLETED`, and `TERMINATED`).

```
          ⟨event-binding⟩ ::= ⟨abstract-event⟩  ‘->’  ⟨concrete-events⟩+
        ⟨concrete-events⟩ ::= ⟨concrete-event⟩  ‘|’  ⟨concrete-events⟩
                            |  ⟨concrete-event⟩
         ⟨concrete-event⟩ ::= ⟨step-state-event⟩
                            |  ⟨exception-throwing-event⟩
                            |  ⟨parameter-def-use-event⟩
       ⟨step-state-event⟩ ::= ⟨step-identifier⟩  ‘:’  ⟨step-state⟩
             ⟨step-state⟩ ::= POSTED|STARTED|RETRACTED
                            |  COMPLETED|TERMINATED|OPTOUT
⟨exception-throwing-event⟩ ::= ⟨step-identifier⟩?  ‘.’  ⟨exception-name⟩
 ⟨parameter-def-use-event⟩ ::= ⟨step-identifier⟩?  ‘.’
                              ⟨parameter-name⟩  ‘:’  ⟨def-use⟩
                ⟨def-use⟩ ::= DEF|USE
```

Figure 3.12. Property Event Binding Syntax

- **Exception Throwing Event** This kind of event occurs when an exception with the name ⟨`exception-name`⟩ is thrown from a step instance identified by ⟨`step-identifier`⟩. The step identifier can be omitted if the property does not care about from which step the exception is thrown.

- **Parameter Def/Use Event** This kind of event occurs when a parameter with the name ⟨`parameter-name`⟩ is defined or used at a step instance identified by ⟨`step-identifier`⟩. Similar to *Exception Throwing Events*, the step identifier in a *parameter def/use event* definition can also be omitted if it does not matter at which step the parameter is defined or used. In Little-JIL, only leaf steps can manipulate parameters and the non-leaf steps are used to pass the parameters. Therefore, the definition or use of a parameter only occurs at a leaf step. There are two issues with the parameter def/use events. First, since the Little-JIL process definition does not define the behavior of a leaf step, we have no way of knowing the exact point where the parameter is defined or used. Therefore, we

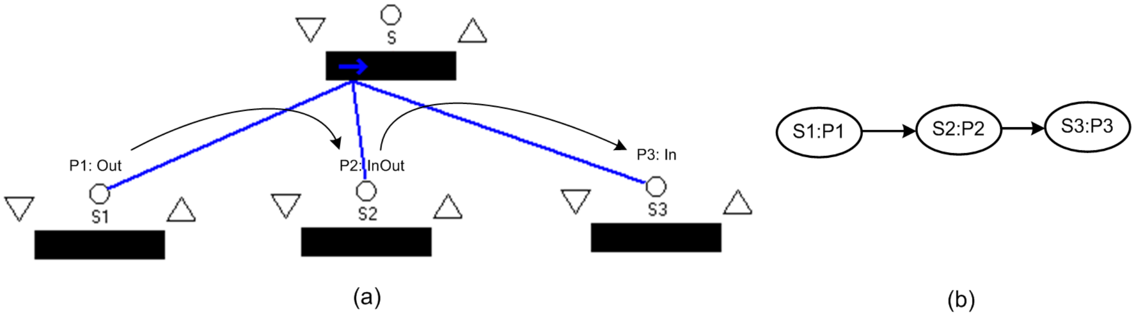


Figure 3.13. Parameter Def/Use Event Example

consider that the parameter is defined when the corresponding leaf step instance is `COMPLETED`, and is used when the corresponding leaf step instance is `STARTED`. Second, when a property refers to the definition or use of a parameter, it usually means "each time the artifact held by that parameter is defined/used". Since the artifact may be propagated through the process, being held by different parameters and being defined or used at different leaf step instances, the analyst has to provide the complete set of bindings, each connecting to one parameter at one leaf step instance.

Figure 3.13(a) shows an example process. Step "$S1$" produces an artifact and passes it out as parameter $P1$. The artifact is passed to the IN/OUT parameter $P2$ at step "$S2$" and then to the IN parameter $P3$ at step "$S3$". Suppose that an event e intends to indicate the definition of this artifact. According to the artifact flow discussed above, one can see that the artifact is created in step "$S1$" and may be changed in step "$S2$". Therefore, the event can be defined as `e -> S1:P1 DEF | S2:P2 DEF`. The artifact flows in a real-world process, however, could be very complex. In addition, an artifact may be passed to parameters with different names. Identifying step instances where an artifact is defined or used manually could be very difficult and prone to errors.

To reduce the burden of the analyst, we construct an artifact flow graph from the process definition. The analyst only needs to provide the binding for a parameter at one step instance. The rest of them then can be automatically inferred based on the artifact flow graph. The formal definition of the artifact flow graph and how it is constructed are discussed in Fault Tree Analysis Chapter. For now, it is enough to know that it is a directed graph representing the artifact flows in a process definition. Figure 3.13(b) shows the artifact flow graph for the example process. To define the event mentioned above, the analyst only needs to provide the property event binding `e -> S2:P2 DEF`, which only contains the definition of one parameter. All parameters that connect to this parameter in the artifact graph can be identified. In this example, those parameters are `S1:P1` and `S3:P3`. `S1:P1` may define the artifact because it is an OUT parameter and `S3:P3` may not define the artifact because it is an IN parameter. Therefore, the given property event binding can be expended to `e -> S1:P1 DEF | S2:P2 DEF`. This property event binding can also be inferred if the analyst provides the property event binding `e -> S1:P1 DEF`.

3.3.3 Property Specification and Property Event Binding Translation

To be verified, the property specification and the property event bindings need to be translated to the property representation accepted by the underlying verifier. A usual way to achieve this is to first translate the property specification into the property representation accepted by the underlying verifier. Then the events in the translated model for the verifier are identified for abstract events based on the property event bindings. Finally, the abstract events in the verifier's property representation are replaced by the corresponding events in the translated model for the verifier. Implementation of this approach could be difficult because identifying the events in the translated model for the abstract events requires the understanding of both the process-to GIMR-model translation and the GIMR-to-verifier-model translation.

In our implementation, we employ a different approach that takes advantage of the predicate feature provided by BIR. The property specification is translated into the property representation accepted by the verifier. Then instead of substituting the abstract events in the verifier's property representation, the property event bindings are translated into predicates in the BIR model. The predicates, when translated to the constructs in the verifier's process model, allow the verifier to recognize the occurrence of abstract events in the verifier's process model.

Propel provides a translator that translates Propel properties into properties accepted by FLAVERS. The property translator for SPIN and other verifiers can be easily implemented. Therefore, we only discuss how to translate property event bindings into BIR predicates here. This translation is implemented in the Little-JIL-to-BIR translator. Remember that a BIR predicate consists a unique *identifier* and a *Boolean expression*. The Boolean expression is defined over two kinds of basic expressions: *thread location tests* and *remote references*. In the property event binding translation, only the *thread location tests* are used. An property event binding is translated into one predicate. The abstract event in the property event binding is directly mapped to the identifier in the predicate. For each concrete event in the property event binding, the Little-JIL-to-BIR translator first finds out the step instances identified by the step identifier in the concrete event. Then it identifies the threads in which the BIR representations of those step instances reside. After that, the exact locations where the concrete event occurs are determined by the type of the concrete event. With the threads and locations, a set of thread location tests can be constructed. Eventually, the conjunction of such thread location tests for all concrete events in the property event binding forms the Boolean expression in the predicate.

As an example, let's look at how to translate property event binding `InfuseBlood -> Infuse Blood STARTED` mentioned earlier. The abstract event *InfuseBlood* is directly mapped to the identifier `InfuseBlood` in the predicate. For the concrete event, the step instance identified by the step identifier is "*Infuse Blood*". The BIR representation of this step instance is embedded in the main thread `T_Perform_Blood_Transfusion`. The location at which this concrete event occurs is the location with label `Infuse_Blood_STARTED`. Therefore the thread location test corresponding to the concrete event is `T_Perform_Blood_Transfusion@Infuse_Blood_STARTED`. As a result, the predicate translated from the property event binding is:

```
InfuseBlood:  T_Perform_Blood_Transfusion@Infuse_Blood_STARTED
```

Similarly, the property event binding `IdentifyPatient -> Identify Patient COMPLETED` is translated to:

```
IdentifyPatient:
      T_Perform_Blood_Transfusion@Identify_Patient_COMPLETED
```

3.3.4 Summary

In our realization of the process verification framework, we used an event-based model for the property specification because it maps nicely to process definition. Since property specifications are usually independent of any particular process definition, we assume that a property is specified over abstract events and map these events to the concrete events in the process based on property event binding specifications. This approach can be easily adapted for verifying processes defined in other languages if an event-based model for property specification is used. For state-based model, a similar approach can be applied. One can define properties in terms of abstract states and map them to concrete states in processes using property state bindings.

3.4 Optimizations

As noted, a human-intensive process often entails substantial amounts of concurrency and communication. It may also incorporate exception handlers to cope with various exceptional situations during the execution of the process. Handling of exceptions often leads to diversion of the normal control flow and causes the process to be more complicated. The state space of the process, therefore, becomes prohibitively large. Despite the fact that verification tools usually have various built-in optimizations, our preliminary experiments showed that direct translation of a process into a verifier's model representation often results in a model that easily exceeds the capability of that verifier. For instance, direct translation of a real-world blood transfusion process (comprised of about 120 steps) results in a BIR model containing about 20K lines of code, 315 threads and 253 Boolean variables. Neither FLAVERS nor SPIN is able to verify it. Therefore, it is necessary to use optimizations to reduce the translated model.

A specific property is usually only concerned with the temporal ordering of a few events in the process. To decide if the property holds, it is often sufficient to restrict the model representation to specific aspects of the process. Based on this observation, we proposed several optimizations that are able to greatly reduce the model, including *Step Abstraction*, *Step Removal*, *Thread Inlining*, *Exception Elimination*, and *Variable Reuse*. Except for *step abstraction*, these optimizations are sound, meaning that a process will not be reported to be consistent with a property unless that is indeed the case. *Step abstraction* is sound for safety properties, but is not sound for liveness properties in general. If the process can always finish, however, *step abstraction* is also sound for liveness properties. These optimizations are also precise in that they will not introduce spurious violations. Spurious violations are property violations in the optimized process that do not correspond to any real trace through the original process.

The optimizations that we have proposed are all Pre-GIMR optimizations. These optimizations are not new. For instance, *step abstraction* and *step removal* are two special cases of the *alphabet refinement* used in FLAVERS [43]. Being performed during the Little-JIL-to-BIR translation (Pre-GIMR), however, allows them to achieve more reduction in the model because they are able to take advantage of the scoping and hierarchical information in the Little-JIL process definition. This high-level information in the process might be lost or scattered over the model after the process is translated to the low-level BIR representation. Moreover, since an optimization might not be provided in all verifiers, implementing it in Little-JIL-to-BIR translation facilitates using different verifiers.

Before describing the details of each optimization, *relevant exceptions* and *relevant steps*, which are steps and exceptions relevant to events in the property, must be identified. This can be achieved by comparing steps and exceptions to the property event bindings for the given property. Specifically, exceptions that match exception throwing events in the property event bindings are relevant exceptions and steps that match step state events in the property event bindings are relevant steps. Steps that throw relevant exceptions are also considered to be relevant steps as are steps that define or use parameters that match parameter def/use events in property event bindings. In the simplified blood transfusion process, for property "*the nurse must identify the patient before infusing the blood*" and the property event bindings shown earlier, step "*Identify Patient*" and " Infuse Blood" are relevant steps because they match the step state event `Identify Patient COMPLETED` and `Infuse Blood STARTED` in the property event bindings respectively. Given the relevant steps and relevant exceptions, the optimizations basically abstract or remove various constructs (e.g. step instances, threads, variables, etc.), without changing the execution ordering of relevant step and relevant exceptions and, thus, without changing the ordering of property events.

3.4.1 Step Abstraction

An unrolled Little-JIL process is essentially a tree composed of steps. Steps relevant to events in the property are scattered throughout the tree. The step abstraction converts sub-trees that do not contain relevant steps into leaf steps. For a sub-tree that does not contain any relevant steps, the only way it can alter the order of events occurring in other part of the process is to throw exceptions. For instance, an exception from the sub-tree without an associated exception handlers at any ancestor could terminate the whole process, preventing the succeeding events from occurring. To preserve such effect, the leaf step that replaces the sub-tree should have the same exception declarations as the root step of the sub-tree. Remember that the exception declarations of a step declare the set of exceptions that may be thrown from the step. The exceptions that are handled by non-rethrown handlers within the sub-tree will not appear in the exception declarations of the root step of the sub-tree. Such exceptions should not be included in the exception declarations of the new leaf step because they have no impact on the occurrence of the relevant events.

In the simplified blood transfusion process, "*Identify Patient*" and "*Infuse Blood*" are the only relevant steps. Since the sub-tree with the root step "*Obtain Patient's Blood Type*" does not contain any relevant step, it can be abstracted to a leaf step. In addition, the exception *PatientBloodTypeUnavailable* is handled within "*Obtain Patient's Blood Type*". Therefore, the abstracted leaf does not throw any exception. Similarly, the sub-tree with the root step "*Check Blood Product*" is also abstracted to a leaf step. Note that the step "*Check Blood Product*" does not handle the exception *BloodExpired* and *WrongBlood* thrown by its sub-steps. Therefore, the abstracted leaf should also throw the exception *BloodExpired* and *WrongBlood.* The process after step abstraction is shown in the Figure 3.14.

Step abstraction is both sound and precise for safety properties. According to the definition of the safety property, a violation trace of a safety property always contains

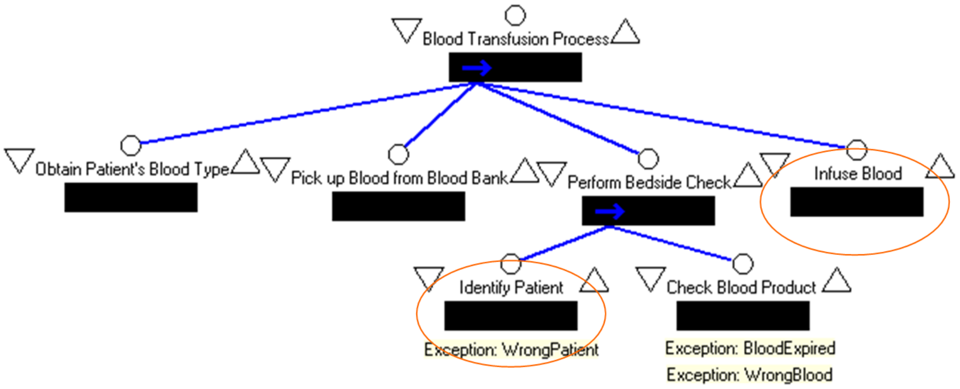


Figure 3.14. Simplified Blood Transfusion Process after Step Abstraction

a finite prefix that leads the property to the violation state [18]. Given one such trace in the original process, a corresponding trace in the abstracted process can always be constructed by replacing the part relevant to the execution of the replaced sub-tree with the execution of the abstracted leaf step. This trace also drives the property to the violation state because the ordering of the property events is conserved in this trace and both removed and inserted nodes do not contain any events in the property. This means that step abstraction is sound in the sense that it will not remove any violation trace in the original process. On the other hand, given a violation trace in the abstracted process, a violation trace in the original process can also be constructed by replacing the execution of abstracted leaf step with a corresponding execution of the sub-tree. This means that no spurious violations that do not correspond to any real traces through the original process will be introduced by step abstraction. Therefore, step abstraction is also precise for safety properties.

Step abstraction, however, is not sound with respect to liveness properties. The reason is that a violation trace of a liveness property may contain infinite loops. Since replacing a sub-tree with a leaf step might eliminate loops in the sub-tree, step abstraction may remove some violation traces through the original process. This is not a serious problem because many real-world processes, especially for the ones in the medical domain that interests us, have a hidden assumption that requires the processes to be able to finish. Under this assumption, the violation traces are finite even for liveness properties. Using the same reasoning for the safety property, it is easy to show that step abstraction becomes both sound and precise for liveness properties under that assumption.

3.4.2 Step Removal

After step abstraction, the Little-JIL process should only contain the following steps: (1) relevant steps and all their ancestors, (2) Leaf steps that are children of a step in (1). Step removal tries to further reduce the size of the process by removing some leaf steps in (2).

Step removal only considers the steps in (2) that do not throw exceptions and do not reference any relevant events. For a pre-requisite step or a post-requisite step, it can never change the execution order of events relevant to the property if it could not throw any exception. Therefore, it is safe to remove it from the process. For an exception handler, since the control flow badge decides the flow of control after the exception handler step is completed, step removal only removes the exception handler step instead of the whole exception handler. For a sub-step that does not throw exceptions, whether to remove it or not depends on the kind of the parent step.

- If the parent step is a sequential step or a parallel step, the sub-step can be removed because it does not have any impact on the execution of the other steps.

- If the parent step is a try step, however, the sub-step that does not throw exceptions should not be removed. According to the semantics of the try step, when a sub-step completes, the try step is completed and the next sub-step will not be executed. The only way that the next sub-step can be invoked is that the previous sub-step throws an exception handled by an exception handler with the *continue* badge. In other words, sub-steps following a sub-step that does

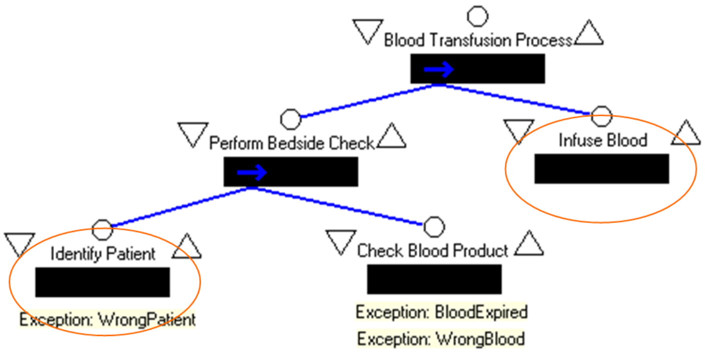


Figure 3.15. Simplified Blood Transfusion Process after Step Removal

not throw exception are not reachable. Therefore, in a well-formed process, a sub-step that does not throw exceptions can only be the last sub-step of a try step. If this sub-step is removed, a *NoMoreAlternative* exception will be thrown when an exception is thrown by the previous sub-step and the corresponding exception handler with the *continue* badge tries to invoke the next sub-step. Such *NoMoreAlternative* will not be raised if the last sub-step is not removed.

- If the parent step is a choice step, it is safe to keep only one such sub-step and remove the others. A choice step is similar to a try step in a sense that they both use the *continue* exception handler to invoke the next alternative, and will throw a *NoMoreAlternative* exception if no alternative can be executed. The difference is that the choice step randomly picks up a non-finished sub-step while the try step always selects the next sub-step. Therefore, a choice step may have multiple sub-steps that do not throw exception. It is easy to see that keeping one such sub-step is enough to preserve the effect of all of them.

In the simplified blood transfusion process after step abstraction, as shown in Figure 3.14, there are five leaf steps. Step "*Identify Patient*" and "*Infuse Blood*" cannot be removed because they are relevant steps. Step "*Check Blood Product*" may throw exceptions, and thus cannot be removed as well. Since step "*Obtain Patient's*

Blood Type" and "*Pick up Blood from Blood Bank*" are sub-steps of a sequential step and do not throw any exception, they will be removed by step removal. This results in the process shown in Figure 3.15.

It is easy to show that step removal is both sound and precise for safety properties using the same reasoning for step abstraction. Furthermore, since the steps removed are leaf steps, no loops in the original process will be removed. Therefore, step removal is both sound and precise for liveness properties as well.

3.4.3 Thread Inlining

During the translation, steps that may potentially run in parallel with other steps, including sub-steps of a parallel step, exception handling steps, and sub-steps of a choice step, are translated into BIR threads. A real-world process often contains many parallel steps and exception handling steps. Translation of the processes, therefore, often end up with a BIR model contains many threads and may run into the state explosion problem. Thread inlining is able to reduce the number of threads by inlining certain threads into other threads. It is based on an observation that the behaviors of the parallel step, the choice step, and the parent step of exception handling steps follow the same pattern: the parent step posts all its child steps (sub-steps or exception handling steps), and then is blocked until all those child steps finish (completed, terminated, opt-outed, or retracted). Although those child steps may run in parallel with each other, the parent step never interleaves with its child steps. Therefore, we can choose one child step and insert its thread into the thread corresponding to the parent step. Inlining threads like this is perfectly safe for both the safety property and the liveness property because it can neither eliminate existing paths from the original process, nor introduce new infeasible paths into it.

After step abstraction and step removal, the simplified blood transfusion process shown in Figure 3.15 does not contain any opportunity for thread inlining because

```
1  loc Check_Blood_Product_STARTED:
2     when true do { } goto Check_Blood_Product_fork_subs;
3  loc Check_Blood_Product_fork_subs:
4     when true do {
5        start(T_Check_Product_Info_Match_Patient_Info);
6     } goto Check_Expiration_Date_begin;
7  loc Check_Expiration_Date_begin:
8  ... // Code for Check Expiration Date
9  loc Check_Expiration_Date_exit:
10    when true do {} goto Check_Blood_Product_join_Sub2;
11 loc Check_Blood_Product_join_Sub2:
12    when true do {
13        join(T_Check_Product_Info_Match_Patient_Info);
14    } goto Check_Blood_Product_terminate_test_Sub2;
15 loc Check_Blood_Product_terminate_test_Sub2:
16    when
17      threadTerminated(
18        T_Check_Product_Info_Match_Patient_Info
19      )
20    do { } goto ⊔;
21 ... // Exception Handling
```

Figure 3.16. BIR Code After Thread Inlining

it only has one thread. For some other properties, however, it is possible that the parallel step "*Check Blood Product*" as well as its two sub-steps are not abstracted or removed. In this case, the thread for one of these two sub-steps can be inlined. Suppose that the thread for "*Check Expiration Date*" is selected to be inlined. The BIR code in Figure 3.16 shows how the thread for "*Check Expiration Date*" is inserted into the thread for the parent parallel step. The fork location (line 3-6) only starts threads for"*Check Product Info Match Patient Info*". Then instead of joining this thread, it goes to the first location in the thread corresponding to "*Check Expiration Date*". When the thread for "*Check Expiration Date*" exits, it goes to the location that starts the join of "*Check Product Info Match Patient Info*" (line 9-10).

The benefit of thread inlining is that it reduces the states that are created during verification. In finite-state verification, a state typically contains a program counter for each thread to keep track of the execution at the thread. The space for storing the program counter for a thread can be saved if the thread is removed. In addition, thread inlining can achieve even more savings for some verification tools such as FLAVERS. In FLAVERS, synchronizations between threads are explicitly captured using concurrency constraints represented as finite-state automata. The state also need to keep the states of those constraints. If a thread for a child step is inlined into the thread for its parent step, the concurrency constraint used to synchronize these two threads can be eliminated, resulting in further reduction of the state size.

3.4.4 Exception Elimination

A robust real-world process usually needs to consider many exceptional situations. Therefore, the process definition usually contains a large number of exceptions, which becomes a major source of the state explosion problem when verifying the process. For a particular property, however, some exceptions are equivalent in a sense that they have the same impact on the execution of the relevant steps or exceptions. To verify such a property, it is enough to keep only one such exception and eliminate the others.

To identify the sets of equivalent exceptions, we adopt a heuristic that checks equivalence of exceptions based on some simple rules. Specifically, two exceptions are considered to be equivalent if:

1. Neither of them is a relevant exception, and

2. They are thrown in the same leaf step. This rule is not as restrictive as it seems to be because after step abstraction, a sub-tree may be replaced by a leaf step that has all the exceptions propagated from the root of the sub-tree. And

3. They don't have exception handlers or they have equivalent exception handlers. For the second case, it is possible that an exception handler re-throws the exception that it catches, causing the exception to have multiple exception handlers

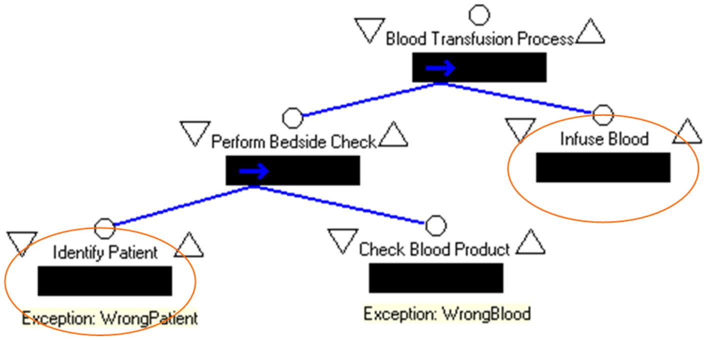


Figure 3.17. Simplified Blood Transfusion Process after Exception Elimination

at different ancestors. In this case, exception handlers at each ancestor step are equivalent. The equivalence of two exception handlers can be checked against the following conditions:

(a) Both handlers have the same kind of control flow badge, and

(b) Neither of them has an exception handling step, or both point to the same exception handling step.

The simplified blood transfusion process after step abstraction and step removal is shown in Figure 3.15. Step "*Check Blood Product*" throws two exceptions: *BloodExpired* and *WrongBlood*. These two exceptions are equivalent because 1) they are not relevant exceptions, 2) they are thrown in the same step, and 3) they don't have exception handlers. Therefore, any one of them can be removed. Figure 3.17 shows the process where *BloodExpired* is removed.

Exception elimination is both sound and precise for all properties. Since it does not introduce any traces to the optimized process, the exception elimination is sound. On the other hand, it is possible that certain violation traces may be removed by the exception elimination. Nevertheless, it is still precise in that for any violation trace in the original process that is removed, an equivalent trace can be constructed from the optimized process. There are two kinds of violation traces that can be removed: the trace where a single eliminated exception is thrown, and the trace where more than one equivalent exception is thrown. For a trace of the first kind, we can always find an equivalent trace in the optimized process by substituting the eliminated exception with the remaining exception. For a trace of the second kind, we can also find the equivalent trace as follows: if the remaining exception is not thrown in the trace, replace one eliminated exception with the remaining exception, and then remove the nodes relevant to all the eliminated exceptions. In both cases, since all eliminated exceptions and the remaining exception are not relevant exceptions, no trace nodes that reference the property events will be removed or introduced. Therefore, the trace constructed from the optimized process is equivalent to the corresponding removed trace with respect to the property.

3.4.5 Variable Reuse

The translated BIR model contains a variety of BIR variables, such as exception variables, parameter variables, variables indicating whether sub-steps of a parallel or choice step finishes or not, etc. Since these variables might be used in different threads, they are defined as global variables. By the variable reuse optimization, the number of variables in the BIR model can be reduced without changing the behavior of the model.

Variable reuse borrows the idea from the *register allocation* [23] in compiler optimizations. When a compiler generates a target program, it is often the case that the number of variables in the target program is much larger than the number of available CPU registers. In this case, some variables have to be saved to memory, and loaded back to registers later if needed. If two variables will not interfere with each other, however, we can use the same register to store both of them without changing the behavior of the program. A variable may interfere with another variable if both may live (hold a value that may be needed in the future) at the same time. Based on this idea, the register allocation tries to allocate as many variables as possible to registers to reduce the expensive memory access operations. It first constructs a interference graph, which is an undirected graph where a node corresponds to a variable in the target program and an edge connects two nodes whose variables may interfere with each other. Then an existing graph coloring algorithm can be applied to identify the minimal number of registers to cover all variables.

In the case of variable reuse, the variables in the original BIR model are analogous to the variables in the target program of the compiler, and the BIR variables in the optimized model are similar to the CPU registers. The differences are the definition of inference relationships and the way to compute them. (To make the following discussion easier, we refer to variables in the original BIR model as "BIR variables" and variables in the optimized model as "register variables".) In variable reuse, two variables are considered to interfere with each other if any one of the following condition is satisfied:

1. *They have different types* Note that even though range types may have different range, they are treated as the same type because we can always use a register variable with the highest range to hold values of those range types.

2. *Any one is relevant to a property event* Some variables may be associated with certain property events. For instance, an exception variable may correspond to an exception throwing event. Share the register variable with other variables could create spurious events in the optimized model.

3. *They may live at the same time* This is the same as in the register allocation.

The first two conditions are easy to decide. The current implementation employs a much simpler approach to check the third condition. Instead of applying the liveness analysis, as in register allocation, this approach takes advantage of scoping information provided by the Little-JIL process. Specifically, each variable is associated with

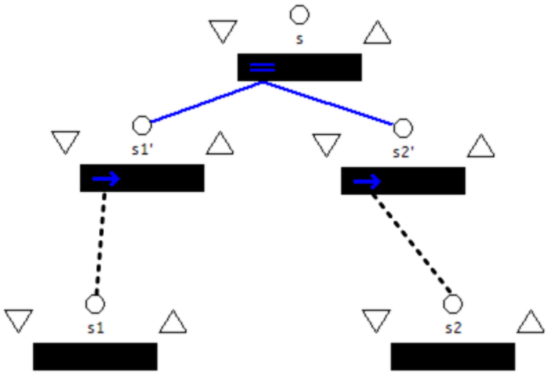


Figure 3.18. Check Whether Two Steps May in Parallel

one or more step instances in the process that indicate the scope in which the variable is alive. For example, an exception variable is associated with the step that throws the corresponding exception and the exception handling step, if one exists. To decide whether two variables may be live at the same time becomes a simple check to determine whether their associated steps may execute in parallel. Given two steps s_1 and s_2, as shown in Figure 3.18, their least common ancestor s can be identified following the tree structure of the process. s_1' is an ancestor of s_1, and s_2' is an ancestor of s_2. Both s_1' and s_2' are children of s. We conservatively assume that s_1 and s_2 may run in parallel if s_1' and s_2' may run in parallel. To decide if s_1' and s_2' may run in parallel, we only need to check if s_1' and s_2' are sub-steps and s is a parallel step, or s_1' and s_2' are exception handling steps. Theoretically, this simple approach is not as precise as liveness analysis. There are two reasons why we prefer to use it however. One is that to perform the liveness analysis, we need to construct the global control flow graph, which is not available during the translation. The other reason is that in our experience, it is quite effective and can lead to great reduction of the number of variables. Once the interference relationships between variables are decided, the variable reuse can find the minimal set of register variables using existing graph coloring algorithms.

For the simplified process shown in Figure 3.17, two exception variables, corresponding to exception *WrongPatient* and *WrongBlood*, are created in the BIR model. Since the parent step is a sequential step, "*Identify Patient*" and "*Check Blood Product*" will never run in parallel. The two exception variables thus will not live at the same time. Since both exceptions are not relevant to the property events, variable reuse only assigns one register variable for them.

After optimizations including step abstraction, step removal, exception elimination and variable reuse, the simplified blood transfusion process is translated to the BIR program shown in the following figure. Comparing this program to the one without optimizations (as shown in Appendix C), it is obvious that those optimizations achieve significant reduction. Three threads (one main thread and two threads corresponding two sub-steps of parallel step "*Check Blood Product*") are reduced to a single main thread, six variables (four exception variables and two done variables for two sub-steps of parallel step "*Check Blood Product*") are reduced to one, and total lines of BIR code is reduced from 217 to 71.

```
process Process_Blood_Transfusion_Process ()
  b_e1: boolean:= false;

  main thread T_Blood_Transfusion_Process
    loc Blood_Transfusion_Process_POSTED:
      when true do { }
        goto Blood_Transfusion_Process_STARTED;
    loc Blood_Transfusion_Process_STARTED:
      when true do { }
        goto Perform_Bedside_Check_POSTED;
    loc Perform_Bedside_Check_POSTED:
      when true do { }
        goto Perform_Bedside_Check_STARTED;
    loc Perform_Bedside_Check_STARTED:
      when true do { }
        goto Identify_Patient_POSTED;
    loc Identify_Patient_POSTED:
      when true do { }
        goto Identify_Patient_STARTED;
    loc Identify_Patient_STARTED:
      when true do { }
        goto Identify_Patient_COMPLETED;
      when true do { b_e1 :=true ; }
        goto Identify_Patient_TERMINATED;
    loc Identify_Patient_COMPLETED:
      when true do { }
        goto Check_Blood_Product_POSTED;
    loc Identify_Patient_TERMINATED:
      when ( b_e1==true ) do ()
        goto Perform_Bedside_Check_TERMINATED;
    loc Check_Blood_Product_POSTED:
      when true do { }
        goto Check_Blood_Product_STARTED;
    loc Check_Blood_Product_STARTED:
      when true do { }
        goto Check_Blood_Product_COMPLETED;
      when true do { b_e1 :=true ; }
        goto Check_Blood_Product_TERMINATED;
    loc Check_Blood_Product_COMPLETED:
      when true do { }
        goto Perform_Bedside_Check_COMPLETED;
    loc Check_Blood_Product_TERMINATED:
      when ( b_e1==true ) do ()
        goto Perform_Bedside_Check_TERMINATED;
    loc Perform_Bedside_Check_COMPLETED:
      when true do { } goto Infuse_Blood_POSTED;
    loc Perform_Bedside_Check_TERMINATED:
      when ( b_e1==true ) do { }
        goto Blood_Transfusion_Process_TERMINATED;
    loc Infuse_Blood_POSTED:
      when true do { } goto Infuse_Blood_STARTED;
    loc Infuse_Blood_STARTED:
      when true do { } goto Infuse_Blood_COMPLETED;
    loc Infuse_Blood_COMPLETED:
      when true do { }
        goto Blood_Transfusion_Process_COMPLETED;
    loc Blood_Transfusion_Process_COMPLETED:
      when true do { }
        goto T_Blood_Transfusion_Process_exit;
    loc Blood_Transfusion_Process_TERMINATED:
       when true do { }
        goto T_Blood_Transfusion_Process_exit;
    loc T_Blood_Transfusion_Process_exit:
```

```
  end T_Blood_Transfusion_Process;

predicates
  IdentifyPatient =
      T_Blood_Transfusion_Process@Identify_Patient_COMPLETED;
  InfuseBlood =
      T_Blood_Transfusion_Process@Infuse_Blood_STARTED;
end Process_Blood_Transfusion_Process;
```

3.4.6 Summary

In this section, we presented several optimizations used to reduce the translated models. Our evaluation, discussed later in this thesis, showed that these optimizations significantly improved the scalability of our implementation of the process verification framework. All properties for two complex real-world processes were verified in a few seconds. Although these optimizations are described based on the Little-JIL and BIR, they are pretty general and can be easily used in the translation of other process definition languages. As noted earlier, all of these optimizations are Pre-GIMR optimizations and have the benefit of applying to different verifiers. Unfortunately, we did not find any opportunity for new Post-GIMR optimizations. As discussed in future work, we plan to investigate and incorporate more optimizations into the process verification framework.

CHAPTER 4

FAULT-TREE ANALYSIS

A human-intensive process consists of tasks that are performed by various agents, including hardware devices and human agents. The process verification framework discussed in the previous chapter primarily detects event sequence errors. It assumes that the tasks are done correctly and is concerned with the ordering of events occurred at some tasks, as specified by the given temporal properties. Due to hardware failure or human error, however, faults may be introduced during the execution of a task. If not detected and corrected, such a fault may propagate to the successors without violating those temporal properties and eventually lead to hazards. "Hazard" is a term used in the system safety analysis community. It is defined as "a state or set of conditions of the system that, together with certain other conditions in the environment, will lead inevitably to an accident" [80]. The accident usually results in harm to people or damage to property. For example, a hazard for a blood transfusion process could be "the blood unit to be transfused to the patient is wrong". This hazard could cause transfusion reaction that endangers the patient's life. Developing such a safety critical process also requires to prevent or control potential hazards. This can be achieved by incorporating various mechanisms to prevent or control faults that could lead to the hazards. For instance, a failure-resistant agent could be assigned to some tasks where major faults could occur. Additionally, consistency checks could be added to well-chosen places in the process to stop the propagation of faults. To add such mechanisms into the process, the process developers need to identify the potential hazards that could occur in the process, as well as the faults that could lead to those hazards. Due to resource limitations or other constraints, the process developers are usually only able to apply the fault prevention mechanisms to the most important faults, which requires assessing and prioritizing the faults. This kind of analysis (i.e. identifying and assessing hazards and faults) is called "hazard analysis" in the system safety analysis community. So far, many hazard analysis techniques have been proposed and successfully applied in several industries [80]. We believe that these techniques can also be adopted to analyze safety critical processes.

In this work, we investigated a selected hazard analysis technique called Fault Tree Analysis (FTA) [131]. Given a potential hazard in a system, FTA deductively identifies events (faults or conditions) in the system that could lead to the hazard and produces a fault tree, which provides a graphical depiction of all possible parallel and sequential combinations of those events. Once a fault tree has been derived, qualitative and quantitative evaluation can be applied to provide information, such as specific sequences and sets of events that are sufficient to cause a hazard and overall system vulnerability. This information can then be used as guidance for improvement of the design or implementation of the system. To apply FTA on processes, we developed a process FTA framework that is able to automatically derive and evaluate fault trees for processes modeled in a process definition language. It also provides supports that help process developers understand the analysis results.

4.1 Overview of Process Fault-Tree Analysis

Fault-tree analysis involves three steps: (1) *defining the system*, (2) *deriving the fault tree*, and (3) *evaluating the fault tree*. Since systems in many domains are defined in informal notations, the fault tree derivation has to be performed manually by a group of experts and therefore is time-consuming, costly and error-prone. Some researches, however, have demonstrated that this step can be automated if the system is completely and precisely defined. As discussed in the background chapter, several template-based approaches have been proposed to automatically derive fault trees from software systems defined in various specification languages. Since processes definitions are essentially software, such template-based approaches may as well be applied to automatically construct fault trees from rigorous process definitions. To show this, we developed an automatic fault tree derivation algorithm for the Little-JIL process definition language. During our research, we encountered three issues that, as far as we know, were not well addressed in the previous software fault tree amalysis approaches.

- **Scalability** A human-intensive process usually involves complex control and artifact flows. The fault tree derived from it can easily become too large to either be understood by humans or be analyzed by fault tree analyzing tools. To address this issue, two optimizations are proposed to reduce the size of the process before the fault tree is derived. In addition, an optimization is applied to remove events from the fault tree after it is derived.

- **Looping Constructs** Processes often contain steps that are executed repeatedly. Fault trees derived from these processes, therefore, may contain loops (i.e. an input and the output of an event connect to the same event). Since a fault tree with loops is no longer a tree, existing techniques for analyzing fault trees cannot be applied to it. To address this issue, we proposed an approach to remove loops from the derived fault trees. As discussed later in this chapter, this approach does not degrade the results.

- **NOT Gate** Traditionally, NOT gates are not used in fault trees because they may introduce difficulties to the evaluation of fault trees. Without NOT gates, however, some relationships between events cannot be captured in the fault tree. This may cause the fault-tree analysis tools to produce inaccurate results. To improve the accuracy of analysis results, we introduced to use NOT gates in our fault tree derivation and evaluation approach.

- **Usability** Once a fault tree has been derived, various qualitative and quantitative evaluation techniques can be applied to evaluate the fault tree. In this work, we investigated the most commonly used technique – *Minimal Cut Set* (MCS) analysis. For complex real-world processes, it is often very difficult to understand how events in a MCS cause the hazard to occur. To address this usability issue, we proposed two complimentary representations that help process developers to easily understand the MCSs produced by the MCS analysis.

The remainder of this chapter is organized as follows. Section 4.2 presents our automatic fault tree derivation algorithm for the Little-JIL process definition language. Section 4.3 gives a brief introduction to the MCS analysis. Section 4.4 discusses the approaches that we proposed to handle four issues in process fault tree analysis mentioned above. Finally, the limitation of our approach is discussed in Section 4.5.

4.2 Automatic Fault Tree Derivation

As discussed in the Background and Related Work chapter, a fault tree shows all the parallel and sequential combinations of events that could lead to the given hazard. The basic elements of a fault tree are *events* and *gates*. Events are used to represent faults, such as component failures, human errors, or other pertinent conditions in the system or environment. *Basic events* are basic initiating faults or conditions. *Undeveloped events* are events that are not developed any further, either because necessary information for deriving the fault tree leading to these events is unavailable or because these events are considered to have insignificant consequence. Basic events and undeveloped events are also called *primary events* because they require no further development. As opposed to primary events, *intermediate events* are events that need to be developed. In a fault tree, events are connected using various *gates*. Each gate connects one or more input events to a single output event.

To derive a fault tree from a process, the given hazard is represented as an intermediate event called the TOP event. Starting with a fault tree that only cotains the TOP event, the fault tree derivation procedure proceeds to expand the fault tree by developing intermediate events in it. An intermediate event is developed by investigating the process to identify the immediate, necessary, and sufficient events that cause this event, and then connecting those events to it via a proper gate. The new events may also be intermediate events that need to be developed. The derivation procedure terminates when no undeveloped intermediate events is left in the fault tree. To automate this derivation procedure, the process has to be precisely specified. In addition, two problems need to be solved:

- How to automatically discover events that could possibly occur in the various steps of the process. This could be difficult because the tasks performed by different steps are quite different, and thus so are the events that could occur in these steps.
- Given an intermediate event, how to automatically identify events that could cause this event and connect them to this events using appropriate gates.

To address these problems, we propose a template-based automatic fault tree derivation approach for Little-JIL process definitions. In this approach, the first problem is solved by classifying events that could possibly occur into several pre-defined types so that they can be identified based on the simple uniform step interfaces specified in Little-JIL process definitions. The second problem is solved by defining a set of templates according to the semantics of the Little-JIL process definition language. Each template is used to develop a particular type of event. With these event types and templates, a simple work-list algorithm can be applied to automatically derive

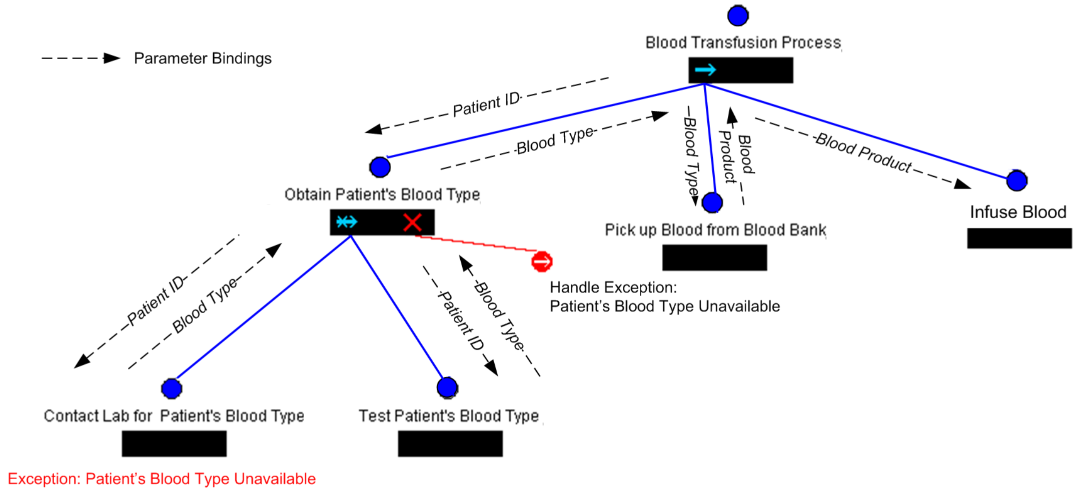

Parameter Bindings
Blood Transfusion Process
Patient ID
Blood Type
Blood Type
Blood Product
Blood Product
Obtain Patient's Blood Type
Pick up Blood from Blood Bank
Infuse Blood
Handle Exception:
Patient's Blood Type Unavailable
Patient ID
Blood Type
Patient ID
Blood Type
Contact Lab for Patient's Blood Type
Test Patient's Blood Type
Exception: Patient's Blood Type Unavailable

Figure 4.1. Simplified Blood Process

fault trees from the Little-JIL process definition. Although the events and templates are specific to the Little-JIL language, the basic idea of this template-based fault tree derivation approach may be applied to automatically construct fault trees for other process definition languages.

In the remainder of this section, we first present a simplified version of a blood transfusion process that will be used as a running example in the discussion. Then we show the set of pre-defined events types. We also discuss a few templates and show how to apply them to develop events. After that, we describe the work-list algorithm that derives the fault trees using the templates. Finally, we discuss three major issues faced by the derivation algorithm as well as our approaches to address these issues.

Figure 4.1 shows the simplified blood transfusion process presented in the Background and Related Work chapter. For simplicity, step "*Perform Bedside Check*" as well as its two sub-steps are removed. In this figure, arrows with dashed lines are used to illustrate the parameter bindings, which play an important role in fault tree derivation. To keep the figure simple, parameters of steps are not displayed. Such parameters, however, can easily be inferred from the parameter bindings. For example, there is a parameter binding for *Patient' ID* from step "*Blood Transfusion Process*" to "*Obtain Patient's Blood Type*". Therefore, both steps must have a parameter *Patient's ID*. A serious hazard that could occur in a blood transfusion process is that the blood product to be transfused to the patient is wrong (e.g., does not match the patient's blood type). This could cause a transfusion reaction that endangers the patient's life. We illustrate how to apply our approach to derive the fault tree for this hazard and show how the analysis result can help to improve the safety of the process.

4.2.1 Events

In fault-tree analysis, events usually represent faults of components in the system. In addition to faults, some conditions in the environment are also considered to be events if they may affect the propagation of faults. Steps are the basic building blocks of Little-JIL process definitions. Based on the simple uniform interfaces of Little-JIL steps, we proposed several pre-defined types of events, such as "*All Inputs are Correct, But S Produces Incorrect Output o*", "*Artifact o is Wrong When S is Posted*" and "*Exception e is Thrown by S*". In this work, we limit our attention to the faults related to artifacts in Little-JIL process definitions. We believe that a large number of interesting faults are artifacts faults or can easily be turned into artifact faults. For example, in many human-intensive processes, some hazards are caused by the delay of certain steps. To capture such faults, we can associate an artifact representing the execution time to each step. Then our approach then can handle unacceptable delays as an unacceptable output artifact value. For the faults that can not be easily turned into artifact faults, our framework is designed in such a way that a new kind of fault can be supported if the type of the event representing that fault is defined and the appropriate template used to develop this type of event is provided.

The events that we proposed fall into three categories.

4.2.1.1 Category 1

A fault may be introduced to a process in two ways. It may be caused by the incorrect execution of a step in the process. It may also be generated outside the process and passed to the process as certain wrong artifact. Category 1 events represent the faults in the first case. The faults in the second case are represented as events of type *Event Type 2: Artifact o is Wrong When S is Posted* defined in Category 2.

Event Type 1: All Inputs are Correct, But S Produces Incorrect Output o

In this type of event, the step S is a leaf step and o is an output artifact of S. In Little-JIL, only a leaf step is able to manipulate its input artifacts and create or change its output artifacts. Whereas non-leaf steps are only used to coordinate their sub-steps and passing artifacts. Without losing generality, we assume that no faults could occur during artifact passing. Unreliable artifact passing can be explicitly modeled using additional leaf steps, so a fault that could occur during artifact passing can be defined as this type of event. Therefore, this type of event can only occur at leaf steps, representing the possibility that designated agents fail to execute those steps as required. Since we do not look into the implementation of leaf steps, this events of this type are always primary events.

4.2.1.2 Category 2

The second category contains events indicating that an artifact is incorrect when a step is at a particular step state, including:

Event Type 2: Artifact o is Wrong When S is Posted

Event Type 3: Artifact o is Wrong When S is Started

Event Type 4: Artifact o is Wrong When S is Retracted

Event Type 5: Artifact o is Wrong When S is Opted-out

Event Type 6: Artifact o is Wrong When S is Completed

Event Type 7: Artifact o is Wrong When S is Terminated by Exception e

These types of events are usually intermediate events that serve as the intermediate nodes on the propagation paths of faults generated by the events in Category 1. The only exception is when the step S in *Event Type 2* is the root step of the process. In this case, the artifact is already wrong when it is passed into the process. Since the error in the artifact is not introduced within this process, such an event cannot be developed based on this process and therefore is treated as a primary event. It should also be noted that *Event Type 7* has an additional parameter "*exception e*". The reason is that a step might throw more than one exception, which may lead to different execution paths of the process. If we do not distinguish different exceptions, the corresponding paths will be mixed together, which may introduce imprecision to the fault tree.

The last type of event in this category does not correspond to any step state defined in the Little-JIL specification. It says:

Event Type 8: Artifact o is Wrong When S is about to be Completed

An event of this type occurs after S is started and before S is completed. This event type is introduced to break the large template for *Event Type 6: Artifact o is Wrong When S is Completed* into four small templates: *Template 6* and *Template 7* for *Event Type 6: Artifact o is Wrong When S is Completed* and *Template 10* and *Template 11* for *Event Type 8: Artifact o is Wrong When S is about to be Completed.* These templates are discussed in Appendix B. Introducing this type of event not only simplifies the code to generate the fault tree, but also makes the templates much easier to describe and understand.

4.2.1.3 Category 3

Since a fault that leads to the hazard along one path may not cause the hazard to occur on another path, it is necessary to introduce events to represent conditions that might change the control flow of the process. In Little-JIL, there are several cases where the flow of control may be changed. The first case is the exception. When an exception is thrown, the control flow is diverted to the exception handler. Therefore, two types of events are introduced to represent whether an exception is thrown or not.

Event Type 9: Exception e is Thrown by S

Event Type 10: Exception e is not Thrown by S

Another case is that an optional step may be opted out by the agent assigned to the step. For instance, suppose a step has a cardinality (2..5), so five instances of the step may be created during the execution. The first two instances are required to be executed. The remaining three instances are optional. So the agent has the choice not to perform those three instances. Therefore, two types of events are used to represent whether a step is opted out nor not.

Event Type 11: Step S is Opted-out

Event Type 12: Step S is not Opted-out

The third case is that some step may be retracted under certain situations. For example, a posted sub-step of a parallel step will be retracted if another sub-step throws an exception. Posted sub-steps of a choice step will be retracted if they are not chosen to be executed. Similarly, two types of events are introduced to represent whether a step is retracted nor not.

Event Type 13: Step S is Retracted

Event Type 14: Step S is not Retracted

Given the interface of a step, the events that could occur in that step can be easily identified. Let's take step "*Contact Lab for Patient's Blood Type*" in the simplified blood transfusion process as an example. Its interface specifies that it has an input parameter *Patient ID* and an output parameter *Blood Type*. It may also throw an exception *Patient's Blood Type Unavailable*. Based on this interface, events that could occur at this step include:

- *All Inputs are Correct, But Contact Lab for Patient's Blood Type Produces Incorrect Blood Type*
- *Patient ID is Wrong When Contact Lab for Patient's Blood Type is Posted*
- *Patient ID is Wrong When Contact Lab for Patient's Blood Type is Started*
- *Patient ID is Wrong When Contact Lab for Patient's Blood Type is about to be Completed*
- *Blood Type is Wrong When Contact Lab for Patient's Blood Type is Completed*
- *Blood Type is Wrong When Contact Lab for Patient's Blood Type is Terminated by Exception Patient's Blood Type Unavailable*
- *Exception Patient's Blood Type Unavailable is Thrown by Contact Lab for Patient's Blood Type*
- *Exception Patient's Blood Type Unavailable is not Thrown by Contact Lab for Patient's Blood Type*

While we can identify all the events in a process in this way, we do not have to do so because a large number of them may have nothing to do with the given hazard. In our fault tree derivation approach, events are generated on the fly by applying various templates and the design of templates ensures that only events that are relevant to the hazard will be generated.

To derive a fault tree, the hazard needs to be specified as an event, usually called the TOP event. In our approach, the TOP event is required to be defined as an event in Category 2. For the simplified blood transfusion process, the hazard of interest is "the blood product to be transfused to the patient is wrong", which can be translated into the event "*Blood Product is Wrong When Infuse Blood is Started*".

4.2.2 Templates

The fault tree derivation is an iterative procedure. A single intermediate event is developed in each iteration. This is done automatically by applying the appropriate pre-defined template. Based on the Little-JIL semantics, we develop a collection of templates, each of which is used to develop a particular type of events.

A template consists of a *requirement* and a *partial fault tree.* The *requirement* specified the conditions under which the template can be applied. The *partial fault tree* demonstrates how to develop an event. The root event of the partial fault tree is the event to be developed. The leaves of the partial fault tree are the immediate and necessary events that could lead to the root event. "Immediate" means that those events should occur at step states immediately preceding the step state associated with the root event (each event is explicitly or implicitly associated with a specific step state, as shown in Section 4.2.1.) This partial fault tree has a generic form, which needs to be instantiated when applied to develop an event. To develop an event, the appropriate template will be identified according to the type of the event as well as certain attributes of the step or the artifact in the event, such as whether the step is a leaf step and whether the artifact is an output parameter of the step. Then the partial fault tree of the template is instantiated according to the attributes of the step and the artifact in the event. Finally, the event is replaced by the partial fault tree instance.

The corresponding templates for different types of events are shown as Table 4.1. Note that one type of event might have more than one template. Each template is applicable for a particular situation. The templates are carefully designed so that for any intermediate event, there is exactly one template that is applicable. Some types of events are always primary events that do not need to be developed. Therefore, no templates are associated with them.

Due to space limitation, we only discuss three selected templates in this section. The detailed description of all the templates is given in Appendix B. In the discussion of templates, we use "child steps" to refer to sub-steps, pre-requisite steps, post-requisite steps, and exception handler steps. Child steps of the same parent step are considered to be “siblings” of each other. Before discussing templates, it is necessary to first give the definitions of the *Reverse Control Flow Graph* and the *Artifact Flow Graph*, which are used when instantiating the partial fault tree in a template.

4.2.2.1 Reverse Control Flow Graph and Artifact Flow Graph

The *Reverse Control Flow Graph (RCFG)* is used to find the previous step states of a given step state. It is a directed graph $G_c =< N_c, E_c >$, where N_c is the set of states of step instances in the process and E_c is the set of edges. There is an edge from n_1 to n_2 iff the step state n_2 may immediately precedes the step state n_1 according to the Little-JIL process definition. In other words, the edge goes in the opposite direction of the ordinary control flow. Figure 4.2 shows the reverse control flow graph of the example process.

The *Artifact Flow Graph (AFG)* is used to decide whether an artifact is data dependent on another artifact. It is a directed graph $G_a =< N_a, E_a >$, where N_a is

Table 4.1. Fault Tree Derivation Templates

Event Type		**Template(s)**
Event Type 1	*All Inputs are Correct, But S Produces Incorrect Output o*	None
Event Type 2	*Artifact o is Wrong When S is Posted*	Template 1, 2
Event Type 3	*Artifact o is Wrong When S is Started*	Template 3
Event Type 4	*Artifact o is Wrong When S is Retracted*	Template 4
Event Type 5	*Artifact o is Wrong When S is Opted-out*	Template 5
Event Type 6	*Artifact o is Wrong When S is Completed*	Template 6, 7
Event Type 7	*Artifact o is Wrong When S is Terminated by Exception e*	Template 8, 9
Event Type 8	*Artifact o is Wrong When S is about to be Completed*	Template 10, 11
Event Type 9	*Exception e is Thrown by S*	None
Event Type 10	*Exception e is not Thrown by S*	Template 12
Event Type 11	*Step S is Opted-out*	None
Event Type 12	*Step S is not Opted-out*	Template 13
Event Type 13	*Step S is Retracted*	None
Event Type 14	*Step S is not Retracted*	Template 14

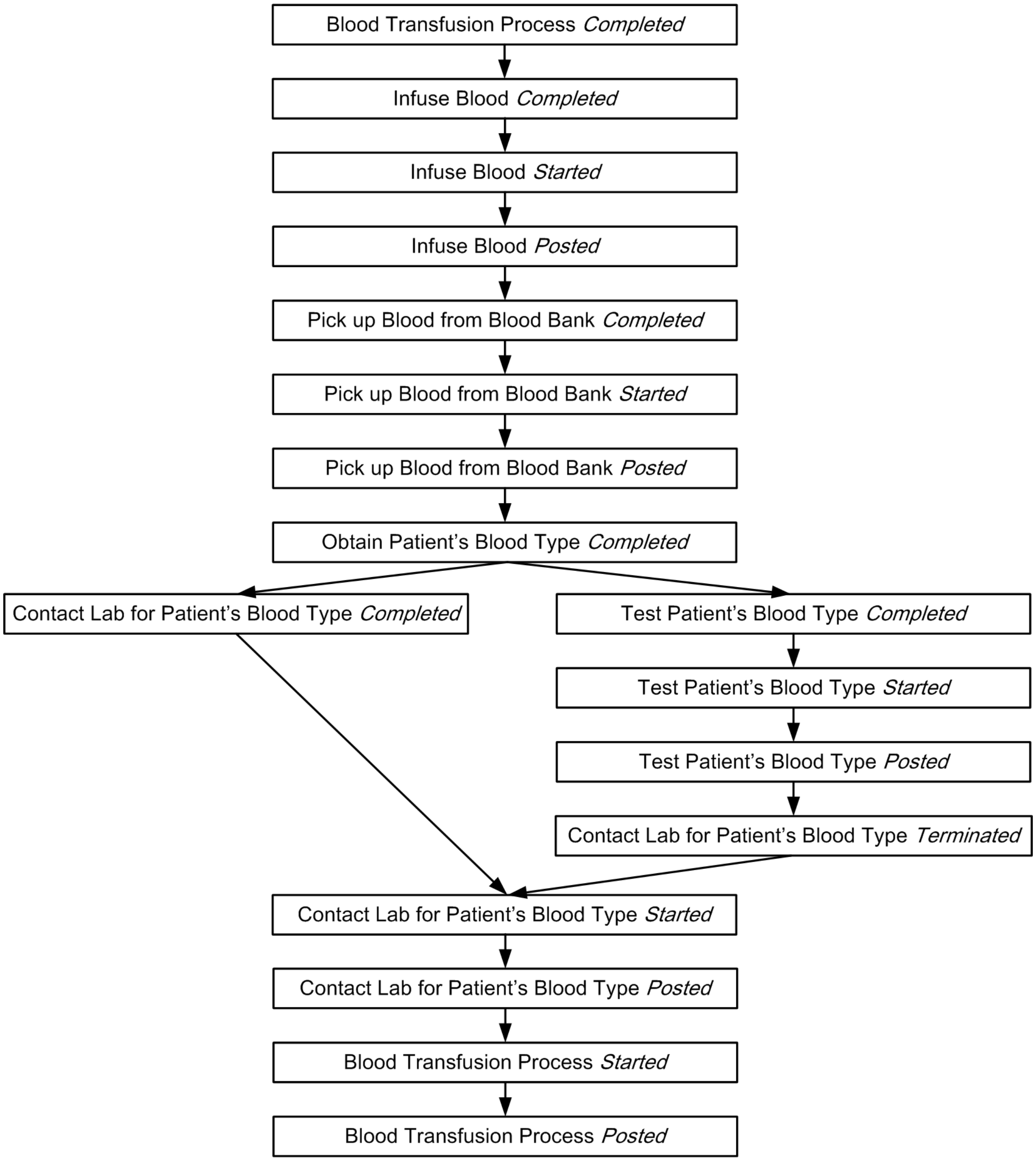

Blood Transfusion Process *Completed*
Infuse Blood *Completed*
Infuse Blood *Started*
Infuse Blood *Posted*
Pick up Blood from Blood Bank *Completed*
Pick up Blood from Blood Bank *Started*
Pick up Blood from Blood Bank *Posted*
Obtain Patient's Blood Type *Completed*
Contact Lab for Patient's Blood Type *Completed*
Test Patient's Blood Type *Completed*
Test Patient's Blood Type *Started*
Test Patient's Blood Type *Posted*
Contact Lab for Patient's Blood Type *Terminated*
Contact Lab for Patient's Blood Type *Started*
Contact Lab for Patient's Blood Type *Posted*
Blood Transfusion Process *Started*
Blood Transfusion Process *Posted*

Figure 4.2. Reverse Control Flow Graph for Simplified Blood Transfusion Process

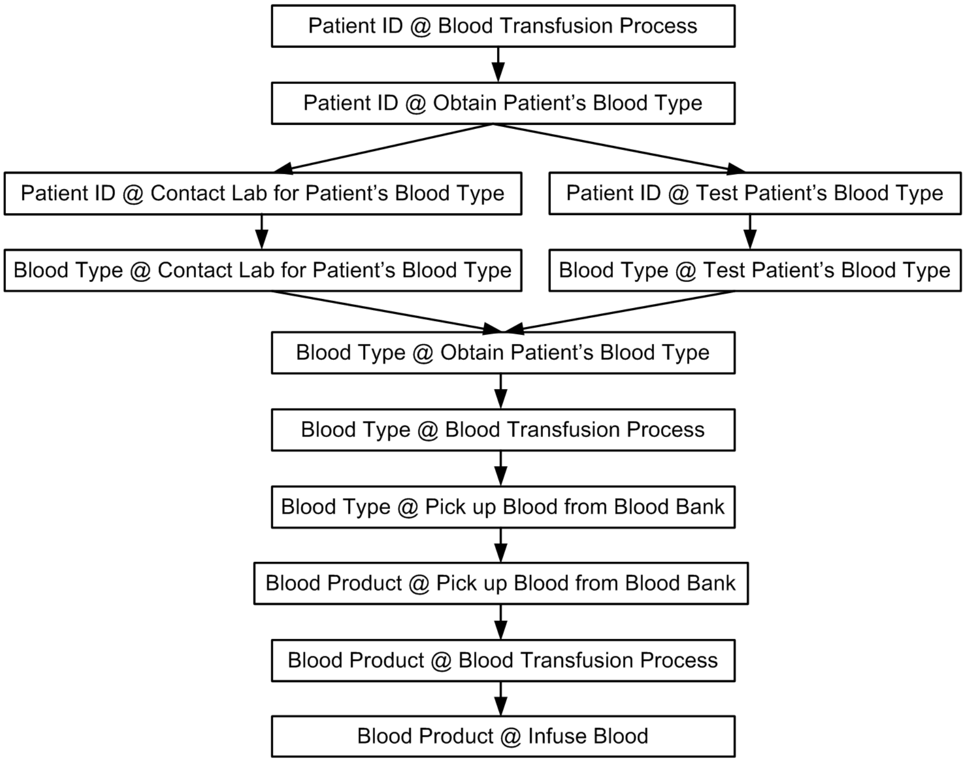

Patient ID @ Blood Transfusion Process
Patient ID @ Obtain Patient's Blood Type
Patient ID @ Contact Lab for Patient's Blood Type
Patient ID @ Test Patient's Blood Type
Blood Type @ Contact Lab for Patient's Blood Type
Blood Type @ Test Patient's Blood Type
Blood Type @ Obtain Patient's Blood Type
Blood Type @ Blood Transfusion Process
Blood Type @ Pick up Blood from Blood Bank
Blood Product @ Pick up Blood from Blood Bank
Blood Product @ Blood Transfusion Process
Blood Product @ Infuse Blood

Figure 4.3. Artifact Flow Graph for Simplified Blood Transfusion Process

the set of artifact instances in the process and E_c is the set of edges. There is an edge from n_1 to n_2 iff the artifact n_2 may have an immediate data dependency on n_1. For instances, suppose both n_1 and n_2 are parameters, n_2 may depend on n_1 if 1) there is a parameter binding indicating that n_1 is passed to n_2, or 2) n_1 is an input parameter and n_2 is an output parameter of the same step. In general, if there is a path from an artifact n_1 to the artifact n_2 in the graph, we know that the fault in n_1 may be propagated to n_2. Figure 4.3 gives the artifact flow graph for the simplified blood transfusion process. Note that parameters of different steps might have the same name. For example, "*Blood Transfusion Process*", "*Pick up Blood from Blood Bank*", and "*Perform Transfusion*" all have a parameter called *Blood Product*. Therefore, we use "Parameter Name @ Step Name" to distinguish them.

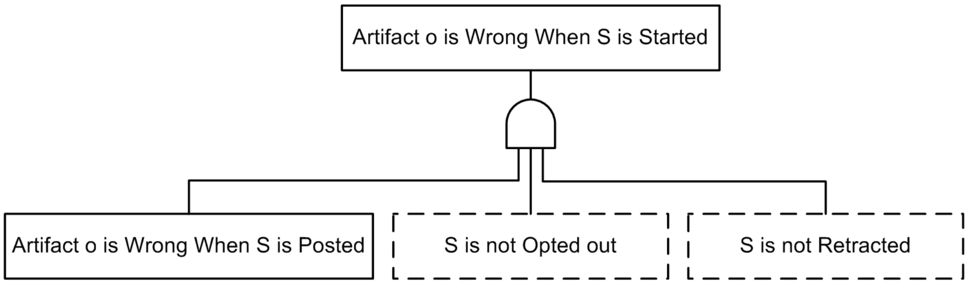

Artifact o is Wrong When S is Started
Artifact o is Wrong When S is Posted
S is not Opted out
S is not Retracted

Figure 4.4. Partial Fault Tree for Template 3

4.2.2.2 Template for *Artifact o is Wrong When S is Started*

The event "*Artifact o is Wrong When S is Started*" is developed by Template 3.

Template 3

Requirement:

- The event e to be developed is *Artifact o is Wrong When S is Started*

Partial Fault Tree:

The partial fault tree for this template is shown in Figure 4.4. In Little-JIL, the only step state that may immediately precedes the state `<S, Started>` is `<S, Posted>`. Since artifacts cannot be changed between these two states, event "*Artifact o is Wrong When S is Started*" can occur only if the artifact o is already wrong when the step S is posted. On the other hand, however, event "*Artifact o is Wrong When S is posted*" does not always lead to "*Artifact o is Wrong When S is Started*". The step S may be a retractable step, e.g. S is a sub-step of a parallel step. S may go from the posted state to the retracted state when another sub-step throws an exception. When this occurs, the faulty o will not be propagated to the state `<S, Started>`. The step S may also be an optional step. The agent responsible for S is allowed to opt out of the posted step S. In this case, event "*Artifact o is wrong when S is posted*" will not lead to "*Artifact o is Wrong When S is Started*" as well. Therefore, to allow the faulty o to be propagated from the posted state to the started state, S

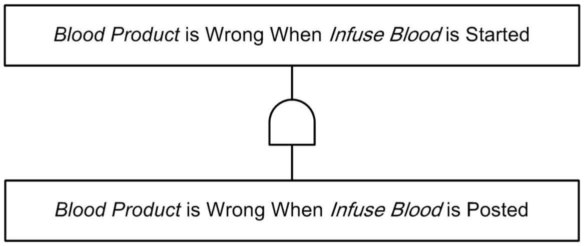

Blood Product is Wrong When *Infuse Blood* is Started
Blood Product is Wrong When *Infuse Blood* is Posted

Figure 4.5. Instantiated Partial Fault Tree for Template 3

should not be retracted or opted-out. Notice that the nodes for event "*Step S is not Opted out*" and "*Step S is not Retracted*" are drawn using dash lines. This means that these two events may or may not be present, depending on if the step S is optional or retractable. "*Step S is not Opted out*" is present iff S is an optional step, and "*Step S is not Retracted*" is present iff S is a retractable step.

This template can be used to develop the TOP event "*Blood Product is Wrong When Infuse Blood is Started*" for the example process. Since step "*Infuse Blood*" is neither optional nor retractable, the TOP event can only be caused by "*Blood Product is Wrong When Infuse Blood is Posted*", as shown in Figure 4.5. This new event is an intermediate event, which can be further developed using a template that we are going to discuss next.

4.2.2.3 Template for *Artifact o is Wrong When S is Posted*

Since the artifact o is already wrong when S is posted, the fault is not introduced by the step S. It must be propagated from another step. Depending on whether the step S is an exception handler or not, the control flows going to S are quite different. Therefore, two templates are introduced to develop this type of events.

Template 1

Requirement:

- The event e to be developed is *Artifact o is Wrong When S is Posted*, and

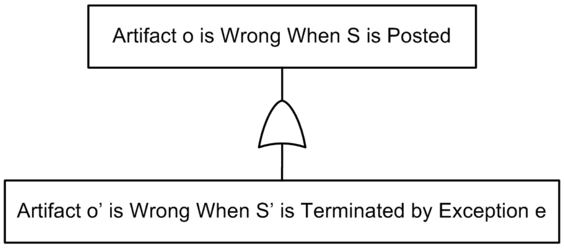

Artifact o is Wrong When S is Posted
Artifact o' is Wrong When S' is Terminated by Exception e

Figure 4.6. Partial Fault Tree for Template 1

- Step S is an exception handler step

Partial Fault Tree:

The partial fault tree for this template is shown in Figure 4.6. If S is an exception handler step, it can only be posted if the corresponding exception is thrown. Therefore, event "*Artifact o is Wrong When S is Posted*" can only be caused by event "*Artifact o' is Wrong When S' is Terminated by Exception e*", where

- e is the exception handled by S;
- S' may throw exception e;
- o' is an output parameter of S' that will be passed to o (via parameter bindings). If no such parameter exists, $o' = o$.

It is possible that the step S' does not have an output parameter o' that is passed to o. o' may be an output of another step S'' that is executed before S' and be passed to the exception handler. It is tempting to look several steps back to identify S'', create an event "*Artifact o' is Wrong When S" is Completed*", and connect this event to the `OR` gate. This, however, violates an important rule in the practice of fault tree derivation: *always look at the immediate causes each time* [131]. Violating this rule could result in overlooking some critical events between event "*Artifact o' is Wrong When S" is Completed*" and "*Artifact o is Wrong When S is Posted*". One might argue that since we derive the fault tree automatically instead of manually, we can avoid overlooking such critical events by considering all possible situations and incorporating them into the templates. The problem is that this will greatly increase the size of the templates and make the derivation algorithm much more complicated. Therefore, we choose to apply *Template 1* and create the temporary event "*Artifact o is Wrong When S' is Terminated by Exception e*" in this case. This event can be interpreted as: although the artifact o is not visible to the step S', it is already wrong and still alive at the point when S' is terminated by exception e. After the fault tree is derived, most of the temporary events introduced in this way can be removed by the fault tree optimization techniques that are discussed later in this chapter.

To apply this template, the step that throws the exception e handled by S needs to be identified. This can be achieved by looking up the successor nodes of the node `<S, Posted>` in the *Reversed Control Flow Graph.* For any successor node that has the form `<S', Terminated>`, we further check if the step S' throws the exception e. If so, S' is what we are looking for. We also need to find out whether there is an output parameter of S' that will be passed to o. Given an output parameter of S, whether this parameter may be passed to o can be decided by checking if there is a path from this parameter to o in the *Artifact Flow Graph.* If there is one such output parameter o', we create an event "*Artifact o' is Wrong When S' is Terminated by Exception e*" and connect it to the `OR` gate. Otherwise, the event "*Artifact o is Wrong When S' is Terminated by Exception e*" will be created instead.

Template 2

Requirement:

- The event e to be developed is *Artifact o is Wrong When S is Posted*
- S is not an exception handler step

Partial Fault Tree:

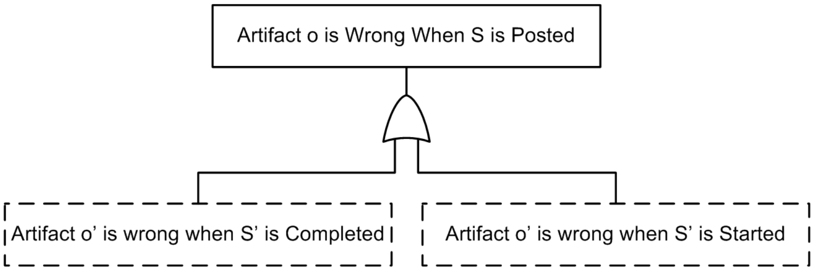

Artifact o is Wrong When S is Posted
Artifact o' is wrong when S' is Completed
Artifact o' is wrong when S' is Started

Figure 4.7. Partial Fault Tree for Template 2

The partial fault tree for this template is shown in Figure 4.7. If S is the root step, the incorrect artifact must be passed in as a parameter to the process. How the parameter went wrong is out of the scope of the process. Therefore, the event is treated as a primary event that does not need to be developed further.

If S is neither an exception handler nor the root step, it may be posted in two cases: it may be posted after its parent step is started or it may be posted after a sibling step is completed. The first case occurs if S is the first child step to be executed, e.g. S is a pre-requisite step, or S is the first sub-step of a sequential step that does not have a pre-requisite step. Suppose S' is the parent step of S, S will be posted as soon as S' is started. Since S' is a non-leaf step, it is not able to change the artifact o. The artifact o must already be wrong at the point when the S' is started. In other words, the event "*Artifact o is Wrong When S is Posted*" could be caused by the event "*Artifact o' is Wrong When S' is Started*", where:

- S' is the parent step of S, and S is the first child to be executed, and
- o' is a parameter of S' that will be passed to o. If no such parameter exists, $o' = o$.

The second case occurs when S is not the first child step to be executed. In this case, S can only be executed after one of its siblings is completed. For example, if S is the second sub-step of a sequential step, it will be posted as soon as the first sub-step is completed. If the first step may throw an exception and the corresponding exception handler has a *continue* badge, S can also be posted after the exception handler step is completed. Note that if this exception handler does have a handler step, S can actually be posted when the first step is terminated. To simplify the template, a dummy handler step is added to an exception handler if it is not associated with a hander step. For steps in the second case, event "*Artifact o is Wrong When S is posted*" may be caused by event "*Artifact o' is Wrong When S' is Completed*", where:

- S' a sibling of S and if S' is completed, S will be executed, and
- o' is an output parameter of S' that will be passed to o. If no such parameter exists, $o' = o$.

To apply this template, we first need to identify the S', which can be done by checking the conditions given above. This could become fairly complicated, especially for the second case. With the help of the *Reversed Control Flow Graph*, however, S' can be identified easily. We start with looking up the successor nodes of the node `<S, Posted>`. For any node that has the form `<S', Started>`, if S' is the parent step of S, S' satisfies the condition for the first case. For any node that has the form `<S', Completed>`, if S' has the same parent as S, S' satisfies the condition for the second case. We then can further check the parameters using the *Artifact Flow Graph* and create an event accordingly.

We now use this template to develop the event "*Blood Product is Wrong When Infuse Blood is Posted*" in the fault tree for the example process. In the *Reversed Control Flow Graph* shown in Figure 4.2, the node `<Infuse Blood, Posted>` only has one successor `<Pick up Blood from Blood Bank, Completed>`. Since "*Pick up Blood from Blood Bank*" and "*Perform Transfusion*" has the same parent step, "*Pick up Blood from Blood Bank*" is eligible for the second case. Next, we need to check if "*Pick up Blood from Blood Bank*" has an output parameter that can be passed to

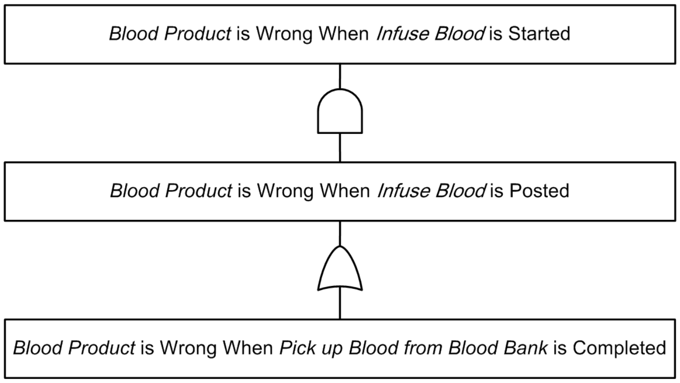

Blood Product is Wrong When *Infuse Blood* is Started
Blood Product is Wrong When *Infuse Blood* is Posted
Blood Product is Wrong When *Pick up Blood from Blood Bank* is Completed

Figure 4.8. Instantiated Partial Fault Tree for Template 2

the parameter *Blood Product* of "*Infuse Blood*". "*Pick up Blood from Blood Bank*" has one output parameter *Blood Product*. This parameter will be passed to *Blood Product* of "*Infuse Blood*" because there is a path from the node `<Blood Product @ Pick up Blood from Blood Bank>` to the node `<Blood Product @ Infuse Blood>` in the *Artifact Flow Graph* shown in Figure 4.3. Therefore, "*Blood Product is Wrong When Infuse Blood is Posted*" can only be caused by the event "*Blood Product is Wrong When Pick up Blood from Blood Bank is Completed*", as shown in Figure 4.8.

4.2.3 Derivation algorithm

The fault tree derivation algorithm for Little-JIL process definitions consists of three phases:

- **Perform Process Unrolling** The state space of a Little-JIL process definition might be infinite because of the recursion and unbound cardinality. The fault tree, on the other hand, is a finite representation. To be able to automatically derive a fault tree from a Little-JIL process definition, we first make the state space of the process finite using the same unrolling technique discussed in the previous chapter. The recursive steps and steps with unbound cardinality are unrolled to step instances up to bounds given by the analyst.

- **Construct RCFG and AFG** To construct the *Reversed Control Flow Graph*, we first build a regular control flow graph from the unrolled process, then reverse all its edges. The *Artifact Flow Graph* can be easily constructed based on the parameter bindings in the unrolled process. Both algorithms can be done in polynomial time.

- **Derive Fault Tree** With the unrolled process, the RCFG, and the AFG, the following work-list algorithm applies the templates to develop the fault tree for the given TOP event.

```
Input: Unrolled Little-JIL process definition, TOP event, RCFG, AFG
Output: Fault Tree
Initialize the visited event vs set to empty;
Initialize work-list wl set to empty;
Add TOP event into the work-list wl;
while wl is not empty do
    Remove an event e from wl;
    if vs contains e then
        continue;
    end
    Find the appropriate template t for e;
    Instantiate t using the RCFG and AFG;
    Replace e with the partial fault tree in the instantiated t;
    foreach leaf e' in the partial fault tree do
        if e' is not a primary event then
            Add e' to the work-list wl;
        end
    end
end
```

Algorithm 1: The Fault Tree Derivation Algorithm

This algorithm is a polynomial time algorithm. Suppose there are n step instances and m artifacts in the unrolled process. There are at most $m * n$ events that will be created. This algorithm ensures that each event will be developed at most once. Therefore it can be finished in $O(m * n)$.

Given the TOP event “*Blood Product is Wrong When Infuse Blood is Started*”, the fault tree derived by this algorithm from the simplified blood transfusion process is shown in Figure 4.9. One might immediately notice that it is not exactly a tree. The event “*Patient ID is Wrong When Contact Lab for Patient’s Blood Type is Started*” serves as an input to two gates. We can certainly expand the graph into a tree by creating two sub-trees and connect them to those two gates respectively. For fault trees derived from real-world processes, however, fully expanded versions are usually much larger than the compact graph representations. While much smaller and easier to understand, the compact graph representation produces the same MCSs as its fully expanded counterpart. Therefore, we choose to use this graph representation. Without causing confusion, we still call these graphs fault trees.

4.3 Evaluating Fault Trees

Once a fault tree is derived, all faults that might cause the hazard to happen can be obtained from the primary events in the fault tree. The process developer can then change the process to control or eliminate occurrence of those faults. For instance, a failure-resistant agent could be assigned to steps where faults could occur. Additionally, consistency check steps could be added to well-chosen places in the process to stop the propagation of faults. In practice, due to resource limitations or other constraints, the process developer usually is only able to apply the fault prevention mechanisms to a few of the most important faults.

Priorities of the faults can be determined by evaluating the derived fault tree [131]. The most important fault tree evaluation technique is the *Minimal Cut Set Analysis*.

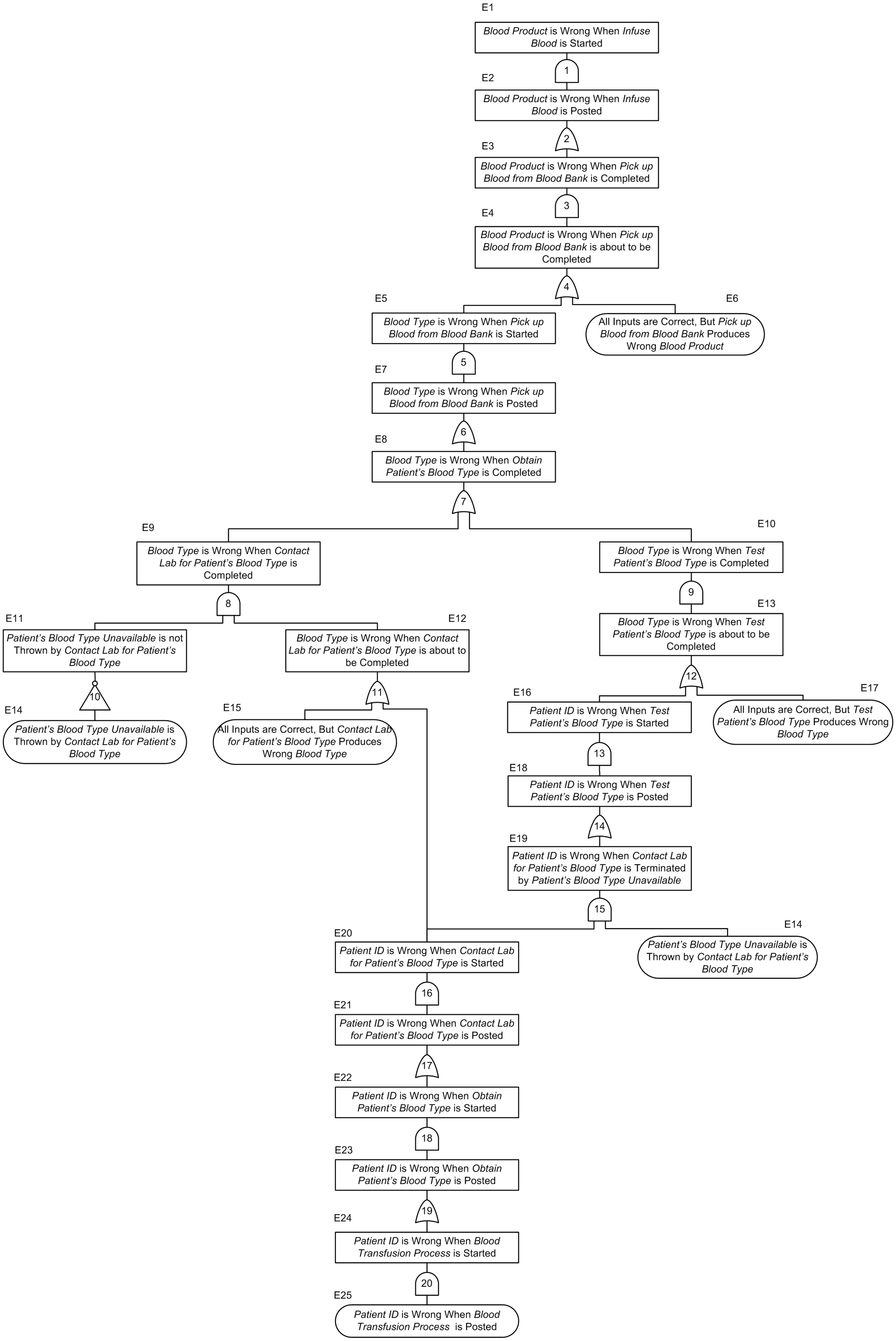

E1
Blood Product is Wrong When Infuse Blood is Started
1
E2
Blood Product is Wrong When Infuse Blood is Posted
2
E3
Blood Product is Wrong When Pick up Blood from Blood Bank is Completed
3
E4
Blood Product is Wrong When Pick up Blood from Blood Bank is about to be Completed
4
E5
Blood Type is Wrong When Pick up Blood from Blood Bank is Started
E6
All Inputs are Correct, But Pick up Blood from Blood Bank Produces Wrong Blood Product
5
E7
Blood Type is Wrong When Pick up Blood from Blood Bank is Posted
6
E8
Blood Type is Wrong When Obtain Patient's Blood Type is Completed
7
E9
Blood Type is Wrong When Contact Lab for Patient's Blood Type is Completed
E10
Blood Type is Wrong When Test Patient's Blood Type is Completed
8
9
E11
Patient's Blood Type Unavailable is not Thrown by Contact Lab for Patient's Blood Type
E12
Blood Type is Wrong When Contact Lab for Patient's Blood Type is about to be Completed
E13
Blood Type is Wrong When Test Patient's Blood Type is about to be Completed
10
11
12
E14
Patient's Blood Type Unavailable is Thrown by Contact Lab for Patient's Blood Type
E15
All Inputs are Correct, But Contact Lab for Patient's Blood Type Produces Wrong Blood Type
E16
Patient ID is Wrong When Test Patient's Blood Type is Started
E17
All Inputs are Correct, But Test Patient's Blood Type Produces Wrong Blood Type
13
E18
Patient ID is Wrong When Test Patient's Blood Type is Posted
14
E19
Patient ID is Wrong When Contact Lab for Patient's Blood Type is Terminated by Patient's Blood Type Unavailable
15
E20
Patient ID is Wrong When Contact Lab for Patient's Blood Type is Started
E14
Patient's Blood Type Unavailable is Thrown by Contact Lab for Patient's Blood Type
16
E21
Patient ID is Wrong When Contact Lab for Patient's Blood Type is Posted
17
E22
Patient ID is Wrong When Obtain Patient's Blood Type is Started
18
E23
Patient ID is Wrong When Obtain Patient's Blood Type is Posted
19
E24
Patient ID is Wrong When Blood Transfusion Process is Started
20
E25
Patient ID is Wrong When Blood Transfusion Process is Posted

Figure 4.9. Fault Tree for Simplified Blood Transfusion Process

It computes a set of *Minimal Cut Sets(MCSs)* from the fault tree. A *cut set* is a set of primary events whose occurrence ensures that the TOP event occurs. An *MCS* is a cut set that cannot be further reduced. In other words, an MCS will not be able to cause the hazard to occur if any event in it is missing. The importance of a MCS can be grossly estimated by its size. Smaller MCSs are considered to be more important than larger ones since fewer faults need to arise to cause the top event to occur. The process developer, therefore, can start with the faults in MCSs with smallest size. If probabilities of all the primary events in the fault tree are provided, statistic analysis can be applied to compute probabilities of MCSs. As the indicator, the probability is more accurate than the size. The quantitative evaluation can even obtain the numeric evaluation of overall system vulnerability to a hazard caused by the occurrence of a particular fault. In this section, we focus on the computation of MCSs.

To compute MCSs from a fault tree, each gate in the fault tree is first translated into a Boolean equation. Consider the fault tree derived from the simplified blood transfusion process (Figure 4.9) as an example. The Boolean equations corresponding to several gates are shown as follows.

Gate 1: $E_1 = E_2$

Gate 2: $E_2 = E_3$

Gate 3: $E_3 = E_4$

Gate 4: $E_4 = E_5 + E_6$

...

Gate 8: $E_9 = E_{11}E_{12}$

...

Given these equations, existing *Boolean Minimization* tools can be applied to get rid of all the intermediate events, except the TOP event, and obtain an equation that only contains the TOP event and primary events. In this equation, the left side is the TOP event and the right side is a formula of primary events in *Conjunctive Normal*

Form. Each clause in this formula corresponds to a MCS, which contains all events in that clause. The final equation computed for the example fault tree is:

$E_1 = E_6 + E_{17} + E_{25} + E_{15} \neg E_{14}$

From this equation, we get four MCSs:

1) $\{E_6\}$

2) $\{E_{17}\}$

3) $\{E_{25}\}$

4) $\{E_{15}, \neg E_{14}\}$

MCS 1, 2, and 3 only contain one event, indicating that the process is exposed to a *single point of failure.* The hazard will definitely occur if either wrong blood product is picked up at the blood bank(E_6), wrong blood type is produced by the blood testing(E_{17}) or the patient ID passed to the process is wrong (E_{25}). Therefore subsequent changes need to be made to remove these single points of failure.

MCS 4 does not indicate a single point of failure because it contains two events E_{15} and $\neg E_{14}$. If we look carefully at these events, however, we can see the problem with this MCS. E_{15} says "*All Inputs are Correct, But Contact Lab for Patient's Blood Type Produces Wrong Blood Type*". Since the agent for step "*Contact Lab for Patient's Blood Type*" already produces the blood type, although it is wrong, the agent will not throw the exception *Patient's Blood Type Unavailable.* Therefore, E_{15} implies $\neg E_{14}$, which represents that the exception *Patient's Blood Type Unavailable* is not thrown by "*Contact Lab for Patient's Blood Type*". We can see that E_{15} is a indeed single point of failure even though the MCS says that it is not. Since our analysis treats leaf steps as black boxes, it is not able to deduce relationships between primary events that occur at leaf steps. Therefore it is up to analysts to identify such relationships and generate more accurate results. Although MCSs computed by our approach may not be minimal, there is one thing that is guaranteed – if a MCS contains only one event, the event must be a single point of failure.

4.4 Process FTA Issues

We encountered four major issues during our study of process fault tree analysis: *scalability*, *looping contructs*, *NOT gates*, and it Useability. In this section, we discuss these issues as well as our approaches to address these issues.

4.4.1 Scalability

A real-world human-intensive process usually involves complex control and artifact flows. The fault tree derived from it can easily become too large either to be easily understood by humans or be analyzed by fault tree evaluation tools. To address this issue, two optimizations, *Step Abstraction* and *Step Removal*, are applied to the unrolled process before the fault tree is derived. Similar to step abstraction and step removal in the process verification framework, *Step Abstraction* abstracts sub-processes into leaf steps and *Step Removal* removes certain steps that do not have impact on the propagation of faults. In addition to these two optimizations, the optimization *Equivalent Event Removal* can be applied to remove certain intermediate events from the fault tree after it is derived. All three optimizations guarantee that the optimized fault tree is equivalent to the original one in the sense that both fault trees generate the same minimal cut sets.

4.4.1.1 Step Abstraction

In the process verification framework, the *Step Abstraction* first identifies a set of relevant steps where property events could occur, and then abstracts sub-processes that do not contains relevant steps into leaf steps. The *Step Abstraction* for fault tree derivation is almost the same as the one for verification, except for the way in which relevant steps are identified. In our approach, the hazard is represented as an incorrect artifact at a specific point in the process. The faults that could cause the hazard to occur can only originate from two sources: a leaf step that produces a wrong output or a wrong input to the root step. The other steps can only change the

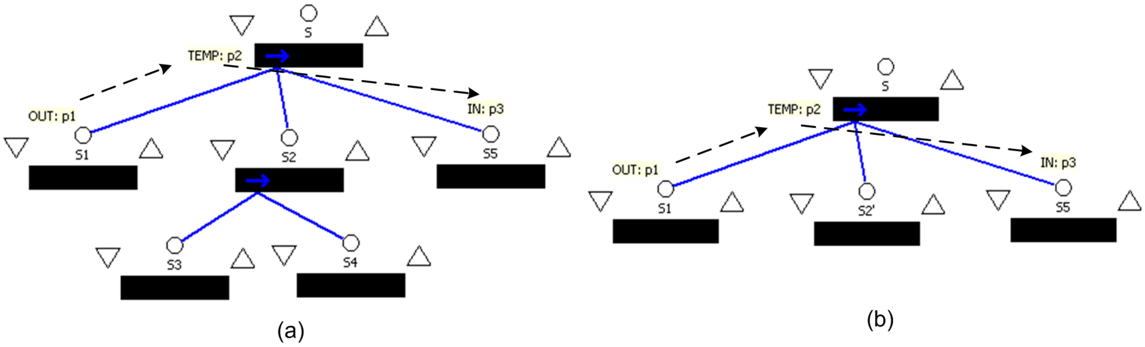


Figure 4.10. Step Abstraction for Fault Tree Derivation

propagation of the faults by throwing exceptions. Therefore, *relevant steps* for fault tree derivation are the root step and the steps that have an `OUT` parameter that may flow to the artifact in the TOP event. As noted earlier, the TOP event is required to be defined as an event in Category 2, which always has an artifact. These relevant steps can be easily identified with the help of the *Artifact Flow Graph*. Once the relevant step set is obtained, any sub-process that does not contain relevant steps can be replaced with a leaf step that throws the same exceptions as the root step of the sub-process. In this way, intermediate events associated with the steps in the sub-process will not be created.

Figure 4.10(a) shows an example. The step S_1 creates the parameter p_1, which is passed to the parameter p_3 of the step S_5 via the temporary parameter p_2. Suppose the hazard is "*Artifact p_3 is Wrong When S_5 is Started*". The only step that has an output parameter that may flow to p_3 is step S_1. Therefore, the hazard can only be caused by the fault "*All Inputs are Correct, But S Produces Wrong p_1*". The sub-process with the root S_2 can never create faults that lead to the hazard. We can safely replace the sub-process with a leaf step S_2', as shown in Figure 4.10(b). This results in a fault tree without intermediate events containing states of step S_3 and S_4. The fault tree derived after the step abstraction is equivalent to the fault tree

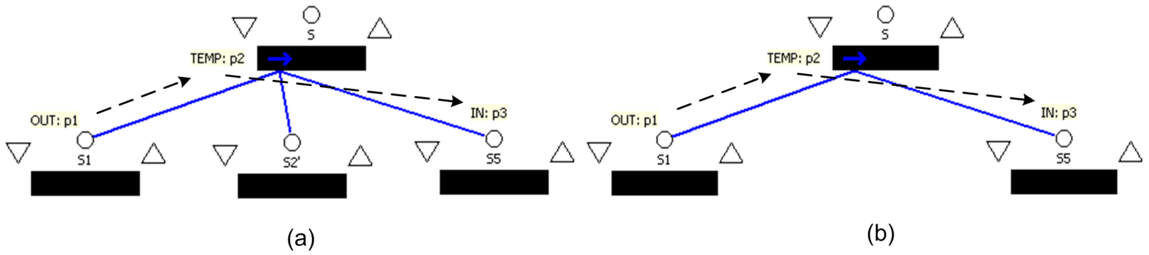


Figure 4.11. Step Removal for Fault Tree Derivation

derived without step abstraction. Both have one MCS: {"*All Inputs are Correct, But S Produces Wrong* p_1"}.

4.4.1.2 Step Removal

Different from the one for process verification, the *Step Removal* for fault tree derivation removes a step only if it satisfies the following conditions:

1. It is not a relevant step, and

2. It does not throw any exceptions

The first condition guarantees that the step will never produce a fault that could lead to the hazard. The second condition ensures that the step cannot change the propagation of faults that may cause the hazard to occur. For the process shown in Figure 4.11(a), we can remove S_2' because it satisfies both conditions. Figure 4.11(b) is the process after removing S_2. It is easy to see that the fault tree derived from this process is equivalent to the fault tree derived from the original process.

4.4.1.3 Equivalent Event Removal

While the previous two optimizations are performed on the process before the fault tree is derived, the *Equivalent Event Removal* optimization is applied to remove certain intermediate events from the fault tree after it is derived. This optimization first partitions the events in the fault tree into different *equivalent groups*. Each

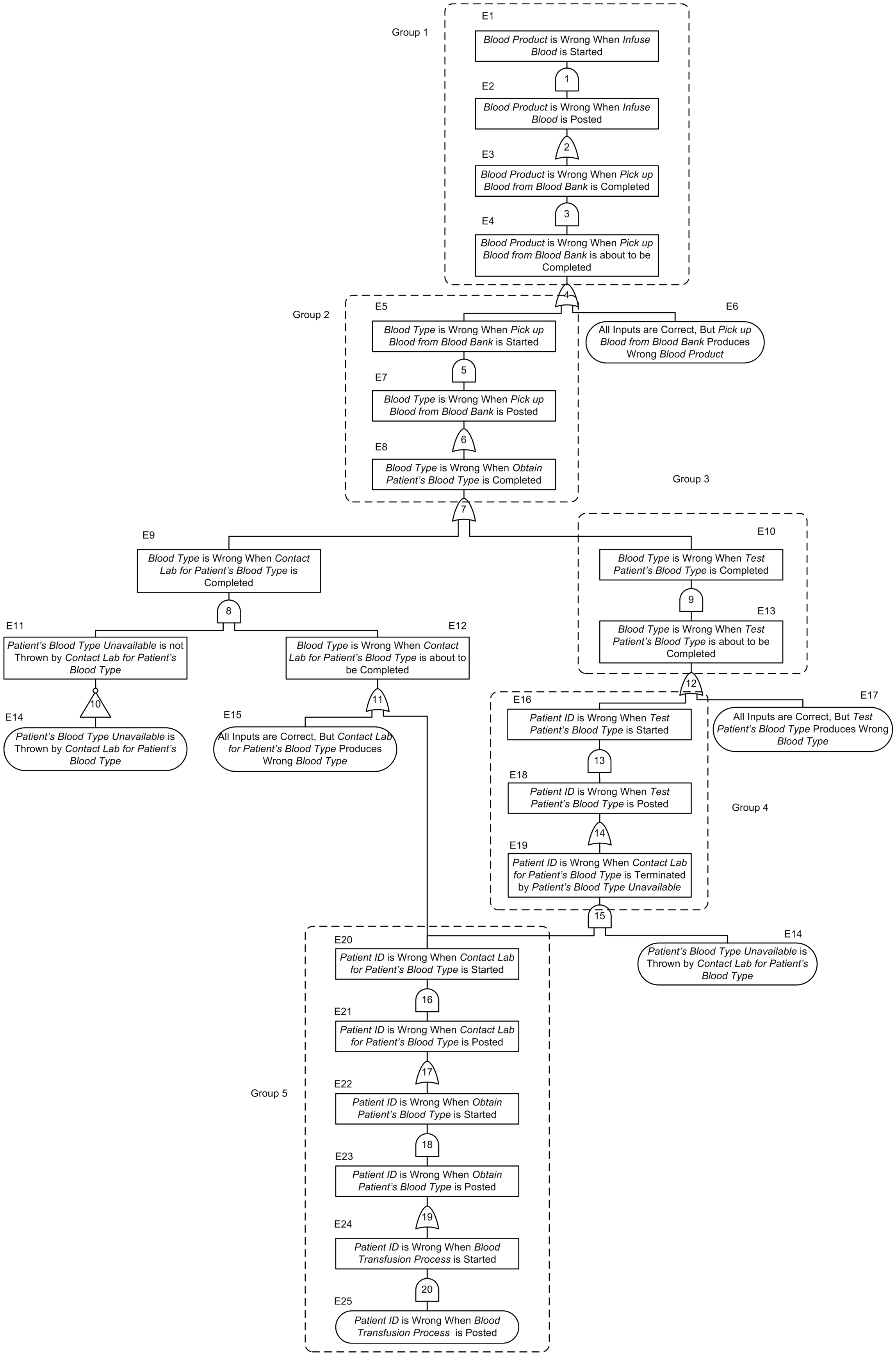

Group 1
E1
Blood Product is Wrong When Infuse Blood is Started
1
E2
Blood Product is Wrong When Infuse Blood is Posted
2
E3
Blood Product is Wrong When Pick up Blood from Blood Bank is Completed
3
E4
Blood Product is Wrong When Pick up Blood from Blood Bank is about to be Completed
4
Group 2
E5
Blood Type is Wrong When Pick up Blood from Blood Bank is Started
E6
All Inputs are Correct, But Pick up Blood from Blood Bank Produces Wrong Blood Product
5
E7
Blood Type is Wrong When Pick up Blood from Blood Bank is Posted
6
E8
Blood Type is Wrong When Obtain Patient's Blood Type is Completed
Group 3
7
E9
Blood Type is Wrong When Contact Lab for Patient's Blood Type is Completed
E10
Blood Type is Wrong When Test Patient's Blood Type is Completed
8
9
E11
Patient's Blood Type Unavailable is not Thrown by Contact Lab for Patient's Blood Type
E12
Blood Type is Wrong When Contact Lab for Patient's Blood Type is about to be Completed
E13
Blood Type is Wrong When Test Patient's Blood Type is about to be Completed
10
11
12
E14
Patient's Blood Type Unavailable is Thrown by Contact Lab for Patient's Blood Type
E15
All Inputs are Correct, But Contact Lab for Patient's Blood Type Produces Wrong Blood Type
E16
Patient ID is Wrong When Test Patient's Blood Type is Started
E17
All Inputs are Correct, But Test Patient's Blood Type Produces Wrong Blood Type
13
E18
Patient ID is Wrong When Test Patient's Blood Type is Posted
Group 4
14
E19
Patient ID is Wrong When Contact Lab for Patient's Blood Type is Terminated by Patient's Blood Type Unavailable
15
E20
Patient ID is Wrong When Contact Lab for Patient's Blood Type is Started
E14
Patient's Blood Type Unavailable is Thrown by Contact Lab for Patient's Blood Type
16
E21
Patient ID is Wrong When Contact Lab for Patient's Blood Type is Posted
17
Group 5
E22
Patient ID is Wrong When Obtain Patient's Blood Type is Started
18
E23
Patient ID is Wrong When Obtain Patient's Blood Type is Posted
19
E24
Patient ID is Wrong When Blood Transfusion Process is Started
20
E25
Patient ID is Wrong When Blood Transfusion Process is Posted

Figure 4.12. Groups in Fault Tree for Simplified Blood Transfusion Process

equivalent group has a *source event* that is a primary event or an event whose input gate connects to event(s) in another group. Except for the source event, each event in an equivalent group occurs if and only if the source event occurs. Note that the idea of *equivalent groups* is similar to *basic blocks* of control flow graphs in compiler optimizations [9]. Figure 4.12 shows the groups identified in the fault tree derived from the simplified blood transfusion process. Each equivalent group is enclosed in a box with the dashed line. To make the figure less crowded, the boxes for groups with a single event are not drawn. Take group 1 as an example. *E4* is the source event. The other three events are included in this group because they may occur if and only if *E4* occurs.

Here, we give an informal description of the algorithm that we use to identify equivalent groups. The algorithm performs a post-order traverse over the given fault tree. Upon visiting an event e, a new group is created to include e if e is a primary event or input gate of e is a NOT gate. Otherwise, e must be an intermediate event whose input gate g is either an AND gate or an OR gate. In this case, the algorithm proceeds to check the input events of g. If all these input events belong to the same group, e is added into this group. Otherwise, a new group is created to include e.

For each equivalent group, the *Equivalent Event Removal* optimization selects certain event(s) and removes the other events in the group. Specifically, the TOP event and primary events are always selected to be kept. If a group does not contain the TOP event or a primary event, any event in the group can be selected. Based on the definition of equivalent groups, it is easy to prove that this optimization will result in a fault tree that is equivalent to the original one. This optimization is able to greatly reduce the size of the fault tree. The problem, however, is that removing intermediate events creates "gaps" between events that may cause the fault tree to be difficult to understand. To solve this problem, events are only temporarily removed when the MCSs are calculated. When the fault tree is displayed to the user, an

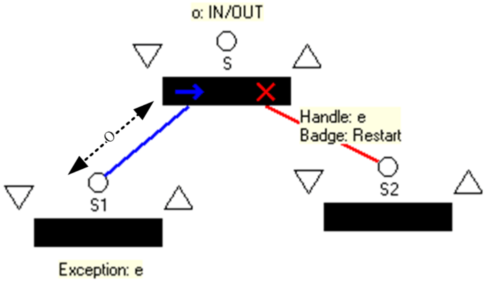


Figure 4.13. Example of Process Containing Loop

equivalent group with more than one event is collapsed into a node. If necessary, the user can expand the node to see the events in the group.

4.4.2 Looping Constructs

Processes often contain steps that are executed repeatedly. Fault trees derived from these processes, therefore, may contain loops (i.e. an input and the output of an event connect to the same event). Figure 4.13 shows a process example. In this process, the sub-step S_1 may throw exception e. The exception is handled by a `restart` handler. Step S has an `INOUT` parameter o that will be passed and changed by S_1. Exception handler step S_2 cannot change o. Since the exception handler is a `restart` handler, S_1 and S_2 might be executed repeatedly. Given the hazard "*Artifact o is Wrong When S is completed*", the fault tree derived by our algorithm is shown in Figure 4.14. For each intermediate event, the template used to develop this event is shown next to the event. Note that there is an edge from the input gate of event e_{14} to event e_7. This edge introduces a loop into the fault tree.

Since a fault tree with loops is no longer a tree, existing techniques for analyzing fault trees cannot be applied to it. To address this issue, we simply remove loops from the derived fault tree f in the following way:

- *Detecting a loop* A depth search is performed on the fault tree f, starting from the hazard event. If an input event of a gate points to an event in the stack,

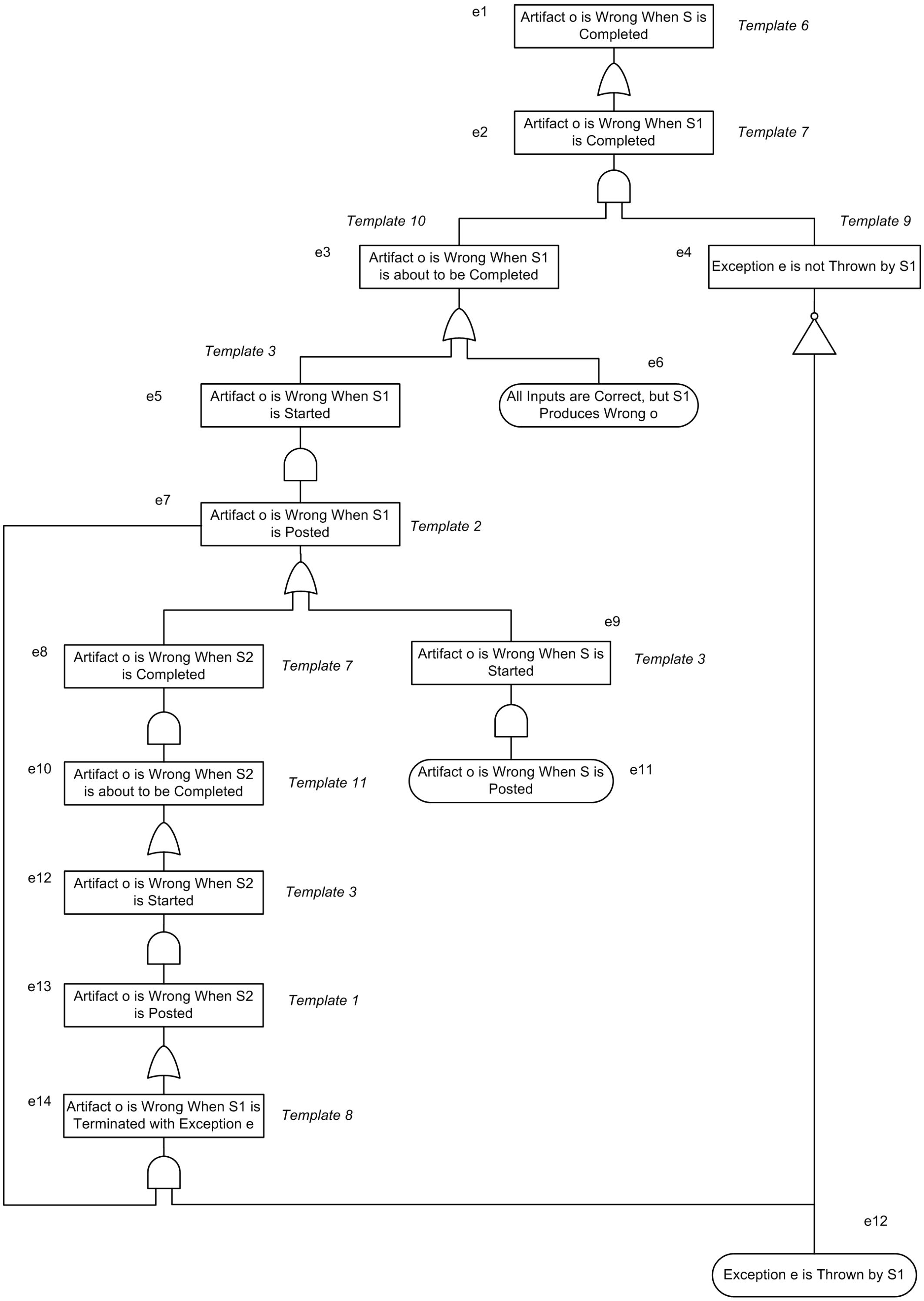

e1
Artifact o is Wrong When S is Completed
Template 6
e2
Artifact o is Wrong When S1 is Completed
Template 7
Template 10
Template 9
e3
Artifact o is Wrong When S1 is about to be Completed
e4
Exception e is not Thrown by S1
Template 3
e6
e5
Artifact o is Wrong When S1 is Started
All Inputs are Correct, but S1 Produces Wrong o
e7
Artifact o is Wrong When S1 is Posted
Template 2
e9
e8
Artifact o is Wrong When S2 is Completed
Template 7
Artifact o is Wrong When S is Started
Template 3
e10
Artifact o is Wrong When S2 is about to be Completed
Template 11
Artifact o is Wrong When S is Posted
e11
e12
Artifact o is Wrong When S2 is Started
Template 3
e13
Artifact o is Wrong When S2 is Posted
Template 1
e14
Artifact o is Wrong When S1 is Terminated with Exception e
Template 8
e12
Exception e is Thrown by S1

Figure 4.14. Fault Tree Containing Loop

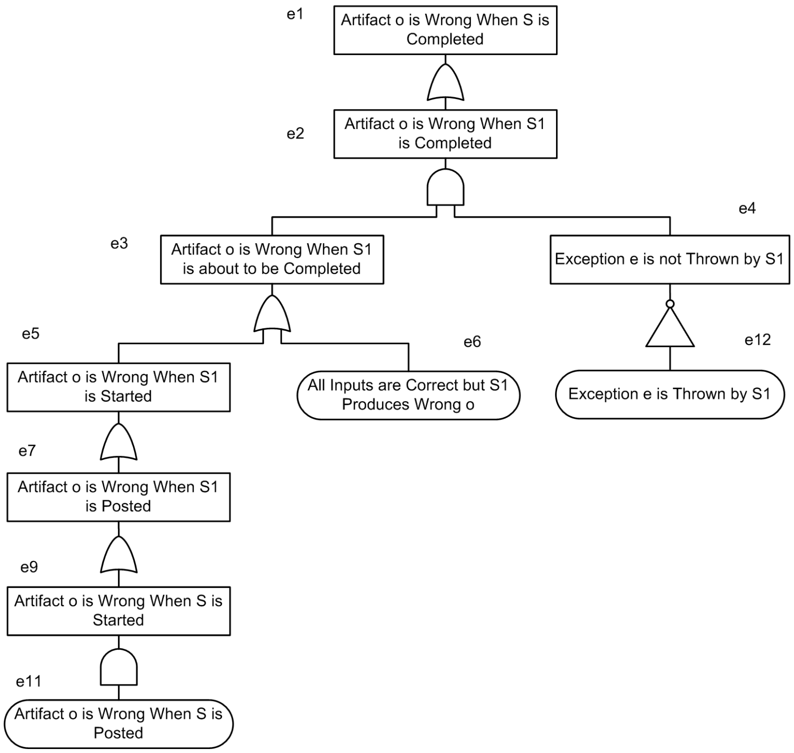

e1
Artifact o is Wrong When S is Completed
e2
Artifact o is Wrong When S1 is Completed
e3
Artifact o is Wrong When S1 is about to be Completed
e4
Exception e is not Thrown by S1
e5
Artifact o is Wrong When S1 is Started
e6
All Inputs are Correct but S1 Produces Wrong o
e12
Exception e is Thrown by S1
e7
Artifact o is Wrong When S1 is Posted
e9
Artifact o is Wrong When S is Started
e11
Artifact o is Wrong When S is Posted

Figure 4.15. Fault Tree without Loop

a loop is detected. In the fault tree shown in Figure 4.14, the loop is detected when the edge from the input gate of event e_{14} to event e_7 is visited.

- *Removing the loop* Before discussing how to remove a loop, we first define an elimination operation $f \backslash e$, where f is a fault tree and e is an event in f. This operation produces another fault tree f' as follows. As discussed in Section 4.3, events in a fault tree can be considered as Boolean variables and gates as Boolean equations. The event e in the operation is first assigned it to `false`. This may cause some other events to be evaluated to `false`. For example, suppose there is an AND gate that has two input events, e and e', and an output event e''. The equation for this gate is $e'' = ee'$. Since e is assigned to `false`, e'' will be evaluated to `false` too according to this equation. This may cause more events to be evaluated to `false`. Removing all events evaluated to `false` from the fault tree f results in f'.

To remove a loop, a new event e is created and the back edge in the loop is connected to this new event. Then the operation $f\backslash e$ is applied to obtain fault tree f'. Removing the loop in the fault tree shown in Figure 4.14 results in the fault tree shown in Figure 4.15.

A loop in a fault tree is caused by certain steps that are executed repeatedly. Removing a loop in the above way basically results in a fault tree corresponding to the case where those steps executed once. Compared to this fault tree, a fault tree corresponding to the case where those steps executed more than once usually produces larger MCSs. Since process developers usually focus on small MCSs, we believe that it should be enough to only consider the fault tree corresponding to the case where those steps executed once. If necessary, however, a fault tree corresponding to the case where those steps executed more than once can be easily created from the fault tree with the loop.

4.4.3 NOT Gate

Traditionally, NOT gates are not used in fault trees because they may introduce difficulties to the evaluation of fault trees. The use of NOT gates will result in a non-coherent fault tree, meaning that the fault tree is no longer a monotone Boolean function [10]. For non-coherent fault trees, the original concept of minimal cut sets does not apply and should be replaced by prime implicants. Usually, computing prime implicants for non-coherent fault trees is much more difficult than computing minimal cut sets for coherent fault trees. Without NOT gates, however, the fault-tree analysis may produce inaccurate results because relationships between many events cannot be captured in the fault tree. Therefore whether to use NOT gates is a trade off between efficiency and accuracy. In the early stage of fault-tree analysis (1960s–1980s), due to

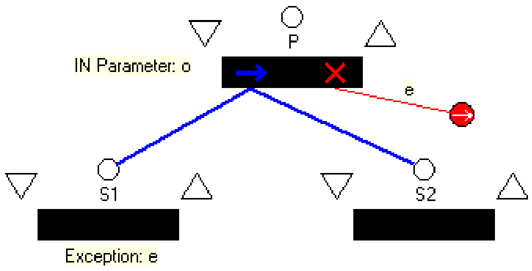


Figure 4.16. Simple Process for Discussing NOT gates

the limitation of computing power and algorithms, fault trees were derived without using NOT gates so that they could be evaluated. Since then, the computing power has been increased significantly. More importantly, recent research has showed that prime implicants for large non-coherent fault trees can be computed using symbolic or heuristic approaches [112] [39] [3]. Because of this progress, many FTA approaches now choose to use NOT gates in order to achieve more accurate results.

In our research, we decided to use NOT gates to capture relationships between certain conditional events. For example, in Template 12, event "*Exception e is not Thrown by S*" is connected to event "*Exception e is Thrown by S*" via a NOT gate. Here, we use a simple example to show why FTA of processes may produce inaccurate results without NOT gates. In the process shown in Figure 4.16, the root step "*P*" is a sequential step with two sub-steps. The first sub-step "*S1*" may throw exception *e*, which will be handle by a *continue* handler. The root step has an input parameter *o* that cannot be changed by its sub-steps. For the hazard "*o is Wrong When S2 is Started*", the fault tree derived without NOT gates is shown in Figure 4.17. The MCSs computed from this fault tree are {*o is Wrong When P is Posted* (E10), *e is Thrown by S1* (E5)} and {*o is Wrong When P is Posted* (E10), *e is not Thrown by S1* (E7)}. These two MCSs fail to indicate the single point of failure – event "*o is Wrong When P is Posted*", which always causes the hazard to occur no matter whether *S1* throws exception *e* or not. The reason is that the relationship between

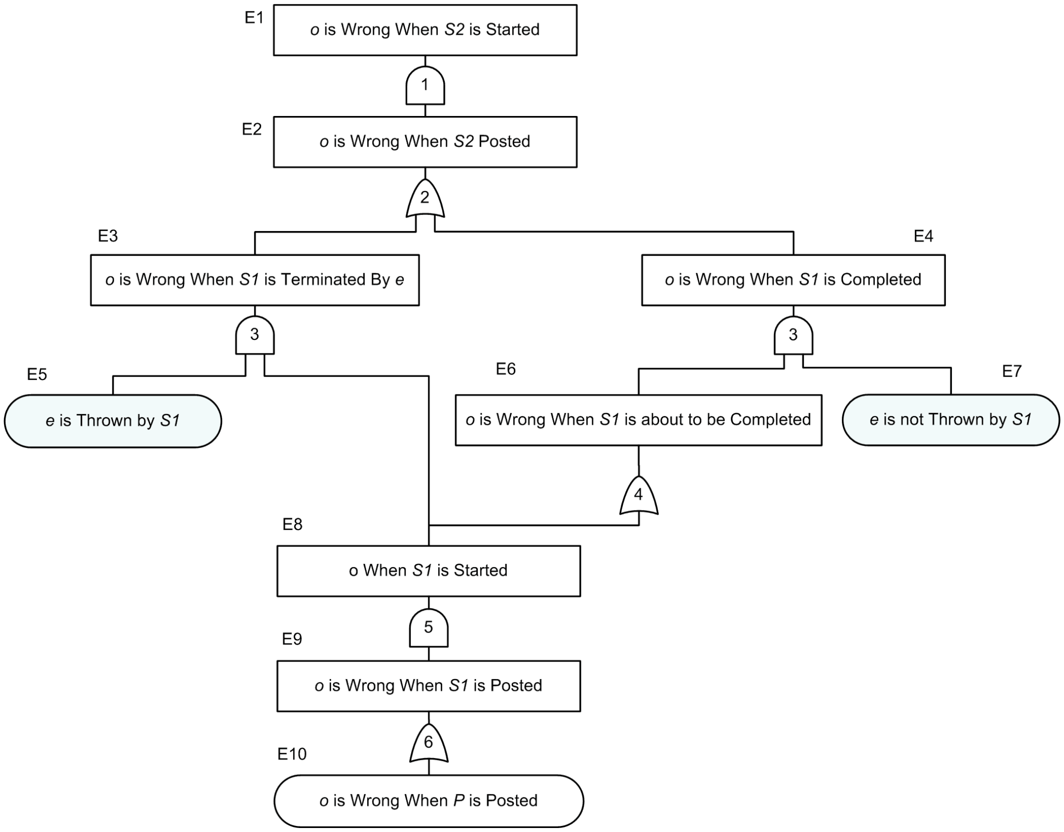

E1
o is Wrong When S2 is Started
1
E2
o is Wrong When S2 Posted
2
E3
o is Wrong When S1 is Terminated By e
E4
o is Wrong When S1 is Completed
3
3
E5
e is Thrown by S1
E6
o is Wrong When S1 is about to be Completed
E7
e is not Thrown by S1
4
E8
o When S1 is Started
5
E9
o is Wrong When S1 is Posted
6
E10
o is Wrong When P is Posted

Figure 4.17. Fault Tree without NOT Gates

event "*e is not Thrown by S1*" and "*e is Thrown by S1*" is not captured in the fault tree. To get the expected MCS {"*o is Wrong When P is Posted*"}, a NOT gate needs to be introduced to connect these two events.

4.4.4 Usability

To control or eliminate a fault in a MCS, the process developer needs to understand how this fault together with the other events in the MCS could cause to hazard to occur. Usually, this can be done by first identifying events from the MCS in the fault tree and then following the fault tree up to the TOP event. The problem is that fault trees derived from real-world processes could be very large. To solve this problem, we generate two representations, *Partial Fault Trees* and a *Process Trace*, which are designed to help process developers to better understand MCSs.

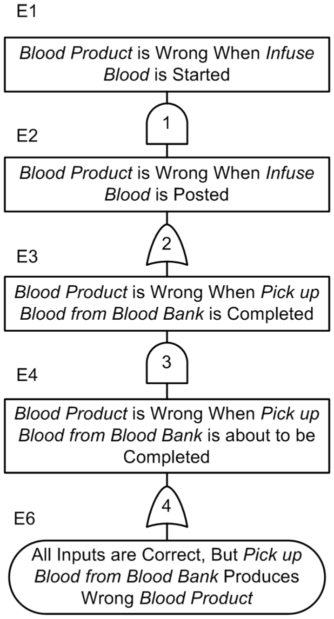

E1
Blood Product is Wrong When Infuse Blood is Started
1
E2
Blood Product is Wrong When Infuse Blood is Posted
2
E3
Blood Product is Wrong When Pick up Blood from Blood Bank is Completed
3
E4
Blood Product is Wrong When Pick up Blood from Blood Bank is about to be Completed
4
E6
All Inputs are Correct, But Pick up Blood from Blood Bank Produces Wrong Blood Product

Figure 4.18. Partial Fault Tree for MCS {E6}

4.4.4.1 Partial Fault Tree

The fault tree derived from a process shows combinations of all events that could lead to the hazard. Large parts of the fault tree actually have nothing to do with the events in a MCS. Therefore, we generate a *partial fault tree* for the MCS by removing events irrelevant to the MCS from the fault tree. Given a MCS m and the fault tree f from which m is computed, an event e is an *irrelevant event* if $f \backslash e$ is still able to produce MCS m. $f \backslash e$ is the elimination operation discussed in Section 4.4.2. Since the partial fault tree does not contain irrelevant events, it is much smaller and is easier to follow. Figure 4.18 shows the partial fault tree for the minimal cut set $\{E6\}$ generated from the fault tree for the simplified blood transfusion process.

4.4.4.2 Process Trace

In many cases, events in a MCS could lead to the hazard along different executions of the process. In a large partial fault tree, it is often difficult to distinguish events occur along one execution from events along another execution. To address this issue, we generate process traces that show how events in a MCS cause the hazard to occur. Details about this trace generation algorithm are given in Chapter *Trace Generation and Visualization.*

4.5 Limitations of Our Approach

To use our process FTA approach, process developers need to be aware of several limitations that may cause the analysis results to be imprecise or even incorrect.

- The completeness a fault tree derived from a Little-JIL process definition by our automatic algorithm depends on the completeness of the process. Thus, in cases where the Little-JIL process definition fails to completely represent steps in a real-world process that have an effect upon critical artifact flows, our algorithm will, accordingly, produce an incomplete fault tree.

- Since the fault tree derivation algorithm only looks at step interfaces, which do not specify how a leaf step produces its outputs from its inputs, it has to assume that any output of a leaf step depends on all its inputs. Thus, leaf steps that do not satisfy this assumption may cause the derived fault tree to contain superfluous sub-trees.

- As shown in the example discussed in Section 4.3, computed MCSs may not be minimal because our analysis is not able to identify the relationships between different primary events.

4.6 Summary

In this chapter, we present our template-based automatic fault tree derivation algorithm for the Little-JIL process definition language. Although the events and templates are specific to the Little-JIL language, the basic idea of this template-based fault tree derivation approach may be applied to automatically construct fault trees for other process definition languages. When applying FTA to real-world, human-intensive processes, we encountered three issues. To address the scaling issue, we proposed three optimizations that are able to significantly reduce the size of fault trees so that they can be understood and evaluated more easily. To address the loop issue, we applied a simple approach to remove loops in fault trees so that they can be analyzed using existing fault tree evaluation techniques. To obtain more accurate results, we chose to use NOT gate to capture relationships between certain conditional events. We believe that these issues may also be found when applying FTA to processes defined in other languages and believe our approaches provide suggestions for addressing such issues. In addition to support for automatic fault tree derivation, we proposed two representations that make MCSs more easily understood, no matter how the fault trees were derived.

CHAPTER 5

TRACE GENERATION AND VISUALIZATION

In software engineering, an *execution* of a software system is a sequence of system states from initialization to termination. A *state* is a configuration of the system where every variable in the system is assigned to a specific value. Here we define a *trace* to be a partial view of an execution. Specifically, a trace is a sequence of states, where the set of states in the trace is a sub-set of the states in the corresponding execution and the order of states in the trace is the same as the order in which those states appear in the execution. Traces have been widely used in many software engineering techniques. For instance, to debug a software system, the developer may follow a particular trace generated by the debugger to diagnose the bugs in the system. System logs, which can be considered as a particular kind of traces, are used to recorded information about certain executions of the system. The system logs help the system administrator to detect abnormal behavior of the system. In system simulation, the analyst needs to study the traces of the simulator to make sure that the simulator correctly captures the behaviors of the real system. The trace also plays an important role in two analysis techniques that we study in this thesis. As mentioned earlier, during process verification, if the property is violated by the process, a counter-example trace will be generated by the verifier to demonstrate how a property is violated. The counter example trace represents one possible execution of the translated model along which the property is violated. For the Fault-Tree Analysis, we generate traces for a MCS to aid the analyst in understanding how events in a MCS may eventually cause the hazard to occur.

In this section, we focus on an important yet often neglected issue: how to make traces more accessible to the analyst. We proposed an approach to generate a high-level Little-JIL trace from a low-level violation trace of the underlying verification tools used by our verification framework. This Little-JIL trace generation approach is designed in a generic way so that it is also able to produce a Little-JIL trace based on a Minimal Cut Set (MCS) computed by the fault tree analysis. In addition, we proposed two visual representations that allow the user to navigate and understand the Little-JIL trace more easily. We believe these approaches are able to greatly enhance the user-friendliness of both process verification and process fault tree analysis. Although the discussion in the remainder of this chapter is based on the Little-JIL process definition language, the underlying ideas of our approaches can be adopted to generate and represent traces in other languages.

5.1 High-Level Trace Generation

In our process verification framework, we translate the Little-JIL process definition into the model used by the underlying verification tool. The counter-example trace produced by the verification tool, therefore, is a trace through this model instead of through the Little-JIL process definition. During the translation, high-level constructs, such as exceptions and channels, are mapped to low-level representations in the model. In addition, steps or artifacts might be removed or abstracted by the optimizations. Therefore, the model often becomes considerably different from the original Little-JIL process definition. As a consequence, the counter-example traces become extremely difficult to understand, even for an expert with deep understanding of the translation and optimization techniques.

To solve this problem, we implemented a tool that is able to generate a corresponding trace through the Little-JIL process definition based on the counter-example trace through the model used by the verification tool. Since the Little-JIL process definition is mapped into the model via a sequence of transformations including translation and various optimizations, the most straight-forward approach is to map the counter-example trace backward, phase by phase, along those transformations. The problem with this approach is that it requires the mapping information for each transformation to be stored. Furthermore, any change in the optimizations or the translation might lead to change in the implementation of the tool. Instead, we chose to perform an A* search [114] on the Little-JIL process definition, where the search goal is the sequence of property events as they appear in the counter-example. Since the Little-JIL trace generated by the A* search contains the same sequence of property events, it will drive the property into the violation state in the same way as the low-level counter-example trace. For this A* search approach, no extra transformation information needs to be stored, and changes in transformations usually will not affect the implementation of tool. Another major advantage of this approach is that we are able to reuse it to generate the Little-JIL traces from a MCS, with a few minor changes as described below.

Since A* search could effectively generate high-level violation traces, one might wonder why we do not use the A* search to verify a property in the first place. The main reason is that A* search is not effective for proving the property is not violated. In this case, the A* search would need to explore the whole state space of the process and can easily run into the state explosion problem.

5.1.1 Introduction to A* Algorithm

Before discussing the high-level trace generation tool, it is necessary to give a brief introduction to the A* search algorithm. The A* search algorithm is a best-first graph search algorithm. Starting from an initial node, it is able to find the least-cost path that satisfies the given goal. The algorithm maintains a work-list that contains the nodes to be examined. During each iteration of the search, one node is taken from the work-list. This node is then examined to see if it meets the goal. If so, the algorithm reconstructs and returns a path from the initial node to this node. Otherwise, the successors of the node are added into the work-list. This is similar to the many search algorithms such as depth-first and breath-first algorithms. What sets A* algorithm apart from these algorithms is the order in which the nodes in the work-list are visited. In the A* algorithm, the order of a node n to be examined is determined by a heuristic function $f(n)$. The node with the smaller $f(n)$ has the higher priority. Therefore, the algorithm always picks a node with the smallest $f(n)$ to examine during each iteration. The heuristic function $f(n)$ for a node n is the sum of two functions: $g(n)$, which is the shortest distance from the initial node to node n, and $h(n)$, which is the estimate of the distance from node n to the goal. An important property of A* search is that it will find the optimal path if the function $h(n)$ is *admissible*. $h(n)$ is admissible if it is guaranteed to never overestimate the actual minimal distance from node n to the goal. Formally, suppose $h'(n)$ is the function that gives the actual minimal cost from node n to the goal, $h(n)$ is admissible if and only if $h(n) \leq h'(n)$ for all node n. The proof of this property can be found in [114]. It should be noted that for a search problem, one may come up with different $h(n)$ functions. The A* algorithm is able to find an optimal path with all these functions as long as they are admissible. The only difference is the performance. Compared to the depth-first algorithm, A* search is usually able to find a shorter path. Compared to the breath-first algorithm, A* algorithm is faster because it often explores fewer states. Therefore, A* search is widely used to solve many practical search problem.

5.1.2 High-Level Trace Generation Tool Architecture

Figure 5.1 shows the architecture of the high-level Little-JIL trace generation tool. There are four major components: a *search engine*, a *goal constructor*, a *goal manager*, and an *Little-JIL event manager*.

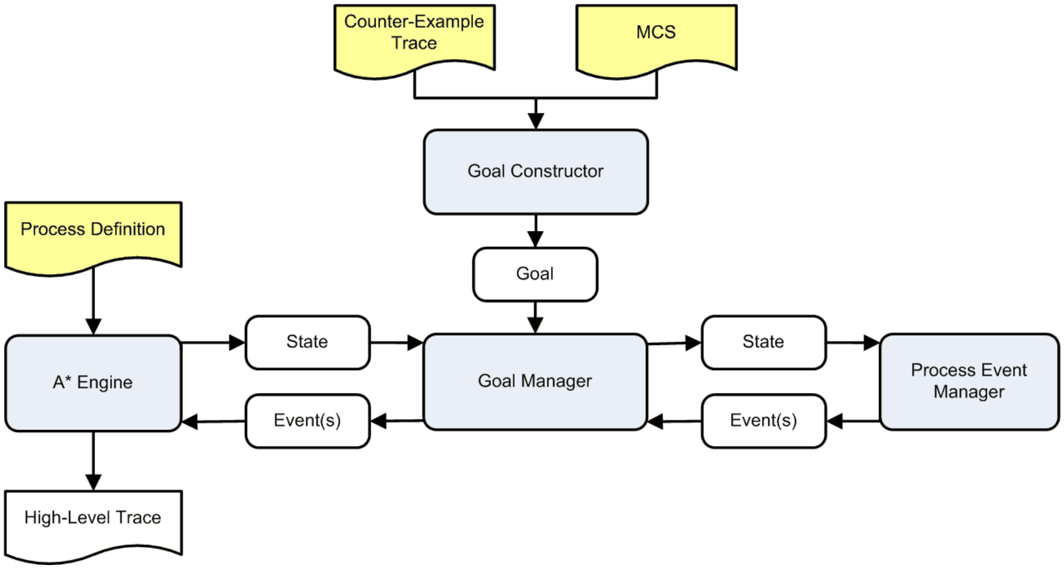


Figure 5.1. Little-JIL Trace Generation Tool Architecture

- The *search engine* applies the A* search algorithm to explore the state space of the given Little-JIL process definition. Each time a process state is visited, the *search engine* consults the *goal manager* to decide what to do next. There are three possible results that may be returned by the *goal manager*. The *goal manager* may indicate that the current process state meets the goal. If so, the *search engine* will reconstruct the Little-JIL trace from the initial process state to the current process state and stops the search. The *goal manager* may also indicate that the goal can never be achieved after the current process state. In this case, the *search engine* simply discards this state and picks up another state from the work-list to visit. The last possible result is that the goal is not yet met by the given process state but may possibly be achieved by its successors. If this is the case, the *search engine* computes the successors of the current process state and adds them onto the work-list.

- The *goal constructor* creates the goal that needs to be achieved by the search. The *goal constructor* for FSV extracts the goal from a counter-example trace, while the *goal constructor* for FTA constructs the goal from a MCS. In our approach, the goal is encoded as a finite-state automaton (FSA) for both FSV and FTA. The labels in this FSA are events: either property events for FSV, or fault tree events for FTA. Since both kinds of events are designed to share the same interface, the goal can be implemented as a generic FSA, independent of the underlying difference between these two kinds of events.

- The *goal manager* makes the decision of what the *search engine* should do about a process state based on the goal. The process state passed from the *search engine* contains a goal state that represents the state of the goal right before this process state. The *goal manager* first forwards the process state to the *Little-JIL event manager* to see whether any event may occur when the process goes into that state. Based on the goal FSA, it then computes the next goal state according to the events returned by the *Little-JIL event manager*. If the next state is the accepting state, it will tell the search engine that the goal is achieved by the process state. If the next state is the violation state, it will tell the *search engine* to discard this process state because no successors of this process state can achieve the goal. When the next goal state is neither the accepting state nor the violation state, the *goal manager* will notify the *search engine* to keep searching the successors of the process state. In this case, some other information will also be return to the *search engine*, including the next goal state and the value of $h(x)$ for the process state. The next goal state will be stored in the successors of the process state. The value of $h(x)$ will be used to compute $f(x)$ for the successors of the process state. Given a process state x, $h(x)$ is defined as the distance between the next goal state corresponding to x and the accepting state in the goal FSA. For an arbitrary FSA, the distances between two states may differ on different paths connecting these states. The goal FSA in our approach, however, guarantees that the distances between a goal state and the accepting state are always the same. We provide a more detailed discussion of $h(x)$ in Section 5.1.2.1 and 5.1.2.2.

- The *Little-JIL event manager* is used to generate event(s) for a given process state. The *search engine* adopts an interleaved model of the Little-JIL process definition. Therefore, only one step state in the process state is changed from the previous process state. We call such a step state in a process state the active step state. The *Little-JIL event manager* checks whether any event may occur when the active step goes into the specific active step state. Although sharing the same interface, the *Little-JIL event manager* for FSV and the one for FTA use different methods to generate the events.

The major part of this framework, including the *search engine* and the *goal manager*, can be shared for both FSV and FTA. The only two components that need to be customized are the *goal constructor* and the *Little-JIL event manager*.

5.1.2.1 Customization for FSV

In this section, we describe the goal constructor that extracts a goal FSA from a given counter-example trace. We also discuss the way in which events are generated by the Little-JIL event manager for FSV. These two customized components allow the search engine to find a Little-JIL trace that violates the property in the same way as the low-level counter-example trace.

5.1.2.1.1 Goal Constructor

Our process verification framework allows the analyst to use various underlying verification tools such as FLAVERS and SPIN. Counter-example traces produced by different verification tools may have different forms. Ignoring the syntax differences, however, a counter-example trace can usually be represented as a sequence of nodes:

$$n_1, n_2, \ldots, n_i(e_1), \ldots n_j(e_2), \ldots, n_k(e_m), \ldots, n_k$$

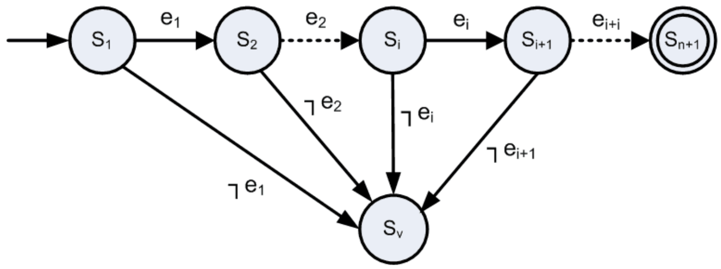


Figure 5.2. Goal FSA from Counter-Example Trace

Each node corresponds to one state of the model fed to the verification tool. Some nodes (i.e. n_i, n_j, and n_k) may be associated with an event in the alphabet of the property. This means that the event associated with a node occurs when the model goes into the state represented by that node. Given a counter-example trace, we can extract a sequence of events:

$$e_1, e_2, \ldots, e_m$$

The order of events in this sequence must remain the same as that in the counter-example trace. This event sequence will drive the property into the violation state. Since the model translated from the Little-JIL process definition is equivalent to the Little-JIL process definition with respect to the property, there must exist a Little-JIL trace that will produce the same event sequence. What we need to do, therefore, is to construct a goal from the event sequence so that the A* search engine can find such a Little-JIL trace. Given the event sequence $e_1, e_2, \ldots, e_n$, the finite state automaton for the goal is shown in Figure 5.2.

States $s_1, s_2, \ldots, s_n$ correspond to events $e_1, e_2, \ldots, e_n$ respectively. s_{n+1} is the accepting state, and s_v is the violation state. At the state $s_i (1 \leq i \leq n)$, only the event e_i can cause the state to move to state s_{i+1}. All the other events in the alphabet will lead to the violation state. Once the goal state goes into the accepting state s_{n+1}, we know that the current Little-JIL trace can produce the given event sequence that violates the property. So the search can stop and return the current Little-JIL trace.

In this goal FSA, there is only one path from any state $s_i(1 \leq i \leq n)$ to the accepting state. Therefore we define $h(s_i)$ as $n-i+1$. For a process state n associated with s_i, any path from n to the process state n_f that satisfies the goal must contain process states associated with $s_{i+1}, \ldots, s_{n+1}$. The actual distance between n and n_f should be at least $n - i + 1$. So this $h(s_i)$ is admissible function that can never overestimate the actual distance.

5.1.2.1.2 Little-JIL Event Manager

The events for FSV are property events. They are generated according to the property event bindings discussed in Chapter 3. An event binding maps a property event to one or more concrete events in the process. There are three kinds of concrete events: *step state events*, *exception throwing events*, and *parameter def/use events*. The step state events can directly map to step states. For example, if there is a property event binding saying "event e occurs if step s goes into the started state", the Little-JIL event manager will generate the event e if the active step state indicates that step s is started. The exception throwing events correspond to `TERMINATED` step states with specific exceptions. For instance, if there is a property event binding saying "event e occurs if step s throws exception ex", the Little-JIL event manager will generate the event e only if the active step state indicates that step s is terminated with exception ex. As discussed in Chapter 3, parameter def/use events can be treated as step `COMPLETED` and `STARTED` states respectively.

5.1.2.2 Customization for FTA

In this section, we discuss how to customize the goal constructor and the Little-JIL event manager for a MCS to allow the search engine to find a Little-JIL trace that shows how events in the MCS cause the hazard to occur.

5.1.2.2.1 Goal Constructor

Unlike the counter-example trace, the MCS contains a set of events that do not have a specific order. Therefore the goal should accept sequences of those events occurring in any order.

The MCS is a minimal set of primary events (or negation of primary events) that could lead to the given hazard. As discussed in Chapter 4, only the following types of basic events may appear in a MCS:

- *Event Type 1: All inputs are correct, but S produces incorrect output o*
- *Event Type 2: Artifact o is wrong when S is posted (S is the root step)*
- *Event Type 9: Exception e is thrown by S*
- *Event Type 11: Step S is opted-out*
- *Event Type 13: Step S is retracted*

Note that event of *Event Type 2* can be a basic event only if the step S is the root step. Otherwise, the event must be caused by another event within the process, and thus cannot be a basic event. To match against the step state events generated by the event manager, we need to do a simple event translation for the events in the MCS. Notice that each basic event shown above is explicitly or implicitly associated with a step state of the process. An event is translated to its corresponding step state event using the following map:

All inputs are correct, but S produces incorrect output o → *Step S is completed*
Artifact o is wrong when S is posted → *Step S is posted*
Exception e is thrown by S → *Step S is terminated by exception e*
Step S is opted-out → *Step S is opted-out*
Step S is retracted → *Step S is retracted*

The translation ignores the artifact information in the first two types of events. This information is not necessary for generating the Little-JIL trace because whether an artifact is wrong or not does not affect the control flow of the process. Suppose there is a trace containing a step state event "*S is completed*", we can interpret this event as "*All inputs are correct, but S produces incorrect output o*" (o is an output of S). This interpretation does not invalidate the trace.

After the event translation, a MCS becomes a set of step state events $\{e_1, e_2, \ldots, e_n\}$. This is an unordered set of events as opposed to the ordered event sequence extracted from the counter-example trace in FSV. Since we want to find a Little-JIL trace along which events in the MCS cause the top event to occur, we should also take the top event into consideration. The top event also needs to be translated into a step state event h. Given the set $\{e_1, e_2, \ldots, e_n\}$ and h, the goal finite state automata satisfies the following conditions:

1. Each state in the goal is a sub-set of $\{e_1, e_2, \ldots, e_n\}$, except the accepting state s_a and the violation state s_v. The initial state s_0 is the set $\{e_1, e_2, \ldots, e_n\}$.

2. Given a state s that is neither s_a nor s_v,

   (a) If s is not an empty set, for each event e in s, there is a transition with the label e from the state s to the state $s' = s - e$. There is also a transition with the label $!e$ from s to the violation state s_v. $!e$ is the negation of event e;

   (b) For the state s that is the empty set, there are a transition labeled h from s to the accepting state s_a and a transition labeled $!h$ from s to the violation state s_v. h here is the hazard event.

3. Given a state s, if an event e does not appear on a transition defined in 2, the event e will not change the state.

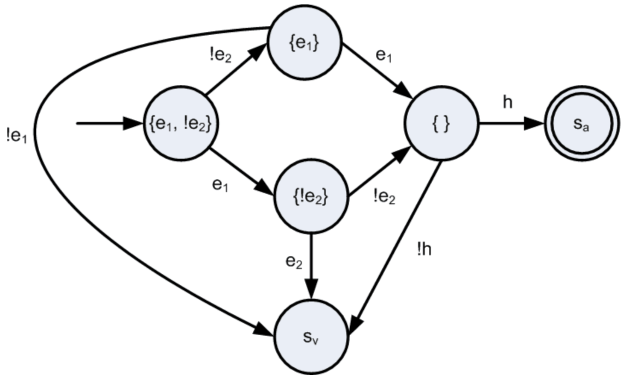


Figure 5.3. Goal FSA from MCS

Figure 5.3 shows the goal for a MCS that contains event e_1 and $!e_2$ after the event translation.

Although there may exist multiple paths from a state to the accepting state, the lengths of those paths are the same. For a state s_i that contains n events, it cannot get to the accepting state unless all n events as well as the hazard event h are encountered. Therefore, $h(s_i) = n + 1$. It is obvious that this $h(s_i)$ is admissible and can never overestimate the actual distance.

5.1.2.2.2 Little-JIL Event Manager

The Little-JIL event manager for FTA is trivial. As shown above, the FSA generated by the goal constructor is defined over step state events. Therefore, the Little-JIL event manager simply returns the event that corresponds to the active step state. For example, Little-JIL event manager will generate the event *Step S is started* if the active step state indicates that step S is started.

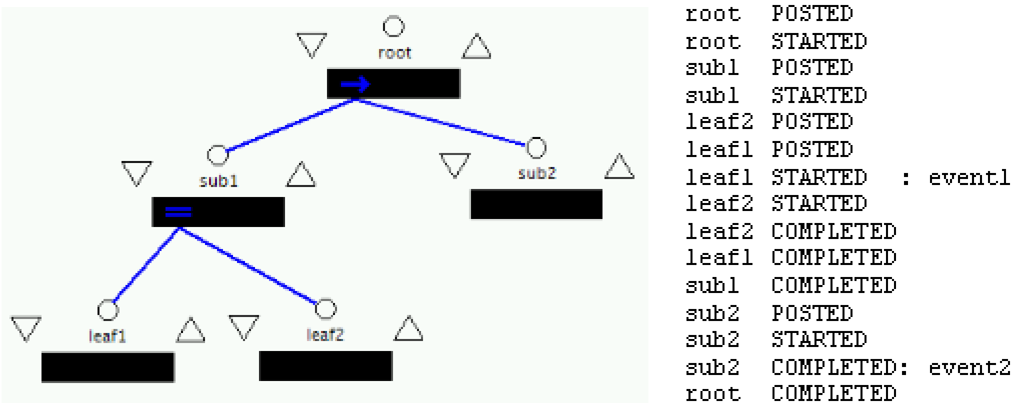
5.1.3 Scaling Problem

Although this approach avoids the difficulties faced by the reverse-mapping approach, it might not work if the counter-example trace contains too few events and the process is very large. In this case, the A* search has to search almost all executions of the Little-JIL process definition and may run out of memory. One possible solution is to automatically select some steps that change the control flow of the process and annotate them with additional events during the translation. This will result in a counter-example trace that contains more events, but should alleviate the problem. In fact, the more events that are added, the faster the high-level trace can be found. However, adding events to the model usually slows down the verification because it increases the model size. For future work, it would be interesting to explore the strategies to minimize total time and perform an experimental evaluation of those strategies.

5.2 Trace Visualization

The Little-JIL trace produced by the high-level trace generation tool, as well as other techniques such as simulation, is usually represented as a sequence of step executions annotated with necessary information, such as events, time, and artifact usage. Since the processes to be analyzed tend to be complex, the traces are often lengthy and verbose. This prevents them from being easily comprehended by the analyst.

To solve this problem, we implemented a generic Little-JIL trace visualization tool that is able to display traces produced by different analysis tools using different graphic representations. Since the traces often carry a lot of information, we suspect that one single representation might not be that helpful. Instead, several complementary representations that display different aspects of the traces might sometimes be a better choice. After investigating various existing techniques for representing the phases and activities of work flows or project executions, such as the Gantt Chart [57], we support two trace visual representations: the *Structured Textual View* and the *Timeline View*. We believe that they are able to help analysts to navigate and understand the Little-JIL traces more easily.



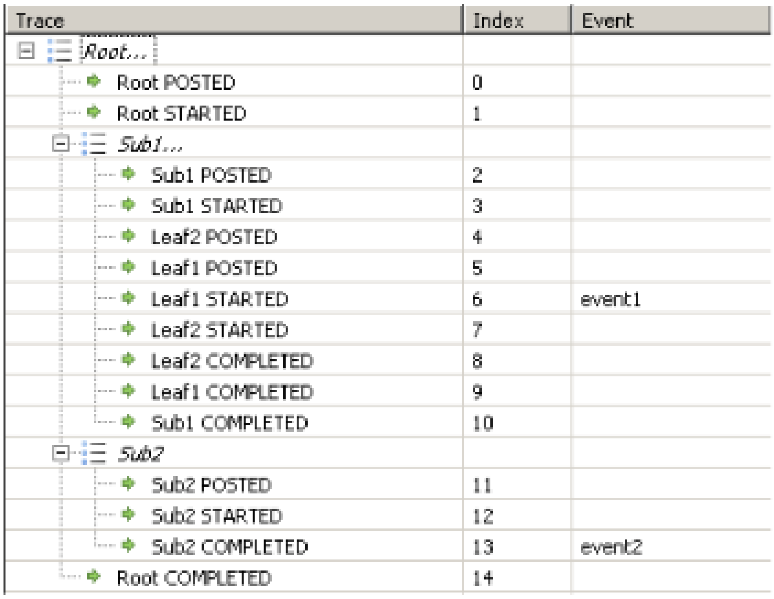
```
root   POSTED
root   STARTED
sub1   POSTED
sub1   STARTED
leaf2  POSTED
leaf1  POSTED
leaf1  STARTED   : event1
leaf2  STARTED
leaf2  COMPLETED
leaf1  COMPLETED
sub1   COMPLETED
sub2   POSTED
sub2   STARTED
sub2   COMPLETED: event2
root   COMPLETED
```

Figure 5.4. Simple Trace Example

To present these two representations, we use the simple example showed in Figure 5.4. The trace on the right is a Little-JIL trace that violates a property of the process on the left. The trace is a sequence of step states, some of which are associated with property events. For example, state `Leaf1 STARTED` in the trace is associated with event *event1*. Note that trace produced by other analyses may be annotated with data other than the events. In the two representations that we discuss, this data is treated in the same way as events.

5.2.1 Structured Textual View

An important feature of Little-JIL is that a process is organized as the hierarchical decomposition of steps. The user can often understand the high-level logic of the process by looking at the first few levels of steps. If necessary, the user can also quickly pinpoint the step of interest by following the hierarchical structure of the process. This hierarchical structure information, however, is buried in the Little-JIL trace. The structured textual representation tries to retrieve the hierarchical structure information as much as possible from the plain trace and present the trace to the user using this information.



Trace	Index	Event
Root...		
Root POSTED	0	
Root STARTED	1	
Sub1...		
Sub1 POSTED	2	
Sub1 STARTED	3	
Leaf2 POSTED	4	
Leaf1 POSTED	5	
Leaf1 STARTED	6	event1
Leaf2 STARTED	7	
Leaf2 COMPLETED	8	
Leaf1 COMPLETED	9	
Sub1 COMPLETED	10	
Sub2		
Sub2 POSTED	11	
Sub2 STARTED	12	
Sub2 COMPLETED	13	event2
Root COMPLETED	14	

Figure 5.5. Structured Textual View

Figure 5.5 shows the structured view of the example trace. The first column is a tree. The leaf nodes are trace nodes from the Little-JIL trace. The non-leaf nodes are called scope nodes that are intended to match the steps in the process. The trace nodes under a scope node constitute a sub-trace. Because of the interleaving, some steps in the process may not have corresponding scope nodes. In the example trace, the trace nodes of step $Leaf1$ and $Leaf2$ interleave with each other because the parent step $Sub1$ is a parallel step. Therefore, we are not able to put them under different scope nodes without changing their order. The second column shows the index for each trace nodes and the third column shows the events corresponding to the trace nodes.

The structured textual view allows the user to collapse or expand scope nodes. When a scope node is collapsed, the index range of the sub-trace corresponding to the scope node will appear in the index column, and the sequence of events in the sub-trace will appear in the event column. The view also allows the user to filter

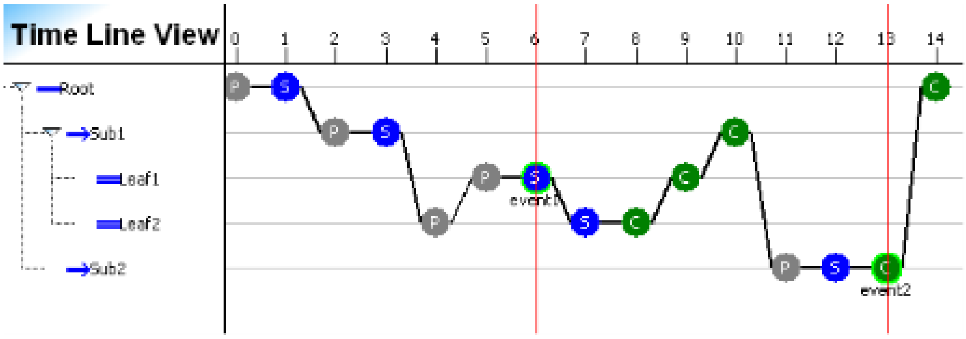


Figure 5.6. Timeline View

certain types of trace nodes. The user, therefore, is able to look into the slice of the trace by filtering out all `POSTED` traces nodes.

5.2.2 Timeline View

The Timeline View consists of two panels. The left panel is the process tree view. The right panel shows the trace nodes in the two dimensional way. The x-axis is the time. Since there is no time information in the trace generated by the process verification, we use the indexes of trace nodes to represent the time. The y-axis is the steps in the left panel. The trace nodes of a step lay on the same horizontal line corresponding to the step on the left panel. Different types of trace nodes are distinguished using different letter and color. For instance, a node with letter *P* and grey color represents a `POSTED` trace node. If a node is associated with an event, the event will be shown above or below the node depending on the position of the node. Moreover, there is a vertical red line crossing every node with event.

The Timeline View also allows the user to collapse non-leaf steps. When a step is collapsed, the trace nodes between the first and the last node of the step become a dash line, as shown in Figure 5.7. The events in-between, however, are still visible.

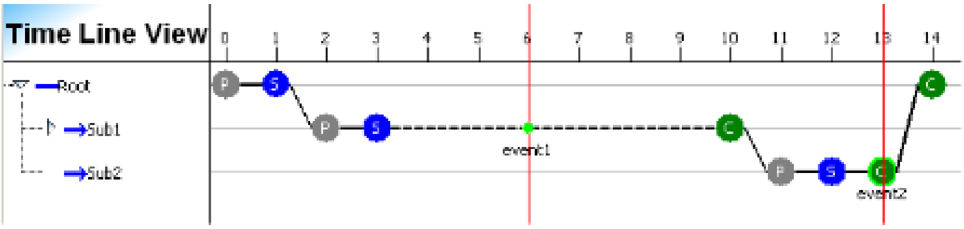


Figure 5.7. Timeline View - Collapsed

5.3 Summary

In this chapter, we discussed our high-level trace generation approach that is able to generate high-level process traces based on counter-example traces through the model used by the verification tool. Instead of mapping the counter-example trace backward along the translation and optimizations, this approach performs an A* search on the process definition with the search goal being the sequence of property events as they appear in the counter-example. One advantage of this approach is that it will not be affected by changes in the translation and optimizations. Another major advantage of this approach is that, with a few minor changes, it can be reused to generate the process traces from MCSs produced by FTA. Due to the complexity in real-world processes, process traces produced by the high-level trace generation tool as well as other techniques could be lengthy and verbose, preventing them from being easily comprehended by analysts. To make process traces more accessible to analysts, we provided support for visualizing process traces using two graphic representations: the *Structured Textual View* and the *Timeline View*. Although the discussion is based on the Little-JIL process definition language, the underlying ideas of our approaches can be adopted to generate and represent traces in other languages.

CHAPTER 6
EVALUATION

In this work, we designed and implemented an analysis framework that is capable of performing Finite-State Verification(FSV) and Fault Tree Analysis(FTA) on Little-JIL process definitions. It is intended to be used to effectively detect defects and weakness in processes. To evaluate this analysis framework, we applied it to the analysis of two non-trivial real-world, human-intensive medical processes: a *In-Patient Blood Transfusion* process and a *Chemotherapy* process. The emphasis of these two case studies has been on demonstrating errors and weaknesses found by the analysis framework. In the remainder of this chapter, we first give a brief introduction to the two selected medical processes. Then we present the two kinds of analysis techniques that we applied. For each kind of analysis, we first describe the methodology employed in applying the analysis, then present the analysis results and discuss the lesson learned from the analysis.

6.1 Introduction to Selected Cases

Medical errors are a major cause of death in our society. A 1999 report from the Institute of Medicine (IOM) [79] estimated that approximately 100,000 people die each year in US hospitals from preventable medical errors. There is ample anecdotal evidence that the complexity of the processes used to administer healthcare is a significant source of the problem. The healthcare literature is replete with documented evidence of such errors as administration of blood of the wrong type, misidentification of patients, and incorrect dosages of potentially lethal medications.

The medical community is aware of these problems and has approached them in a number of ways. One principal approach has been to devise mandated procedures for carrying out many healthcare activities, especially those identified as being particularly high-risk. Mandated procedures are generally described in considerable detail, sometimes in documents that are dozens of pages long. These documents consist largely of natural language text, often supplemented by diagrams. These documents are the basis for both the training of medical professionals and the actual processes performed in hospitals. Despite the care that went into the creation of such documents, as well as other safety measures, a subsequent IOM study [110] revealed that error rates in hospitals had not declined significantly in the five-year period following the initial IOM report.

Examination of documents used to describe medical processes suggests several reasons why such documents have proven to be inadequate. Documents describing such processes as blood transfusion (e.g., [134], [135]) provide good examples of the problem. Despite attempts to be complete, they contain terms that are poorly defined and inconsistently used, and important details are often missing, especially details for handling special cases that might arise. Recognizing such limitations, the medical community has tried to employ a number of modeling representations, but these are usually based upon such formalisms as data flow graphs that make it relatively cumbersome to represent the handling of exceptional cases or complicated concurrency and synchronization. As a result, these representations generally fail to represent the full complexity of these processes.

Indeed, the many diverse circumstances under which activities like blood transfusions must take place require processes of considerable complexity. Moreover, a medical process often requires coordinating the efforts of many different parties, often performing their activities in parallel. The complexity of a process increases the risk of defects. This suggests that the finite-state verification and fault-tree analysis used to analyze complex, distributed systems might be effective in analyzing medical processes. Therefore, in our case study we chose two medical processes: an *In-Patient Blood Transfusion* process and a *Chemotherapy* process. More specifically, the two selected processes are real-world processes actually being performed in the hospital, instead of processes defined in a textbook or medical reference. This is because we hope that our analysis can directly help the medical professionals to identify errors and weakness in the processes that they perform.

It should be noted that although both cases are medical processes, this work also suggests the applicability of our approach to human-intensive, safety-critical processes in other problem domains.

6.1.1 In-Patient Blood Transfusion Process

The blood transfusion process is used to administrate blood or blood product to a patient's bloodstream. It is a common, high-risk, human-intensive medical intervention. Despite strict regulation by the US Food and Drug Administration as well as healthcare accreditation agencies, the error rate in blood transfusion is significant and believed to be underreported [64]. The blood transfusion process that we are going to analyze is an *In-Patient Blood Transfusion* process, which is a blood transfusion process specific for patients who are hospitalized overnight. Although usually following the similar high-level steps, the in-patient blood transfusion processes practiced in different hospitals or medical institutes may vary in details. In our case study, we choose to model and analyze the one employed in the Baystate Medical Center in Massachusetts.

Figure 6.1 shows the top-level diagram of this process defined in Little-JIL. The full process is provided in Appendix D. The simplified blood transfusion process used as the running example in the discussion of our process analysis framework is a highly simplified version of this in-patient blood transfusion process, where several

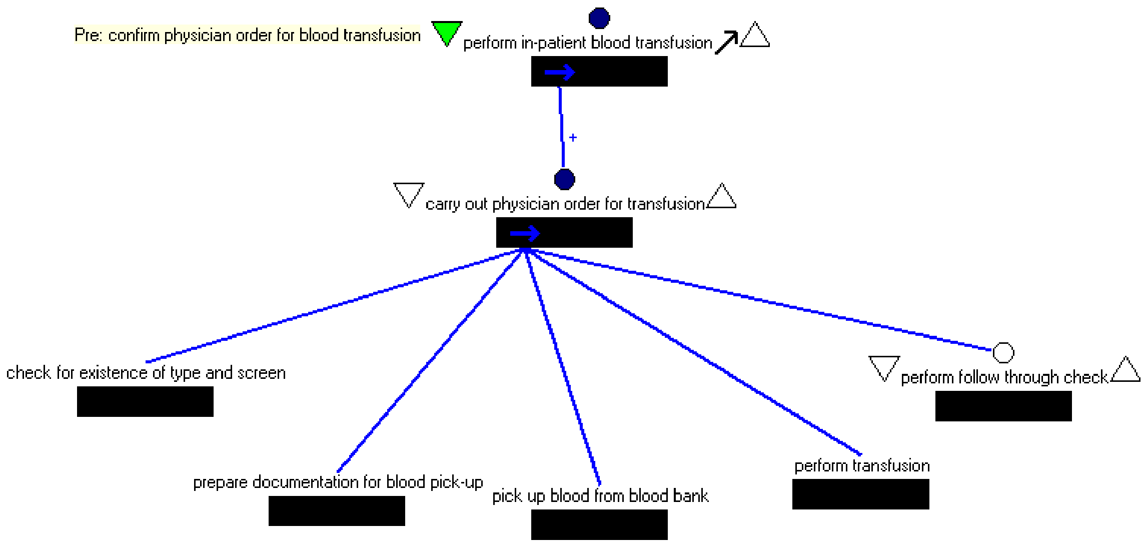

Pre: confirm physician order for blood transfusion
perform in-patient blood transfusion
+
carry out physician order for transfusion
check for existence of type and screen
prepare documentation for blood pick-up
pick up blood from blood bank
perform transfusion
perform follow through check

Figure 6.1. In-Patient Blood Transfusion Process

sub-processes are abstracted into leaf steps and most of the checks and exception handlers are removed.

An in-patient blood transfusion process cannot start unless there is a blood transfusion order from a physician. Therefore, the root step "*perform in-patient blood transfusion* has a pre-requisite step "*confirm physician order for blood transfusion.* This step checks whether a blood transfusion order is presented. If no order is presented, an exception "*NoPhysicianOrder* will be thrown and the whole process is terminated.

One blood transfusion order may require blood products to be transfused to the patient several times. The step "*carry out physician order for transfusion*" models the process that needs to be followed each time the blood products are transfused. There is a cardinality + adjacent to the edge between step "*perform in-patient blood transfusion*" and step "*carry out physician order for transfusion*", which means that "*carry out physician order for transfusion*" will be done one or more times, as prescribed in the physician order. Since "*perform in-patient blood transfusion*" is a sequential step (note the right arrow in the left hand side of the step bar), instances of "*carry out* *physician order for transfusion*" must be executed sequentially. The step "*carry out physician order for transfusion*" has five sub-steps. Since it is a sequential step, once it is started, five sub-steps will be executed one by one, from left to right. The first sub-step "*check for existence of type and screen*" obtains the blood type and screen for the patient. Then the step "*prepare documentation for blood pick-up*" creates the blood pick up document based on the blood type and screen. Given the pick up document, blood products can be obtain from the blood bank following the step "*pick up blood from blood bank*". In the step "*perform transfusion*", the blood products are transfused to the patient. The step "*perform follow through check*" performs the necessary last check on the patient. Note that all four sub-steps except "*perform follow through check*" do not have triangles attached to their step bars, which means that they are references. Each of them refers to a sub-process defined in another diagram.

6.1.2 Chemotherapy Process

Chemotherapy is the use of chemical substances to treat disease. In its modern-day use, it refers primarily to the administration of cytotoxic drugs to treat cancer. Chemotherapy medications are typically highly toxic, and thus it is of overriding importance to be sure that the right patient receives the right medications in the right dosages at the right times. To assure this, elaborate processes are carried out that integrate the efforts of such diverse medical personnel as doctors, nurses, pharmacists, and clerical workers. Chemotherapy processes aim to speed the flow of treatment, while assuring that errors do not occur. Checks are in place to guard against committing such errors. Similar to the in-patient blood transfusion process, we also selected a chemotherapy process employed in the Baystate Medical Center.

Figure 6.2 is the top-level diagram of chemotherapy process defined in Little-JIL. The entire Little-JIL process definition has more than 250 steps and is provided in Appendix H. The root step "*chemotherapy process*" has two sub-steps. The first

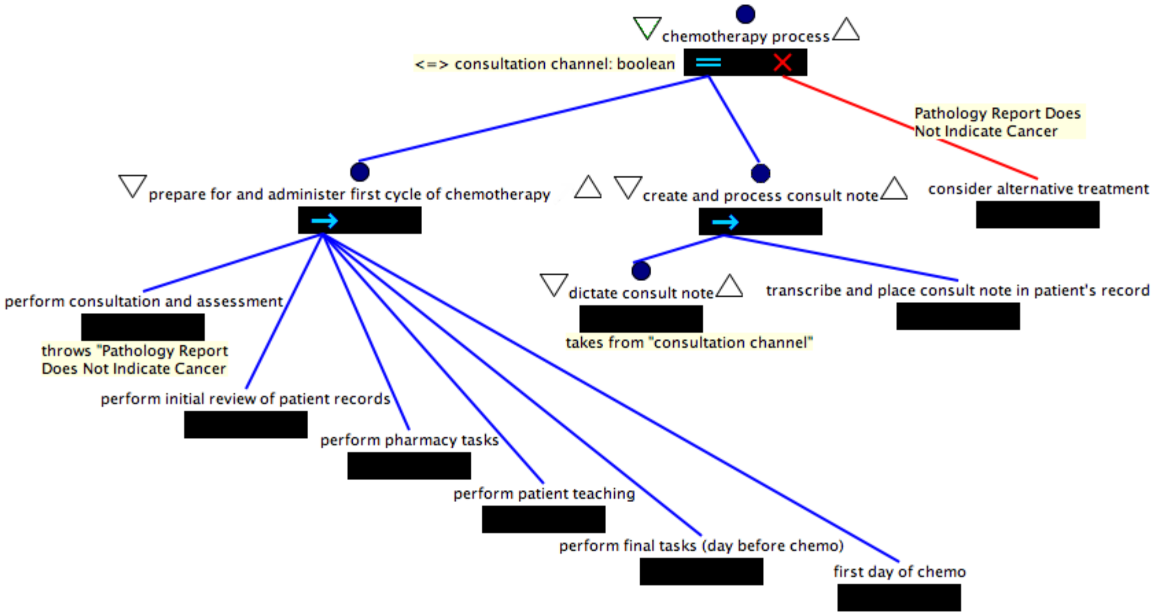

chemotherapy process
<=> consultation channel: boolean
Pathology Report Does Not Indicate Cancer
prepare for and administer first cycle of chemotherapy
create and process consult note
consider alternative treatment
perform consultation and assessment
throws "Pathology Report Does Not Indicate Cancer
dictate consult note
takes from "consultation channel"
transcribe and place consult note in patient's record
perform initial review of patient records
perform pharmacy tasks
perform patient teaching
perform final tasks (day before chemo)
first day of chemo

Figure 6.2. Chemotherapy Process

sub-step, "*prepare for and administer first cycle of chemotherapy*", is further decomposed into six sub-steps to be executed in sequence. These six sub-steps capture the major stages of the chemotherapy process. Although the agent assignments are not given in this diagram, "*perform consultation and assessment*" is done by a Medical Doctor(MD); "*perform initial review of patient records*" by a Practice Registered Nurse(RN) and a Triage Medical Assistant; "*perform pharmacy task*" by a Pharmacist; "*perform patient teaching*" by a Nurse Practitioner; "*perform final tasks (day before chemo)*" by a Pharmacist and a Clinic RN; and "*first day of chemo*" is done again by a Pharmacist and a Clinic RN. In the step "*perform consultation and assessment*", the doctor may determine that the patient's pathology report does not indicate cancer. In this case, the *Pathology Report Does Not Indicate Cancer* exception is thrown (the decomposition of the "*perform consultation and assessment*" step is not shown due to space limitations). The exception propagates up the tree until it reaches the root step that has a matching handler "*consider alternative treatment*".

Thus, control is transferred to "*consider alternative treatment*" and appropriate action is taken.

The second sub-step of the root step is "*create and process consult note*". It is further decomposed into two sub-steps: "*dictate consult note*" performed by a Medical Doctor(MD) and "*transcribe and place consult note in patient's record*" by a Medical Records Clerk. Since the consult note is primarily used for billing and legal purposes and does not directly affect the patient's treatment, the doctor may choose to dictate the consult note right after evaluating the patient or later, while the tasks in "*prepare for and administer first cycle of chemotherapy*" are already underway. Therefore the root is modeled as a parallel step so that the "*dictate consult note*" step can potentially execute in parallel with the step "*prepare for and administer first cycle of chemotherapy*". At the same time, the channel "*consultation channel*" imposes the additional restriction that the doctor cannot dictate the consult note before evaluating the patient's condition — the step "*dictate consult note*" takes a parameter from the "*consultation channel*" and thus cannot start until "*perform patient consultation*" (not shown for lack of space), which is a sub-step of "*perform consultation and assessment*", completes and writes a parameter to the "*consultation channel*".

6.2 Methodology

Each case study involves four phases. We first rigorously specify the process in Little-JIL language. Then the properties collected during the process specification are defined using PROPEL. Given the Little-JIL process definition and the PROPEL property specifications, the FSV tool is used to verify the process definition against the property specifications. During the verification, the process definition and property specification may need to be changed to eliminate the violations found by the FSV tool. Once there are no more violations, we finally apply the FTA to look for the weakness in the process.

6.2.1 Process Specification

As mentioned above, we intend to analyze the process that is actually being followed in the hospital, instead of the process defined in a textbook or medical reference. To elicit such process, we met regularly with a group of medical professionals who perform the real process. The medical professionals are responsible for describing the processes and reviewing the material created by us. And we are responsible for defining the process in Little-JIL. The process is first described by the medical professionals in natural language, sometimes complemented by flow graphs. We then try to capture the description using Little-JIL language. Due to misunderstandings, the Little-JIL process definition may not correctly capture the real process. Therefore, we need to explain the process that we specified to the medical professionals. To do this, we employ a Little-JIL process description generation tool to automatically create the detailed description of the process definition. The description explains every details of the process definition in a disciplined natural language that can be easily understood by the medical professionals. The medical professionals then study this description to check for any inconsistency. With their feedbacks, we modify the process definition accordingly and generate a new description. This iteration ends when both sides agree that there is no inconsistency between the process definition and the real process.

6.2.2 Property Specification

The FSV can only identify errors that violate the given properties. To detect more errors, we need to come up with as many properties as possible. The initial set of properties are collected from the requirements given in regulations and references. While the process is being defined, it is also not unusual for goals or high-level requirements to be mentioned. These requirements are recorded into the requirement set. After the process definition has begun to stabilize, meetings are then held to elicit the set of requirements into properties that can be verified by the verification framework. The requirements are usually initially stated informally in natural language. We first work with the medical professionals to agree on a glossary of terms that are used to more systematically describe the requirements, but still in natural language. After agreement is reached on these statements, we work closely with the medical professionals to develop the detailed property specifications using PROPEL. Similar to the process definition, property specification is also an iterative process, which ends when the property specifications are believed to correctly formulate the requirements.

6.2.3 Finite-State Verification

As mentioned earlier, the formal properties are defined in terms of abstract events. Before verifying such properties, we work together with the medical professionals to specify event bindings that map abstract events in the properties to the concrete events. After the process definition, property specifications, and property event bindings are ready, we apply the FSV tools to evaluate whether the process definition is consistent with the formal properties. It is often the case that the FSV tools find problems in the process definition and in the property specifications as well as in the process. We carefully record these problems during the verification, and consult the medical professionals before making changes to the process definition, the property specifications, or to the actual process to eliminate those problems. To make sure that no error is introduced by the changes, we re-run the verification on the modified process definition and property specifications. This iterative procedure completes when no more problems are detected.

6.2.4 Fault-Tree Analysis

After the verification is completed, the fault-tree analysis is applied on the process definition. The hazards that we are interested are straight-forward for both processes that we investigate. For the in-patient blood transfusion process, the hazard is that the blood products to be transfused to the patient are wrong. For the chemotherapy process, the hazard is that the chemo drugs to be administered are incorrect. For a process, we first define the hazard as the top event. With the top event, our FTA tool automatically derives the fault tree from the process definition. It also computes the MCSs for the fault tree. We carefully investigate the MCSs to identify the weaknesses of the process. The results are presented to the medical professionals as guidance for them to further improve the process.

6.3 Results and Discussion

Each of the four phases may detect various problems and errors. Since this thesis focuses on the process analysis, we only present and discuss the results of the finite-state verification and fault-tree analysis here.

6.3.1 Finite-State Verification

The complete verification reports for the two processes can be found in Appendix E and Appendix I. In this section, we first summarize the verification results of the two processes. Then we discuss various kinds of problems and errors identified during the verification in details.

For the in-patient blood transfusion process, 60 properties were specified. 5 of these cannot be verified because they are timing properties. Another 6 cannot be verified because we could not find appropriate bindings for them. Among the remaining 49 properties that were verified, 41 properties were violated. For the chemotherapy process, 57 properties were specified. 13 of these cannot be verified because we could not find appropriate bindings for them. Among the remaining 44 properties that were verified, 11 properties were violated.

The errors found during the verification fall into three categories: *errors in the process definitions*, *errors in the property specifications*, and *errors in the real processes.*

6.3.1.1 Errors in the process definitions

The majority of errors fall into this category. Some of these were exposed when we were trying to define the event bindings for certain properties. There simply were no steps in the process definition that could be bound to the abstract events used to specify those properties. In some cases, this was because the part of the process intended to address the issue being captured by a property was simply missing and had to be added. For instance, in the in-patient blood transfusion process, there was no step that can be bound to the event "*review patient history*" in the property "*Must review patient history before infusing each unit of blood product*". This pointed out that an important step "*review patient history*" as well as the exception handler "*handle previously unrecognized patient problem*" to handle the exception *ProblemFoundInPatientHistory* thrown by this step were missing. In other cases, the problem was that certain steps needed to be further decomposed to identify the sub-steps that should be bound to the property events. For example, there is an event "*review patient history*" in the property "*Must review patient history before infusing each unit of blood product*". However, we were not able to identify a step in the in-patient blood transfusion process that related to this event. This problem showed that the leaf step "*assess patient*" needed to be decomposed to add a sub-step "*review patient history*".

The actual verification helped us to find errors in the process definitions. Some of these were so subtle that they remained undiscovered despite careful inspection by both medical professionals and us. For example, the in-patient blood transfusion process definition specified that if discrepancies occur during the "*verify blood product*" step, then "*verify blood product*" is to be terminated and a *FailedBloodProductCheck* exception is to be thrown. The handler "*handle failed blood product check*" requires the nurse to send the blood product back to the blood bank and obtain a replacement blood product. The exception continuation badge of this handler was specified to be *continue*, which implied that, after this exception has been handled in this way, the process continues with the nurse signing the blood bank form after obtaining the replacement product. This process thus violated the property "*Must verify the blood product before it is transfused to the patient*", where the event "*verify the blood product*" is bound to completion of the step "*verify blood product*". It is clear that this would introduce the potential for serious medical error, and in the actual process the nurse verifies the patient identification and the product identification again after obtaining the replacement product. Thus, attempting to verify this property indicated that the exception's continuation badge was wrong and needed to be changed to *restart*.

6.3.1.2 Errors in the property specification

The verification also helped us to discover several errors in the property specification. Some properties were specified incorrectly because we overlooked certain exceptional situations under which the properties are not required to hold. For instance, a violation was initially found when we verified the in-patient blood transfusion process against the property "*Obtain Patient's stated name and birth date before infusing each unit of blood product*". The violation showed that when the patient is unable to communicate to the nurse (i.e. the patient is in coma), an exception *PatientUnableToCommunicate* will be thrown. The exception handler then directs the nurse to obtain the patient's name and birth date in another way. In this case, the event "*obtain patient's stated name and birth date*" will not occur. It is obvious that this property is not required for this exceptional situation. So we changed the property specification to require the property to hold only if the exception *PatientUnableToCommunicate* is not thrown. And verification showed no violation for the modified property.

In addition, a few properties turned out to be unnecessary. In the in-patient blood transfusion process, the step "*verify blood product information*" has four sub-steps that check the different aspects of the blood product: "*verify product tag matches patient ID bracelet*", "*verify product tag matches product label*", "*check product expiration date & time*", and "*confirm product type on product tag matches that on patient record*". There were several properties that required these sub-steps to be executed in a specific order. The process definition, however, defines the parent step "*verify blood product information*" to be a parallel step. This means that four substeps may be executed in any order, which violates those properties. As we consulted the medical professionals, they confirmed that in the real-world process, it does not matter in which order those checks are performed, as long as they are completed before the blood product is infused to the patient. Those properties only reflect the choice of the person who suggested such properties, and should not be applied to other practitioners.

6.3.1.3 Errors in the real processes

The most interesting error that we found is an error in the real-world chemotherapy process. In attempting to verify the property "*Consult note must be present in patient's record before chemotherapy can be administered*", we bound the event "*consult note is present in patient's record*" to the *COMPLETED* state of step "*file consult note in patient's record*", and the event "*administer chemotherapy*" to the *STARTED* state of step "*administer chemo drug*". The verifier reported that the chemotherapy process may violate this property, and it produced a trace of a valid execution of the process where "*administer chemo drug* is started before "*file consult note in patient's record*" is completed. To better understand the violation trace, let's look at Figure 6.2. Step "*file consult note in patient's record*" is a sub-step of step "*transcribe and place consult note in patient's record*", and step "*administer chemo drug*" appears in the sub-process with the root step "*first day of chemo*". Since the root step "*chemotherapy process*" is a parallel step, step "*administer chemo drug*" may indeed be executed prior to step "*file consult note in patient's record*". After talking to the medical professionals, we found out that the medical records clerk who files the consult note into the patient's record assumes the clinic RNs check whether the consult note is present in the patient's record on the first day of chemotherapy. The clinic RNs, however, said that they do not perform that check. Therefore, it is possible that this error occurs in the real process. Although it probably will not lead to an accident, it may cause serious legal issues if an accident happens. Such errors exist because the process is very complicated, involving different people who have conflicting assumptions about others' responsibilities. Without formally specifying and verifying the process. Such conflicts may remain undiscovered despite careful inspection by the medical professionals.

The verification process involves defining the event bindings, running the verifier, examining violation traces to identify the cause of errors, and consulting medical professionals to eliminate errors. All these steps make the verification process very lengthy. As we have seen, except the one in the real chemotherapy process, all errors that we identified are in the formal process definitions and property specifications. This is often the case when verifying software systems — much of the early effort of verification is devoted to finding errors in the model and the properties. For human-intensive processes, like two medical processes we investigated, this may be more significant since the initial artifacts from which process definition and properties are derived are less concrete and precise than, say, source code.

Nevertheless, the effort invested on the verification, even for the blood transfusion process where no errors in real process were found, still provided great benefits. The direct benefit is that the verification greatly improves the accuracy of formal process definitions, which are also intended to be used for other types of analysis (i.e. FTA), simulation, and possibly even guidance in the clinical setting. With process definitions that do not accurately reflect the real processes, these approaches may produce invalid results or cause dangerous consequences. Another benefit is that verification provides a solid foundation for process improvement in the future. As noted earlier, real-world processes usually undergo continuous changes to cope with changed requirements, discovered defects, inefficiencies, or the need to reduce costs, etc. The modifications to the processes are usually made incrementally, with changes introduced to address specific goal. To ensure that no errors are introduced by the change, one needs to incorporate the change in the formal process definition and verify it against previously verified properties. Often, the property event bindings can be reused, perhaps with some changes to small parts of the process. The number of violations detected, if any, should be much smaller than those detected for the original process definition. Therefore, the verification process will usually be easier than that of the first round.

As mentioned earlier, the emphasis of verifying the two processes has been on demonstrating that finite-state verification can be automated and applied to analyze human-intensive processes. Although we did not perform a careful evaluation of how optimizations improve the performance, verification of the two processes showed that the optimizations can greatly scale up the verification framework. In our evaluation, we used FLAVERS as the underlying verification tool in our process verification framework. The verification was performed on a Dell Inspiron E1505 laptop with Intel Core Duo CPU. Without applying the optimizations, the translated models for both processes caused FLAVERS running out memory with the maximum heap size set to 1GB (FLAVERS is a Java program). When all the optimizations were enabled, every property for both processes was verified in 1 minute.

6.3.2 Fault-Tree Analysis

For the In-Patient Transfusion process, we performed the FTA for the hazard "*the blood unit to be transfused to the patient are wrong*". On a Dell Inspiron E1505 laptop with Intel Core Duo CPU, it took our fault tree derivation tool 238 seconds to derive the fault tree for this hazard. The fault tree contains 344 gates and 405 events (including 61 primary events and 344 intermediate events). Among the primary events, there are 25 fault events that could cause the hazard to occur and 36 conditional events that decide the propagation of those fault events. The *Equivalent Event Removal* optimization removed 263 intermediate events and 263 gates, resulting in a fault tree with 142 events and 81 gates, as shown in Appendix F. To see whether there is single point of failure in the process, MCSs need to be computed from the fault tree. It took another 106 seconds to compute 37 MCSs, which are listed in Appendix G. One might notice that there are more MCSs than primary fault events. The reason is that one fault event may appear in more than one MCS, representing different scenarios in which the fault event could cause the hazard to occur. As discussed in Chapter 4, an MCS may contain spurious conditional events that need to be removed. After manually removing those spurious events, all MCSs still contain at least 6 events. Therefore, this process is not exposed to any single point of failures with respect the given hazard.

For the Chemotherapy process, the hazard that we analyzed is "*the chemo drug to be administered to the patient is wrong*". It took 369 seconds to derive the fault tree for this hazard. The fault tree contains 1345 gates and 1425 events (including 80 primary events and 1345 intermediate events). Among the primary events, there are 52 fault events and 28 conditional events. The *Equivalent Event Removal* optimization removed 1174 intermediate events and 1174 gates, resulting in a fault tree with 251 events and 171 gates, as shown in Appendix J. 52 MCSs were computed in 810 seconds. There is an MCS that contains only one event "*Step handle LabelAndOrderDontMatch(call pharmacy) produces wrong ChemoDrug*". Step "*handle LabelAndOrderDontMatch(call pharmacy)*" is an exception handling step that is invoked if any discrepancy between the chemo drug label and the chemo order is detected. This step, therefore, needs to identify the error, in the chemo drug or in the chemo order, that cause the discrepancy. If it turns out that the chemo drug is wrong, the step needs to make a change to the chemo drug to fix the error. Since this step is performed by a human agent, errors may be introduced into the chemo drug during the execution of this step, as indicated by the fault event in the MCS. The high-level process trace generated from this MCS shows that once step "*handle LabelAndOrderDontMatch(call pharmacy)*" is completed, the process will proceed to the next step and eventually execute step "*administer chemo drug*" because this exception handling step is associated with a *continue* badge. Moreover, there is no other step to check the chemo drug between this step and step "*administer chemo drug*". In other words, event "*Step handle LabelAndOrderDontMatch(call pharmacy) produces wrong ChemoDrug*" will always cause the hazard to occur. This exposes a single point of failure in the process definition.

6.4 Summary

We evaluated the proposed process analysis framework by applying our implementation of the process analysis framework to two real-world medical processes: an *In-Patient Blood Transfusion* process and a *Chemotherapy* Process. The results showed that the framework can be used to effectively detect the defects in such real-world human-intensive processes. The FSV detected a large number of errors in the process definitions and the property specifications. More interestingly, it identified a subtle error in the chemotherapy process itself. Through FTA, we discovered a single point of failure in the chemotherapy process.

CHAPTER 7

CONCLUSION AND FUTURE WORK

Processes are used in almost every domain in our society. For example, products are produced by various manufacturing processes; software is developed following software development processes; government or business services are provided via established processes; healthcare is delivered through different medical processes. Such real-world processes are often undergoing improvements in order to achieve various goals, such as dealing with changed requirements, eliminating defects, increasing the quality of the products, and reducing costs. Analysis should play a critical role in process improvement. Identifying and evaluating the root causes of the problem (i.e. failing to meet changed requirements, discovered defects and low quality) require careful analysis of the process. In addition, since errors may be introduced by changes to the process, re-analysis should be carried out to detect such errors before the modified process is deployed. To date, the analysis support for such processes is very limited. If done at all, it is usually performed manually and can be time-consuming, costly and error-prone. Analysis of human-intensive processes, where the human contributions require considerable domain expertise and have a significant impact on the success or failure of the overall mission, are of particular concern because they can be extremely complex and prone to errors.

In this thesis, we investigate how two selected safety analysis techniques — *Finite-State Verification (FSV)* and *Fault Tree Analysis (FTA)* — can be automated and applied to analyze human-intensive processes. We developed an analysis framework that is capable of performing both FSV and FTA on processes specified in a process definition language with precisely defined semantics. For FSV, we proposed a general process FSV framework that is intended to address three practical issues: flexibility, scalability and usability. To address the flexibility issue, a two-phase translation approach was proposed to facilitate using different existing verification tools. To address the scalability issue, several optimizations are proposed to reduce the size of the translated models. To address the usability issue, automatic support is provided to translate properties and violation traces. For FTA, we implemented a template-based approach to automatically derive fault trees from Little-JIL process definitions. When applying this approach to derive fault trees from real-world processes, we encountered three issues: scalability, looping construct and NOT gate. To address the scalability issue, we proposed three optimizations that are able to significantly reduce the size of fault trees so that they can be understood and evaluated more easily. To address the looping issue, we applied a simple approach to remove loops in fault trees so that they can be analyzed using existing fault tree evaluation tools. To obtain more accurate results, we chose to use NOT gates to capture relationships between certain conditional events. In addition to the support for automatic fault tree derivation, we also proposed two representations — partial fault tree and process trace — to make MCSs more easily understood. This process analysis framework has been implemented for the selected Little-JIL process definition language and been applied to analyze two real-world human-intensive processes: an In-Patient Blood Transfusion process and a Chemotherapy process. The results showed that the process analysis framework can be used to effectively detect the defects in real-world human-intensive processes.

In the remainder of this chapter, we discuss several directions for future research.

7.1 Finite-State Verification

To verify real-world processes, scalability is always one of the most important issues that need to be addressed. In our process verification framework, we proposed several optimizations that are able to take advantage of aspects of the structure of the process definitions to reduce the size of the models. Our evaluation showed that the optimizations were able to greatly improve the performance of the verification of two selected medical processes. But those two processes are relatively simple. As we extend our work to consider processes involving many medical professionals carrying out many activities in parallel, we expect that the time and memory resources required will limit the applicability of our analysis frameworks. Therefore, we need to look for more optimizations to further scale up our analysis frameworks. We plan to investigate existing optimizations used in software verification to see whether we can incorporate them into our process verification framework. We will also look for domain-specific optimizations that take advantage of special features of processes in specific domains.

7.2 Fault Tree Analysis

The fault tree derivation algorithm that we implemented may produce inaccurate results. Since a Little-JIL process definition only defines the interface of a leaf step, which does not specify how the leaf step produces its outputs from its inputs. Therefore, the fault tree derivation algorithm has to assume that any output of a leaf step depends on all its inputs. Thus, leaf steps that do not satisfy this assumption may cause the derived fault tree to contain superfluous sub-trees. Evaluation of such fault trees may result in inaccurate results. In addition, as shown in the example discussed in Section 4.3, computed MCSs may not be minimal because our fault tree derivation algorithm is not able to identify the relationships between different primary events. To improve the precision of FTA, we plan to extend the fault tree derivation algorithm to allow analysts to precisely specify the dependencies between outputs and inputs for certain leaf steps as well as relationships between different primary events.

The fault tree derivation algorithm is a polynomial algorithm that is able to derive fault trees from complex real-world processes. The bottleneck is in the understanding and evaluation of derived fault trees. In our approach, we proposed three optimizations to reduce the size of fault trees so that they can be understood and evaluated more easily. To further reduce the size of fault trees, we intend to consider a partition approach that decomposes a large fault tree into several smaller sub-trees that are easier to understand. To evaluate the fault tree, one should be able to evaluate the sub-trees individually and then combine or summarize the results.

7.3 Trace Generation and Visualization

In our approach, we use an A* search algorithm to generate a high-level process traces from a low-level counter-example traces produced by verification tools or a MCS created by FTA. Although this approach avoids the difficulties faced by the reverse-mapping approach, it might not work if the counter-example trace or the MCS contains too few events and the process is very large. In this case, the A* search has to search almost all executions of the Little-JIL process definition and may run out of memory. One possible solution is to automatically select some steps that change the control flow of the process and annotate them with additional events during the translation. This will result in a counter-example trace that contains more events, but should alleviate the problem. In fact, the more events that are added, the faster the high level trace can be found. However, adding events to the model usually slows down the verification because it increases the model size. For future work, it would be interesting to explore the strategies to minimize total time and perform an experimental evaluation of those strategies.

7.4 Failure Mode and Effects Analysis

In this work, we adapted two analysis techniques to analyze processes. The next technique that we start to investigate is the Failure Mode and Effects Analysis (FMEA), also referred to as Failure Mode Effect and Criticality Analysis (FMECA) [120]. It is a hazard analysis technique that can be used to evaluate the impact of an individual failure mode of a component on the overall system. FMEA was originally developed by the US military in 1949, described in Military procedure MIL-P-1629 [130], to evaluate the effect of system and equipment failures. It was then used by NASA in the early sixties for the Apollo Project. In the early 1980s, FMEA was introduced to automotive industry in the U.S. The application led to an industry-wide FMEA standard SAE J-1739 [29] developed by Chrysler Corporation, Ford Motor Company and General Motors Corporation. Since then, the generic nature of the method assisted the rapid spreading of FMEA to various other industries, such as aerospace industry and chemical industry. Besides manufacturing industries, many service providers, such as heath care providers [42], also apply FMEA to assess and improve their service delivery processes. Introduced by Reifer in 1979 [111], FMEA began to be used to analyze software based systems. Details about various software FMEA approaches can be found in [101].

In FMEA, A *failure mode* at a component basically refers to a fault that could occur at the component. FMEA usually involves five steps:

1. *Modeling the system.* The system model is usually specified using a flow chart. It is also used to prescribe the boundary and granularity of the analysis.

2. *Identifying failure modes.* For each component in the system model, all possible ways the component may go wrong need to be considered and represented as failure modes.

3. *Determining failure effects.* An inductive procedure is used to determine all possible hazards that could eventually be caused by each component failure mode. Given a failure mode, the procedure identifies its immediate effects based on the underlying dependency and interactions between components. Those failure effects are represented as failure modes at successor or next higher level components, which in turn need to be further developed. This iteration continues until the hazards are reached.

4. *Prioritizing failure modes* Failure modes then can be prioritized by their risks. There are a number of different methods that can be used to evaluate the risk of a failure mode based on the severities of the hazards caused by it as well as other data such as probability and detectability of a failure mode.

5. *Identifying potential causes and proposing actions.* For failure modes with high priorities, potential causes are identified and actions that can be applied to eliminate or reduce those causes are proposed.

The information produced during the FMEA, such as failure modes, hazards caused by those failure modes, risks of failure modes, causes, and actions, are recorded in a worksheet. This worksheet can then be used as guidance for improving the design or implementation of the system.

Similar to the FTA, the traditional FMEA is also conducted manually by a team with comprehensive knowledge of the system being analyzed. It is obvious that step (1), (4), and (5) cannot be automated. Thus, we focus on developing an approach that is able to automatically identify failure modes and determine failure effects. As a matter of fact, the failure modes in FMEA and the events in FTA both refer to the faults in components. Therefore, we can use the set of predefined events in the automatic fault tree derivation to represent the failure modes. Templates used in automatic fault tree derivation can also be used to determine failure effects of a given failure mode. As mentioned above, in each iteration of the procedure to determine failure effects, the analyst need to identify other failure modes that could be caused by a given failure mode. This can be automated by finding all templates whose leafs contain the given failure modes. Since a template is a tree where the leaf cause the root to occur, the roots of identified templates are the failure modes that could be caused by the given failure mode. We have demonstrated the feasibililty of this idea by implementing it in a prototype. This prototype was able to automatically produce the FMEA worksheet from a simplified blood transfusion process defined in the Little-JIL process definition language [132]. As future work, we plan to improve this prototype and apply it on complex real-world, human-intensive processes.

7.5 Interplay of Different Analysis Techniques

All three analysis techniques (FSV, FTA, FMEA) have their own limits. They seem to complement each other. We expect that more comprehensive results can be produced by integrating these analysis techniques. In FSV, the kinds of errors can be detected depend on the properties verified. In other words, errors could not be found unless the corresponding property is verified. Therefore, the analyst needs to think of as many properties as possible. This usually is a very difficult task. During our preliminary experiments, we observed that a large part of the properties address the checks for the faults occurring in steps. Therefore, the faults produced by FTA or FMEA may help the analyst to think of more properties. On the other hand, by detecting errors in the process definitions, FSV can help to increase the credibility of the results produced by FTA and FMEA. Since both FTA and FMEA are performed on the process definitions, such errors may cause FTA and FMEA to produce invalid results. FTA and FMEA can also complement each other. For FTA, it is often difficult to come up with all important hazards. On the contrary, a problem for FMEA is that it might be a challenge to identify all possible failure modes in the system. Therefore, approaches such as [83] try to solve this problem by combining FTA and FMEA. In these approaches, the primary events identified by FTA serve as the inputs for FMEA. And hazards produced by FMEA may in turn be used to find more primary events by FTA. Similar to these approaches, we also plan to investigate how to combine FTA and FMEA to achieve more complete results.

Ultimately we envisage the development of a process environment in which a spectrum of process analysis techniques are smoothly integrated with the process definition tool. Such a support environment would hopefully lead to a systematic and well-reasoned approach to process improvement.

APPENDIX A

LITTLE-JIL PROCESS TRANSLATION TEMPLATES

Posting Template

This type of template translates the behaviors after a step is *POSTING* and before it is *POSTED*. Since such behaviors are the same for all steps, only one template is defined for this type.

After a step is *POSTING* and before it is *POSTED*, the step may pass in parameters and obtain artifacts from channels. Since the Little-JIL process definition does not define the behavior of a leaf step, in our translation the behaviors of leaf steps are abstracted using a conservative model (see A). Our implementation is based on Little-JIL 1.4. In this version, a parameter may only affect the flow of control in a leaf step. Since such effect is subsumed by the abstract model of the leaf step, parameters and parameter passing can be ignored. Therefore, the posting template only translates the behavior to obtain artifacts from channels.

As discussed in Chapter 3, a channel is simply translated into a Boolean variable that indicates whether the channel is empty. Little-JIL provides four operators to obtain an artifact from a channel: *Blocking Take*, *Non-Blocking Take*, *Blocking Read* and *Non-Blocking Read*. Here we use a simple example to explain how these operators are translated by the posting template. Suppose that there is a step `S` that has an input parameters `p` and `p` gets an artifact from a channel `channel`.

Suppose that operator *Blocking Take* is used to obtain an artifact from `channel` to `p`. If the channel is not empty, an artifact is removed from the channel and is put into the parameter. Otherwise, the step is not posted until an artifact becomes

```
loc S_POSTING:
   when true do { } goto S_channel_take;
loc S_channel_take:
   when ( channel == false ) do { }
     goto S_channel_take;
   when ( channel != false )
     do { channel := false; }
     goto S_POSTED;
```

Figure A.1. BIR for Posting Template - Channel Blocking Take

```
loc S_POSTING:
   when true do { } goto S_channel_nbtake;
loc S_channel_nbtake:
   when true
     do { channel := false; }
     goto S_POSTED;
```

Figure A.2. BIR for Posting Template - Channel Non-Blocking Take

```
loc S_POSTING:
   when true do { } goto S_channel_read;
loc S_channel_read:
   when ( channel == false ) do { }
     goto S_channel_read;
   when ( channel != false ) do { }
     goto S_POSTED;
```

Figure A.3. BIR for Posting Template - Channel Blocking Read

```
loc S_POSTING:
   when true do { } goto S_channel_nbread;
loc S_channel_nbread:
   when true do { } goto S_POSTED;
```

Figure A.4. BIR for Posting Template - Channel Non-Blocking Read

available in the channel. The BIR code generated by the posting template for step `S` is shown in Figure A.1. Once in the *POSTING* state, the step proceeds to obtain an artifact from the channel `channel` (line 3-8). It first checks whether the channel is empty (`channel == false`). If so, it keeps going back to check again until the channel is not empty. If the channel is not empty, the channel is set to be empty (`channel := false`) and the step `S` goes into the *POSTED* state.

Figure A.2 shows the case where operator *Non-Blocking Take* is used. If the channel is not empty, an artifact is removed from the channel and is put into the parameter. Unlike *Blocking Take*, *Non-Blocking Take* does not block the step even if the channel is empty. Therefore, the BIR code does not check whether the channel is empty. It simpy sets the channel to be empty and posts the step.

Figure A.3 shows the case where operator *Blocking Read* is used. *Blocking Read* is similar to *Blocking Take* except that the artifact is copy to the parameter and is not removed from the channel. Therefore, if the channel is not empty, the BIR code simply posts the step without setting the channel to be empty.

Figure A.4 shows the case where operator *Blocking Read* is used. In this case, no matter whether the channel is empty or not, the BIR code posts the step without setting the channel to be empty.

Posted Template

This type of template translates the behaviors after a step is *POSTED*. After posted, a step may be opted-out or retracted under some circumstances. If it is not opted-out or retracted, the step is started. The only template of this type simply uses appropriate *Opted-Out Template* and *Retracted Template* to translate behaviors that check whether the step should be opted-out or retracted.

```
1 // For Sub1:
2    loc Sub1_test_retract:
3       when (S_choice != 0) do { } goto Sub1_RETRACTED;
4       when (S_choice == 0) do { } goto Sub1_test_optout;
5    loc Sub1_RETRACTED:
6       when true do { } goto T_Sub1_end;
7
8 // For Sub2:
9    loc Sub2_test_retract:
10      when (S_choice != 1) do { } goto Sub2_RETRACTED;
11      when (S_choice == 1) do { } goto Sub2_test_optout;
12   loc Sub2_RETRACTED:
13      when true do { } goto T_Sub2_end;
```

Figure A.5. BIR Representation for Choice Sub-Step Retracted Template

Retracted Template

This type of template translates the behavior that checks whether a step should be retracted. In Little-JIL, only sub-steps if a choice step or sub-steps of a parallel step may be retracted. Thereofre, two templates are defined: *Choice Sub-Step Retracted Template* handles the sub-steps of choice steps; *Parallel Sub-Step Retracted Template* handles the sub-steps of parallel steps.

Choice Sub-Step Retracted Template

A choice step allows agents to select one of several sub-steps to perform. A sub-step of a choice step will be retracted if it is not the one chosen to be executed. Suppose that there is a choice step `S` has two sub-step `Sub1` and `Sub2`.

Figure A.5 shows the BIR code generated by *Choice Sub-Step Retracted Template* for `Sub1` and `Sub2`. This piece of code checks whether `Sub1` should be retracted. The first location (line 2-4) compares the variable `S_choice` against constant `0`. `S_choice` keeps the index of the sub-step that is chosen to be executed. The constant `0` is the index of the sub-step `Sub1`. If `S_choice` is not `0`, `Sub1` is not chosen to be executed. Therefore it goes directly into the *Retracted* state, which ends the thread `T_Sub1`.

```
1  // For Sub1:
2     loc Sub1_test_retract:
3        when  (e1 || e2) do { } goto Sub1_RETRACTED;
4        when !(e1 || e2) do { } goto Sub1_test_optout;
5     loc Sub1_RETRACTED:
6        when true do { } goto T_Sub1_end;
7
8  // For Sub2:
9     loc Sub2_test_retract:
10       when true do { } goto Sub2_test_optout;
```

Figure A.6. BIR Representation for Parallel Sub-Step Retracted Template

Otherwise, Sub1 continues to execute the code created by *Opted-Out Template* that checks whether this sub-step is opted-out.

Parallel Sub-Step Retracted Template

For a posted sub-step of a parallel step, it should be retracted if any exception is thrown by other sub-steps. Suppose that there is a choice step S has two sub-step Sub1 and Sub2. Sub2 may thrown exception e1 and e2. Figure A.5 shows the BIR code generated by *Parallel Sub-Step Retracted Template* for Sub1 and Sub2. In this code, Boolean variable e1 or e2 becomes true if the corresponding exception is thrown. Line 3-4 checks whether exception e1 or e2 is thrown. If so, Sub1 is retracted. Otherwise, Sub1 continues to check if it is opted-out. Since Sub1 does not throw any exception, Sub2 will never be retracted. Therefore, Sub2 goes directly to check if it is opted-out.

Opted-Out Template

This type of template translates the behavior that checks whether a step should be opted-out. Only one template is defined for this type. Suppose step S is an optional step. It is up to the agent assigned to S to decide whether to opt out this step. Since such decision making logic is not defined in the Little-JIL process, we conservatively

```
loc S_test_optout:
    when true do { } goto S_STARTED;
    when true do { } goto S_OPTED_OUT;
```

Figure A.7. BIR Representation for Opted-Out Template

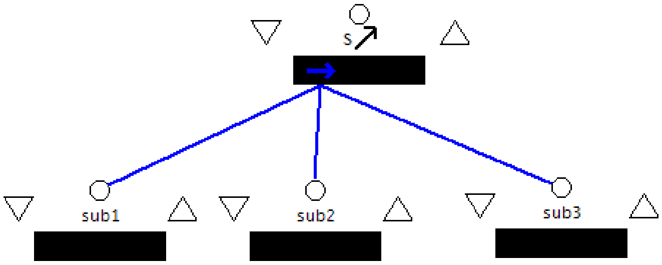


Figure A.8. Sequential Step Example

translate it into a non-deterministic choice. The BIR code generated by *Opted-Out Template* is shown in Figure A.7.

Started Template

This type of template translates the behaviors after a step is started. Corresponding to five step kinds, five templates are defined for this type: *Sequential Step Started Template*, *Try Step Started Template*, *Parallel Step Started Template*, *Choice Step Started Template*, and *Leaf Step Started Template*.

Sequential Step Started Template

The *Sequential Step Started Template* is used to construct the behavior of a sequential step after it has been started. When a sequential step is started, its sub-steps will be executed one by one from left to right. And the sequential step is only successfully completed after all of its sub-steps are successfully completed.

Figure A.8 shows a sequential step `S` that has three sub-steps: `Sub1`, `Sub2`, `Sub3`. The BIR code produced by *Sequential Step Started Template* is given in the Figure A.9. When the sequential step `S` is started, the first sub-step `Sub1` goes into

```
1  loc S_STARTED:
2     when true do { } goto Sub1_POSTING;
3  ... // Sub1
4  loc Sub1_COMPLETED:
5     when true do { } goto Sub2_POSTING;
6  ... // Sub2
7  loc Sub2_COMPLETED:
8     when true do { } goto Sub3_POSTING;
9  ... // Sub3
10 loc Sub3_COMPLETED:
11    when true do { } goto S_COMPLETING;
```

Figure A.9. BIR Representation for Sequential Started Template

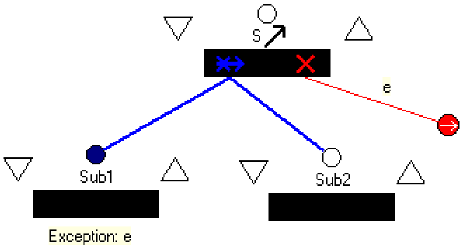


Figure A.10. Try Step Example

the *Posting* state (line 1-2). When `Sub1` is completed, the next sub-step is posting (line 4-5). Similarly, `Sub3` is posting as soon as `Sub2` is completed (line 7-8). And the sequential step `S` goes into the *Completing* state when its last sub-step `Sub3` is completed (line 10-11). In this example, those sub-steps do not throw any exceptions. If a sub-step may throw exceptions, BIR code must be inserted to check and handle those exceptions. This is achieved by invoking the *exception handling template*.

Try Step Started Template

The *Try Step Started Template* is used to construct the behavior of a try step after it has been started. When a try step is started, it tries to execute its sub-steps one by one from left to right. The try step is successfully completed if any of its sub-steps is successfully completed.

```
1   loc S_STARTED:
2      when true do { } goto Sub1_POSTING;
3   ... // Sub1
4   loc Sub1_COMPLETED:
5      when true do { } goto S_COMPLETING;
6   ... // Sub1 cont.
7   loc Sub1_TERMINATED:
8      when true do { } goto S_handle_exceptions1;
9   loc S_handle_exceptions1:
10     when b_e do { b_e:=false; } goto Sub2_POSTING;
11  loc Sub2_POSTING:
12  ... // Sub2
13  loc Sub2_COMPLETED:
14     when true do { } goto S_COMPLETING;
```

Figure A.11. BIR Representation for Try Started Template

Figure A.10 shows a try step `S` that has two sub-steps `Sub1` and `Sub2`. `Sub1` may throw exception `e` that is handled by an exception handler with the *continue* badge. The BIR code produced by applying *Try Step Started Template* is given in the Figure A.11. When the try step `S` is started, the first sub-step `Sub1` goes into the *Posting* state (line 1-2). When `Sub1` is completed, the parent step `S` is completing (line 4-5). If `Sub1` throws exception `e`, it is terminated and the corresponding exception handler is executed. The exception handling code from line 9-10 is created by the *Exception Handling Template*. *Try Step Started Template* sets the target for the transition at line 10 to `Sub2_POSTING`. This means that after the exception `e` is handled, the next sub-step `Sub2` is posting because the exception handler has a *continue* badge. The try step `S` goes into the *Completing* state when its last sub-step `Sub2` is completed (line 14-15).

Parallel Step Started Template

The *Parallel Step Started Template* is applied to construct the behavior of a parallel step after it has been started. It is quite different from the *Sequential Step Started Template* that we just discussed. When a parallel step is started, all its sub-steps will

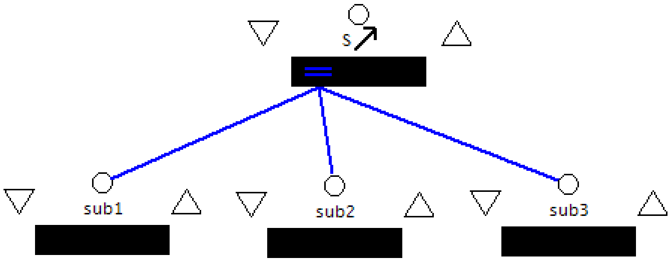


Figure A.12. Parallel Step Example

be posted at the same time. Then the sub-steps can be executed in any (possibly interleaved) order. And the parallel step is only completed after all of its sub-steps has successfully completed.

Figure A.12 shows a parallel step `S` that has three sub-steps: `Sub1`, `Sub2`, `Sub3`. The corresponding BIR representation is given in the Figure A.13. Before the *Started Parallel Step Template* is applied, the sub-steps of the parallel step have already been translated into different BIR threads. In this example, thread `T_Sub1`, `T_Sub2` and `T_Sub3` are created for sub-step `Sub1`, `Sub2`, and `Sub3` respectively. When the parallel step `S` is started, three threads of its sub-steps are started by the `start` action (line 3-8). Then the parallel step waits until these thread to finish (line 9-20). Note that each `join` action on a thread is followed by a `threadTerminated` test for that thread. This is a constraint imposed by the BIR language specification.

Choice Step Started Template

The *Choice Step Started Template* is applied to construct the behavior of a choice step after it has been started. Once started, a choice step allows the agent to select a sub-step to execute among its sub-steps. A choice step is considered completed only after one of its sub-steps have completed. Once a sub-step is selected to be executed, the other sub-steps are retracted. Handlers with *continue* badge are used to execute another sub-step if the selected sub-step fails. More specifically, when a handler with

```
loc S_STARTED:
   when true do { } goto S_fork_subs;
loc S_fork_subs:
   when true do {
      start(T_Sub1);
      start(T_Sub2);
      start(T_Sub3);
   } goto S_join_Sub1;
loc S_join_Sub1:
   when true do { join(T_Sub1); } goto S_test_Sub1;
loc S_terminate_test_Sub1:
   when threadTerminated(T_Sub1) do { } goto S_join_Sub2;
loc S_join_Sub2:
   when true do { join(T_Sub2); } goto S_test_Sub2;
loc S_terminate_test_Sub2:
   when threadTerminated(T_Sub2) do { } goto S_join_Sub3;
loc S_join_Sub3:
   when true do { join(T_Sub3); } goto S_test_Sub3;
loc S_terminate_test_Sub3:
   when threadTerminated(T_Sub3) do { } goto S_COMPLETED;
...
```

Figure A.13. BIR Representation for the Parallel Started Template

continue badge is executed, all the retracted sub-steps will be posted again and one of them is selected to be performed.

In our approach, sub-steps of choice steps are translated into threads. This is not so intuitive because the choice is supposed to choose only one sub-step to execute each time. The problem is that the choice step does not restrict the order of its sub-steps. In another word, sub-steps of a choice step may execute in any order though not in parallel. One straight-forward translation option is to enumerate all possible such orders in the BIR code. This could easily blow up the BIR code if the choice step has many sub-steps. Instead, we decide to translate sub-steps of a choice step into BIR thread and force them to run one after another. This is achieved by adding a *choice variable* that indicates which sub-step is selected. In addition, each sub-step is assigned one *done variable* that represents whether the sub-step is done or not. The choice variable is randomly assigned to the index of a single sub-step at the beginning of each iteration. Then all sub-step threads are started. The thread of a sub-step first tests if it has been done. If so, the thread ends immediately. Otherwise the thread proceeds to check the sub-step index against the choice variable. Only the one whose index equals to the value of the choice variable will continue. And other threads simply retract their corresponding sub-steps. The major advantage of this approach is that it produces more compact BIR code, greatly relieving the burden of the model construction for verification tools.

Figure A.14 shows a choice step `S` that has two sub-steps `Sub1` and `Sub2`. Both sub-steps could throw exception `e` which is handled by a hander with *continue* badge. This allows the one sub-step to be executed if the other one fails. The BIR produced by applying *Choice Step Started Template* is presented in Figure A.15. When the choice step `S` is started, the done variables for both sub-steps are initialized to be `false` (line 3-7). Then the choice variable of `S` is non-deterministically assigned to be `0` or `1` (line 8-10). If `S_choice` is `0`, `Sub1` is selected to be executed. And if it is `1`, `Sub2`

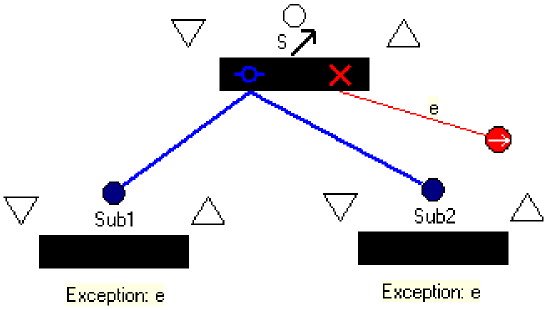


Figure A.14. Choice Step Example

is selected. The next location (line 11-17) checks whether `Sub1` is selected. If `Sub1` is not selected (`S_choice!=0`), `Sub2` should be checked. If `Sub1` is selected and it is not done yet (`(S_choice==0)&&!Sub1_done`), the sub-step threads will be forked. In the case when `Sub1` is selected but it has already done (`(S_choice==0)&&Sub1_done`), it should not be executed again. Therefore, control flow goes back to choose the choice variable of `S`. Line 18-24 checks `Sub2` in a similar way. One may readily notice that in line 25-38, both sub-step threads are forked. According to the semantic of choice steps, however, only one sub-step that has not finished should be executed in each iteration. This constraint is actually enforced in the individual sub-step thread. Figure A.16 presents the sub-step thread for `Sub1`. Line 2-4 shows that `Sub1` can be posted only if it is not done yet. Otherwise, the whole sub-step thread ends. The code that makes sure that `Sub1` should be retracted if it is not selected is not shown here. This piece of code is generated by *Choice Sub-Step Retracted Template* as discussed above. In line 6-9, the done variable `Sub1_done` is set to `true` if `Sub1` is completed or terminated, indicating that `Sub1` is done. If `Sub1` is retracted, however, the done variable is not set to `true` because it could be posted again by the exception handler with *continue* badge. Although not shown in this example, the done variable should also be set to true if the sub-step is opted-out.

```
loc S_STARTED:
   when true do { } goto S_init;
loc S_init:
   when true do {
      Sub1_done := false;
      Sub2_done := false;
   } goto S_choose;
loc S_choose:
   when true do { S_choice := 0; } goto S_check_Sub1;
   when true do { S_choice := 1; } goto S_check_Sub1;
loc S_check_Sub1:
   when ( S_choice != 0 )
      do { } goto S_check_Sub2;
   when ( (S_choice == 0) && !Sub1_done)
      do { } goto S_fork_subs;
   when ( (S_choice == 0) && Sub1_done )
      do { } goto S_choose;
loc S_check_Sub2:
   when ( S_choice != 1 )
      do { } goto S_fork_subs;
   when ( (S_choice == 1) && !Sub2_done)
      do { } goto S_fork_subs;
   when ( (S_choice == 1) && Sub2_done )
      do { } goto S_choose;
loc S_fork_subs:
   when true do {
      start(T_Sub1);
      start(T_Sub2);
   } goto S_join_Sub1;
loc S_join_Sub1:
   when true do { join(T_Sub1); } goto S_test_Sub1;
loc S_terminate_test_Sub1:
   when threadTerminated(T_Sub1) do { } goto S_join_Sub2;
loc S_join_Sub2:
   when true do { join(T_Sub2); } goto S_test_Sub2;
loc S_terminate_test_Sub2:
   when threadTerminated(T_Sub2)
      do { } goto S_handle_exceptions;
```

Figure A.15. BIR Representation for the Choice Started Template

```
thread T_Sub1
   loc Sub1_begin:
      when !(Sub1_done) do {  } goto Sub1_POSTING;
      when Sub1_done do {  } goto Sub1_end;
   ... // Sub1
   loc Sub1_COMPLETED:
      when true do { Sub1_done := true; } goto Sub1_end;
   loc Sub1_TERMINATED:
      when true do { Sub1_done := true; } goto Sub1_end;
   loc Sub1_RETRACTED:
      when true do { } goto Sub1_end;
   loc Sub1_end:
      when true do { exit(T_Sub1); } goto Sub1_exit;
   loc Sub1_exit:
end T_Sub1;
```

Figure A.16. BIR Representation for Sub-step of Choice Step

Leaf Step Started Template

In a Little-JIL process, leaf steps are performed by the assigned agents once they are started. The behaviors of agents, however, are not modeled in the Little-JIL process definition. The process definition only specifies the interfaces of a leaf step, which declares the parameters used by the step as well as exceptions that might be thrown by the step. Based on the interface, we translate the started behavior of a leaf step into a BIR model that conservatively captures such behavior.

Supposed that there is a leaf step `S` that might throw exception `e1` and `e2`. Figure A.17 shows the BIR code generated by *Leaf Step Started Template* for this step. There are three transitions in this location. All of them have the condition `true`. This means that when the execution reaches the location, one transition will be nondeterministically selected to be executed. In the first transition (line 2), no exception is thrown and the step `S` is completing. In the second transition (line 3), the exception `e1` is thrown and the step `S` is terminated. In the third transition (line 4), the exception `e2` is thrown that also terminates `S`.

```
loc S_STARTED:
    when true do { } goto S_COMPLETING;
    when true do { e1 := true; } goto S_TERMINATED;
    when true do { e2 := true; } goto S_TERMINATED;
```

Figure A.17. BIR Representation for Leaf Step Started Template

To handle the rare situations where the precise model of the started behavior of a leaf step is necessary, we support customized started behavior models. The analyst provides the BIR model for a leaf step that defines how this step goes from the *STARTED* state into the *COMPLETING* and *TERMINATED* states. When applying *Leaf Step Started Template* for such a step, the customized model is returned. For a leaf step without a customized model, the default model shown above will be created.

Completing Template

This type of template translates the behaviors after a step is *COMPLETING* and before it is *COMPLETED*. Such behaviors are the same for all steps, so only one template is defined for this type.

After a step is *COMPLETING* and before it is *COMPLETED*, the step may pass out parameters and put artifacts to channels. Similar to the posting template, the completing template only translates the channel operations.

Operator *Write* is the only operator that can be used to put an artifact to a channel. Suppose that there is a step `S` that puts an artifact to a channel `channel` using the *Write* operator. The BIR code after applying the template is shown in Figure A.18. Once in the *COMPLETING* state, the step proceeds to write an artifact to the channel `channel`. Since the *Operator* never blocks the execution of a step, the channel is simply set to be non-empty (`channel := true`).

```
  loc S_COMPLETING:
     when true do { } goto S_channel_write;
  loc S_channel_write:
     when true do {
        channel := true;
     } goto S_COMPLETED;
```

Figure A.18. BIR Representation for Completing Template Example

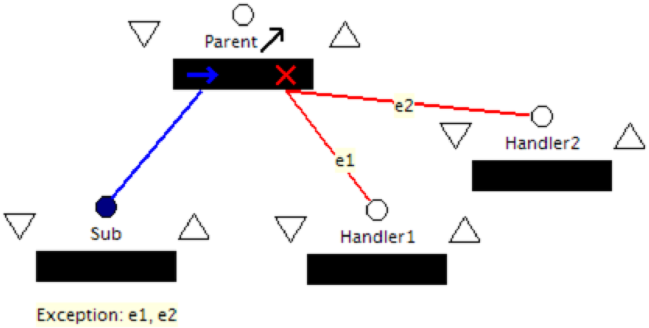


Figure A.19. Exception Handling Example

Exception Handling Template

The *Exception Handling Template* is invoked by various *Started Templates* to generate the code to handle exceptions thrown by the sub-steps. When a sub-step throws an exception, the matching exception handler associated with the parent step will be invoked to handle the exception. If the handler is associated with an exception handler step, the handler step will be executed. The exception handler also has a control-flow badge indicating how the step catching the exception executes after the handler step finishes. Since this exception handling mechanism is the same for all kinds of steps, only one *Exception Handling Template* is defined.

Figure A.19 shows an exception handling example. The sub-step `Sub` may throw two exceptions: `e1` and `e2`. The handler used to handle exception `e1` has a handler step `Handler1` and a *continue* control flow badge. And the handler used to handle exception `e2` has a handler step `Handler2` and a *rethrow* control flow badge. Figure A.20 shows the BIR code that will be executed once `Sub1` is terminated. In this code, `b_e1`

```
loc Parent_fork_T_Handler1:
   when (b_e1==true) do {
      start(T_Handler1);
   } goto Parent_fork_T_Handler2;
   when (b_e1==false) do {
   } goto Parent_fork_T_Handler2;
loc Parent_fork_T_Handler2:
   when (b_e2==true) do {
      start(T_Handler2);
   } goto Parent_fork_T_Handler2;
   when (b_e2==false) do {
   } goto Parent_join_T_Handler1;
loc Parent_join_T_Handler1:
   when (b_e1==true) do {
      join(T_Handler1);
   } goto Parent_terminate_test_T_Handler1;
   when (b_e1==false) do {
   } goto Parent_join_T_Handler2;
loc Parent_terminate_test_T_Handler1:
   when threadTerminated(T_Handler1) do {
   } goto Parent_control_rethrow;
loc Parent_join_T_Handler2:
   when (b_e2==true) do {
      join(T_Handler2);
   } goto Parent_terminate_test_T_Handler2;
   when (b_e2==false) do {
   } goto Parent_control_rethrow;
loc Parent_control_rethrow:
   when (b_e2==true) do {  } goto Parent_TERMINATED;
   when (b_e2==false) do {  } goto Parent_control_continue;
loc Parent_control_continue:
   when (b_e1==true) do { b_e1:=false; } goto ⊔;
   when (b_e1==false) do { } goto ⊔;
```

Figure A.20. BIR Representation for the Exception Handling Template

and `b_e2` are Boolean variables used to encode exception `e1` and `e2` respectively. Since exception handlers may be executed in parallel, the handler steps are translated into BIR threads. `T_Handler1` and `T_Handler2` are BIR threads corresponding to handler step `Handler1` and `Handler2`. The first part of the code (line 1-12) tests whether exceptions are thrown or not one by one. If an exception is thrown, the thread translated from the corresponding exception handler step is started. Otherwise, the next exception is checked. The next part (13-27) joins the threads of exception handling steps if they have been started. The last part (line 28-33) directs the flow of control according to the control-flow badges of the exception handlers. In this example, exception `e2` is tested first because its handler `Handler2` has a *rethrow* badge, which has a higher priority than the *continue* badge of `Handler1`. One might notice that while exception variable `b_e1` is set to `false` (line 32), exception variable `b_e1` remains to be `true` (line 29). This is because exception `e2` is rethrown by its handler. The exception variable should remain `true` until it is handled by a non-rethrown handler associated with any ancestor of the parent step. Another thing needed to be mentioned is that two targets of the last location are empty. The reason is that they may go to different locations for different kinds of parent step. For instance, if the parent step is a try step, the first transformation should go to the posted location of the next sub-step, and the second one should go to the completed location of the parent step. But if the parent step is a sequential step, both should go to the posted location of the next sub-step. These targets, therefore, are left empty and will be filled up by the *Started Template* that invokes the *Exception Handling Template.*

APPENDIX B

FAULT TREE DERIVATION TEMPLATES

Table B.1. Fault Tree Derivation Templates

Event Type		**Template(s)**
Event Type 1	*All Inputs are Correct, But S Produces Incorrect Output o*	None
Event Type 2	*Artifact o is Wrong When S is Posted*	Template 1, 2
Event Type 3	*Artifact o is Wrong When S is Started*	Template 3
Event Type 4	*Artifact o is Wrong When S is Retracted*	Template 4
Event Type 5	*Artifact o is Wrong When S is Opted-out*	Template 5
Event Type 6	*Artifact o is Wrong When S is Completed*	Template 6, 7
Event Type 7	*Artifact o is Wrong When S is Terminated by Exception e*	Template 8, 9
Event Type 8	*Artifact o is Wrong When S is about to be Completed*	Template 10, 11
Event Type 9	*Exception e is Thrown by S*	None
Event Type 10	*Exception e is not Thrown by S*	Tempate 12
Event Type 11	*Step S is Opted-out*	None
Event Type 12	*Step S is not Opted-out*	Template 13
Event Type 13	*Step S is Retracted*	None
Event Type 14	*Step S is not Retracted*	Template 14

Templates for *Artifact o is Wrong When S is Posted*

Since the artifact o is already wrong when S is posted, the fault is not introduced by step S. It must be propagated from the previous step. Depending on whether step S is an exception handler or not, the flow of control goes to S is quite different. To make the templates simpler, we use two separate templates to develop this type of events.

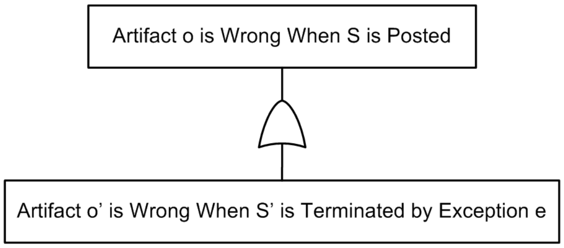

Artifact o is Wrong When S is Posted
Artifact o' is Wrong When S' is Terminated by Exception e

Figure B.1. Template 1

Template 1

Requirement:

- The event e to be developed is *Artifact o is Wrong When S is Posted*, and
- Step S is an exception handler step

Partial Fault Tree:

The partial fault tree for this template is shown in Figure B.1. If S is an exception handler step, it can only be posted if the corresponding exception is thrown. Therefore, the event "*Artifact o is Wrong When S is Posted*" can only be caused by the event "*Artifact o' is Wrong When S' is Terminated by Exception e*", where

- e is the exception handled by S;
- S' may throw exception e;
- o' is an output parameter of S' that will be passed to o (via parameter bindings). If no such parameter exists, $o' = o$.

It is possible that the step S' does not have an output parameter o' that is passed to o. o' may be an output of another step S'' that is executed before S' and be passed to the exception handler. It is tempting to look several steps back to identify S'', create an event "*Artifact o' is Wrong When S" is Completed*", and connect this event to the `OR` gate. This, however, violates an important rule in the practice of fault tree derivation: *always look at the immediate causes each time* [131]. Violating this rule could result in overlooking some critical events between event “*Artifact o’ is Wrong When S” is Completed*” and “*Artifact o is Wrong When S is Posted*”. One might argue that since we derive the fault tree automatically instead of manually, we can avoid overlooking such critical events by considering all possible situations and incorporating them into the templates. The problem is that this will greatly increase the size of the templates and make the derivation algorithm much more complicated. Therefore, we choose to apply *Template 1* and create the temporary event “*Artifact o is Wrong When S’ is Terminated by Exception e*” in this case. This event can be interpreted as: although the artifact o is not visible to the step S', it is already wrong and still alive at the point when S' is terminated by exception e. After the fault tree is derived, most of the temporary events introduced in this way can be removed by the fault tree optimization techniques discussed in Chapter 4.

Template 2

Requirement:

- The event e to be developed is *Artifact o is Wrong When S is Posted*
- S is not an exception handler step

Partial Fault Tree:

The partial fault tree for this template is shown in Figure B.2. If S is the root step, the incorrect artifact must be passed in as a parameter to the process. How the parameter went wrong is out of the scope of the process. Therefore, the event is treated as a basic event and does not need to be developed further.

If S is neither an exception handler nor the root step, it may be posted in two cases: it may be posted after its parent step is started or it may be posted after a sibling step is completed. The first case occurs if S is the first child step to be executed, e.g. S is a pre-requisite step, or S is the first sub-step of a sequential step

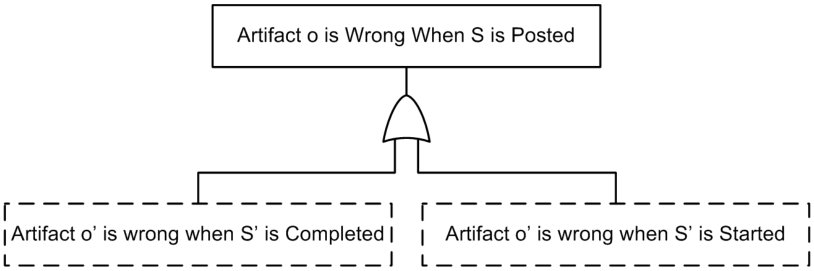

Artifact o is Wrong When S is Posted
Artifact o' is wrong when S' is Completed
Artifact o' is wrong when S' is Started

Figure B.2. Template 2

that does not have a pre-requisite step. Suppose S' is the parent step of S, S will be posted as soon as S' is started. Since S' is a non-leaf step, it is not able to change the artifact o. The artifact o must already be wrong at the point when the S' is started. In other words, event "*Artifact o is Wrong When S is Posted*" could be caused by event "*Artifact o' is Wrong When S' is Started*", where:

- S' is the parent step of S, and S is the first child to be executed, and
- o' is a parameter of S' that will be passed to o. If no such parameter exists, $o' = o$.

The second case occurs when S is not the first child step to be executed. In this case, S can only be executed after one of its siblings is completed. For example, if S is the second sub-step of a sequential step, it will be posted as soon as the first sub-step is completed. If the first step may throw an exception and the corresponding exception handler has a *continue* badge, S can also be posted after the exception handler step is completed. Note that if this exception handler does have a handler step, S can actually be posted when the first step is terminated. To simplify the template, a dummy handler step is added to an exception handler if it is not associated with a hander step. For steps in the second case, event "*Artifact o is Wrong When S is posted*" may be caused by event "*Artifact o' is Wrong When S' is Completed*", where:

- S' a sibling of S and if S' is completed, S will be executed, and

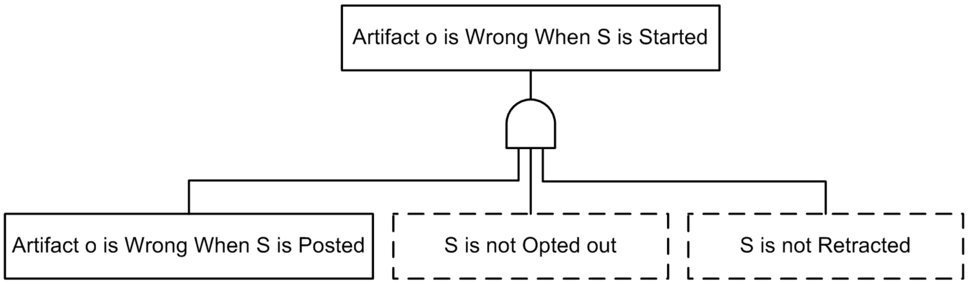

Artifact o is Wrong When S is Started
Artifact o is Wrong When S is Posted
S is not Opted out
S is not Retracted

Figure B.3. Template 3

- o' is an output parameter of S' that will be passed to o. If no such parameter exists, $o' = o$.

Template for *Artifact o is Wrong When S is Started*

Template 3

Requirement:

- The event e to be developed is *Artifact o is Wrong When S is Started*

Partial Fault Tree:

The partial fault tree for this template is shown in Figure B.3. In Little-JIL, the only step state that may immediately precedes the state `<S, Started>` is `<S, Posted>`. Since artifacts cannot be changed between these two states, event "*Artifact o is Wrong When S is Started*" can occur only if the artifact o is already wrong when the step S is posted. On the other hand, however, event "*Artifact o is Wrong When S is posted*" does not always lead to "*Artifact o is Wrong When S is Started*". The step S may be a retractable step, e.g. S is a sub-step of a parallel step. S may go from the posted state to the retracted state when another sub-step throws an exception. When this occurs, the faulty o will not be propagated to the state `<S, Started>`. The step S may also be an optional step. The agent responsible for S is allowed to opt out of the posted step S. In this case, event "*Artifact o is wrong when S is*

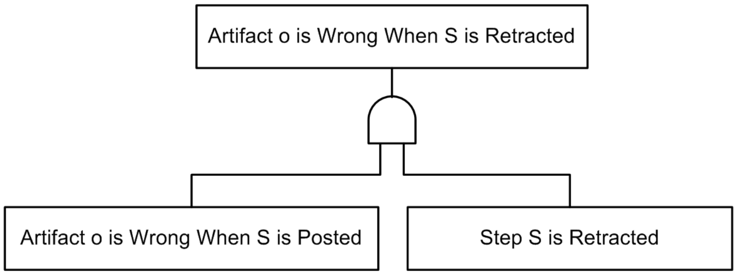

Artifact o is Wrong When S is Retracted
Artifact o is Wrong When S is Posted
Step S is Retracted

Figure B.4. Template 4

posted" will not lead to "*Artifact o is Wrong When S is Started*" as well. Therefore, to allow the faulty o to be propagated from the posted state to the started state, S should not be retracted or opted-out. Notice that the nodes for event "*Step S is not Opted out*" and "*Step S is not Retracted*" are drawn using dash lines. This means that these two events may or may not be present, depending on if the step S is optional or retractable. "*Step S is not Opted out*" is present iff S is an optional step, and "*Step S is not Retracted*" is present iff S is a retractable step.

Templates for *Artifact o is Wrong When S is Retracted*

Template 4

Requirement:

- The event e to be developed is *Artifact o is Wrong When S is Retracted*

Partial Fault Tree:

The immediate predecessor of the step state `<S, Retracted>` is `<S, Posted>`. Since retracted step can never make changes to its parameters, the artifact o must have already been wrong when S is posted. Moreover, the control flow only goes from `<S, Posted>` to `<S, Retracted>` if step S is retracted. This results in the template shown in Figure B.4.

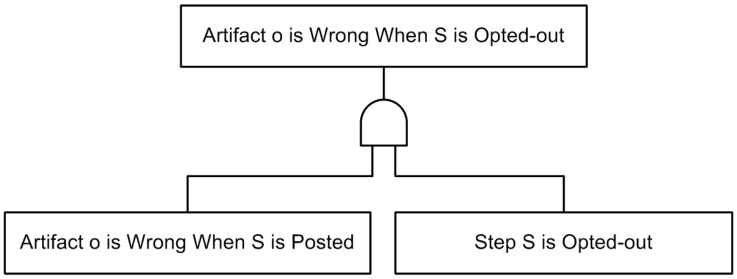

Artifact o is Wrong When S is Opted-out
Artifact o is Wrong When S is Posted
Step S is Opted-out

Figure B.5. Template 5

Templates for ***Artifact o is Wrong When S is Opted-out***

Template 5

Requirement:

- The event e to be developed is *Artifact o is Wrong When S is Opted-out*

Partial Fault Tree:

The immediate predecessor of the step state `<S, Opted-out>` is also `<S, Posted>`. Similar to the retracted step, opted-out step does not make changes to its parameters. Therefore, the artifact o must already be wrong when S is posted. Moreover, the control flow only goes from `<S, Posted>` to `<S, Opted-out>` if step S is retracted. This results in the template shown in Figure B.5.

Templates for ***Artifact o is Wrong When S is Completed***

Template 6

Requirement:

- The event e to be developed is *Artifact o is Wrong When S is Completed*, and
- S is not a leaf step

Partial Fault Tree:

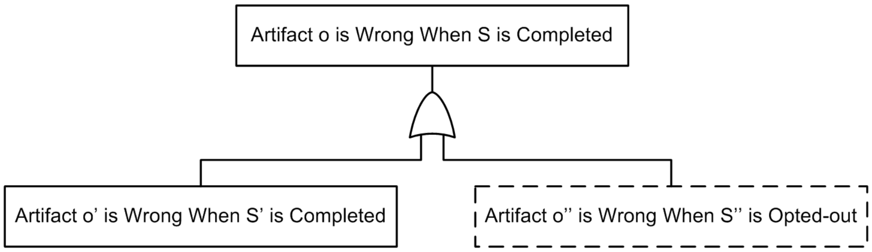

Artifact o is Wrong When S is Completed
Artifact o' is Wrong When S' is Completed
Artifact o'' is Wrong When S'' is Opted-out

Figure B.6. Template 6

If S is not a leaf step, S may be completed immediately after certain child step is completed. For example, a sequential step is completed as soon as its last sub-step is completed. Suppose this step has an exception handler with a `complete` badge, it may also be completed when the exception handler step is completed. In this case, event "*Artifact o is Wrong When S is Completed*" is caused by event "*Artifact o' is Wrong When S' is Completed*", where:

- S' is a child step of S and if it is completed, S will be completed, and
- o' is a parameter of S' that will be passed to o. If no such parameter exists, $o' = o$.

There may be more than one child steps that are qualified. For each such child step, an event "*Artifact o' is Wrong When S' is Completed*" needs to be created and connected to the `OR` gate.

If step S has optional sub-steps, it could be completed after one of its optional sub-step is opted-out. In this case, event "*Artifact o is Wrong When S is Completed* is caused by event "*Artifact o" is Wrong When S" is Opted-out*", where:

- S'' is an optional sub-step of S and if it is completed, S will be completed, and
- o'' is a parameter of S'' that will be passed to o. If no such parameter exists, $o'' = o$.

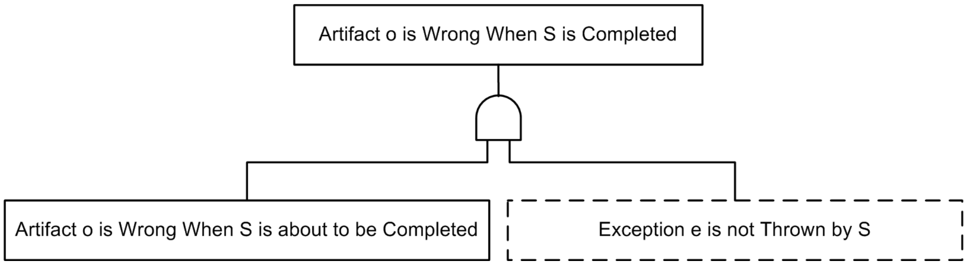

Artifact o is Wrong When S is Completed
Artifact o is Wrong When S is about to be Completed
Exception e is not Thrown by S

Figure B.7. Template 7

Figure B.6 shows the template that considers both cases.

Template 7

Requirement:

- The event e to be developed is *Artifact o is Wrong When S is Completed*, and
- S is a leaf step

Partial Fault Tree:

If S is a leaf step, event "*Artifact o is wrong when S is completed*" occurs if the artifact o is already wrong when step S is about to be completed, and no exception is thrown by S. If any exception is thrown, the faulty artifact o will not be propagated to the state `<S, Completed>`. Figure B.7 shows the template.

S may throw more than one exception. For each exception e that could be thrown by S, a new event "*Exception e is not thrown by S*" is created. If S does not throw any exception, event "*Exception e is not thrown by S*" will not be created.

Templates for *Artifact o is Wrong When S is Terminated by Exception e*

Template 8

Requirement:

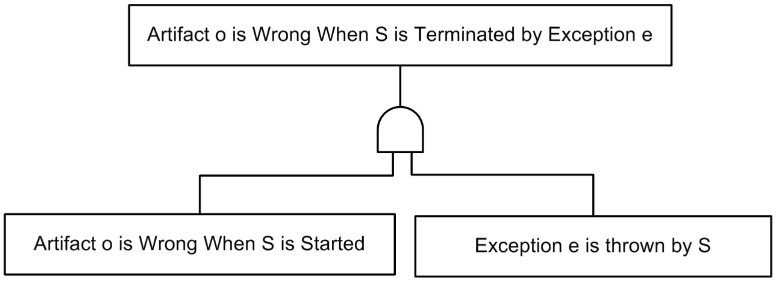

Artifact o is Wrong When S is Terminated by Exception e
Artifact o is Wrong When S is Started
Exception e is thrown by S

Figure B.8. Template 8

- The event e to be developed is *Artifact o is Wrong When S is Terminated by Exception e*, and
- S is a leaf step

Partial Fault Tree:

Figure B.8 shows the partial fault tree for this template. According to the copy-in-copy-out semantics of parameter passing in Little-JIL, if a leaf step is terminated, the parameter that it changes will not be passed out. For the case that o is an `OUT` parameter of S, o can be wrong only if S changes o incorrectly. This error, however, well not be propagated out of S if S is terminated. Therefore "*Artifact o is wrong when S is terminated by exception e*" should never occur. To handle this, we mark the event as infeasible. The infeasible events will be pruned from the fault tree after the whole fault tree is derived.

For the other cases (o is an `IN/INOUT` parameter of S, or o is not a parameter of S), Event "*Artifact o is wrong when S is terminated by exception e*" can occur only when the artifact o is already wrong when S is started and the exception e is thrown by S.

Template 9

Requirement:

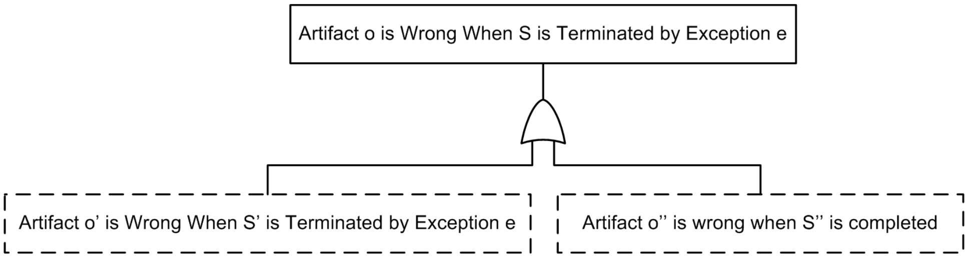

Artifact o is Wrong When S is Terminated by Exception e
Artifact o' is Wrong When S' is Terminated by Exception e
Artifact o'' is wrong when S'' is completed

Figure B.9. Template 9

- The event e to be developed is *Artifact o is Wrong When S is Terminated by Exception* e, and
- S is a not leaf step

Partial Fault Tree:

Figure B.9 shows the partial fault tree for this template. If S is not a leaf step, it is usually terminated by exception e when the exception s is propagated from a child step. For example, a handler step throws exception e, or a sub-step throws e that does not have a corresponding exception hander. For this case, event "*Artifact o is wrong when S is terminated by exception e*" may be caused by event "*Artifact o is wrong when S' is terminated by exception e*" is created, where:

- S' is a child step of S and when it throws exception e, S is terminated;
- o' is a parameter of S' that will be passed to o. If no such parameter exists, $o' = o$.

Suppose S has an exception handler with `rethrow` badge to handle e, S could be terminated immediately after the exception handler step is completed. For this case, event "*Artifact o is wrong when S is terminated by exception e*" may be caused by event "*Artifact o'' is wrong when S'' completed*" is created, where:

- S'' is an exception handler step associated with `rethrow` exception handler that handles exception e, and

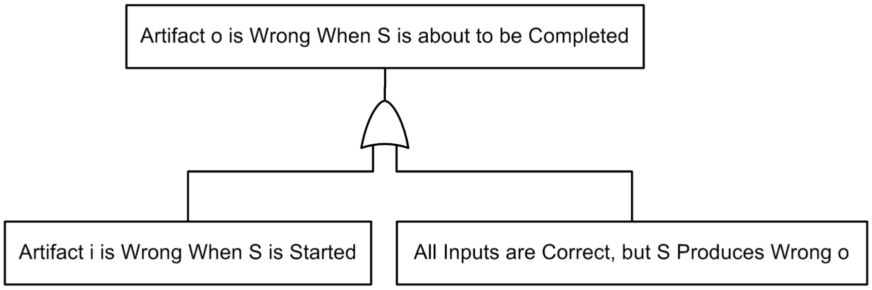

Artifact o is Wrong When S is about to be Completed
Artifact i is Wrong When S is Started
All Inputs are Correct, but S Produces Wrong o

Figure B.10. Template 10

- o'' is a parameter of S'' that will be passed to o. If no such parameter exists, $o'' = o$.

Templates for *Artifact o is Wrong When S is about to be Completed*

Template 10

Requirement:

- The event e to be developed is *Artifact o is Wrong When S is about to be Completed*, and
- o is an output parameter of S

Partial Fault Tree:

Note that step S in event "*Artifact o is wrong when S is about to be Completed*" should always be a leaf step because this event can only be introduced by **Template 7**, which requires step S to be a leaf step.

If o is an output parameter of S, S may change o. We conservatively assume that o depends on all inputs of S. Therefore, incorrect o may be caused by incorrect input, or incorrect execution of S, as shown in Figure B.10. In event "*Artifact i is Wrong When S is Started*", S is the same step as the one in "*Artifact o is Wrong When S*

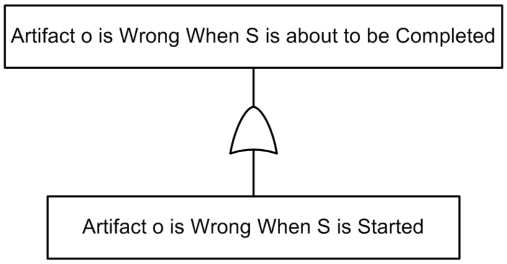

Artifact o is Wrong When S is about to be Completed
Artifact o is Wrong When S is Started

Figure B.11. Template 11

is about to be Completed", and i is an input of S. For each input i of S, one such event needs to be created. In event "*All Inputs are Correct, but S Produces Wrong o*", artifact o and step S are the same step as the ones in "*Artifact o is Wrong When S is about to be Completed*".

Template 11

Requirement:

- The event e to be developed is *Artifact o is Wrong When S is about to be Completed*, and
- o is not an output parameter of S

Partial Fault Tree:

Since o is not an output parameter of S, S cannot change o. Therefore, o is wrong simply because it is already wrong before S is started, as shown in Figure B.11. The artifact o and step S in the new event "*Artifact o is Wrong When S is Started*" are the same as the ones in event "*Artifact o is Wrong When S is about to be Completed*".

Templates for *Exception e is not Thrown by S*

Template 12

Requirement:

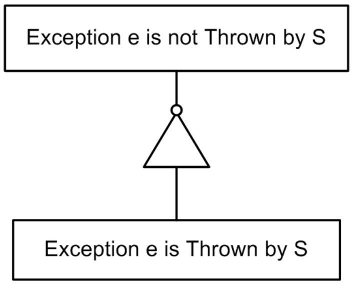


Figure B.12. Template 12

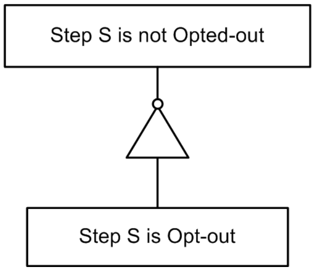


Figure B.13. Template 13

- The event e to be developed is *Exception e is not Thrown by S*

Partial Fault Tree:

As shown in Figure B.12, this template is straight-forward. "*Exception e is not Thrown by S*" simply means `NOT` "*Exception e is Thrown by S*".

Templates for ***Step S is not Opted-out***

Template 13

Requirement:

- The event e to be developed is *Step S is not Opted-out*

Partial Fault Tree:

As shown in Figure B.13, "*Step S is not Opted-out*" simply means `NOT` "*Step S is Opted-out*".

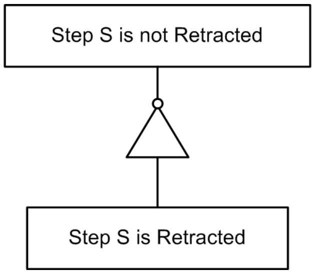


Figure B.14. Template 14

Templates for ***Step S is not retracted***

Template 14

Requirement:

- The event e to be developed is *Step S is not Opted-out*

Partial Fault Tree:

As shown in Figure B.14, "*Step S is not retracted*" simply means `NOT` "*Step S is retracted*".

APPENDIX C

UNOPTIMIZED BIR PROGRAM FOR SIMPLIFIED BLOOD TRANSFUSION PROCESS

```
process Process_Blood_Transfusion_Process ()
  // Exception variable for PatientBloodTypeUnavailable
  b_PatientBloodTypeUnavailable := false;
  // Exception variable for WrongPatient
  b_WrongPatient := false;
  // Exception variable for BloodExpired
  b_BloodExpired := false;
  // Exception variable for WrongBlood
  b_WrongBlood   := false;

  // Done variable for 'Check Expiration Date'
  b_CheckExpirationDateDone := false;
  // Done variable for Check 'Product Info Match Patient Info'
  b_CheckProductInfoMatchPatientInfo := false;

  main thread T_Blood_Transfusion_Process
    loc Blood_Transfusion_Process_POSTED:
      when true do { }
        goto Blood_Transfusion_Process_STARTED;
    loc Blood_Transfusion_Process_STARTED:
      when true do { }
        goto Obtain_Patients_Blood_Type_POSTED;
    loc Obtain_Patients_Blood_Type_POSTED:
      when true do { }
        goto Obtain_Patients_Blood_Type_STARTED;
    loc Obtain_Patients_Blood_Type_STARTED:
      when true do { }
        goto Check_Lab_for_Patients_Blood_Type_POSTED;
    loc Check_Lab_for_Patients_Blood_Type_POSTED:
      when true do { }
        goto Check_Lab_for_Patients_Blood_Type_STARTED;
    loc Check_Lab_for_Patients_Blood_Type_STARTED:
      when true do { }
        goto Check_Lab_for_Patients_Blood_Type_COMPLETED;
      when true do { b_PatientBloodTypeUnavailable := true; }
```

```
        goto Check_Lab_for_Patients_Blood_Type_TERMINATED;
    loc Check_Lab_for_Patients_Blood_Type_COMPLETED:
      when true do { }
        goto Obtain_Patients_Blood_Type_COMPLETED;
    loc Check_Lab_for_Patients_Blood_Type_TERMINATED:
      when true do { }
        goto Obtain_Patients_Blood_Type_control_continue:
    loc Obtain_Patients_Blood_Type_control_continue:
      when ( b_PatientBloodTypeUnavailable == true )
        do { b_PatientBloodTypeUnavailable := false; }
        goto Test_Patients_Blood_Type_POSTED;
    loc Test_Patients_Blood_Type_POSTED:
      when true do { }
        goto Test_Patients_Blood_Type_STARTED;
    loc Test_Patients_Blood_Type_STARTED:
      when true do { }
        goto Test_Patients_Blood_Type_COMPLETED;
    loc Test_Patients_Blood_Type_COMPLETED:
      when true do { }
        goto Obtain_Patients_Blood_Type_COMPLETED;
    loc Obtain_Patients_Blood_Type_COMPLETED:
      when true do { }
        goto Pick_up_Blood_from_Blood_Bank_POSTED;
    loc Pick_up_Blood_from_Blood_Bank_POSTED:
      when true do { }
        goto Pick_up_Blood_from_Blood_Bank_STARTED;
    loc Pick_up_Blood_from_Blood_Bank_STARTED:
      when true do { }
        goto Pick_up_Blood_from_Blood_Bank_COMPLETED;
    loc Pick_up_Blood_from_Blood_Bank_COMPLETED:
      when true do { }
        goto Perform_Bedside_Check_POSTED;
    loc Perform_Bedside_Check_POSTED:
      when true do { }
        goto Perform_Bedside_Check_STARTED;
    loc Perform_Bedside_Check_STARTED:
      when true do { }
        goto Identify_Patient_POSTED;
    loc Identify_Patient_POSTED:
      when true do { }
        goto Identify_Patient_STARTED:
    loc Identify_Patient_STARTED:
      when true do { }
        goto Identify_Patient_COMPLETED;
      when true do { b_WrongPatient := true; }
        goto Identify_Patien_TERMINATED;
    loc Identify_Patient_COMPLETED:
     when true do { }
      goto Check_Blood_Product_POSTED;
    loc Identify_Patien_TERMINATED:
     when true do { }
      goto Perform_Bedside_Check_control_rethrow:
    loc Perform_Bedside_Check_control_rethrow:
     when ( b_WrongPatient == true ) do ()
      goto Perform_Bedside_Check_TERMINATED;
    loc Check_Blood_Product_POSTED:
     when true do { }
      goto Check_Blood_Product_STARTED;
    loc Check_Blood_Product_STARTED:
     when true do { }
      goto Check_Blood_Product_fork_subs:
    loc Check_Blood_Product_fork_subs:
     when true do {
       start(T_Check_Expiration_Date);
       start(T_Check_Product_Info_Match_Patient_Info);
     } goto Check_Blood_Product_join_Sub1;
    loc Check_Blood_Product_join_Sub1:
     when true do { join(T_Check_Expiration_Date); }
      goto Check_Blood_Product_terminate_test_Sub1;
    loc Check_Blood_Product_terminate_test_Sub1:
     when threadTerminated(T_Check_Expiration_Date) do { }
      goto Check_Blood_Product_join_Sub2;
    loc Check_Blood_Product_join_Sub2:
     when true do {
        join(T_Check_Product_Info_Match_Patient_Info);
     } goto Check_Blood_Product_terminate_test_Sub2;
    loc Check_Blood_Product_terminate_test_Sub2:
     when
      threadTerminated(
       T_Check_Product_Info_Match_Patient_Info
      )
     do { } goto Check_Blood_Product_coontrol_rethrow;
    loc Check_Blood_Product_coontrol_rethrow:
     when ( b_BloodExpired != true &&
            b_WrongBlood   != true )
      do { }
      goto Check_Blood_Product_COMPLETED;
     when ( b_BloodExpired == true ||
            b_WrongBlood   == true)
      do { }
      goto Perform_Bedside_Check_control_rethrow2;
    loc Check_Blood_Product_COMPLETED:
      when true do { }
        goto Perform_Bedside_Check_COMPLETED;
    loc Perform_Bedside_Check_COMPLETED:
      when true do { }
        goto Infuse_Blood_POSTED;
    loc Perform_Bedside_Check_control_rethrow2:
      when ( b_BloodExpired == true ||
             b_WrongBlood   == true )
        do { }
        goto Perform_Bedside_Check_TERMINATED;
    loc Perform_Bedside_Check_TERMINATED:
      when true do { }
        goto Blood_Transfusion_Process_control_rethrow:
    loc Blood_Transfusion_Process_control_rethrow:
      when ( b_WrongPatient == true ||
             b_BloodExpired == true ||
             b_WrongBlood   == true )
        do { }
        goto Blood_Transfusion_Process_TERMINATED
    loc Infuse_Blood_POSTED:
      when true do { } goto Infuse_Blood_STARTED;
    loc Infuse_Blood_STARTED:
      when true do { } goto Infuse_Blood_COMPLETED;
    loc Infuse_Blood_COMPLETED:
      when true do { }
        goto Blood_Transfusion_Process_COMPLETED;
    loc Blood_Transfusion_Process_COMPLETED:
      when true do { }
        goto T_Blood_Transfusion_Process_exit;
    loc Blood_Transfusion_Process_TERMINATED:
      when true do { }
        goto T_Blood_Transfusion_Process_exit;
    loc T_Blood_Transfusion_Process_exit:
  end T_Blood_Transfusion_Process;

  thread T_Check_Expiration_Date
    loc Check_Expiration_Date_test_retract:
      when ( b_WrongBlood == true ) do { }
        goto Check_Expiration_Date_RETRACTED;
      when ( b_WrongBlood != true ) do { }
        goto Check_Expiration_Date_POSTED:
    loc Check_Expiration_Date_POSTED:
      when true do { }
        goto Check_Expiration_Date_STARTED;
    loc Check_Expiration_Date_STARTED:
      when true do { }
    goto Check_Expiration_Date_COMPLETED;
   when true do { b_BloodExpired := true; }
    goto Check_Expiration_Date_TERMINATED;
  loc Check_Expiration_Date_COMPLETED:
   when true do { } goto T_Check_Expiration_exit;
  loc Check_Expiration_Date_TERMINATED:
   when true do { } goto T_Check_Expiration_exit;
  loc Check_Expiration_Date_RETRACTED:
   when true do { } goto T_Check_Expiration_exit;
  loc T_Check_Expiration_exit:
 end T_Check_Expiration_Date;

 thread T_Check_Product_Info_Match_Patient_Info
  loc Check_Product_Info_Match_Patient_test_retract:
   when ( b_BloodExpired == true ) do { }
    goto Check_Product_Info_Match_Patient_RETRACTED;
   when ( b_BloodExpired != true ) do { }
    goto Check_Product_Info_Match_Patient_POSTED:
  loc Check_Product_Info_Match_Patient_POSTED:
   when true do { }
    goto Check_Product_Info_Match_Patient_STARTED;
  loc Check_Product_Info_Match_Patient_STARTED:
   when true do { }
    goto Check_Product_Info_Match_Patient_COMPLETED;
   when true do { b_WrongBlood := true; }
    goto Check_Product_Info_Match_Patient_TERMINATED;
  loc Check_Product_Info_Match_Patient_COMPLETED:
   when true do { }
    goto T_Check_Product_Info_Match_Patient_Info_exit;
  loc Check_Product_Info_Match_Patient_TERMINATED:
   when true do { }
    goto T_Check_Product_Info_Match_Patient_Info_exit;
  loc Check_Product_Info_Match_Patient_RETRACTED:
   when true do { }
    goto T_Check_Product_Info_Match_Patient_Info_exit;
  loc T_Check_Product_Info_Match_Patient_Info_exit:
 end T_Check_Product_Info_Match_Patient_Info;

predicates
 IdentifyPatient =
   T_Blood_Transfusion_Process@Identify_Patient_COMPLETED;
 InfuseBlood =
   T_Blood_Transfusion_Process@Infuse_Blood_STARTED;
end Process_Blood_Transfusion_Process;
```

APPENDIX D

IN-PATIENT BLOOD TRANSFUSION PROCESS

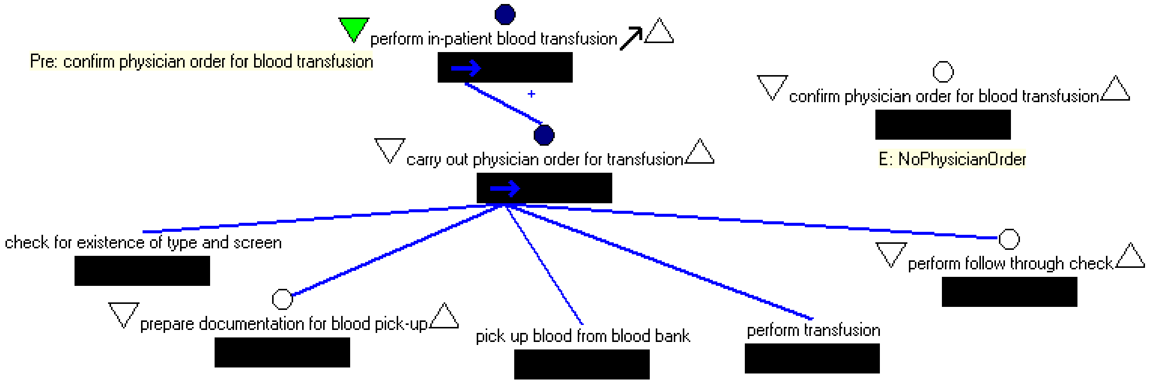

perform in-patient blood transfusion
Pre: confirm physician order for blood transfusion
+
confirm physician order for blood transfusion
E: NoPhysicianOrder
carry out physician order for transfusion
check for existence of type and screen
perform follow through check
prepare documentation for blood pick-up
pick up blood from blood bank
perform transfusion

Figure D.1. Diagram “perform in-patient blood transfusion”

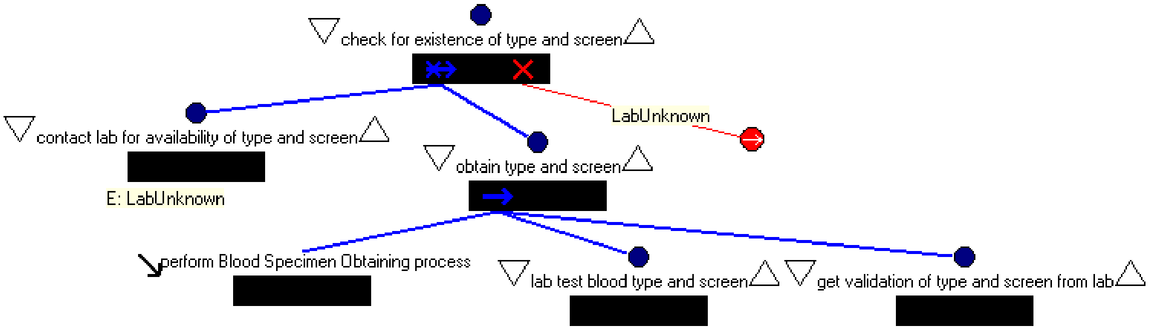

check for existence of type and screen
LabUnknown
contact lab for availability of type and screen
E: LabUnknown
obtain type and screen
perform Blood Specimen Obtaining process
lab test blood type and screen
get validation of type and screen from lab

Figure D.2. Diagram “check for existence of type and screen”

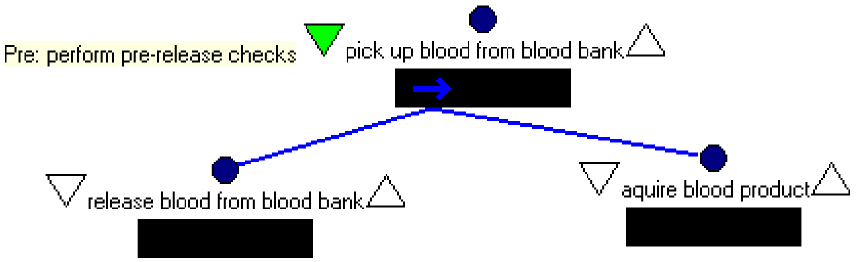

Pre: perform pre-release checks
pick up blood from blood bank
release blood from blood bank
aquire blood product

Figure D.3. Diagram “pick up blood from blood bank”

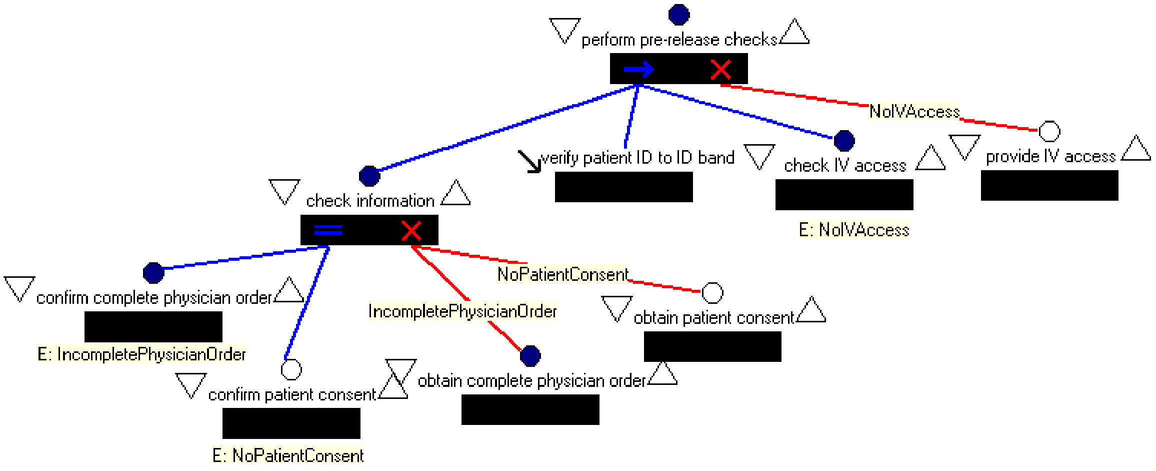


Figure D.4. Diagram "perform pre-release checks"

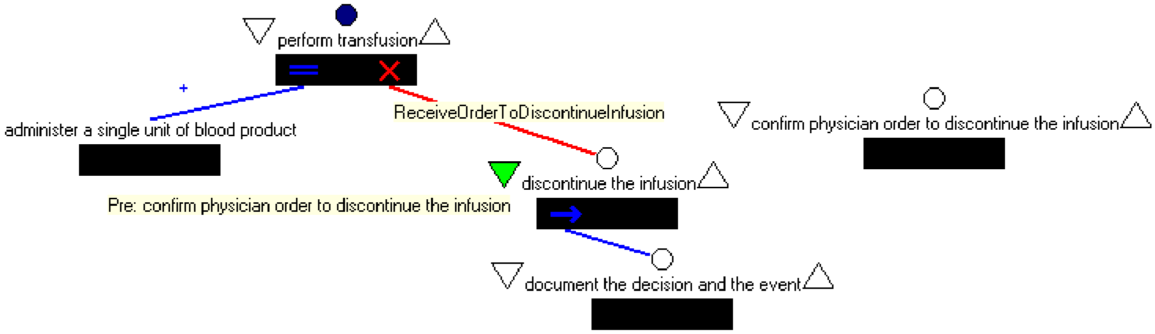

perform transfusion
+
administer a single unit of blood product
ReceiveOrderToDiscontinueInfusion
confirm physician order to discontinue the infusion
discontinue the infusion
Pre: confirm physician order to discontinue the infusion
document the decision and the event

Figure D.5. Diagram "perform transfusion"

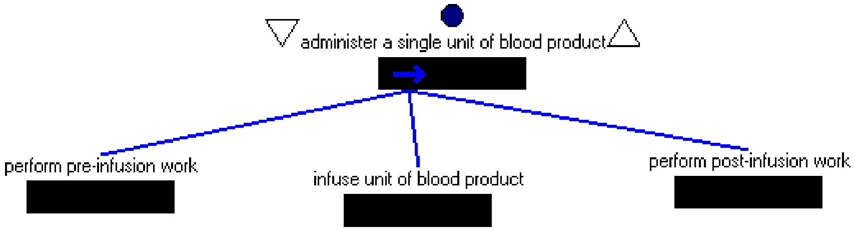

administer a single unit of blood product
perform pre-infusion work
infuse unit of blood product
perform post-infusion work

Figure D.6. Diagram "administer a single unit of blood product"

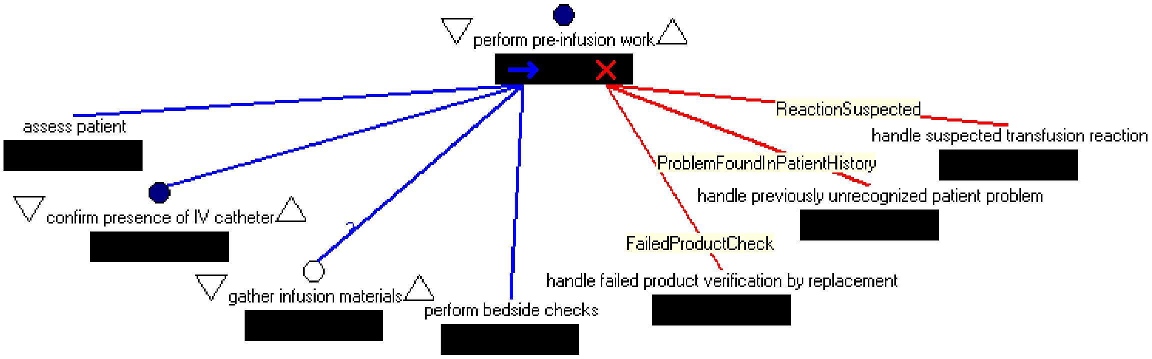

perform pre-infusion work
ReactionSuspected
handle suspected transfusion reaction
assess patient
ProblemFoundInPatientHistory
handle previously unrecognized patient problem
confirm presence of IV catheter
FailedProductCheck
?
handle failed product verification by replacement
gather infusion materials
perform bedside checks

Figure D.7. Diagram "perform pre-infusion work"

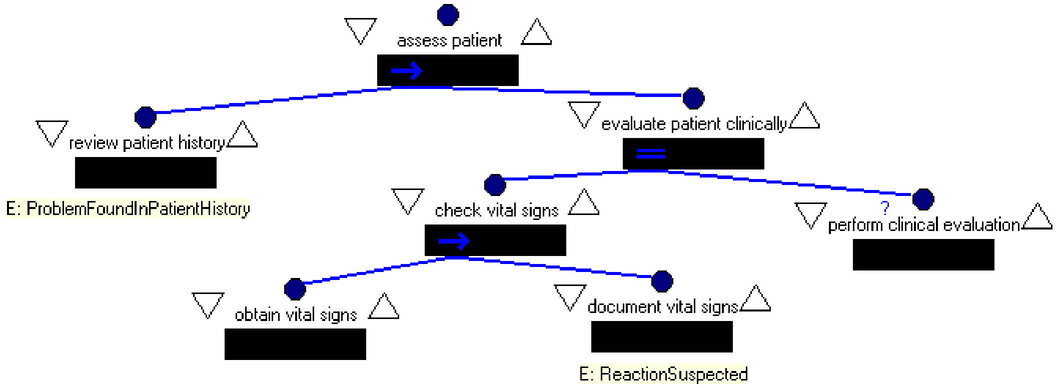


Figure D.8. Diagram "assess patient"

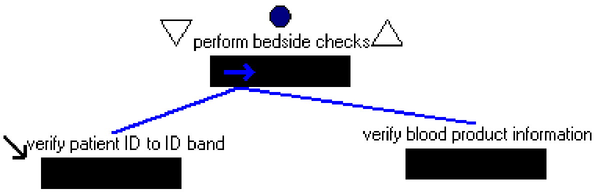

perform bedside checks
verify patient ID to ID band
verify blood product information

Figure D.9. Diagram "perform bedside checks"

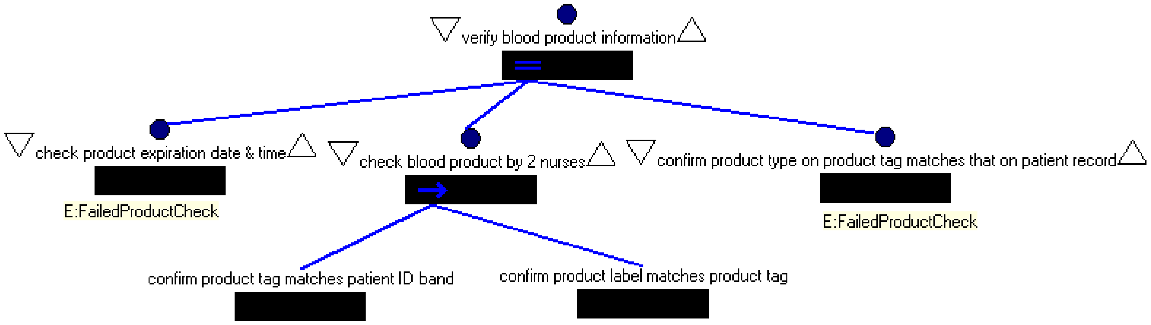

verify blood product information
check product expiration date & time
E:FailedProductCheck
check blood product by 2 nurses
confirm product type on product tag matches that on patient record
E:FailedProductCheck
confirm product tag matches patient ID band
confirm product label matches product tag

Figure D.10. Diagram “verify blood product information”

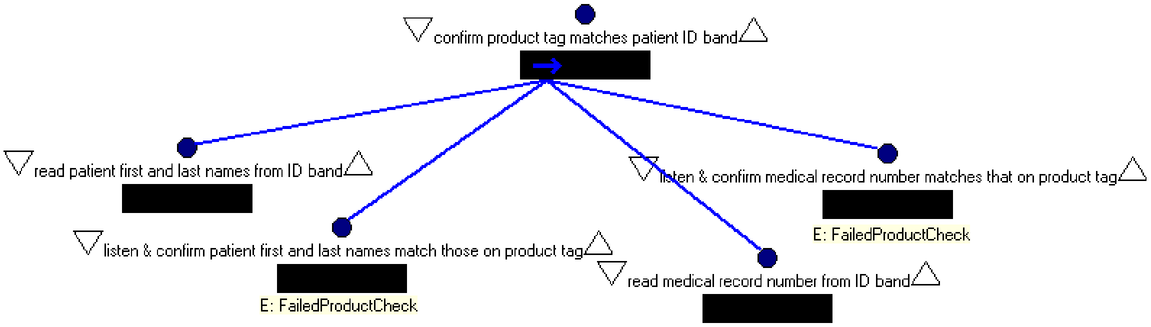

confirm product tag matches patient ID band
read patient first and last names from ID band
listen & confirm medical record number matches that on product tag
E: FailedProductCheck
listen & confirm patient first and last names match those on product tag
E: FailedProductCheck
read medical record number from ID band

Figure D.11. Diagram “confirm product tag matches patient ID band”

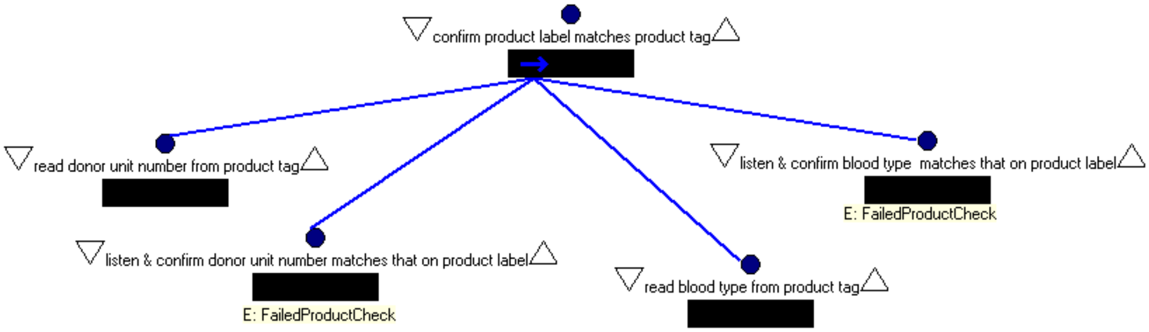

confirm product label matches product tag
read donor unit number from product tag
listen & confirm blood type matches that on product label
E: FailedProductCheck
listen & confirm donor unit number matches that on product label
E: FailedProductCheck
read blood type from product tag

Figure D.12. Diagram “confirm product label matches product tag”

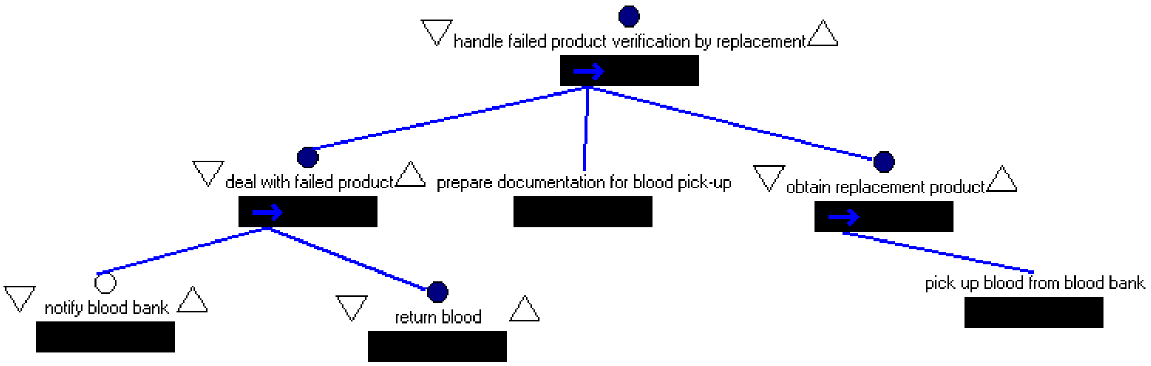

handle failed product verification by replacement
deal with failed product
prepare documentation for blood pick-up
obtain replacement product
notify blood bank
return blood
pick up blood from blood bank

Figure D.13. Diagram "handle failed product verification by replacement"

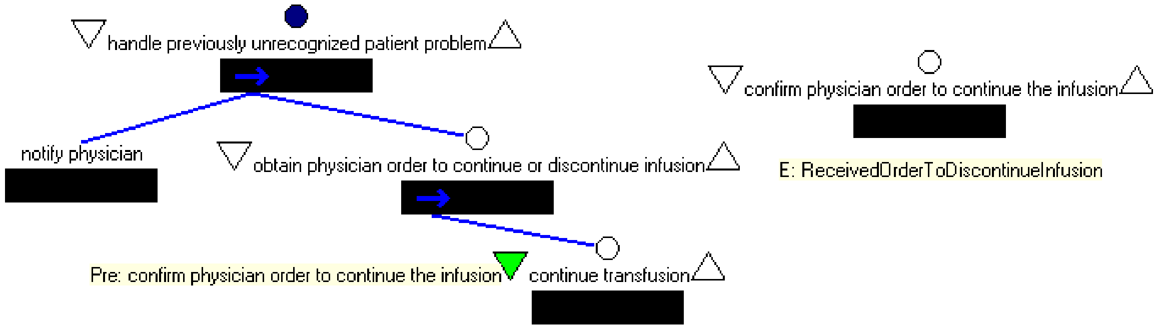

handle previously unrecognized patient problem
confirm physician order to continue the infusion
notify physician
obtain physician order to continue or discontinue infusion
E: ReceivedOrderToDiscontinueInfusion
Pre: confirm physician order to continue the infusion
continue transfusion

Figure D.14. Diagram "handle previously unrecognized patient problem"

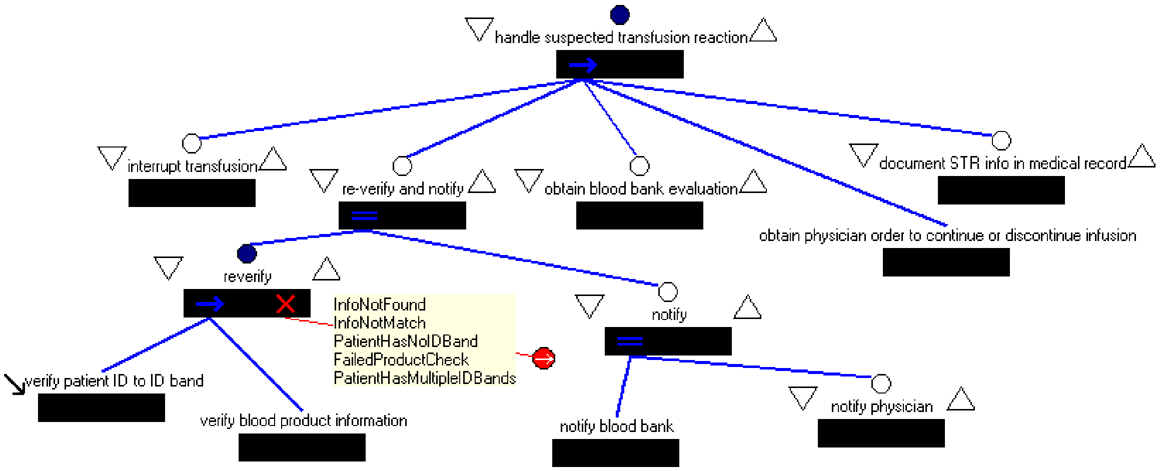

handle suspected transfusion reaction
interrupt transfusion
re-verify and notify
obtain blood bank evaluation
document STR info in medical record
obtain physician order to continue or discontinue infusion
reverify
notify
InfoNotFound
InfoNotMatch
PatientHasNoIDBand
FailedProductCheck
PatientHasMultipleIDBands
verify patient ID to ID band
verify blood product information
notify blood bank
notify physician

Figure D.15. Diagram "handle suspected transfusion reaction"

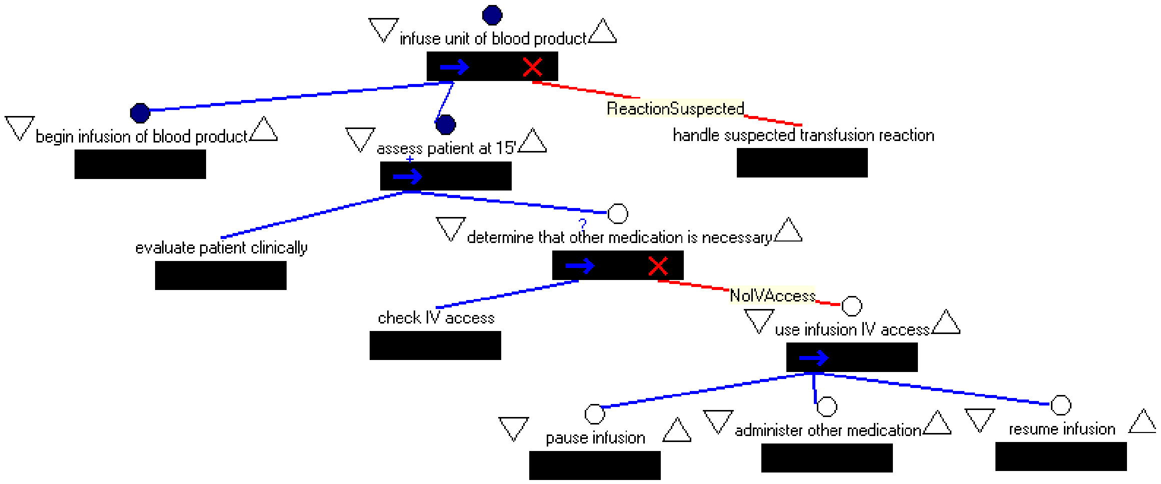

infuse unit of blood product
begin infusion of blood product
assess patient at 15'
ReactionSuspected
handle suspected transfusion reaction
evaluate patient clinically
?
determine that other medication is necessary
check IV access
NoIVAccess
use infusion IV access
pause infusion
administer other medication
resume infusion

Figure D.16. Diagram "infuse unit of blood product"

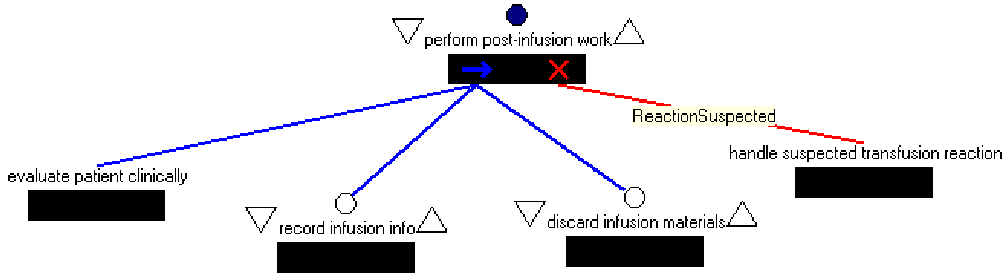

perform post-infusion work
ReactionSuspected
handle suspected transfusion reaction
evaluate patient clinically
record infusion info
discard infusion materials

Figure D.17. Diagram "perform post-infusion work"

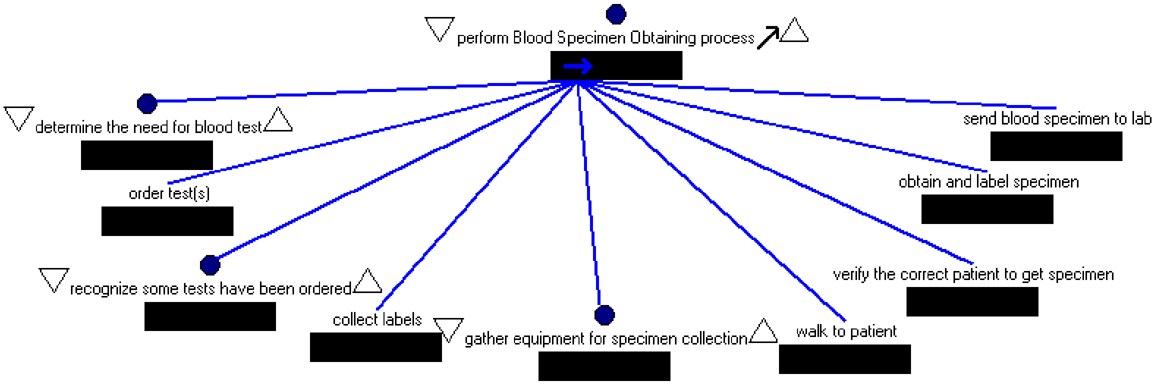

perform Blood Specimen Obtaining process
determine the need for blood test
send blood specimen to lab
order test(s)
obtain and label specimen
recognize some tests have been ordered
verify the correct patient to get specimen
collect labels
gather equipment for specimen collection
walk to patient

Figure D.18. Diagram "perform Blood Specimen Obtaining process"

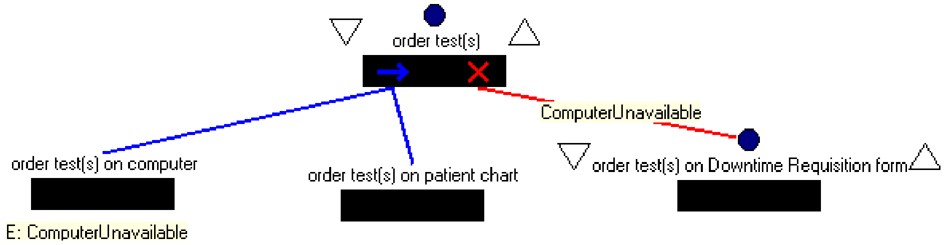

order test(s)
ComputerUnavailable
order test(s) on computer
order test(s) on patient chart
order test(s) on Downtime Requisition form
E: ComputerUnavailable

Figure D.19. Diagram “order test(s)”

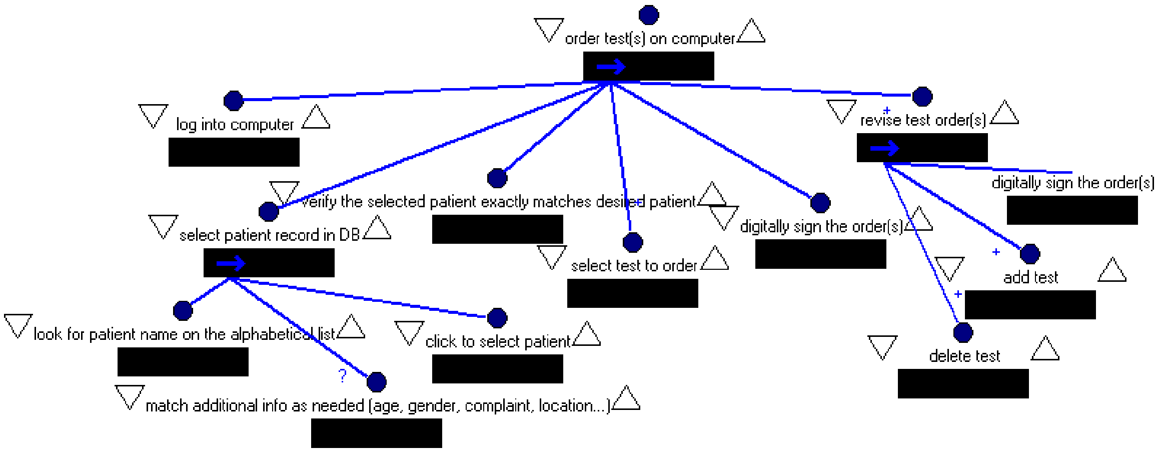

order test(s) on computer
log into computer
revise test order(s)
+
digitally sign the order(s)
verify the selected patient exactly matches desired patient
select patient record in DB
digitally sign the order(s)
select test to order
+
add test
+
delete test
look for patient name on the alphabetical list
click to select patient
?
match additional info as needed (age, gender, complaint, location...)

Figure D.20. Diagram “order test(s) on computer”

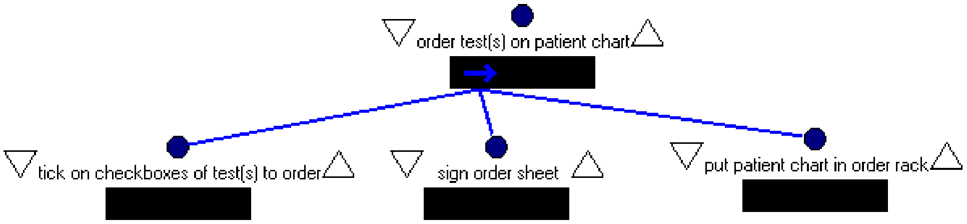

order test(s) on patient chart
tick on checkboxes of test(s) to order
sign order sheet
put patient chart in order rack

Figure D.21. Diagram “order test(s) on patient chart”

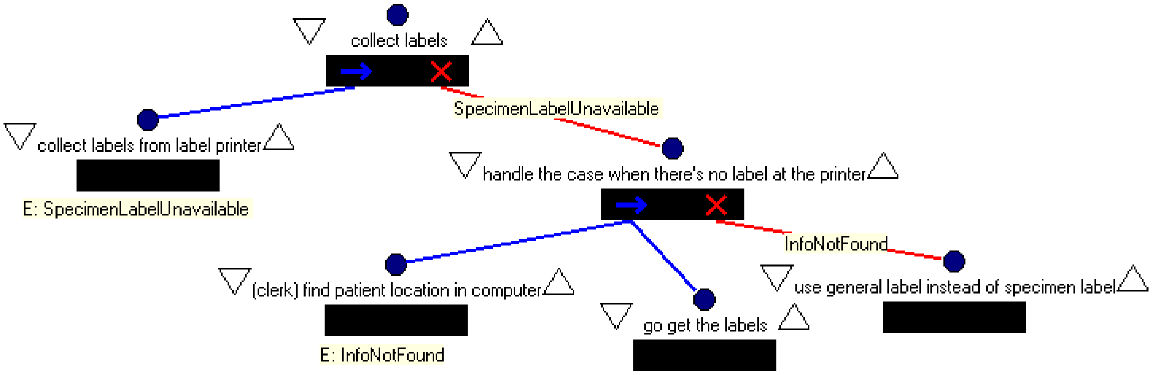

collect labels
SpecimenLabelUnavailable
collect labels from label printer
E: SpecimenLabelUnavailable
handle the case when there's no label at the printer
InfoNotFound
(clerk) find patient location in computer
E: InfoNotFound
go get the labels
use general label instead of specimen label

Figure D.22. Diagram “collect labels”

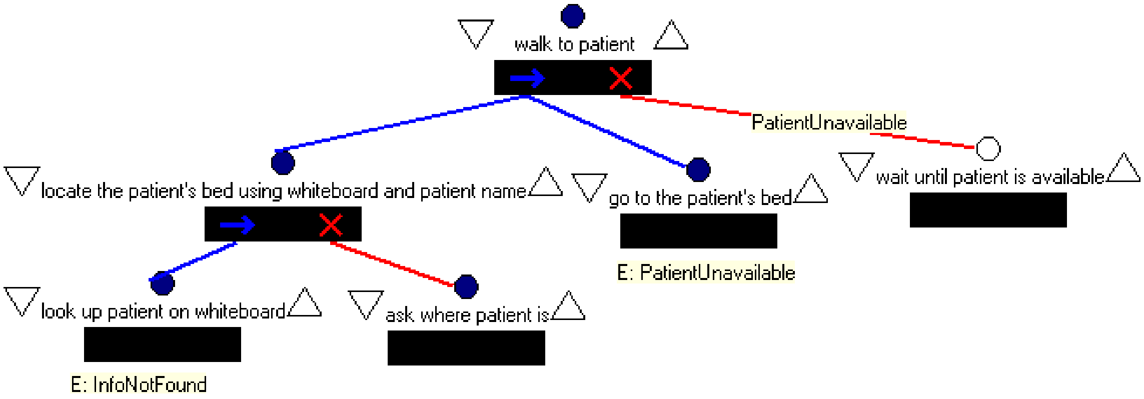

walk to patient
PatientUnavailable
locate the patient's bed using whiteboard and patient name
go to the patient's bed
E: PatientUnavailable
wait until patient is available
look up patient on whiteboard
E: InfoNotFound
ask where patient is

Figure D.23. Diagram “walk to patient”

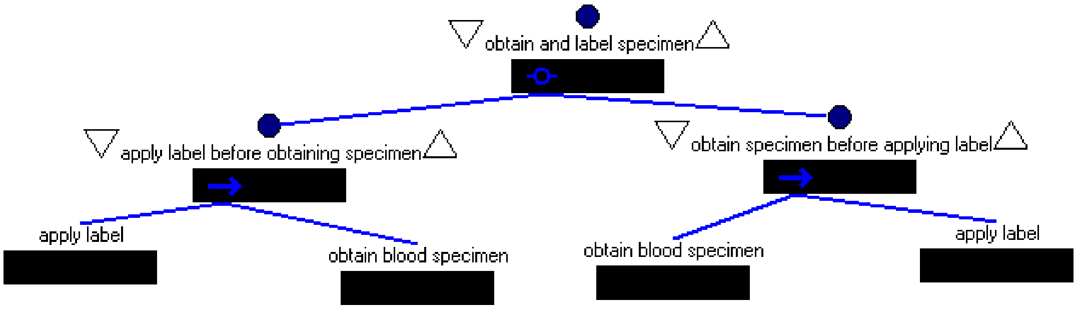

obtain and label specimen
apply label before obtaining specimen
obtain specimen before applying label
apply label
obtain blood specimen
obtain blood specimen
apply label

Figure D.24. Diagram “obtain and label specimen”

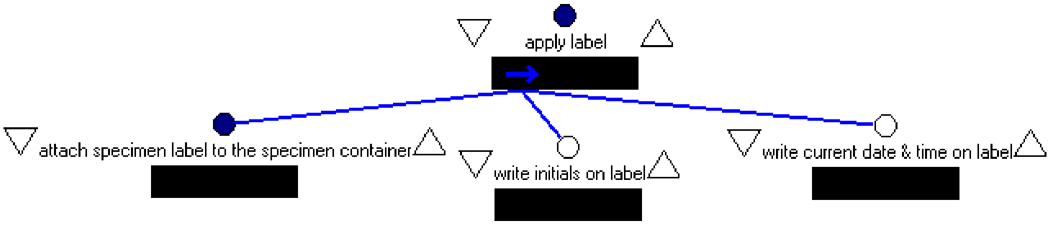

apply label
attach specimen label to the specimen container
write initials on label
write current date & time on label

Figure D.25. Diagram "apply label"

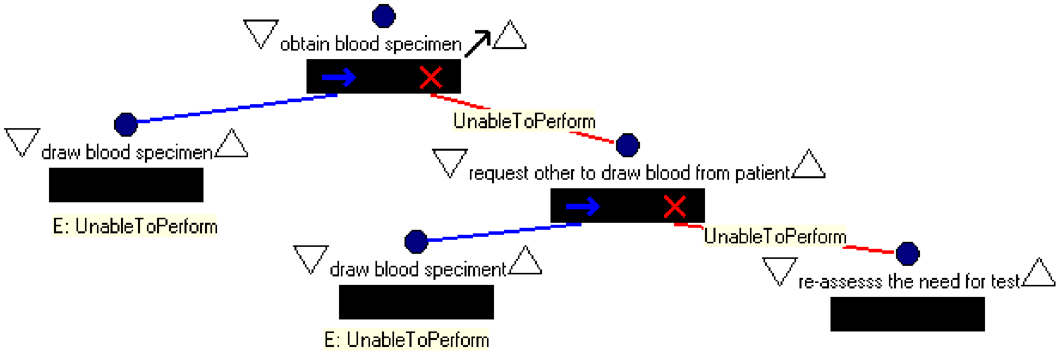


Figure D.26. Diagram "obtain blood specimen"

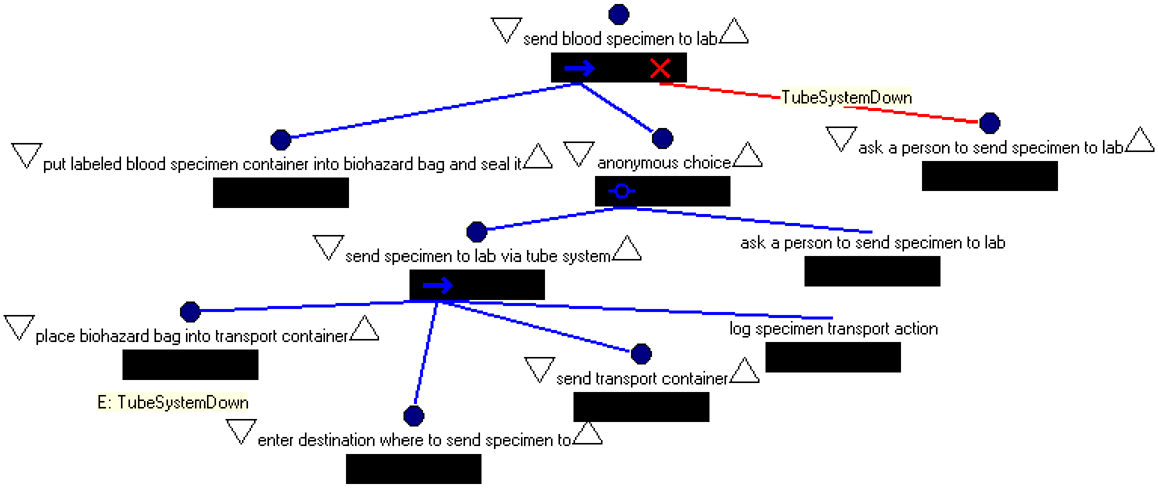

send blood specimen to lab
TubeSystemDown
put labeled blood specimen container into biohazard bag and seal it
anonymous choice
ask a person to send specimen to lab
send specimen to lab via tube system
ask a person to send specimen to lab
place biohazard bag into transport container
log specimen transport action
E: TubeSystemDown
send transport container
enter destination where to send specimen to

Figure D.27. Diagram "send blood specimen to lab"

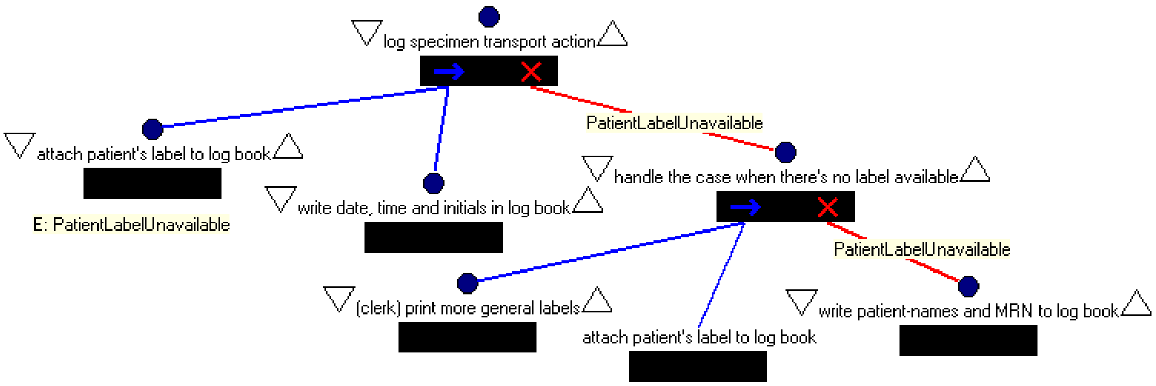

log specimen transport action
PatientLabelUnavailable
attach patient's label to log book
E: PatientLabelUnavailable
write date, time and initials in log book
handle the case when there's no label available
PatientLabelUnavailable
(clerk) print more general labels
attach patient's label to log book
write patient-names and MRN to log book

Figure D.28. Diagram “log specimen transport action”

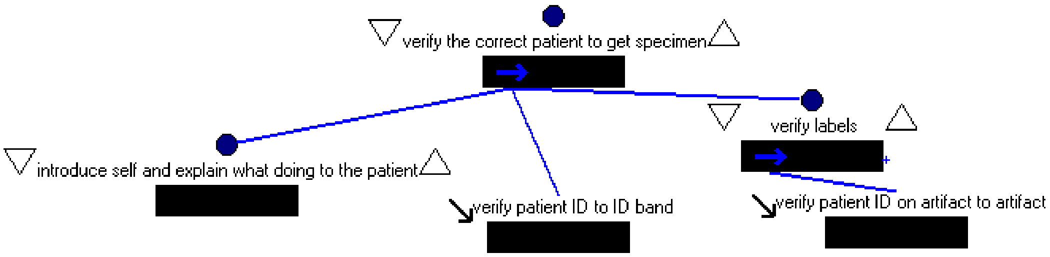

verify the correct patient to get specimen
verify labels
introduce self and explain what doing to the patient
verify patient ID to ID band
verify patient ID on artifact to artifact

Figure D.29. Diagram “verify the correct patient to get specimen”

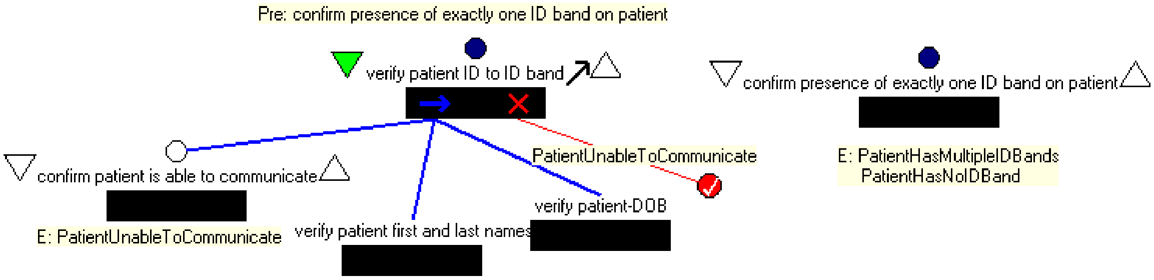

Pre: confirm presence of exactly one ID band on patient
verify patient ID to ID band
confirm presence of exactly one ID band on patient
PatientUnableToCommunicate
E: PatientHasMultipleIDBands
PatientHasNoIDBand
confirm patient is able to communicate
verify patient-DOB
E: PatientUnableToCommunicate
verify patient first and last names

Figure D.30. Diagram “verify patient ID to ID band”

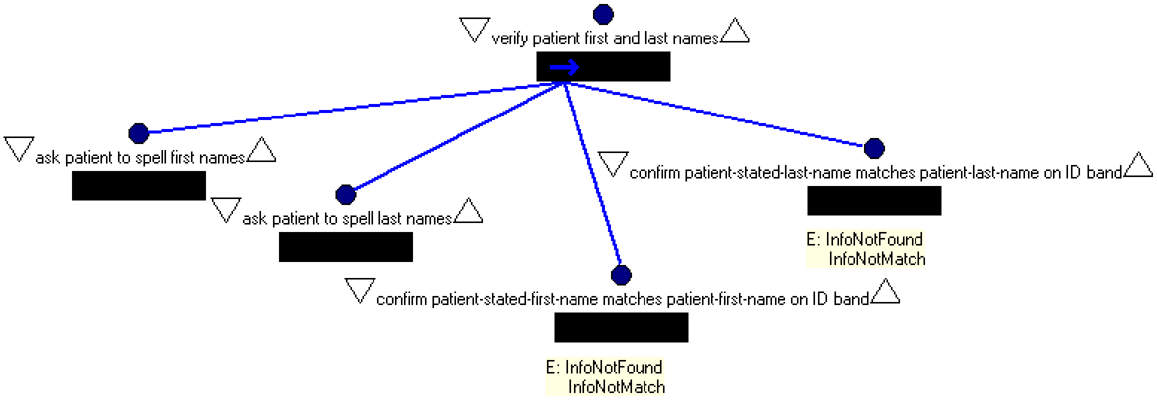

verify patient first and last names
ask patient to spell first names
ask patient to spell last names
confirm patient-stated-last-name matches patient-last-name on ID band
E: InfoNotFound
InfoNotMatch
confirm patient-stated-first-name matches patient-first-name on ID band
E: InfoNotFound
InfoNotMatch

Figure D.31. Diagram "verify patient first and last names"

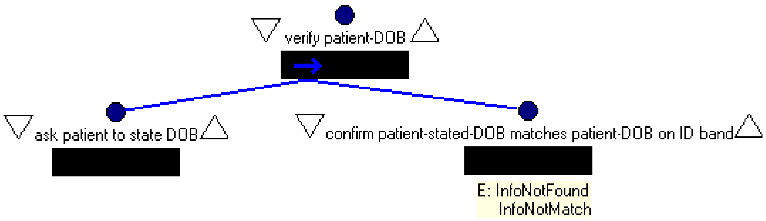


Figure D.32. Diagram "verify patient-DOB"

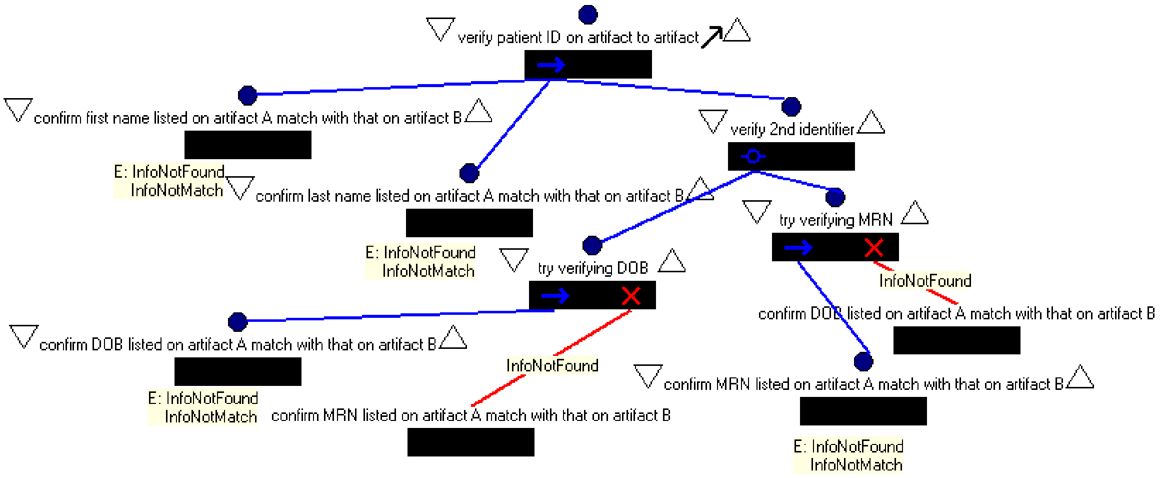

verify patient ID on artifact to artifact
confirm first name listed on artifact A match with that on artifact B
E: InfoNotFound
InfoNotMatch
confirm last name listed on artifact A match with that on artifact B
E: InfoNotFound
InfoNotMatch
verify 2nd identifier
try verifying MRN
InfoNotFound
confirm DOB listed on artifact A match with that on artifact B
try verifying DOB
confirm DOB listed on artifact A match with that on artifact B
E: InfoNotFound
InfoNotMatch
InfoNotFound
confirm MRN listed on artifact A match with that on artifact B
confirm MRN listed on artifact A match with that on artifact B
E: InfoNotFound
InfoNotMatch

Figure D.33. Diagram "verify patient ID on artifact to artifact"

APPENDIX E

BLOOD TRANSFUSION PROCESS VERIFICATION REPORT

A Checks Done Before Obtaining Unit of Blood Product

A.1 If the blood bank doesn't have a type & screen for the patient, obtain a blood specimen

Iteration 1:

- Event Binding:

  receive physician order for blood transfusion →

  confirm physician order for blood transfusion COMPLETED

  obtain blood specimen →

  obtain blood specimen STARTED

  discover blood bank doesn't have type and screen →

  contact lab for availability of type and screen

  throws exception *LabUnknown*

- Result: violation found. The property requires that the scope event "*receive physician order for blood transfusion*" must happen. However, if there is no physician order, the exception *NoPhycisianOrder* will terminate "*confirm physician order for blood transfusion* as well as the whole process. In this case, the event "*receive physician order for blood transfusion*" will never occur. In fact, the property is only required to hold as long as the event "*receive physician order for blood transfusion*" occurs. The scope event "*receive physician order for blood transfusion*" itself is not required to occur. The property specification is too strong.

- Change: in the property specification, the answer to the question "*is receive physician order for blood transfusion required?*" is changed to "*No*". [1]

Iteration 2:

- Event Binding: after Iteration 1, the step "*obtain blood specimen*" has been replaced by a sub-process "*perform Blood Specimen Obtaining process*". So the event bindings are changed accordingly.

  receive physician order for blood transfusion →

    confirm physician order for blood transfusion COMPLETED

  obtain blood specimen →

    perform Blood Specimen Obtaining process STARTED

  discover blood bank doesn't have type and screen →

    contact lab for availability of type and screen

      throws exception "*LabUnknown*"

- Result: no violation found.

A.2 Assess the patient for appropriate I.V. access before picking up units of blood product

Iteration 1:

- Event Binding:

  pick up blood product →

    pick up blood from blood bank STARTED

  assess patient for appropriate I.V. access →

    check IV access COMPLETED

    provide IV access COMPLETED

[1] The Propel has changed the scope question tree after the verification. The question "*is receive physician order for blood transfusion required?*" is no longer exist.

- Result: no violation found.

B Administration of a Unit of Blood Product

B.1 Must locate the patient"s consent document before performing a blood transfusion

Iteration 1:

- Event Binding:

  perform transfusion →

  perform transfusion STARTED

  confirm presence of informed consent →

  confirm patient consent COMPLETED

- Result: violation found. The property requires that the event "*confirm presence of informed consent*" must happen at least once. However, if there is no patient consent form, the exception *NoPatientConsent* will terminate the step "*confirm patient consent*" as well as whole process. So the event "*confirm presence of informed consent*" will never occur. This violation indicates that an important exception handling step to handle the exception *NoPatientConsent* is missing.

- Change: an exception handler "*obtain patient consent*" that handles the *NoPatientConsent* exception is added.

Iteration 2:

- Event Binding:

  perform transfusion →

  perform transfusion STARTED

  confirm presence of informed consent →

confirm patient consent COMPLETED

obtain patient consent COMPLETED

- Result: no violation found.

B.2 Must review patient history before infusing each unit of blood product

Iteration 1:

- Event Binding: could not find the binding for the event "*review patient history*".

- Change: redefine the step "*assess patient*" by adding new substep "*review patient history*" which occurs before "*evaluate patient clinically*". Also, add exception handler "*handle previously unrecognized patient problem*" to handle the exception *ProblemFoundInPatientHistory* thrown by the new substep.

Iteration 2:

- Event Binding:

  Infuse a single unit of blood →

  infuse unit of blood product STARTED

  review patient history →

  review patient history COMPLETED

- Result: no violation found.

B.3 If previously unrecognized patient problem found in history, ask physician for instructions before infusing each unit of blood product

Iteration 1:

- Event Binding: could not identify the binding for the event "*previously unrecognized patient problem found in history*".

- Change: see change in B.2.

Iteration 2:

- Event Binding:

  infuse a single unit of blood →

  infuse unit of blood product STARTED

  ask physician for instructions →

  obtain physician order to continue or discontinue infusion STARTED

  find previously unrecognized patient problem →

  review patient history throws exception *ProblemFoundInPatientHistory*

- Result: no violation found.

B.4.1 Confirm presence of ID band before infusing each unit of blood product

Iteration 1:

- Event Binding: in the process, the step "*check patient's ID bracelet*" could throw two exceptions: *MissingArmband* and *NameMismatch*, which mean that this step actually consists of two tasks: "*confirm the presence of an ID band*" and "*verify ID band and patient's stated name and birth date match*". Properties in C section refer to two different events, which correspond to those two tasks respectively. Therefore, the analyst suggested to add more details into "*check patient's ID bracelet*".

- Change: the step "*check patient's ID bracelet*" is replaced by a detailed patient identification verification sub-process.

Iteration 2:

- Event Binding:

  infuse a unit of blood product →
      infuse unit of blood product STARTED
  confirm presence of ID band →
      confirm presence of exactly one ID band on patient COMPLETED

- Result: no violation found.

B.4.2 Obtain patient's stated name and birth date before infusing each unit of blood product

Iteration 1:

- Event Binding:

  infuse a unit of blood product →
      infuse unit of blood product STARTED
  obtain patient's stated name and birth date →
      obtain patient name and birth date COMPLETED

- Result: violation found. In the process definition, the nurse only need to obtain both the patient name and the birthday when the patient armband is missing or does not match the stated name. The violation trace shows that if the patient's armband is present, the nurse only obtains the patient's name. This indicates an error in the process definition.

- Change: the step "*check patient's ID bracelet*" is replaced by a detailed patient identification verification sub-process that has steps that obtain the patient name and birthday.

Iteration 2:

Note: in the modified process, obtain patient's stated name and birthday is done by two different steps: "*ask patient to spell first and last names*" and "*ask patient to state DOB*". Therefore, two separate sets of binding are verified.

- Event Binding a:

  infuse a unit of blood product →
      infuse unit of blood product STARTED
  obtain patient's stated name and birth date →
      ask patient to spell first and last names COMPLETED

- Event Binding b:

  infuse a unit of blood product →
      infuse unit of blood product STARTED
  obtain patient's stated name and birth date →
      ask patient to state DOB COMPLETED

- Result: violations found for both sets of bindings. The traces show that the step "*confirm patient is able to communicate*" throws an exception *PatientUnableToCommunicate*. The associated exception hander simply completes the parent step "*verify patient ID to ID band*". Therefore, "*ask patient to spell first and last names* and "*verify patient-DOB*", which follow step "*confirm patient is able to communicate*", are skipped and eventually the step "*infuse unit of blood product*" is started. The violations indicate that property specification was not correctly defined. It should only be require if the patient is able to communicate.

- Change: an exceptional event "*patient unable to communicate*" is added to the property. If this event occur, the property is not required to be satisfied.

Iteration 3:

Again, two separate sets of binding are verified.

- Event Binding a:

  patient unable to communicate →

  confirm patient is able to communicate

  throws exception *PatientUnableToCommunicate*

  infuse a unit of blood product →

  infuse unit of blood product STARTED

  obtain patient's stated name and birth date →

  ask patient to spell first and last names COMPLETED

- Event Binding b:

  patient unable to communicate →

  confirm patient is able to communicate

  throws exception *PatientUnableToCommunicate*

  infuse a unit of blood product →

  infuse unit of blood product STARTED

  obtain patient's stated name and birth date →

  ask patient to state DOB COMPLETED

- Result: no violation found for both sets of bindings.

B.4.3 Verify ID band matches patient's stated name and birth date before infusing each unit of blood product

Iteration 1:

- Event Binding: could not find a step that can be bound to the event "verify ID band matches patient's stated name and birth date".

- Changes: the step "*check patient's ID bracelet*" is replaced by a detailed patient identification verification sub-process that has steps that obtain the patient name and birthday.

Iteration 2:

Note: in the new process, verify ID band matches patient's stated name and birth date is done by three different steps: "*confirm patient-stated-first-name matches patient-first-name on ID band*", "*confirm patient-stated-last-name matches patient-last-name on ID band*" and "*confirm patient-stated-DOB matches patient-DOB on ID band*". Therefore, three separate sets of binding are verified.

- Event Binding a:

  infuse a unit of blood product →

  infuse unit of blood product STARTED

  verify ID band and patient's stated name and birth date match →

  confirm patient-stated-first-name matches patient-first-name on ID band COMPLETED

- Event Binding b:

  infuse a unit of blood product →

  infuse unit of blood product STARTED

  verify ID band and patient's stated name and birth date match →

  confirm patient-stated-last-name matches patient-last-name on ID band COMPLETED

- Event Binding c:

  infuse a unit of blood product →

  infuse unit of blood product STARTED

  verify ID band and patient's stated name and birth date match →

  confirm patient-stated-DOB matches patient-DOB on ID band COMPLETED

- Result: Violation found. Similar to B.4.2.

- Change: an exceptional event "*patient unable to communicate*" is added to the property. If this event occur, the property is not required to be satisfied.

Iteration 3:

Again, three separate sets of binding are verified.

- Event Binding a:

  patient unable to communicate →
      confirm patient is able to communicate
          throws exception *PatientUnableToCommunicate*
  infuse a unit of blood product →
      infuse unit of blood product STARTED
  verify ID band and patient's stated name and birth date match →
      confirm patient-stated-first-name matches patient-first-name
          on ID band COMPLETED

- Event Binding b:

  patient unable to communicate →
      confirm patient is able to communicate
          throws exception *PatientUnableToCommunicate*
  infuse a unit of blood product →
      infuse unit of blood product STARTED
  verify ID band and patient's stated name and birth date match →
      confirm patient-stated-last-name matches patient-last-name
          on ID band COMPLETED

- Event Binding c:

patient unable to communicate →

confirm patient is able to communicate
throws exception *PatientUnableToCommunicate*

infuse a unit of blood product →

infuse unit of blood product STARTED

verify ID band and patient's stated name and birth date match →

confirm patient-stated-DOB matches patient-DOB on ID band COMPLETED

- Result: No violation found for all three sets of bindings.

B.4.4 Verify ID band matches physician order for blood transfusion before infusing each unit of blood product

Iteration 1:

- Event Binding: could not find a step that can be bound to the event "*verify ID band matches physician order*". There is a step "*verify physician order*". However, it seems to only check the completeness of the physician order because it could only throw the exception *IncompletePhysicianOrder* and the handler says "*obtain complete physician order*".

- Change: after consulting the professional, "*verify physician order*" is renamed to "*confirm complete physician order*" because "*verify physician order*" here only checks the completeness of the order. And in the Baystate Medical Center, the physician order is printed onto the tag affixed to blood product. Therefore, the step "*confirm product tag matches patient ID band*" can be considered to verify ID band matches physician order as well.

Iteration 2:

- Event Binding:

verify ID band and physician order for blood transfusion match →

confirm product tag matches patient ID band COMPLETED

infuse a unit of blood product →

infuse unit of blood product STARTED

- Result: No violation found.

B.4.5a (B.4.1 → B.4.2) Confirm presence of ID before obtaining patient's stated name and birth date (Before infusing each unit of blood product)

Iteration 1:

- Event Binding:

confirm presence of ID band →

confirm the presence of an ID bracelet COMPLETED

infuse a unit of blood product →

infuse unit of blood product STARTED

obtain patient's stated name and birth date →

request patient name COMPLETED

unit of blood product arrives →

pick up blood from blood bank COMPLETED

- Result: No violation found.

Iteration 2:

Note: the property needs to be re-verified because the related steps have been changed. In the modified process, "*obtain patient's stated name and birth date*" correspond to two steps: "*ask patient to state first and last names*" and "*ask patient to state DOB*". Therefore, two sets of binding are verified.

- Event Binding a:

  confirm presence of ID band →

  confirm presence of exactly one ID band on patient COMPLETED

  infuse a unit of blood product →

  infuse unit of blood product STARTED

  obtain patient's stated name and birth date →

  ask patient to spell first and last names COMPLETED

  unit of blood product arrives →

  pick up blood from blood bank COMPLETED

- Event Binding b:

  confirm presence of ID band →

  confirm presence of exactly one ID band on patient COMPLETED

  infuse a unit of blood product →

  infuse unit of blood product STARTED

  obtain patient's stated name and birth date →

  ask patient to state DOB COMPLETED

  unit of blood product arrives →

  pick up blood from blood bank COMPLETED

- Result: No violation found for both sets of bindings.

B.4.5b (B.4.2 → B.4.3) Obtain patient's stated name and birth date before verifying ID band matches patient's stated name and birth date (before infusing each unit of blood product)

Iteration 1:

- Event Binding:

infuse a unit of blood product →

infuse unit of blood product STARTED

obtain patient's stated name and birth date →

obtain patient name and birth date COMPLETED

unit of blood product arrives →

pick up blood from blood bank COMPLETED

verify ID band and patient's stated name and birth date match →

check patient's ID bracelet STARTED

- Result: Violation found. The violation trace shows that if the patient is unconscious, step "*obtain patient name and birth date*" will throw an exception *PatientUnconsious*. The handler for this exception is a simple *continue* handler, which means that the next step "*check patient's ID bracelet*" will be executed even without obtaining patient's name and birthday. More details should be added to the handler.

- Change: Major revision of patient id verification sub-process.

Iteration 2:

Note: "*verify ID band and patient's stated name and birth date match*" correspond to three steps: "*confirm patient-stated-first-name matches patient-first-name on ID band*", "*confirm patient-stated-last-name matches patient-last-name on ID band*", "*confirm patient-stated-DOB matches patient-DOB on ID band*". Therefore, three sets of binding are verified.

- Event Binding a:

infuse a unit of blood product →

infuse unit of blood product STARTED

obtain patient's stated name and birth date →

ask patient to spell first and last names COMPLETED

unit of blood product arrives →

pick up blood from blood bank COMPLETED

verify ID band and patient's stated name and birth date match →

confirm patient-stated-first-name matches patient-first-name on ID band STARTED

- Event Binding b:

infuse a unit of blood product →

infuse unit of blood product STARTED

obtain patient's stated name and birth date →

ask patient to spell first and last names COMPLETED

unit of blood product arrives →

pick up blood from blood bank COMPLETED

verify ID band and patient's stated name and birth date match →

confirm patient-stated-last-name matches patient-last-name on ID band STARTED

- Event Binding c:

infuse a unit of blood product →

infuse unit of blood product STARTED

obtain patient's stated name and birth date →

ask patient to spell first and last names COMPLETED

unit of blood product arrives →

pick up blood from blood bank COMPLETED

verify ID band and patient's stated name and birth date match →

confirm patient-stated-DOB matches patient-DOB on ID band STARTED

- Result: no violation found for all three sets of bindings.

B.4.5c (B.4.3 → B.4.4) Verify ID band matches patient's stated name and birth date before verifying physician order for blood transfusion (Before infusing each unit of blood product)

Iteration 1:

- Event Binding: could not find a step that can be bound to the event "*verify ID band matches physician order*". There is a step "*verify physician order.* This step, however, seems to only check the completeness of the physician order because it could only throw the exception *IncompletePhysicianOrder* and the handler says "*obtain complete physician order*".

- Change: after consulting the professional, "*verify physician order*" is renamed to "*confirm complete physician order*" because "*verify physician order*" here only checks the completeness of the order. And in the Baystate Medical Center, the physician order is printed onto the tag affixed to blood product. Therefore, the step "*confirm product tag matches patient ID band*" can be considered to verify ID band matches physician order as well.

Iteration 2:

Note: "*verify ID band and patient's stated name and birth date match*" correspond to three steps: "*confirm patient-stated-first-name matches patient-first-name on ID band*", "*confirm patient-stated-last-name matches patient-last-name on ID band*", "*confirm patient-stated-DOB matches patient-DOB on ID band*". Therefore, three sets of binding are verified.

- Event Binding a:

  infuse a unit of blood product →

  infuse unit of blood product STARTED

unit of blood product arrives →

    pick up blood from blood bank COMPLETED

verify ID band and patient's stated name and birth date match →

    confirm patient-stated-first-name matches patient-first-name

        on ID band STARTED

verify ID band and physician order for blood transfusion match →

    confirm product tag matches patient ID band COMPLETED

- Event Binding b:

infuse a unit of blood product →

    infuse unit of blood product STARTED

unit of blood product arrives →

    pick up blood from blood bank COMPLETED

verify ID band and patient's stated name and birth date match →

    confirm patient-stated-last-name matches patient-last-name

        on ID band STARTED

verify ID band and physician order for blood transfusion match →

    confirm product tag matches patient ID band COMPLETED

- Event Binding c:

infuse a unit of blood product →

    infuse unit of blood product STARTED

unit of blood product arrives →

    pick up blood from blood bank COMPLETED

verify ID band and patient's stated name and birth date match →

    confirm patient-stated-DOB matches patient-DOB

        on ID band STARTED

verify ID band and physician order for blood transfusion match →

confirm product tag matches patient ID band COMPLETED

- Result: violations found for all three sets of bindings. Similar to B.4.2.

- Change: an exceptional event "*patient unable to communicate*" is added to the property. If this event occur, the property is not required to be satisfied.

Iteration 3:

- Event Binding a:

patient unable to communicate →

confirm patient is able to communicate

throws exception *PatientUnableToCommunicate*

infuse a unit of blood product →

infuse unit of blood product STARTED

unit of blood product arrives →

pick up blood from blood bank COMPLETED

verify ID band and patient's stated name and birth date match →

confirm patient-stated-first-name matches patient-first-name on ID band STARTED

verify ID band and physician order for blood transfusion match →

confirm product tag matches patient ID band COMPLETED

- Event Binding b:

patient unable to communicate →

confirm patient is able to communicate

throws exception *PatientUnableToCommunicate*

infuse a unit of blood product →

```
    infuse unit of blood product STARTED
unit of blood product arrives →
    pick up blood from blood bank COMPLETED
verify ID band and patient's stated name and birth date match →
    confirm patient-stated-last-name matches patient-last-name
        on ID band STARTED
verify ID band and physician order for blood transfusion match →
    confirm product tag matches patient ID band COMPLETED
```

- Event Binding c:

```
patient unable to communicate →
    confirm patient is able to communicate
        throws exception PatientUnableToCommunicate
infuse a unit of blood product →
    infuse unit of blood product STARTED
unit of blood product arrives →
    pick up blood from blood bank COMPLETED
verify ID band and patient's stated name and birth date match →
    confirm patient-stated-DOB matches patient-DOB
        on ID band STARTED
verify ID band and physician order for blood transfusion match →
    confirm product tag matches patient ID band COMPLETED
```

- Result: no violation found for all three sets of bindings.

B.5.1 Verify ID band matches tag affixed to the unit of blood product (Before infusing each unit of blood product)

Iteration 1:

- Event Binding:

  verify ID band and tag affixed to unit of blood product match →
  verify product tag matches patient ID bracelet COMPLETED
  infuse a unit of blood product →
  infuse unit of blood product STARTED

- Result: violation found. The following figure shows the simplified version of the process. The step "*verify product tag matches patient ID bracelet*" is a sub-step of the parallel step "*verify blood product information*". When another sub-step "*check product expiration date & time*" is started before "*verify product tag matches patient ID bracelet*" and throws an exception *ProductExpired*, "*verify product tag matches patient ID bracelet*" will be retracted, and the handler "*handle unit of blood product expiration*" will be executed. In "*handle unit of blood product expiration*", the expired blood unit is discarded, a replacement blood unit is obtained and infused. The problem is that this handler has a *continue* badge. Therefore, the next step "*infuse unit of blood product*" will be executed and "*verify product tag matches patient ID bracelet*" is still retracted.

- Change: "*verify blood product*" is replaced by a detailed product verification sub-process. In the sub-process, both *Product Expired* and *Failed Product Check* exceptions are now the same, they're handled identically, and handled inside "*verify blood product*", so when there's an exception, new blood will be obtained from blood bank, and all the verification process should be restarted.

Iteration 2:

- Event Binding: the same as iteration 1.

- Result: no violation found.

B.5.2 Verify tag affixed to the unit of blood product (Before infusing each unit of blood product)

Iteration 1:

- Event Binding:

  verify tag affixed to unit of blood product and
  unit of blood product match →
  confirm product label matches product tag COMPLETED
  infuse a unit of blood product →
  infuse unit of blood product STARTED

- Result: no violation found.

B.5.3 Verify that unit of blood product has not expired (Before infusing each unit of blood product)

Iteration 1:

- Event Binding:

  verify that unit of blood product has not expired →
  check product expiration date & time COMPLETED
  infuse a unit of blood product →
  infuse unit of blood product STARTED

- Result: no violation found.

B.5.4a (B.5.1 → B.5.2) Verify ID band matches tag affixed to unit of blood product before verifying tag matches unit of blood product (Before infusing each unit of blood product)

Iteration 1:

- Event Binding:

verify ID band and tag affixed to unit of blood product match →
verify product tag matches patient ID bracelet COMPLETED
verify tag affixed to unit of blood product and
unit of blood product match →
verify product tag matches product label STARTED
unit of blood product arrives →
pick up blood from blood bank COMPLETED
infuse a unit of blood product →
infuse unit of blood product STARTED

- Result: violation found. Since "*verify product tag matches patient ID bracelet*" and "*verify product tag matches product label*" are sub-steps of the parallel step "*verify blood product information*", they could occur in any order. This property is too strong.

- Change: this property is removed.

B.5.4b (B.5.2 → B.5.3) Verify tag matches unit of blood product before verifying that unit of blood product has not expired (Before infusing each unit of blood product)

Iteration 1:

- Event Binding:

verify tag affixed to unit of blood product and
unit of blood product match →
verify product tag matches product label COMPLETED
verify that unit of blood product has not expired →

check product expiration date & time STARTED

unit of blood product arrives →

pick up blood from blood bank COMPLETED

infuse a unit of blood product →

infuse unit of blood product STARTED

- Result: violation found. Similar to B.5.4a, Since "*verify product tag matches patient ID bracelet*" and "*verify product tag matches product label*" are sub-steps of the parallel step "*verify blood product information*", they could occur in any order. This property is too strong.

- Change: this property is removed.

B.6 Assess patient's baseline single-unit status (Before infusing each unit of blood product)

Iteration 1:

- Event Binding:

assess patient's baseline single unit status →

assess patient STARTED

infuse a unit of blood product →

infuse unit of blood product STARTED

- Result: no violation found.

B.7a (B.4.4 → B.5.1) Verify ID band matches physician order for blood transfusion before verifying ID band matches tag affixed to the unit of blood product (Before infusing each unit of blood product)

Iteration 1:

- Event Binding:

  verify ID band and physician order for blood transfusion match →
  verify physician order COMPLETED
  verify tag affixed to unit of blood product and
  unit of blood product match →
  verify product tag matches product label STARTED
  unit of blood product arrives →
  pick up blood from blood bank COMPLETED
  infuse a unit of blood product →
  infuse unit of blood product STARTED

- Results: violation found. This property has a scope between "*unit of blood product arrives*" and "*infuse a unit of blood product*". "*verify physician order*" is done before "*unit of blood product arrives*". But between "*unit of blood product arrives*" and "*infuse a unit of blood product*", only "*verify ID band and physician order for blood transfusion match*" could occur.

- Change: major revision of patient and product verification sub-process.

Iteration 2:

Note: at Baystate, physician order is printed on product tag. Therefore, "*verify ID band matches physician order for blood transfusion*" and "*verifying ID band matches tag affixed to the unit of blood product*" are the same. Therefore, this property does not need to be verified for this blood transfusion process.

B.7b (B.5.3 → B.6) Verify unit of blood product has not expired before assessing patient's baseline single-unit status (Before infusing each unit of blood product)

Iteration 1:

- Event Binding:

verify that unit of blood product has not expired →
    check product expiration date & time COMPLETED
assess patient's baseline single unit status →
    assess patient STARTED
unit of blood product arrives →
    pick up blood from blood bank COMPLETED
infuse a unit of blood product →
    infuse unit of blood product STARTED

- Results: violation found. During "*perform pre-infusion work*, "*access patient*" is done before "*verify blood product information*", which is the parent step of "*check product expiration date & time*".

- Change: this property is removed because it is not necessary to require "*verify unit of blood product has not expired*" must happen before "*assessing patient's baseline single-unit status*".

B.8 If patient's baseline single-unit assessment is problematic, ask physician for instructions before infusing unit of blood product (Before infusing each unit of blood product)

Iteration 1:

- Event Binding:

find problematic patient assessment →
    assess patient TERMINATED
ask physician for instructions →
    confirm physician order to continue transfusion COMPLETED |

confirm physician order to discontinue transfusion COMPLETED

infuse a unit of blood product →

infuse unit of blood product STARTED

- Results: no violation found.

B.9 Infusion of blood product must begin within 30 minutes after being picked up from blood bank

- Discussion: this is a timing property that cannot be verified by our process verification framework.

B.10 If a unit of blood product expires, it cannot be infused into a patient

Iteration 1:

- Event Binding:

expiration date on a unit of blood product is exceeded →

check product expiration date & time TERMINATED

infuse a unit of blood product →

infuse unit of blood product STARTED

- Results: violation found. Note that the events in this property have certain relationship among them: they are all related to the same unit of blood. However, the current verifier cannot distinguish events related to different blood unit. The violation trace shows that when one unit of blood is found to be expired, the nurse go to pick up a new unit of blood and transfuse the new unit of blood. This is obvious a false negative result. To eliminated this false violation, one can manually do an artifact flow analysis. For this property, "*infuse a unit of blood product*" does not have a binding because, according to the artifact flow analysis, after a unit of blood is expired, it will be discarded.

- Change: the professional suggested to change the property to "*if a unit of blood product expires, must pick up (new blood product) from blood bank*".

Iteration 2:

- Event Binding:

  expiration date on a unit of blood product is exceeded →
  check product expiration date & time TERMINATED
  pick up a unit of blood product from blood bank →
  pick up blood from blood bank COMPLETED
  infuse a unit of blood product →
  infuse unit of blood product STARTED

- Results: violation found. In the process, the step "*check product expiration date & time*" appears in two places: one during "*perform bedside checks*" and the other "*handle suspected transfusion reaction*". The exception *FailProductCheck* from the first one is handled by "*handle failed product verification by replacement*", which has a sub-step "*pick up blood from blood bank*". For the second one, however, the exception is handled by a continue handler without handling step.

- Change: attach the handler "*handle failed product verification by replacement*" to the handler that handles the exception from the second one.

Iteration 3:

- Event Binding: the same as Iteration 2.

  expiration date on a unit of blood product is exceeded →
  check product expiration date & time TERMINATED
  pick up a unit of blood product from blood bank →

pick up blood from blood bank COMPLETED

infuse a unit of blood product →

infuse unit of blood product STARTED

- Results: no violation found.

B.11 Must assess patient’s 15-minute single-unit status after 15 minutes of the infusion has passed

- Discussion: this is a timing property that cannot be verified by our process verification framework.

B.12 Must assess patient’s post-single-unit status after the infusion is completed

Iteration 1:

- Event Binding:

record infusion information →

record infusion info COMPLETED

discard transfusion materials →

discard infusion materials COMPLETED

assess patient’s post single unit status →

perform post-infusion work.assess patient STARTED

infuse a unit of blood product →

infuse unit of blood product COMPLETED

- Results: no violation found. Note that the event is bound to ‘*assess patient’s post single unit status*” to step “*assess patient*”, which is a sub-step of “*perform post-infusion work*” instead of all references to “*assess patient*”.

B.14a If patient infusion exceeds the infusion time limit, must stop infusion

- Discussion: in the process, there is no exception about the infusion exceeding the time limit. More detail might need to be added to the process.

B.14b If patient infusion exceeds the infusion time limit, must ask blood bank for further instructions

- Discussion: in the process, there is no exception about the infusion exceeding the time limit. More detail might need to be added to the process.

B.15 If the blood bank or the physician instructs that the infusion be discontinued, the infusion must be discontinued

Iteration 1:

- Event Binding:

  receive blood bank OR physician instruction to discontinue
  the infusion of a unit of blood product →
  confirm physician order to discontinue transfusion COMPLETED
  discontinue the infusion of a unit of blood product →
  discontinue transfusion STARTED

- Results: no violation found.

B.16 If an infusion has been stopped, it must either be resumed or be discontinued

Iteration 1:

- Event Binding:

resume or discontinue the infusion of a unit of blood product →

discontinue transfusion COMPLETED |

continue transfusion COMPLETED

stop the infusion of unit of blood product →

infuse unit of blood product STARTED

- Results: no violation found.

C. Handling a Suspected Transfusion Reaction

C.1 If patient has a suspected transfusion reaction, the infusion must be stopped immediately

- Discussion: this property cannot be verified because it is a timing property.

C.2.1 If patient has a suspected transfusion reaction, must confirm the presence of an ID band

Iteration 1:

- Event Binding:

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

confirm the presence of an ID band →

confirm the presence of an ID band STARTED

- Results: no violation found.

C.2.2 If patient has a suspected transfusion reaction, must obtain patient's stated name and birth date

Iteration 1:

- Event Binding:

  suspect patient is having a transfusion reaction →

  Exception *ReactionSuspected* is thrown by

  infuse unit of blood product

  obtain patient's stated name and birth date →

  obtain patient name and birth date STARTED

- Results: violation found. Similar to B.4.2, when the exception *ReactionSuspected* is thrown, the handler "*handle suspected transfusion reaction*" will reverify the patient' identification. Normally only the patient's name will be obtained by the step "*request patient name*". The step "*obtain patient name and birth date*" will be executed only if anything goes wrong during checking the patient's name.

- Change: major revision of patient id verification sub-process:

Iteration 2:

Note: "*obtain patient's stated name and birth date*" correspond to two steps: "*ask patient to state first and last names*" and "*ask patient to state DOB*". Therefore, two sets of binding are verified.

- Event Binding a:

  suspect patient is having a transfusion reaction →

  Exception *ReactionSuspected* is thrown by

  infuse unit of blood product

  obtain patient's stated name and birth date →

  ask patient to state first and last names STARTED

- Event Binding b:

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

obtain patient's stated name and birth date →

ask patient to state DOB STARTED

- Change: an exceptional event "*patient unable to communicate*" is added to the property. If this event occur, the property is not required to be satisfied.

Iteration 3:

- Event Binding a:

patient unable to communicate →

confirm patient is able to communicate

throws exception *PatientUnableToCommunicate*

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

obtain patient's stated name and birth date →

ask patient to state first and last names STARTED

- Event Binding b:

patient unable to communicate →

confirm patient is able to communicate

throws exception *PatientUnableToCommunicate*

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

obtain patient's stated name and birth date →

ask patient to state DOB STARTED

- Result: no violation found for both sets of bindings.

C.2.3 If patient has a suspected transfusion reaction, must verify ID band matches patient's stated name and birth date

Iteration 1:

- Event Binding:

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

verify ID band and patient's stated name and birth date match →

check patient's ID bracelet STARTED

- Result: no violation found.

Iteration 2:

Note: "*verify ID band and patient's stated name and birth date match*" correspond to three steps: "*confirm patient-stated-first-name matches patient-first-name on ID band*", "*confirm patient-stated-last-name matches patient-last-name on ID band*", "*confirm patient-stated-DOB matches patient-DOB on ID band*". Therefore, three sets of binding are verified.

- Event Binding a:

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

verify ID band and patient's stated name and birth date match →

confirm patient-stated-first-name matches patient-first-name

on ID band STARTED

- Event Binding b:

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

verify ID band and patient's stated name and birth date match →

confirm patient-stated-last-name matches patient-last-name

on ID band STARTED

- Event Binding c:

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

verify ID band and patient's stated name and birth date match →

confirm patient-stated-DOB matches patient-DOB

on ID band STARTED

- Result: violations found for all three sets of binding. The violations occur when the patient is not able to communicate.

- Change: an exceptional event "*patient unable to communicate*" is added to the property. If this event occur, the property is not required to be satisfied.

Iteration 3:

- Event Binding a:

```
patient unable to communicate →
    confirm patient is able to communicate
        throws exception PatientUnableToCommunicate
suspect patient is having a transfusion reaction →
    Exception ReactionSuspected is thrown by
        infuse unit of blood product
verify ID band and patient's stated name and birth date match →
    confirm patient-stated-first-name matches patient-first-name
        on ID band STARTED
```

- Event Binding b:

```
patient unable to communicate →
    confirm patient is able to communicate
        throws exception PatientUnableToCommunicate
suspect patient is having a transfusion reaction →
    Exception ReactionSuspected is thrown by
        infuse unit of blood product
verify ID band and patient's stated name and birth date match →
    confirm patient-stated-last-name matches patient-last-name
        on ID band STARTED
```

- Event Binding c:

```
patient unable to communicate →
    confirm patient is able to communicate
        throws exception PatientUnableToCommunicate
suspect patient is having a transfusion reaction →
    Exception ReactionSuspected is thrown by
```

infuse unit of blood product

verify ID band and patient's stated name and birth date match →

confirm patient-stated-DOB matches patient-DOB

on ID band STARTED

- Result: no violations found for all three sets of bindings.

C.2.4a (C.2.1 → C.2.2) Confirm presence of ID band before obtaining patient's stated name and birth date

Iteration 1:

- Event Binding:

obtain patient's stated name and birth date →

obtain patient name and birth date STARTED

request patient name STARTED

confirm presence of ID band →

confirm the presence of an ID bracelet COMPLETED

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

resume or discontinue the infusion of a unit of blood product →

discontinue transfusion STARTED |

continue transfusion STARTED

- Result: this property holds if two handlers in "*reverify*" are *complete* handlers or *continue* handlers. Otherwise, there will be violations. See B.5.1 and B.16.

- Change: major revision of production verification sub-process.

Iteration 2:

Note: "*obtain patient's stated name and birth date*" correspond to two steps: "*ask patient to state first and last names*" and "*ask patient to state DOB*". Therefore, two sets of binding are verified.

- Event Binding a:

  obtain patient's stated name and birth date →
  ask patient to state first and last names STARTED

  confirm presence of ID band →
  confirm the presence of an ID bracelet COMPLETED

  suspect patient is having a transfusion reaction →
  Exception *ReactionSuspected* is thrown by
  infuse unit of blood product

  resume or discontinue the infusion of a unit of blood product →
  discontinue transfusion STARTED |
  continue transfusion STARTED

- Event Binding b:

  obtain patient's stated name and birth date →
  ask patient to state DOB STARTED

  confirm presence of ID band →
  confirm the presence of an ID bracelet COMPLETED

  suspect patient is having a transfusion reaction →
  Exception *ReactionSuspected* is thrown by
  infuse unit of blood product

  resume or discontinue the infusion of a unit of blood product →
  discontinue transfusion STARTED |
  continue transfusion STARTED

- Result: no violation found. Note that "*obtain patient's stated name and birth date*" is bound to the STARTED state.

C.2.4b (C.2.2 → C.2.3) Obtain patient's stated name and birth date before verifying ID band matches patient's stated name and birth date

Iteration 1:

- Event Binding:

  obtain patient's stated name and birth date →

  obtain patient name and birth date COMPLETED

  request patient name COMPLETED

  verify ID band and patient's stated name and birth date match →

  check patient's ID bracelet STARTED

  suspect patient is having a transfusion reaction →

  Exception *ReactionSuspected* is thrown by

  infuse unit of blood product

  resume or discontinue the infusion of a unit of blood product →

  discontinue transfusion STARTED |

  continue transfusion STARTED

- Result: this property holds if two handlers in "*reverify*" are *complete* handlers or *continue* handlers. Otherwise, there will be violations. See B.5.1 and B.16.

- Change: major revision of production verification sub-process.

Iteration 2:

Note "*verify ID band and patient's stated name and birth date match*" correspond to three steps: "*confirm patient-stated-first-name matches patient-first-name on ID band*", "*confirm patient-stated-last-name matches patient-last-name on ID band*", “*confirm patient-stated-DOB matches patient-DOB on ID band*”. Therefore, three sets of binding are verified.

- Event Binding a:

  obtain patient’s stated name and birth date →

  ask patient to spell first and last names COMPLETED

  verify ID band and patient’s stated name and birth date match →

  confirm patient-stated-first-name matches patient-first-name on ID band STARTED

  suspect patient is having a transfusion reaction →

  Exception *ReactionSuspected* is thrown by *infuse unit of blood product*

  resume or discontinue the infusion of a unit of blood product →

  obtain physician order to continue or discontinue infusion STARTED

- Event Binding b:

  obtain patient’s stated name and birth date →

  ask patient to spell first and last names COMPLETED

  verify ID band and patient’s stated name and birth date match →

  confirm patient-stated-last-name matches patient-last-name on ID band STARTED

  suspect patient is having a transfusion reaction →

  Exception *ReactionSuspected* is thrown by *infuse unit of blood product*

  resume or discontinue the infusion of a unit of blood product →

  obtain physician order to continue or discontinue infusion STARTED

- Event Binding c:

obtain patient's stated name and birth date →

ask patient to state DOB COMPLETED

verify ID band and patient's stated name and birth date match →

confirm patient-stated-DOB matches patient-DOB on ID band STARTED

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

resume or discontinue the infusion of a unit of blood product →

obtain physician order to continue or discontinue infusion STARTED

- Results: no violation found for all three sets of binding.

C.3.1 If patient has a suspected transfusion reaction, must verify ID band matches tag affixed to the unit of blood product

Iteration 1:

- Event Binding:

verify ID band and tag affixed to unit of blood product match →

verify product tag matches patient ID bracelet STARTED

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

- Result: violation found. To handle the exception *ReactionSuspected*, during reverifying the blood unit, the step "*check product expiration date & time*" might be executed before "*verify product tag matches patient ID bracelet*". If "*check product expiration date & time*" throws the exception *FailedProductCheck*, "*verify product tag matches patient ID bracelet*" will not be executed.

- Change: the property is changed to "*If patient has a suspected transfusion reaction, must verify ID band matches tag affixed to the unit of blood product unless exception FailedProductCheck is thrown*".

Iteration 2:

- Event Binding:

verify ID band and tag affixed to unit of blood product match →
confirm product tag matches patient ID band STARTED
suspect patient is having a transfusion reaction →
Exception *ReactionSuspected* is thrown by
infuse unit of blood product
FailedProductCheck is thrown →
Exception *FailedProductCheck* is thrown by
check product expiration date & time |
Exception *FailedProductCheck* is thrown by
confirm product type on product tag matches that on patient record

- Result: no violation found.

C.3.2 If patient has a suspected transfusion reaction, must verify tag affixed to the unit of blood product

Iteration 1:

- Event Binding:

verify tag affixed to unit of blood product and
unit of blood product match →
verify product tag matches product label STARTED
suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by
infuse unit of blood product

- Result: violation found. Similar to C.3.1, to handle the exception *ReactionSuspected*, during reverifying the blood unit, the step “*check product expiration date & time*” might be executed before “*verify product tag matches product label*”. If “*check product expiration date & time*” throws the exception *FailedProductCheck*, “*verify product tag matches product label*” will not be executed.
- Change: the property is changed to “*If patient has a suspected transfusion reaction, must verify tag affixed to the unit of blood product unless exception FailedProductCheck is thrown*”.

Iteration 2:

- Event Binding:

verify tag affixed to unit of blood product and
unit of blood product match →
confirm product label matches product tag STARTED
suspect patient is having a transfusion reaction →
Exception *ReactionSuspected* is thrown by
infuse unit of blood product
FailedProductCheck is thrown →
Exception *FailedProductCheck* is thrown by
confirm product tag matches patient ID band |
Exception *FailedProductCheck* is thrown by
check product expiration date & time |
Exception *FailedProductCheck* is thrown by
confirm product type on product tag matches that on patient record

- Result: no violation found.

C.3.3 If patient has a suspected transfusion reaction, must verify that unit of blood product has not expired

Iteration 1:

- Event Binding:

  verify that unit of blood product has not expired →
  check product expiration date & time STARTED
  suspect patient is having a transfusion reaction →
  Exception *ReactionSuspected* is thrown by
  infuse unit of blood product

- Result: violation found. Similar to C.3.1 and C.3.2, to handle the exception *ReactionSuspected*, during reverifying the blood unit, the step "*verify product tag matches product label*" might be executed before "*check product expiration date & time*". If "*verify product tag matches product label*" throws the exception *FailedProductCheck*, "*check product expiration date & time*" will not be executed.

- Change: the property is changed to "*If patient has a suspected transfusion reaction, must verify that unit of blood product has not expired unless exception FailedProductCheck is thrown*".

Iteration 2:

- Event Binding:

  verify that unit of blood product has not expired →
  check product expiration date & time STARTED
  suspect patient is having a transfusion reaction →
  Exception *ReactionSuspected* is thrown by
  infuse unit of blood product

FailedProductCheck is thrown →

Exception *FailedProductCheck* is thrown by

check blood product by 2 nurses |

Exception *FailedProductCheck* is thrown by

confirm product type on product tag matches that on patient record

- Result: no violation found.

C.3.4a (C.3.1 → C.3.2) Verify ID band matches tag affixed to the unit of blood product before verifying tag matches the unit of blood product

Iteration 1:

- Event Binding:

verify ID band and tag affixed to unit of blood product match →

verify product tag matches patient ID bracelet COMPLETED

verify tag affixed to unit of blood product and

unit of blood product match →

verify product tag matches product label STARTED

resume or discontinue the infusion of a unit of blood product →

discontinue transfusion STARTED |

continue transfusion STARTED

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

- Result: violation found. Since "*verify product tag matches patient ID bracelet*" and "*verify product tag matches product label*" are sub-steps of the parallel step "*verify blood product information*", they could occur in any order.

- Change: this property is removed.

C.3.4b (C.3.2 → C.3.3) Verify tag matches the unit of blood product before verifying that unit of blood product has not expired

Iteration 1:

- Event Binding:

  verify that unit of blood product has not expired →
  check product expiration date & time STARTED
  verify tag affixed to unit of blood product and
  unit of blood product match →
  verify product tag matches product label COMPLETED
  resume or discontinue the infusion of a unit of blood product →
  discontinue transfusion STARTED |
  continue transfusion STARTED
  suspect patient is having a transfusion reaction →
  Exception *ReactionSuspected* is thrown by
  infuse unit of blood product

- Result: violation found. Similar to C.3.4a, since "*check product expiration date & time*" and "*verify product tag matches product label*" are sub-steps of the parallel step "*verify blood product information*", they could occur in any order.

- Change: this property is removed.

C.4 If patient has a suspected transfusion reaction, ask physician and blood bank for further instructions

Iteration 1:

- Event Binding:

ask physician and blood bank for further instructions →

obtain blood bank evaluation STARTED |

obtain physician order for treatment plan STARTED

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

- Result: this property holds if two handlers in "*reverify*" are *complete* handlers or *continue* handlers. Otherwise, there will be violations. See B.5.1 and B.16.
- Change: major revision of production verification sub-process.

Iteration 2:

Note: the event "*ask physician and blood bank for further instructions*" correspond to two steps: "*obtain physician order to continue or discontinue infusion*" and "*obtain blood bank evaluation*". Therefore, two sets of binding are verified.

- Event Binding a:

ask physician and blood bank for further instructions →

obtain physician order to continue or discontinue infusion STARTED

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

- Event Binding b:

ask physician and blood bank for further instructions →

obtain blood bank evaluation STARTED

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

- Result: no violation found.

C.5a (C.1 → C.2.1) Must stop the infusion before confirming the presence of an ID band

Iteration 1:

- Event Binding:

confirm presence of ID band →
  confirm the presence of an ID bracelet STARTED
stop the infusion of a unit of blood product →
  interrupt transfusion COMPLETED
resume or discontinue the infusion of a unit of blood product →
  discontinue transfusion STARTED |
  continue transfusion STARTED
suspect patient is having a transfusion reaction →
  Exception *ReactionSuspected* is thrown by
    infuse unit of blood product

- Results: this property holds if two handlers in "*reverify*" are *complete* handlers or *continue* handlers. Otherwise, there will be violations. See B.5.1 and B.16.

- Change: major revision of production verification sub-process.

Iteration 2:

- Event Binding:

confirm presence of ID band →
  confirm presence of exactly one ID band on patient STARTED
stop the infusion of a unit of blood product →

interrupt transfusion COMPLETED

resume or discontinue the infusion of a unit of blood product →

obtain physician order to continue or discontinue infusion STARTED

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

- Results: no violation found.

C.5b (C.2.3 → C.3.1) Must verify ID band matches patient's stated name and birth date before verifying tag affixed to the unit of blood product

Iteration 1:

- Event Binding:

verify ID band and patient's stated name and birth date match →

check patient's ID bracelet COMPLETED

verify ID band and tag affixed to the unit of blood product match →

confirm product tag matches patient ID band STARTED

resume or discontinue the infusion of a unit of blood product →

discontinue transfusion STARTED |

continue transfusion STARTED

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

- Results: no violation found.

Iteration 2:

Note: "*verify ID band and patient's stated name and birth date match*" correspond to three steps: "*confirm patient-stated-first-name matches patient-first-name on ID* *band*”, “*confirm patient-stated-last-name matches patient-last-name on ID band*”, “*confirm patient-stated-DOB matches patient-DOB on ID band*”. Therefore, three sets of binding are verified.

- Event Binding a:
  - *verify ID band and patient's stated name and birth date match* →
    - *confirm patient-stated-first-name matches patient-first-name on ID band* COMPLETED
  - *verify ID band and tag affixed to the unit of blood product match* →
    - *confirm product tag matches patient ID band* STARTED
  - *resume or discontinue the infusion of a unit of blood product* →
    - *obtain physician order to continue or discontinue infusion* STARTED
  - *suspect patient is having a transfusion reaction* →
    - Exception *ReactionSuspected* is thrown by *infuse unit of blood product*
- Event Binding b:
  - *verify ID band and patient's stated name and birth date match* →
    - *confirm patient-stated-last-name matches patient-last-name on ID band* COMPLETED
  - *verify ID band and tag affixed to the unit of blood product match* →
    - *confirm product tag matches patient ID band* STARTED
  - *resume or discontinue the infusion of a unit of blood product* →
    - *obtain physician order to continue or discontinue infusion* STARTED
  - *suspect patient is having a transfusion reaction* →
    - Exception *ReactionSuspected* is thrown by *infuse unit of blood product*

- Event Binding c:

  verify ID band and patient's stated name and birth date match →

  confirm patient-stated-DOB matches patient-DOB

  on ID band COMPLETED

  verify ID band and tag affixed to the unit of blood product match →

  confirm product tag matches patient ID band STARTED

  resume or discontinue the infusion of a unit of blood product →

  obtain physician order to continue or discontinue infusion STARTED

  suspect patient is having a transfusion reaction →

  Exception *ReactionSuspected* is thrown by

  infuse unit of blood product

- Results: violations found. They occur when the patient is not able to communicate.

- Change: an exceptional event "*patient unable to communicate*" is added to the property. If this event occur, the property is not required to be satisfied.

Iteration 3:

- Event Binding a:

  patient unable to communicate→

  confirm patient is able to communicate

  throws exception *PatientUnableToCommunicate*

  verify ID band and patient's stated name and birth date match →

  confirm patient-stated-first-name matches patient-first-name

  on ID band COMPLETED

  verify ID band and tag affixed to the unit of blood product match →

confirm product tag matches patient ID band STARTED

resume or discontinue the infusion of a unit of blood product →

obtain physician order to continue or discontinue infusion STARTED

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

- Event Binding b:

patient unable to communicate→

confirm patient is able to communicate

throws exception *PatientUnableToCommunicate*

verify ID band and patient's stated name and birth date match →

confirm patient-stated-last-name matches patient-last-name on ID band COMPLETED

verify ID band and tag affixed to the unit of blood product match →

confirm product tag matches patient ID band STARTED

resume or discontinue the infusion of a unit of blood product →

obtain physician order to continue or discontinue infusion STARTED

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

- Event Binding c:

patient unable to communicate→

confirm patient is able to communicate

throws exception *PatientUnableToCommunicate*

verify ID band and patient's stated name and birth date match →

confirm patient-stated-DOB matches patient-DOB
on ID band COMPLETED

verify ID band and tag affixed to the unit of blood product match →
confirm product tag matches patient ID band STARTED

resume or discontinue the infusion of a unit of blood product →
obtain physician order to continue or discontinue infusion STARTED

suspect patient is having a transfusion reaction →
Exception *ReactionSuspected* is thrown by
infuse unit of blood product

- Results: no violation found.

C.5c (C.3.3 → C.4) Must verify that unit of blood product has not expired before asking physician and blood bank for further instructions

Iteration 1:

- Event Binding:

verify that unit of blood product has not expired →
check product expiration date & time COMPLETED

ask physician and blood bank for further instructions →
obtain physician order for treatment plan STARTED |
obtain blood bank evaluation STARTED

resume or discontinue the infusion of a unit of blood product →
discontinue transfusion STARTED
continue transfusion STARTED

suspect patient is having a transfusion reaction →
Exception *ReactionSuspected* is thrown by
infuse unit of blood product

- Results: this property holds if two handlers in "*reverify*" are *complete* handlers or *continue* handlers. Otherwise, there will be violations. See B.5.1 and B.16.

Iteration 2:

- Event Binding:

  verify that unit of blood product has not expired →
  check product expiration date & time COMPLETED
  ask physician and blood bank for further instructions →
  ask physician and blood bank for further instructions STARTED
  suspect patient is having a transfusion reaction →
  Exception *ReactionSuspected* is thrown by
  infuse unit of blood product

- Result: violation found. During the re-verification, the step "*check product expiration date & time*" is a sub-step of a parallel step "*verify blood product information*". When the other sub-step "*verify blood product information*" throws exception *FailedProductCheck*, the step "*check product expiration date & time*" can be retracted. Since the re-verification step has a continue exception handler for *FailedProductCheck*, the "*obtain physician order to continue or discontinue infusion*" will be executed.

- Change: the property is changed to "*Must verify that unit of blood product has not expired before asking physician and blood bank for further instructions unless exception FailedProductCheck is thrown*".

Iteration 3:

- Event Binding:

verify that unit of blood product has not expired →

check product expiration date & time COMPLETED

ask physician and blood bank for further instructions →

ask physician and blood bank for further instructions STARTED

suspect patient is having a transfusion reaction →

Exception *ReactionSuspected* is thrown by

infuse unit of blood product

FailedProductCheck is thrown →

Exception *FailedProductCheck* is thrown by

check blood product by 2 nurses |

Exception *FailedProductCheck* is thrown by

confirm product type on product tag matches that on patient record

- Result: no violation found.

C.6 If an infusion has been stopped due to a suspected transfusion reaction, if the blood bank orders specimens, specimens must be obtained

- Discussion: there are no enough details about the blood bank ordering specimens.

D. Checks That are Done Before Obtaining Specimen(s) from a patient

D.1 Confirm presence of ID band before obtaining a specimem (before obtaining a specimen)

Iteration 1:

- Event Binding: could not find a step that can be bound to the event "*receive order to obtain a specimen*".

- Change: more detailed "*perform Blood Specimen Obtaining process*" sub-process is elicited.

Iteration 2:

- Event Binding:

  receive order to obtain a specimen →

  order test(s) COMPLETED

  confirm presence of ID band →

  confirm presence of exactly one ID band on patient COMPLETED

  obtain specimen →

  obtain blood specimen STARTED

  send specimen to blood bank →

  send specimen to lab via tube system STARTED

- Result: no violation found.

D.2 Obtain patient's stated name and birth date before obtaining a specimen

Iteration 1:

- Event Binding: could not find a step that can be bound to the event "*receive order to obtain a specimen*".

- Change: more detailed "*perform Blood Specimen Obtaining process*" sub-process is elicited.

Iteration 2:

- Event Binding:

send specimen to blood bank →

send specimen to lab via tube system STARTED

receive order to obtain a specimen →

order test(s) COMPLETED

obtain patient's stated name and birth date →

ask patient to spell first and last names COMPLETED

obtain specimen →

obtain blood specimen STARTED

- Result: violation found. Similar to the property B.4.2, the step "*confirm patient is able to communicate*" throws an exception *PatientUnableToCommunicate*. The associated exception hander simply completes the parent step "*verify patient ID to ID band*". Therefore, "*ask patient to spell first and last names*" is skipped and eventually the step "*obtain blood specimen*" is started.

- Change: an exceptional event "*patient unable to communicate*" is added to the property. If this event occur, the property is not required to be satisfied.

Iteration 3:

- Event Binding:

patient unable to communicate→

confirm patient is able to communicate

throws exception *PatientUnableToCommunicate*

send specimen to blood bank →

send specimen to lab via tube system STARTED

receive order to obtain a specimen →

order test(s) COMPLETED

obtain patient's stated name and birth date →

ask patient to spell first and last names COMPLETED

obtain specimen →

obtain blood specimen STARTED

- Result: no violation found.

D.3 Verify ID band matches patient stated name and birth date before obtaining a specimen

Iteration 1:

- Event Binding: could not find a step that can be bound to the event "*receive order to obtain a specimen*".

- Change: more detailed "*perform Blood Specimen Obtaining process*" sub-process is elicited.

Iteration 2:

Note: "*verify ID band and patient's stated name and birth date match*" actually corresponds to three steps: "*confirm patient-stated-DOB matches patient-DOB on ID band*", "*confirm patient-stated-first-name matches patient-first-name on ID band*", and "*confirm patient-stated-last-name matches patient-last-name on ID band*". So three sets of binding need to be verified.

- Event Binding a:

send specimen to blood bank →

send specimen to lab via tube system STARTED

receive order to obtain a specimen →

order test(s) COMPLETED

obtain specimen →

obtain blood specimen STARTED

verify ID band and patient's stated name and birth date match →

confirm patient-stated-DOB matches patient-DOB

on ID band COMPLETED

- Event Binding b:

send specimen to blood bank →

send specimen to lab via tube system STARTED

receive order to obtain a specimen →

order test(s) COMPLETED

obtain specimen →

obtain blood specimen STARTED

verify ID band and patient's stated name and birth date match →

confirm patient-stated-first-name matches patient-first-name

on ID band COMPLETED

- Event Binding c:

send specimen to blood bank →

send specimen to lab via tube system STARTED

receive order to obtain a specimen →

order test(s) COMPLETED

obtain specimen →

obtain blood specimen STARTED

verify ID band and patient's stated name and birth date match →

confirm patient-stated-last-name matches patient-last-name

on ID band COMPLETED

- Result: violation found. Similar to the property B.4.2.

- Change: an exceptional event "*patient unable to communicate*" is added to the property. If this event occur, the property is not required to be satisfied.

Iteration 3:

- Event Binding a:

patient unable to communicate→

confirm patient is able to communicate

throws exception *PatientUnableToCommunicate*

send specimen to blood bank →

send specimen to lab via tube system STARTED

receive order to obtain a specimen →

order test(s) COMPLETED

obtain specimen →

obtain blood specimen STARTED

verify ID band and patient's stated name and birth date match →

confirm patient-stated-DOB matches patient-DOB on ID band COMPLETED

- Event Binding b:

patient unable to communicate→

confirm patient is able to communicate

throws exception *PatientUnableToCommunicate*

send specimen to blood bank →

send specimen to lab via tube system STARTED

receive order to obtain a specimen →

order test(s) COMPLETED

obtain specimen →

obtain blood specimen STARTED

verify ID band and patient's stated name and birth date match →

confirm patient-stated-first-name matches patient-first-name on ID band COMPLETED

- Event Binding c:

patient unable to communicate→

confirm patient is able to communicate throws exception *PatientUnableToCommunicate*

send specimen to blood bank →

send specimen to lab via tube system STARTED

receive order to obtain a specimen →

order test(s) COMPLETED

obtain specimen →

obtain blood specimen STARTED

verify ID band and patient's stated name and birth date match →

confirm patient-stated-last-name matches patient-last-name on ID band COMPLETED

- Result: no violation found.

D.4 Verify order to obtain a specimen before obtaining a specimen

Iteration 1:

- Event Binding: could not find a step related to the event "*verify ID band and order to obtain a specimen match*".

- Change: more detailed "*perform Blood Specimen Obtaining process*" sub-process is elicited.

Iteration 2:

Note: In Baystate Medical Center, the order to obtain a specimen is print on the blood specimen label. Therefore, "*verify ID band and order to obtain a specimen match*" is the same as verifying patient ID band against the specimen label.

- Event Binding:

  send specimen to blood bank →
  send specimen to lab via tube system STARTED
  receive order to obtain a specimen →
  order test(s) COMPLETED
  obtain specimen →
  obtain blood specimen STARTED
  verify ID band and order to obtain a specimen match →
  verify labels COMPLETED

- Result: no violation found.

D.5 Verify specimen container label before obtaining a specimen

Iteration 1:

- Event Binding: could not find a step related to the event "*verify ID band and specimen container label match*".

- Change: more detailed "*perform Blood Specimen Obtaining process*" sub-process is elicited.

Iteration 2:

- Event Binding:

send specimen to blood bank →

send specimen to lab via tube system STARTED

receive order to obtain a specimen →

order test(s) COMPLETED

obtain specimen →

obtain blood specimen STARTED

verify ID band and order to obtain a specimen match →

verify labels COMPLETED

- Result: no violation found.

D.6a (D.1 → D.2) Confirm presence of ID band before obtaining patient's stated name and birth date (before obtaining a specimen)

Iteration 1:

- Event Binding:

confirm presence of ID band →

confirm the presence of an ID bracelet COMPLETED

obtain patient's stated name and birth date →

request patient name STARTED |

obtain patient name and birth date STARTED

receive order to obtain a specimen →

order test(s) COMPLETED

obtain specimen →

obtain blood specimen STARTED

- Result: violation found. The violation trace shows that when "*confirm the presence of an ID bracelet*" fails, it throws an exception *MissingArmband*. The handler "*provide new ID bracelet*" will obtain patient's stated name and birth date. In this case, one can obtain patient's stated name and birth date without completing the step "*confirm the presence of an ID bracelet*".

- Change: more detailed "*perform Blood Specimen Obtaining process*" sub-process is elicited.

Iteration 2:

Two sets of binding are verified.

- Event Binding a:

    confirm presence of ID band →

    confirm presence of exactly one ID band on patient COMPLETED

    obtain patient's stated name and birth date →

    ask patient to spell first and last names STARTED

    receive order to obtain a specimen →

    order test(s) COMPLETED

    obtain specimen →

    obtain blood specimen STARTED

- Event Binding b:

    confirm presence of ID band →

    confirm presence of exactly one ID band on patient COMPLETED

    obtain patient's stated name and birth date →

    ask patient to state DOB STARTED

    receive order to obtain a specimen →

    order test(s) COMPLETED

    obtain specimen →

    obtain blood specimen STARTED

- Result: no violation found.

D.6b (D.2 → D.3) Obtain patient's info before verifying ID band matches patient's stated name and birth date

Iteration 1:

- Event Binding:

  verify ID band and patient's stated name and birth date match →
  check patient's ID bracelet STARTED
  obtain patient's stated name and birth date →
  request patient name COMPLETED |
  obtain patient name and birth date COMPLETED
  receive order to obtain a specimen →
  order test(s) COMPLETED
  obtain specimen →
  obtain blood specimen STARTED

- Result: violation found. Similar to the property D.2, the violation trace shows that if the patient is unconscious, step "*request patient name*" will throw an exception *PatientUnconsious*. The handler for this exception is a simple *continue* handler, which means that the event "*verify ID band and patient's stated name and birth date match*" will be occur even without obtaining patient's name and birthday.

- Change: more detailed "*perform Blood Specimen Obtaining process*" sub-process is elicited.

Iteration 2:

Three sets of binding are verified.

- Event Binding a:

verify ID band and patient's stated name and birth date match →

confirm patient-stated-first-name matches patient-first-name on ID band STARTED

obtain patient's stated name and birth date →

ask patient to spell first and last names COMPLETED

receive order to obtain a specimen →

order test(s) COMPLETED

obtain specimen →

obtain blood specimen STARTED

- Event Binding b:

verify ID band and patient's stated name and birth date match →

confirm patient-stated-last-name matches patient-last-name on ID band STARTED

obtain patient's stated name and birth date →

ask patient to spell first and last names COMPLETED

receive order to obtain a specimen →

order test(s) COMPLETED

obtain specimen →

obtain blood specimen STARTED

- Event Binding c:

confirm presence of ID band →

confirm patient-stated-DOB matches patient-DOB on ID band COMPLETED

obtain patient's stated name and birth date →

ask patient to state DOB STARTED

receive order to obtain a specimen →

  order test(s) COMPLETED

obtain specimen →

  obtain blood specimen STARTED

- Result: no violation found.

D.6c (D.3 → D.4) Verify ID band matches patient's stated name and birth date before verifying order to obtain a specimen

Iteration 1:

- Event Binding: could not find a step related to event "*verifying order to obtain a specimen*".

D.6d (D.4 → D.5) Verify order to obtain a specimen before verifying specimen container label

Iteration 1:

- Event Binding: could not find a step related to event "*verifying order to obtain a specimen*".

D.7 Verify order to obtain a specimen before applying a specimen container label

Iteration 1:

- Event Binding: could not find a step related to the event "*verify ID band and order to obtain specimen match*".

D.8 Verify specimen container label before applying a specimen container label

Iteration 1:

- Event Binding: could not find a step related to the event "*verify ID band and specimen container label match*".

- Change: more detailed "*perform Blood Specimen Obtaining process*" sub-process is elicited.

Iteration 2:

- Event Binding:

  send specimen to blood bank →
  send specimen to lab via tube system STARTED
  receive order to obtain a specimen →
  order test(s) COMPLETED
  apply specimen container label →
  apply label STARTED
  verify ID band and order to obtain a specimen match →
  verify labels COMPLETED

- Result: no violation found.

D.9 Verify ID band matches order to obtain a specimen before verifying specimen container label (before applying a specimen container label)

Iteration 1:

- Event Binding: could not find a step related to the event "*Verify ID band matches order to obtain a specimen*".

D.10a Nothing can occur between obtaining a specimen and applying a specimen container label

- Discussion: cannot verify this kind of property.

D.10b Nothing can occur between applying a specimen container label and obtaining a specimen

- Discussion: cannot verify this kind of property.

D.11 After obtaining a specimen, must apply specimen container label if it hasn't been done yet

Iteration 1:

- Event Binding: could not find a step related to the event "*discover specimen container is not yet labeled*".

APPENDIX F

BLOOD TRANSFUSION PROCESS FAULT TREE

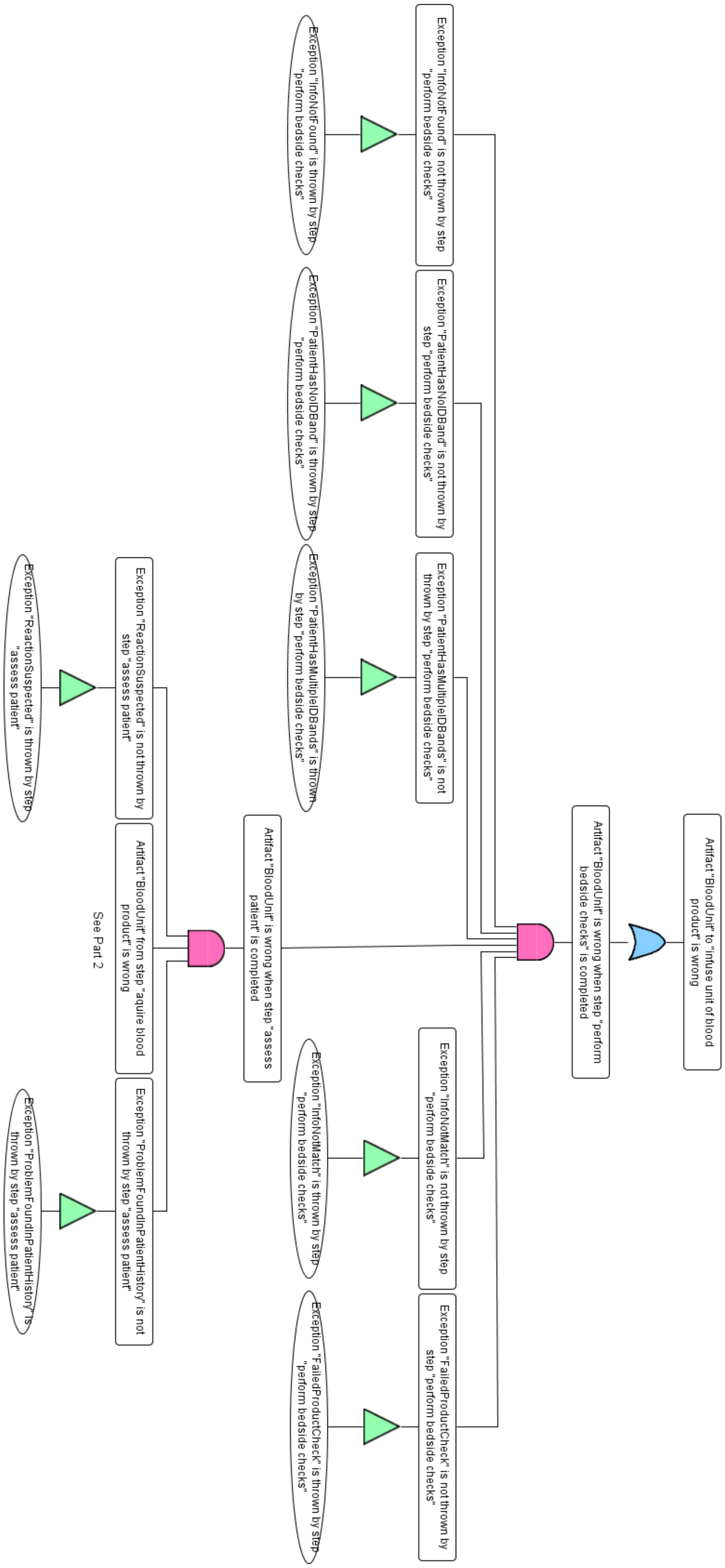

Artifact "BloodUnit" to "infuse unit of blood product" is wrong
Artifact "BloodUnit" is wrong when step "perform bedside checks" is completed
Exception "InfoNotFound" is not thrown by step "perform bedside checks"
Exception "InfoNotFound" is thrown by step "perform bedside checks"
Exception "PatientHasNoIDBand" is not thrown by step "perform bedside checks"
Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks"
Exception "PatientHasMultipleIDBands" is not thrown by step "perform bedside checks"
Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"
Artifact "BloodUnit" is wrong when step "assess patient" is completed
Exception "ReactionSuspected" is not thrown by step "assess patient"
Exception "ReactionSuspected" is thrown by step "assess patient"
Artifact "BloodUnit" from step "aquire blood product" is wrong
See Part 2
Exception "ProblemFoundInPatientHistory" is not thrown by step "assess patient"
Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"
Exception "InfoNotMatch" is not thrown by step "perform bedside checks"
Exception "InfoNotMatch" is thrown by step "perform bedside checks"
Exception "FailedProductCheck" is not thrown by step "perform bedside checks"
Exception "FailedProductCheck" is thrown by step "perform bedside checks"

Figure F.1. Blood Transfusion Process Fault Tree Part 1

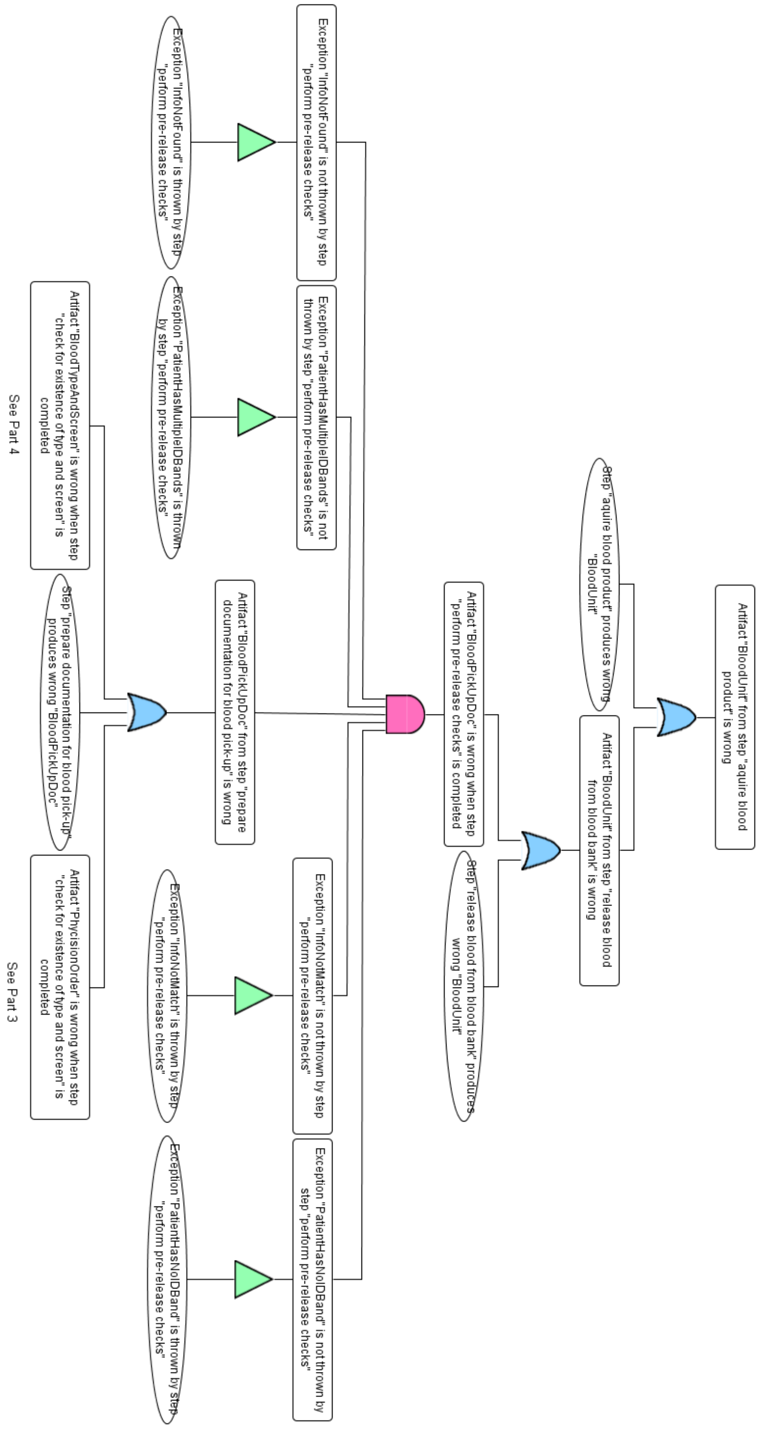

Artifact "BloodUnit" from step "aquire blood product" is wrong
Step "aquire blood product" produces wrong "BloodUnit"
Artifact"BloodUnit" from step "release blood from blood bank" is wrong
Artifact "BloodPickUpDoc" is wrong when step "perform pre-release checks" is completed
Step "release blood from blood bank" produces wrong "BloodUnit"
Exception "InfoNotFound" is not thrown by step "perform pre-release checks"
Exception "InfoNotFound" is thrown by step "perform pre-release checks"
Exception "PatientHasMultipleIDBands" is not thrown by step "perform pre-release checks"
Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"
Artifact "BloodPickUpDoc" from step "prepare documentation for blood pick-up" is wrong
Exception "InfoNotMatch" is not thrown by step "perform pre-release checks"
Exception "InfoNotMatch" is thrown by step "perform pre-release checks"
Exception "PatientHasNoIDBand" is not thrown by step "perform pre-release checks"
Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"
Artifact "BloodTypeAndScreen" is wrong when step "check for existence of type and screen" is completed
See Part 4
Step "prepare documentation for blood pick-up" produces wrong "BloodPickUpDoc"
Artifact "PhycisionOrder" is wrong when step "check for existence of type and screen" is completed
See Part 3

Figure F.2. Blood Transfusion Process Fault Tree Part 2

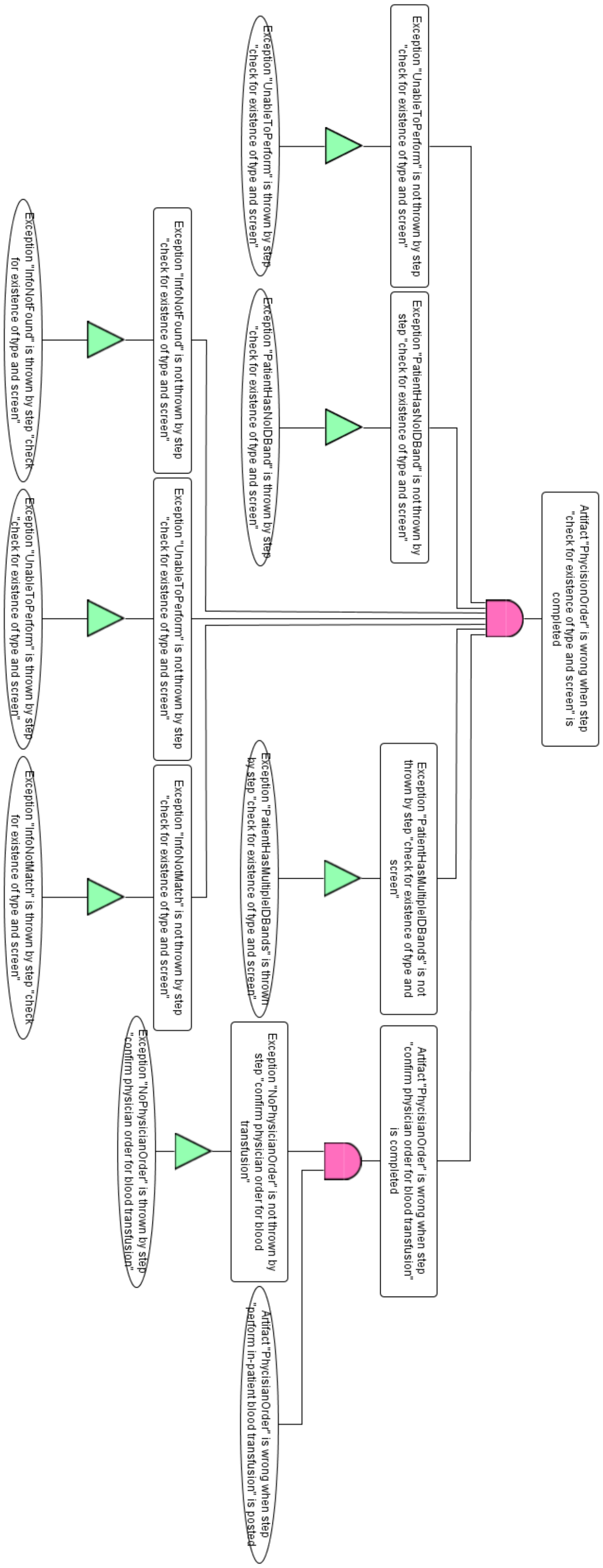

Artifact "PhycisionOrder" is wrong when step "check for existence of type and screen" is completed
Exception "UnableToPerform" is not thrown by step "check for existence of type and screen"
Exception "UnableToPerform" is thrown by step "check for existence of type and screen"
Exception "PatientHasNoIDBand" is not thrown by step "check for existence of type and screen"
Exception "PatientHasNoIDBand" is thrown by step "check for existence of type and screen"
Exception "InfoNotFound" is not thrown by step "check for existence of type and screen"
Exception "InfoNotFound" is thrown by step "check for existence of type and screen"
Exception "UnableToPerform" is not thrown by step "check for existence of type and screen"
Exception "UnableToPerform" is thrown by step "check for existence of type and screen"
Exception "PatientHasMultipleIDBands" is not thrown by step "check for existence of type and screen"
Exception "PatientHasMultipleIDBands" is thrown by step "check for existence of type and screen"
Exception "InfoNotMatch" is not thrown by step "check for existence of type and screen"
Exception "InfoNotMatch" is thrown by step "check for existence of type and screen"
Artifact "PhysicianOrder" is wrong when step "confirm physician order for blood transfusion" is completed
Exception "NoPhysicianOrder" is not thrown by step "confirm physician order for blood transfusion"
Exception "NoPhysicianOrder" is thrown by step "confirm physician order for blood transfusion"
Artifact "PhycisianOrder" is wrong when step "perform in-patient blood transfusion" is posted

Figure F.3. Blood Transfusion Process Fault Tree Part 3

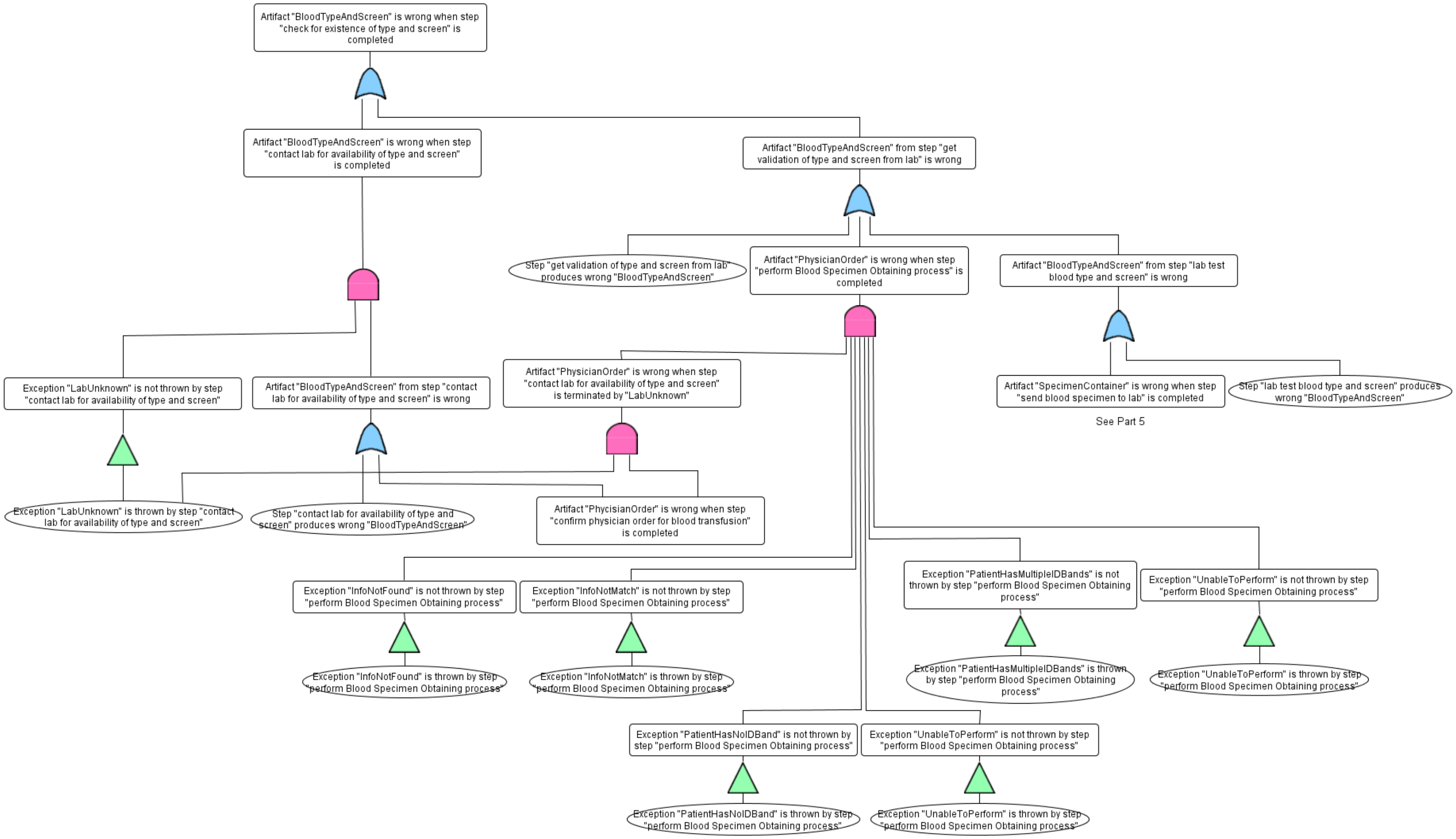

Artifact "BloodTypeAndScreen" is wrong when step "check for existence of type and screen" is completed
Artifact "BloodTypeAndScreen" is wrong when step "contact lab for availability of type and screen" is completed
Artifact "BloodTypeAndScreen" from step "get validation of type and screen from lab" is wrong
Step "get validation of type and screen from lab" produces wrong "BloodTypeAndScreen"
Artifact "PhysicianOrder" is wrong when step "perform Blood Specimen Obtaining process" is completed
Artifact "BloodTypeAndScreen" from step "lab test blood type and screen" is wrong
Exception "LabUnknown" is not thrown by step "contact lab for availability of type and screen"
Artifact "BloodTypeAndScreen" from step "contact lab for availability of type and screen" is wrong
Artifact "PhysicianOrder" is wrong when step "contact lab for availability of type and screen" is terminated by "LabUnknown"
Artifact "SpecimenContainer" is wrong when step "send blood specimen to lab" is completed
Step "lab test blood type and screen" produces wrong "BloodTypeAndScreen"
See Part 5
Exception "LabUnknown" is thrown by step "contact lab for availability of type and screen"
Step "contact lab for availability of type and screen" produces wrong "BloodTypeAndScreen"
Artifact "PhycisianOrder" is wrong when step "confirm physician order for blood transfusion" is completed
Exception "InfoNotFound" is not thrown by step "perform Blood Specimen Obtaining process"
Exception "InfoNotMatch" is not thrown by step "perform Blood Specimen Obtaining process"
Exception "PatientHasMultipleIDBands" is not thrown by step "perform Blood Specimen Obtaining process"
Exception "UnableToPerform" is not thrown by step "perform Blood Specimen Obtaining process"
Exception "InfoNotFound" is thrown by step "perform Blood Specimen Obtaining process"
Exception "InfoNotMatch" is thrown by step "perform Blood Specimen Obtaining process"
Exception "PatientHasMultipleIDBands" is thrown by step "perform Blood Specimen Obtaining process"
Exception "UnableToPerform" is thrown by step "perform Blood Specimen Obtaining process"
Exception "PatientHasNoIDBand" is not thrown by step "perform Blood Specimen Obtaining process"
Exception "UnableToPerform" is not thrown by step "perform Blood Specimen Obtaining process"
Exception "PatientHasNoIDBand" is thrown by step "perform Blood Specimen Obtaining process"
Exception "UnableToPerform" is thrown by step "perform Blood Specimen Obtaining process"

Figure F.4. Blood Transfusion Process Fault Tree Part 4

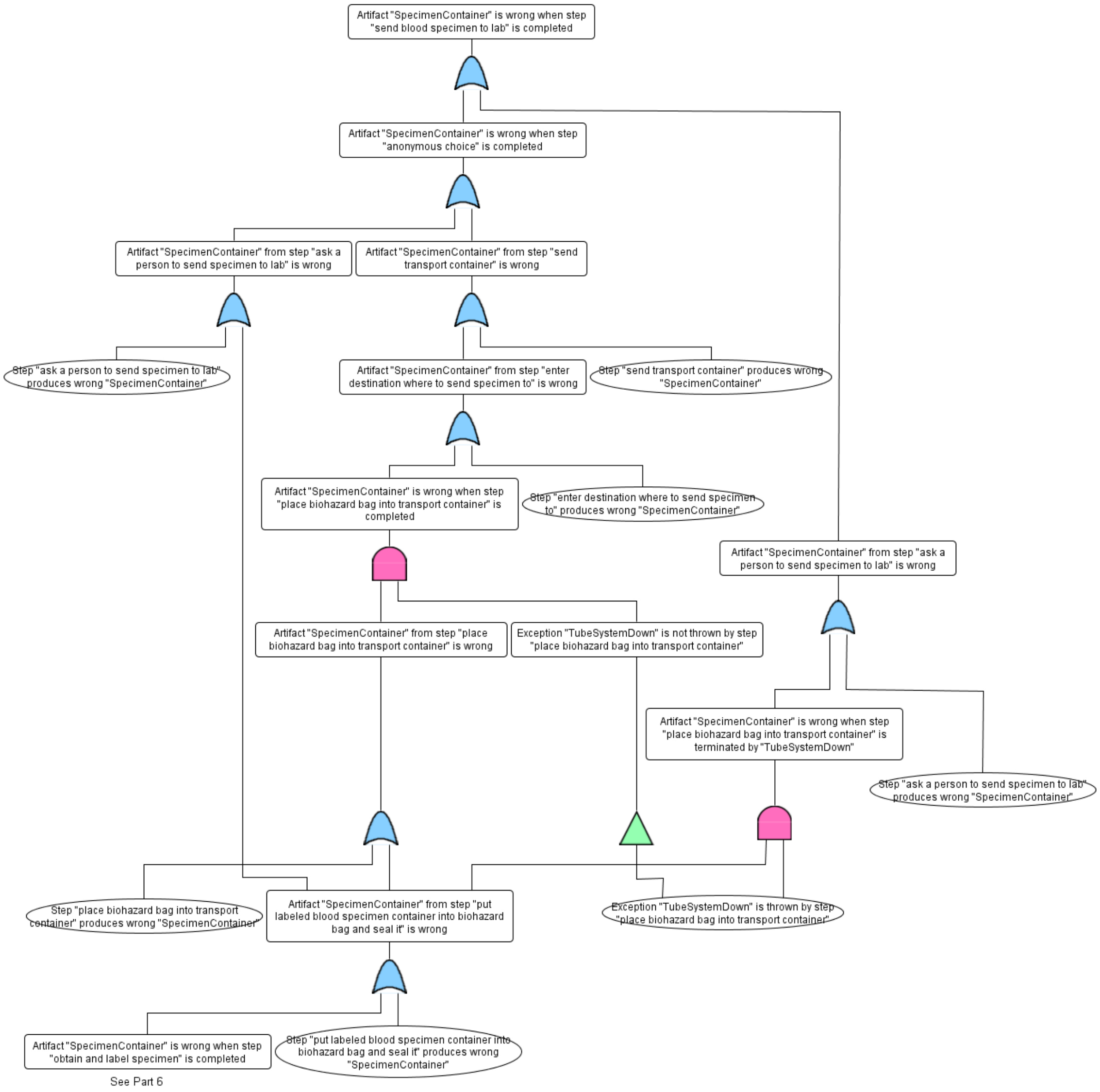

Artifact "SpecimenContainer" is wrong when step "send blood specimen to lab" is completed
Artifact "SpecimenContainer" is wrong when step "anonymous choice" is completed
Artifact "SpecimenContainer" from step "ask a person to send specimen to lab" is wrong
Artifact "SpecimenContainer" from step "send transport container" is wrong
Step "ask a person to send specimen to lab" produces wrong "SpecimenContainer"
Artifact "SpecimenContainer" from step "enter destination where to send specimen to" is wrong
Step "send transport container" produces wrong "SpecimenContainer"
Artifact "SpecimenContainer" is wrong when step "place biohazard bag into transport container" is completed
Step "enter destination where to send specimen to" produces wrong "SpecimenContainer"
Artifact "SpecimenContainer" from step "ask a person to send specimen to lab" is wrong
Artifact "SpecimenContainer" from step "place biohazard bag into transport container" is wrong
Exception "TubeSystemDown" is not thrown by step "place biohazard bag into transport container"
Artifact "SpecimenContainer" is wrong when step "place biohazard bag into transport container" is terminated by "TubeSystemDown"
Step "ask a person to send specimen to lab" produces wrong "SpecimenContainer"
Step "place biohazard bag into transport container" produces wrong "SpecimenContainer"
Artifact "SpecimenContainer" from step "put labeled blood specimen container into biohazard bag and seal it" is wrong
Exception "TubeSystemDown" is thrown by step "place biohazard bag into transport container"
Artifact "SpecimenContainer" is wrong when step "obtain and label specimen" is completed
See Part 6
Step "put labeled blood specimen container into biohazard bag and seal it" produces wrong "SpecimenContainer"

Figure F.5. Blood Transfusion Process Fault Tree Part 5

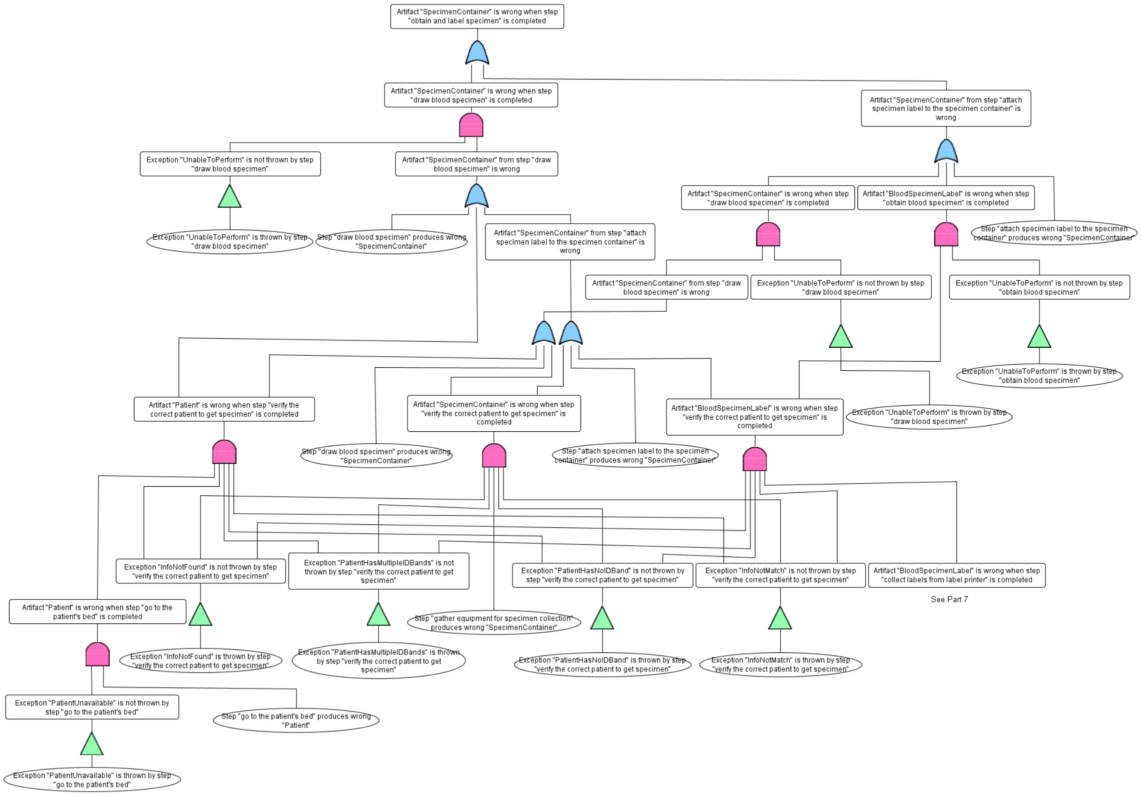

Artifact "SpecimenContainer" is wrong when step "obtain and label specimen" is completed
Artifact "SpecimenContainer" is wrong when step "draw blood specimen" is completed
Artifact "SpecimenContainer" from step "attach specimen label to the specimen container" is wrong
Exception "UnableToPerform" is not thrown by step "draw blood specimen"
Artifact "SpecimenContainer" from step "draw blood specimen" is wrong
Artifact "SpecimenContainer" is wrong when step "draw blood specimen" is completed
Artifact "BloodSpecimenLabel" is wrong when step "obtain blood specimen" is completed
Exception "UnableToPerform" is thrown by step "draw blood specimen"
Step "draw blood specimen" produces wrong "SpecimenContainer"
Artifact "SpecimenContainer" from step "attach specimen label to the specimen container" is wrong
Step "attach specimen label to the specimen container" produces wrong "SpecimenContainer"
Artifact "SpecimenContainer" from step "draw blood specimen" is wrong
Exception "UnableToPerform" is not thrown by step "draw blood specimen"
Exception "UnableToPerform" is not thrown by step "obtain blood specimen"
Exception "UnableToPerform" is thrown by step "obtain blood specimen"
Artifact "Patient" is wrong when step "verify the correct patient to get specimen" is completed
Artifact "SpecimenContainer" is wrong when step "verify the correct patient to get specimen" is completed
Artifact "BloodSpecimenLabel" is wrong when step "verify the correct patient to get specimen" is completed
Exception "UnableToPerform" is thrown by step "draw blood specimen"
Step "draw blood specimen" produces wrong "SpecimenContainer"
Step "attach specimen label to the specimen container" produces wrong "SpecimenContainer"
Exception "InfoNotFound" is not thrown by step "verify the correct patient to get specimen"
Exception "PatientHasMultipleIDBands" is not thrown by step "verify the correct patient to get specimen"
Exception "PatientHasNoIDBand" is not thrown by step "verify the correct patient to get specimen"
Exception "InfoNotMatch" is not thrown by step "verify the correct patient to get specimen"
Artifact "BloodSpecimenLabel" is wrong when step "collect labels from label printer" is completed
See Part 7
Artifact "Patient" is wrong when step "go to the patient's bed" is completed
Step "gather equipment for specimen collection" produces wrong "SpecimenContainer"
Exception "InfoNotFound" is thrown by step "verify the correct patient to get specimen"
Exception "PatientHasMultipleIDBands" is thrown by step "verify the correct patient to get specimen"
Exception "PatientHasNoIDBand" is thrown by step "verify the correct patient to get specimen"
Exception "InfoNotMatch" is thrown by step "verify the correct patient to get specimen"
Exception "PatientUnavailable" is not thrown by step "go to the patient's bed"
Step "go to the patient's bed" produces wrong "Patient"
Exception "PatientUnavailable" is thrown by step "go to the patient's bed"

Figure F.6. Blood Transfusion Process Fault Tree Part 6

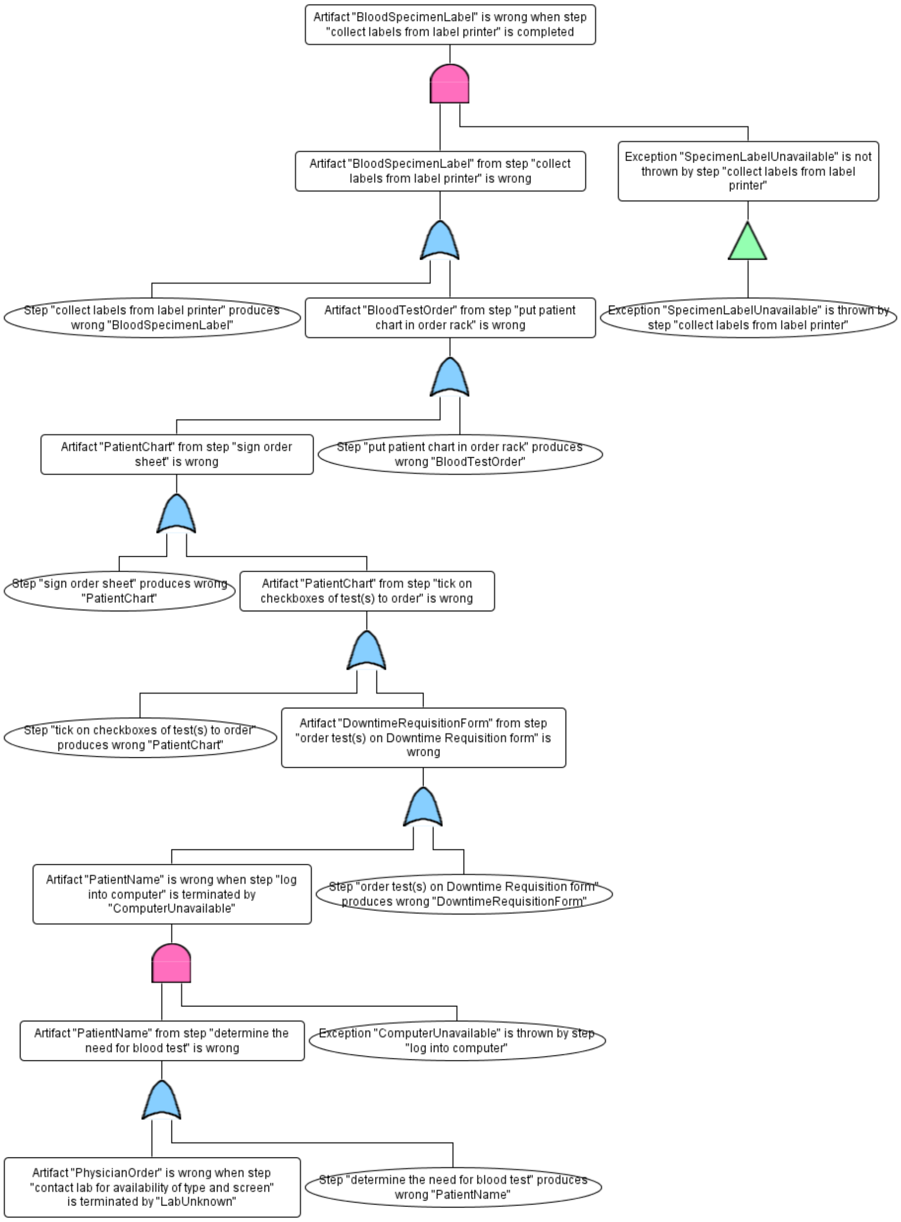

Artifact "BloodSpecimenLabel" is wrong when step "collect labels from label printer" is completed
Artifact "BloodSpecimenLabel" from step "collect labels from label printer" is wrong
Exception "SpecimenLabelUnavailable" is not thrown by step "collect labels from label printer"
Step "collect labels from label printer" produces wrong "BloodSpecimenLabel"
Artifact "BloodTestOrder" from step "put patient chart in order rack" is wrong
Exception "SpecimenLabelUnavailable" is thrown by step "collect labels from label printer"
Artifact "PatientChart" from step "sign order sheet" is wrong
Step "put patient chart in order rack" produces wrong "BloodTestOrder"
Step "sign order sheet" produces wrong "PatientChart"
Artifact "PatientChart" from step "tick on checkboxes of test(s) to order" is wrong
Step "tick on checkboxes of test(s) to order" produces wrong "PatientChart"
Artifact "DowntimeRequisitionForm" from step "order test(s) on Downtime Requisition form" is wrong
Artifact "PatientName" is wrong when step "log into computer" is terminated by "ComputerUnavailable"
Step "order test(s) on Downtime Requisition form" produces wrong "DowntimeRequisitionForm"
Artifact "PatientName" from step "determine the need for blood test" is wrong
Exception "ComputerUnavailable" is thrown by step "log into computer"
Artifact "PhysicianOrder" is wrong when step "contact lab for availability of type and screen" is terminated by "LabUnknown"
Step "determine the need for blood test" produces wrong "PatientName"

Figure F.7. Blood Transfusion Process Fault Tree Part 7

APPENDIX G

BLOOD TRANSFUSION PROCESS MCSS

Total Number of MCSs: 37

MCS 1 (8 events):

{

Step "aquire blood product" produces wrong "BloodUnit",

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 2 (8 events):

{

Step "release blood from blood bank" produces wrong "BloodUnit",

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step

"assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 3 (12 events):

{

Step "prepare documentation for blood pick-up" produces wrong "BloodPickUpDoc",

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)

}

MCS 4 (12 events):

{

Step “get validation of type and screen from lab” produces wrong “BloodTypeAndScreen”,

!(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),

!(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform pre-release checks”),

!(Exception “ReactionSuspected” is thrown by step “assess patient”),

!(Exception “ProblemFoundInPatientHistory” is thrown by step “assess patient”),

!(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),

!(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform bedside checks”),

!(Exception “InfoNotFound” is thrown by step “perform bedside checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)

}

MCS 5 (12 events):

{

Step “lab test blood type and screen” produces wrong “BloodTypeAndScreen”,

!(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),

!(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform pre-release checks”),

!(Exception “ReactionSuspected” is thrown by step “assess patient”),

!(Exception “ProblemFoundInPatientHistory” is thrown by step “assess patient”),

!(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),

!(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform bedside checks”),

!(Exception “InfoNotFound” is thrown by step “perform bedside checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)

}

MCS 6 (12 events):

{

Step “ask a person to send specimen to lab” produces wrong “SpecimenContainer”,

!(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),

!(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform pre-release checks”),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 7 (12 events):

{

Step "put labeled blood specimen container into biohazard bag and seal it" produces wrong "SpecimenContainer",

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform

bedside checks”),

!(Exception “InfoNotFound” is thrown by step “perform bedside checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)

}

MCS 8 (12 events):

{

Step “attach specimen label to the specimen container” produces wrong “SpecimenContainer”,

!(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),

!(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform pre-release checks”),

!(Exception “ReactionSuspected” is thrown by step “assess patient”),

!(Exception “ProblemFoundInPatientHistory” is thrown by step “assess patient”),

!(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),

!(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform bedside checks”),

!(Exception “InfoNotFound” is thrown by step “perform bedside checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)

}

MCS 9 (12 events):

{

Step "send transport container" produces wrong "SpecimenContainer",

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 10 (12 events):

{

Step "enter destination where to send specimen to" produces wrong "SpecimenContainer",

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step

"perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 11 (12 events):

{

Step "ask a person to send specimen to lab" produces wrong "SpecimenContainer",

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 12 (13 events):

{

Step "contact lab for availability of type and screen" produces wrong "BloodTypeAndScreen",

!(Exception "LabUnknown" is thrown by step "contact lab for availability of type and screen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 13 (13 events):

{

Step "draw blood specimen" produces wrong "SpecimenContainer",

!(Exception "UnableToPerform" is thrown by step "draw blood specimen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 14 (13 events):

{

Step “attach specimen label to the specimen container” produces wrong “SpecimenContainer”,

!(Exception “UnableToPerform” is thrown by step “draw blood specimen”),

!(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),

!(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform pre-release checks”),

!(Exception “ReactionSuspected” is thrown by step “assess patient”),

!(Exception “ProblemFoundInPatientHistory” is thrown by step “assess patient”),

!(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),

!(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform bedside checks”),

!(Exception “InfoNotFound” is thrown by step “perform bedside checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)

}

MCS 15 (13 events):

{

Step “draw blood specimen” produces wrong “SpecimenContainer”,

!(Exception “UnableToPerform” is thrown by step “draw blood specimen”),

!(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),

!(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step

"perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 16 (13 events):

{

Step "place biohazard bag into transport container" produces wrong "SpecimenContainer",

!(Exception "TubeSystemDown" is thrown by step "place biohazard bag into transport container"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception “ProblemFoundInPatientHistory” is thrown by step “assess patient”),

!(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),

!(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform bedside checks”),

!(Exception “InfoNotFound” is thrown by step “perform bedside checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)

}

MCS 17 (14 events):

{

Artifact “PhycisianOrder” is wrong when step “perform in-patient blood transfusion” is posted,

!(Exception “NoPhysicianOrder” is thrown by step “confirm physician order for blood transfusion”),

!(Exception “LabUnknown” is thrown by step “contact lab for availability of type and screen”),

!(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),

!(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform pre-release checks”),

!(Exception “ReactionSuspected” is thrown by step “assess patient”),

!(Exception “ProblemFoundInPatientHistory” is thrown by step “assess patient”),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 18 (17 events):

{

Step "gather equipment for specimen collection" produces wrong "SpecimenContainer",

!(Exception "InfoNotMatch" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "verify the correct patient to get specimen"),

!(Exception "InfoNotFound" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasNoIDBand" is thrown by step "verify the correct patient to get specimen"),

!(Exception "UnableToPerform" is thrown by step "draw blood specimen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 19 (17 events):

{

Step "gather equipment for specimen collection" produces wrong "SpecimenContainer",

!(Exception "InfoNotMatch" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "verify the correct patient to get specimen"),

!(Exception "InfoNotFound" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasNoIDBand" is thrown by step "verify the correct patient to get specimen"),

!(Exception "UnableToPerform" is thrown by step "draw blood specimen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step

"perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 20 (18 events):

{

Step "collect labels from label printer" produces wrong "BloodSpecimenLabel",

!(Exception "SpecimenLabelUnavailable" is thrown by step "collect labels from label printer"),

!(Exception "InfoNotMatch" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "verify the correct patient to get specimen"),

!(Exception "InfoNotFound" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasNoIDBand" is thrown by step "verify the correct patient to get specimen"),

!(Exception "UnableToPerform" is thrown by step "obtain blood specimen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 21 (18 events):

{

Step "order test(s) on Downtime Requisition form" produces wrong "DowntimeRequisitionForm",

!(Exception "SpecimenLabelUnavailable" is thrown by step "collect labels from label printer"),

!(Exception "InfoNotMatch" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "verify the

correct patient to get specimen”),

!(Exception “InfoNotFound” is thrown by step “verify the correct patient to get specimen”),

!(Exception “PatientHasNoIDBand” is thrown by step “verify the correct patient to get specimen”),

!(Exception “UnableToPerform” is thrown by step “obtain blood specimen”),

!(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),

!(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform pre-release checks”),

!(Exception “ReactionSuspected” is thrown by step “assess patient”),

!(Exception “ProblemFoundInPatientHistory” is thrown by step “assess patient”),

!(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),

!(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform bedside checks”),

!(Exception “InfoNotFound” is thrown by step “perform bedside checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)

}

MCS 22 (18 events):

{

Step “sign order sheet” produces wrong “PatientChart”,

!(Exception “SpecimenLabelUnavailable” is thrown by step “collect labels

```
        from label printer”),
    !(Exception “InfoNotMatch” is thrown by step “verify the correct patient
        to get specimen”),
    !(Exception “PatientHasMultipleIDBands” is thrown by step “verify the
        correct patient to get specimen”),
    !(Exception “InfoNotFound” is thrown by step “verify the correct patient
        to get specimen”),
    !(Exception “PatientHasNoIDBand” is thrown by step “verify the
        correct patient to get specimen”),
    !(Exception “UnableToPerform” is thrown by step “obtain blood specimen”),
    !(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),
    !(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),
    !(Exception “PatientHasMultipleIDBands” is thrown by step
        “perform pre-release checks”),
    !(Exception “PatientHasNoIDBand” is thrown by step “perform
        pre-release checks”),
    !(Exception “ReactionSuspected” is thrown by step “assess patient”),
    !(Exception “ProblemFoundInPatientHistory” is thrown by step
        “assess patient”),
    !(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),
    !(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),
    !(Exception “PatientHasMultipleIDBands” is thrown by step “perform
        bedside checks”),
    !(Exception “InfoNotFound” is thrown by step “perform bedside checks”),
    !(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)
}
```

MCS 23 (18 events):

{

Step "tick on checkboxes of test(s) to order" produces wrong "PatientChart",

!(Exception "SpecimenLabelUnavailable" is thrown by step "collect labels from label printer"),

!(Exception "InfoNotMatch" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "verify the correct patient to get specimen"),

!(Exception "InfoNotFound" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasNoIDBand" is thrown by step "verify the correct patient to get specimen"),

!(Exception "UnableToPerform" is thrown by step "draw blood specimen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 24 (18 events):

{

Step "put patient chart in order rack" produces wrong "BloodTestOrder",

!(Exception "SpecimenLabelUnavailable" is thrown by step "collect labels from label printer"),

!(Exception "InfoNotMatch" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "verify the correct patient to get specimen"),

!(Exception "InfoNotFound" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasNoIDBand" is thrown by step "verify the correct patient to get specimen"),

!(Exception "UnableToPerform" is thrown by step "obtain blood specimen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 25 (18 events):

{

Step "tick on checkboxes of test(s) to order" produces wrong "PatientChart",

!(Exception "SpecimenLabelUnavailable" is thrown by step "collect labels from label printer"),

!(Exception "InfoNotMatch" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "verify the correct patient to get specimen"),

!(Exception "InfoNotFound" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasNoIDBand" is thrown by step "verify the correct patient to get specimen"),

!(Exception "UnableToPerform" is thrown by step "obtain blood specimen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform

pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 26 (18 events):

{

Step "collect labels from label printer" produces wrong "BloodSpecimenLabel",

!(Exception "SpecimenLabelUnavailable" is thrown by step "collect labels from label printer"),

!(Exception "InfoNotMatch" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "verify the correct patient to get specimen"),

!(Exception "InfoNotFound" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasNoIDBand" is thrown by step "verify the correct patient to get specimen"),

!(Exception "UnableToPerform" is thrown by step "draw blood specimen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform pre-release checks”),

!(Exception “ReactionSuspected” is thrown by step “assess patient”),

!(Exception “ProblemFoundInPatientHistory” is thrown by step “assess patient”),

!(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),

!(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform bedside checks”),

!(Exception “InfoNotFound” is thrown by step “perform bedside checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)

}

MCS 27 (18 events):

{

Step “go to the patient’s bed” produces wrong “Patient”,

!(Exception “InfoNotMatch” is thrown by step “verify the correct patient to get specimen”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “verify the correct patient to get specimen”),

!(Exception “InfoNotFound” is thrown by step “verify the correct patient to get specimen”),

!(Exception “PatientHasNoIDBand” is thrown by step “verify the correct patient to get specimen”),

!(Exception "PatientUnavailable" is thrown by step "go to the patient's bed"),

!(Exception "UnableToPerform" is thrown by step "draw blood specimen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step
"perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform
pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step
"assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform
bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 28 (18 events):

{

Step "put patient chart in order rack" produces wrong "BloodTestOrder",

!(Exception "SpecimenLabelUnavailable" is thrown by step "collect labels
from label printer"),

!(Exception "InfoNotMatch" is thrown by step "verify the correct patient
to get specimen"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "verify the

correct patient to get specimen"),

!(Exception "InfoNotFound" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasNoIDBand" is thrown by step "verify the correct patient to get specimen"),

!(Exception "UnableToPerform" is thrown by step "draw blood specimen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 29 (18 events):

{

Step "go to the patient's bed" produces wrong "Patient",

!(Exception "InfoNotMatch" is thrown by step "verify the correct patient

```
        to get specimen”),
    !(Exception “PatientHasMultipleIDBands” is thrown by step “verify the
        correct patient to get specimen”),
    !(Exception “InfoNotFound” is thrown by step “verify the correct patient
        to get specimen”),
    !(Exception “PatientHasNoIDBand” is thrown by step “verify the
        correct patient to get specimen”),
    !(Exception “PatientUnavailable” is thrown by step “go to the patient’s bed”),
    !(Exception “UnableToPerform” is thrown by step “draw blood specimen”),
    !(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),
    !(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),
    !(Exception “PatientHasMultipleIDBands” is thrown by step
        “perform pre-release checks”),
    !(Exception “PatientHasNoIDBand” is thrown by step “perform
        pre-release checks”),
    !(Exception “ReactionSuspected” is thrown by step “assess patient”),
    !(Exception “ProblemFoundInPatientHistory” is thrown by step
        “assess patient”),
    !(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),
    !(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),
    !(Exception “PatientHasMultipleIDBands” is thrown by step “perform
        bedside checks”),
    !(Exception “InfoNotFound” is thrown by step “perform bedside checks”),
    !(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)
}
```

MCS 30 (18 events):

{

Step “sign order sheet” produces wrong “PatientChart”,

!(Exception “SpecimenLabelUnavailable” is thrown by step “collect labels from label printer”),

!(Exception “InfoNotMatch” is thrown by step “verify the correct patient to get specimen”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “verify the correct patient to get specimen”),

!(Exception “InfoNotFound” is thrown by step “verify the correct patient to get specimen”),

!(Exception “PatientHasNoIDBand” is thrown by step “verify the correct patient to get specimen”),

!(Exception “UnableToPerform” is thrown by step “draw blood specimen”),

!(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),

!(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform pre-release checks”),

!(Exception “ReactionSuspected” is thrown by step “assess patient”),

!(Exception “ProblemFoundInPatientHistory” is thrown by step “assess patient”),

!(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),

!(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform bedside checks”),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 31 (18 events):

{

Step "order test(s) on Downtime Requisition form" produces wrong "DowntimeRequisitionForm",

!(Exception "SpecimenLabelUnavailable" is thrown by step "collect labels from label printer"),

!(Exception "InfoNotMatch" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "verify the correct patient to get specimen"),

!(Exception "InfoNotFound" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasNoIDBand" is thrown by step "verify the correct patient to get specimen"),

!(Exception "UnableToPerform" is thrown by step "draw blood specimen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step

"assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 32 (19 events):

{

Artifact "PhycisianOrder" is wrong when step "perform in-patient blood transfusion" is posted,

!(Exception "NoPhysicianOrder" is thrown by step "confirm physician order for blood transfusion"),

!(Exception "InfoNotMatch" is thrown by step "perform Blood Specimen Obtaining process"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform Blood Specimen Obtaining process"),

!(Exception "InfoNotFound" is thrown by step "perform Blood Specimen Obtaining process"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform Blood Specimen Obtaining process"),

!(Exception "UnableToPerform" is thrown by step "perform Blood Specimen Obtaining process"),

!(Exception "UnableToPerform" is thrown by step "perform Blood Specimen Obtaining process"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 33 (19 events):

{

Exception "ComputerUnavailable" is thrown by step "log into computer",

Step "determine the need for blood test" produces wrong "PatientName",

!(Exception "SpecimenLabelUnavailable" is thrown by step "collect labels from label printer"),

!(Exception "InfoNotMatch" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "verify the correct patient to get specimen"),

!(Exception “InfoNotFound” is thrown by step “verify the correct patient to get specimen”),

!(Exception “PatientHasNoIDBand” is thrown by step “verify the correct patient to get specimen”),

!(Exception “UnableToPerform” is thrown by step “draw blood specimen”),

!(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),

!(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform pre-release checks”),

!(Exception “ReactionSuspected” is thrown by step “assess patient”),

!(Exception “ProblemFoundInPatientHistory” is thrown by step “assess patient”),

!(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),

!(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform bedside checks”),

!(Exception “InfoNotFound” is thrown by step “perform bedside checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)

}

MCS 34 (19 events):

{

Exception “ComputerUnavailable” is thrown by step “log into computer”,

Step “determine the need for blood test” produces wrong “PatientName”,

!(Exception “SpecimenLabelUnavailable” is thrown by step “collect labels

```
        from label printer”),
    !(Exception “InfoNotMatch” is thrown by step “verify the correct patient
        to get specimen”),
    !(Exception “PatientHasMultipleIDBands” is thrown by step “verify the
        correct patient to get specimen”),
    !(Exception “InfoNotFound” is thrown by step “verify the correct patient
        to get specimen”),
    !(Exception “PatientHasNoIDBand” is thrown by step “verify the
        correct patient to get specimen”),
    !(Exception “UnableToPerform” is thrown by step “obtain blood specimen”),
    !(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),
    !(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),
    !(Exception “PatientHasMultipleIDBands” is thrown by step
        “perform pre-release checks”),
    !(Exception “PatientHasNoIDBand” is thrown by step “perform
        pre-release checks”),
    !(Exception “ReactionSuspected” is thrown by step “assess patient”),
    !(Exception “ProblemFoundInPatientHistory” is thrown by step
        “assess patient”),
    !(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),
    !(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),
    !(Exception “PatientHasMultipleIDBands” is thrown by step “perform
        bedside checks”),
    !(Exception “InfoNotFound” is thrown by step “perform bedside checks”),
    !(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)
}
```

MCS 35 (19 events):

{

Artifact "PhycisianOrder" is wrong when step "perform in-patient blood transfusion" is posted,

!(Exception "NoPhysicianOrder" is thrown by step "confirm physician order for blood transfusion"),

!(Exception "InfoNotMatch" is thrown by step "check for existence of type and screen"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "check for existence of type and screen"),

!(Exception "InfoNotFound" is thrown by step "check for existence of type and screen"),

!(Exception "PatientHasNoIDBand" is thrown by step "check for existence of type and screen"),

!(Exception "UnableToPerform" is thrown by step "check for existence of type and screen"),

!(Exception "UnableToPerform" is thrown by step "check for existence of type and screen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception "InfoNotFound" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform pre-release checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform pre-release checks"),

!(Exception "ReactionSuspected" is thrown by step "assess patient"),

!(Exception "ProblemFoundInPatientHistory" is thrown by step "assess patient"),

!(Exception "FailedProductCheck" is thrown by step "perform bedside checks"),

!(Exception "InfoNotMatch" is thrown by step "perform bedside checks"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "perform bedside checks"),

!(Exception "InfoNotFound" is thrown by step "perform bedside checks"),

!(Exception "PatientHasNoIDBand" is thrown by step "perform bedside checks")

}

MCS 36 (20 events):

{

Exception "ComputerUnavailable" is thrown by step "log into computer",

Artifact "PhycisianOrder" is wrong when step "perform in-patient blood transfusion" is posted,

!(Exception "NoPhysicianOrder" is thrown by step "confirm physician order for blood transfusion"),

!(Exception "SpecimenLabelUnavailable" is thrown by step "collect labels from label printer"),

!(Exception "InfoNotMatch" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasMultipleIDBands" is thrown by step "verify the correct patient to get specimen"),

!(Exception "InfoNotFound" is thrown by step "verify the correct patient to get specimen"),

!(Exception "PatientHasNoIDBand" is thrown by step "verify the correct patient to get specimen"),

!(Exception "UnableToPerform" is thrown by step "obtain blood specimen"),

!(Exception "InfoNotMatch" is thrown by step "perform pre-release checks"),

!(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform pre-release checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform pre-release checks”),

!(Exception “ReactionSuspected” is thrown by step “assess patient”),

!(Exception “ProblemFoundInPatientHistory” is thrown by step “assess patient”),

!(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),

!(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),

!(Exception “PatientHasMultipleIDBands” is thrown by step “perform bedside checks”),

!(Exception “InfoNotFound” is thrown by step “perform bedside checks”),

!(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)

}

MCS 37 (20 events):

{

Exception “ComputerUnavailable” is thrown by step “log into computer”,

Artifact “PhycisianOrder” is wrong when step “perform in-patient blood transfusion” is posted,

!(Exception “NoPhysicianOrder” is thrown by step “confirm physician order for blood transfusion”),

!(Exception “SpecimenLabelUnavailable” is thrown by step “collect labels from label printer”),

!(Exception “InfoNotMatch” is thrown by step “verify the correct patient to get specimen”),

```
    !(Exception “PatientHasMultipleIDBands” is thrown by step “verify the
            correct patient to get specimen”),
    !(Exception “InfoNotFound” is thrown by step “verify the correct patient
            to get specimen”),
    !(Exception “PatientHasNoIDBand” is thrown by step “verify the
            correct patient to get specimen”),
    !(Exception “UnableToPerform” is thrown by step “draw blood specimen”),
    !(Exception “InfoNotMatch” is thrown by step “perform pre-release checks”),
    !(Exception “InfoNotFound” is thrown by step “perform pre-release checks”),
    !(Exception “PatientHasMultipleIDBands” is thrown by step
            “perform pre-release checks”),
    !(Exception “PatientHasNoIDBand” is thrown by step “perform
            pre-release checks”),
    !(Exception “ReactionSuspected” is thrown by step “assess patient”),
    !(Exception “ProblemFoundInPatientHistory” is thrown by step
            “assess patient”),
    !(Exception “FailedProductCheck” is thrown by step “perform bedside checks”),
    !(Exception “InfoNotMatch” is thrown by step “perform bedside checks”),
    !(Exception “PatientHasMultipleIDBands” is thrown by step “perform
            bedside checks”),
    !(Exception “InfoNotFound” is thrown by step “perform bedside checks”),
    !(Exception “PatientHasNoIDBand” is thrown by step “perform bedside checks”)
}
```

APPENDIX H

CHEMOTHERAPY PROCESS

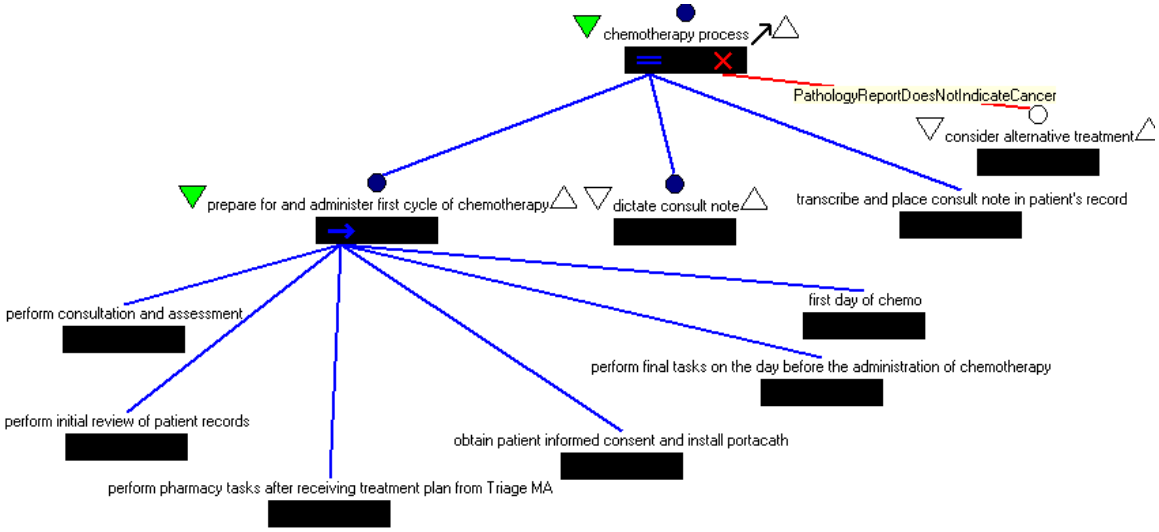

chemotherapy process
PathologyReportDoesNotIndicateCancer
consider alternative treatment
prepare for and administer first cycle of chemotherapy
dictate consult note
transcribe and place consult note in patient's record
perform consultation and assessment
first day of chemo
perform final tasks on the day before the administration of chemotherapy
perform initial review of patient records
obtain patient informed consent and install portacath
perform pharmacy tasks after receiving treatment plan from Triage MA

Figure H.1. Diagram "chemotherapy process"

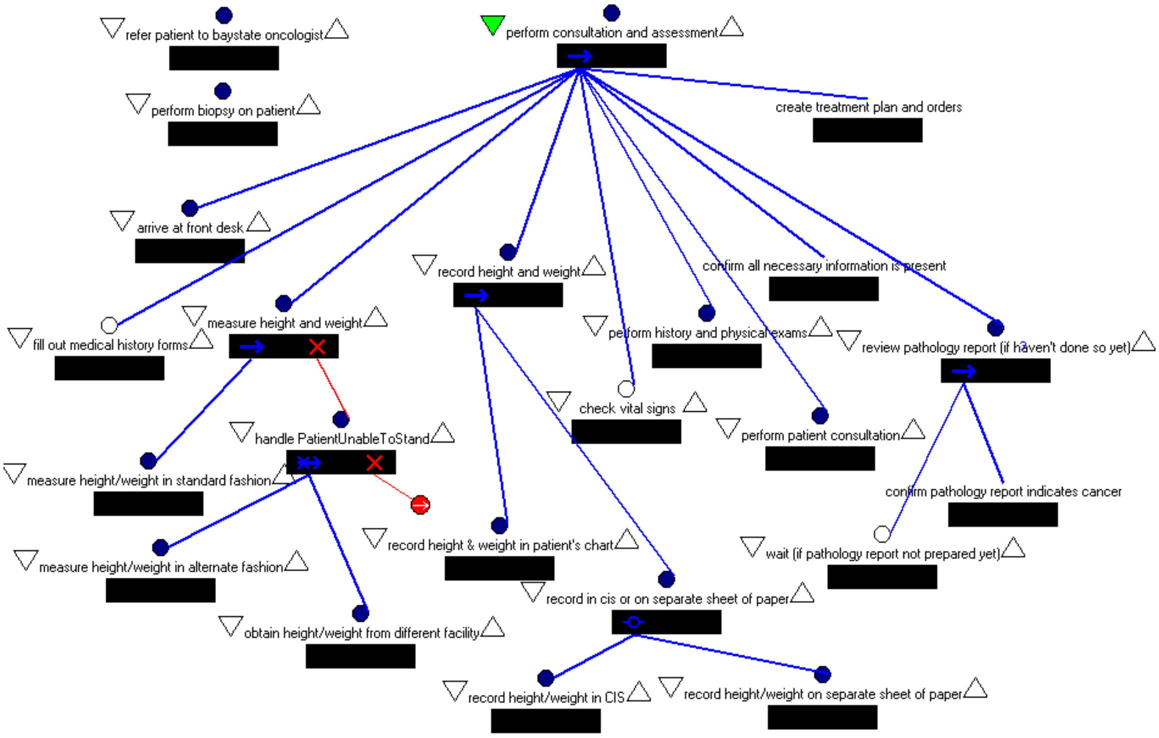

refer patient to baystate oncologist
perform consultation and assessment
perform biopsy on patient
create treatment plan and orders
arrive at front desk
record height and weight
confirm all necessary information is present
fill out medical history forms
measure height and weight
perform history and physical exams
review pathology report (if haven't done so yet)
check vital signs
handle PatientUnableToStand
perform patient consultation
measure height/weight in standard fashion
confirm pathology report indicates cancer
record height & weight in patient's chart
wait (if pathology report not prepared yet)
measure height/weight in alternate fashion
record in cis or on separate sheet of paper
obtain height/weight from different facility
record height/weight in CIS
record height/weight on separate sheet of paper

Figure H.2. Diagram "perform consultation and assessment"

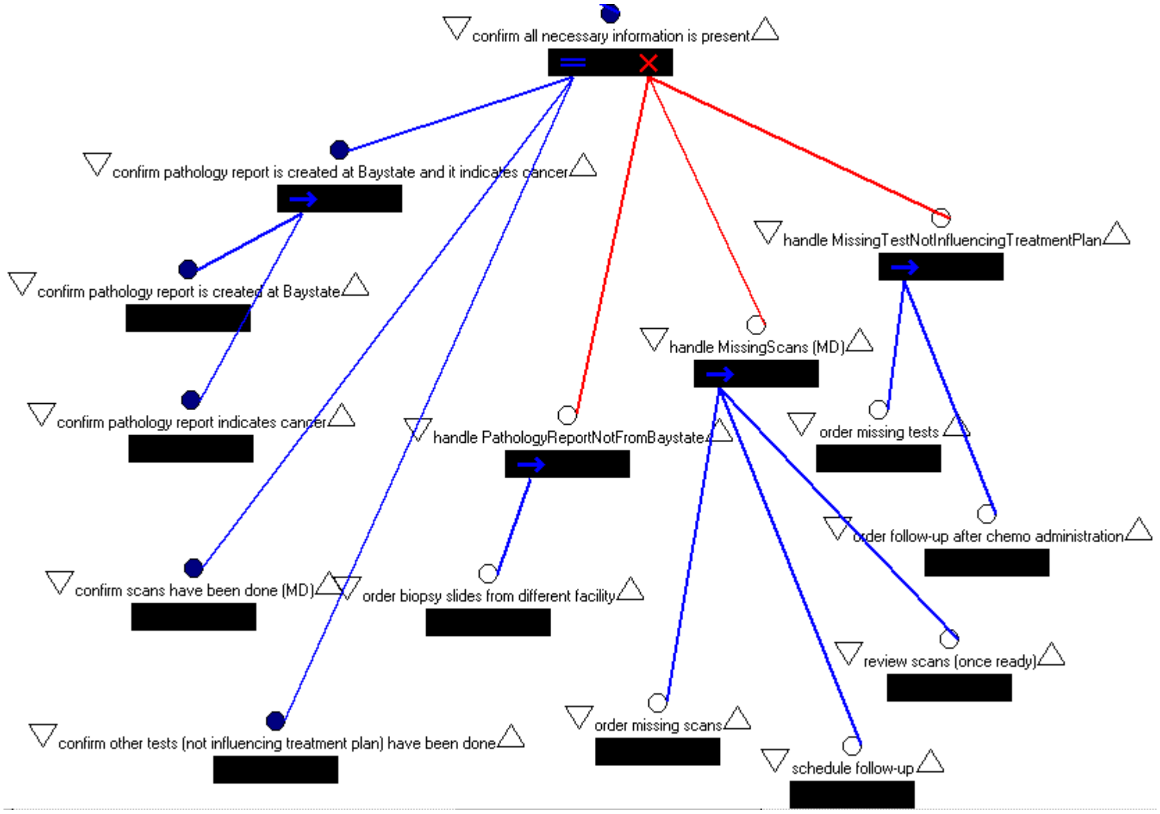

confirm all necessary information is present
confirm pathology report is created at Baystate and it indicates cancer
handle MissingTestNotInfluencingTreatmentPlan
confirm pathology report is created at Baystate
handle MissingScans (MD)
confirm pathology report indicates cancer
handle PathologyReportNotFromBaystate
order missing tests
order follow-up after chemo administration
confirm scans have been done (MD)
order biopsy slides from different facility
review scans (once ready)
order missing scans
confirm other tests (not influencing treatment plan) have been done
schedule follow-up

Figure H.3. Diagram "confirm all necessary information is present"

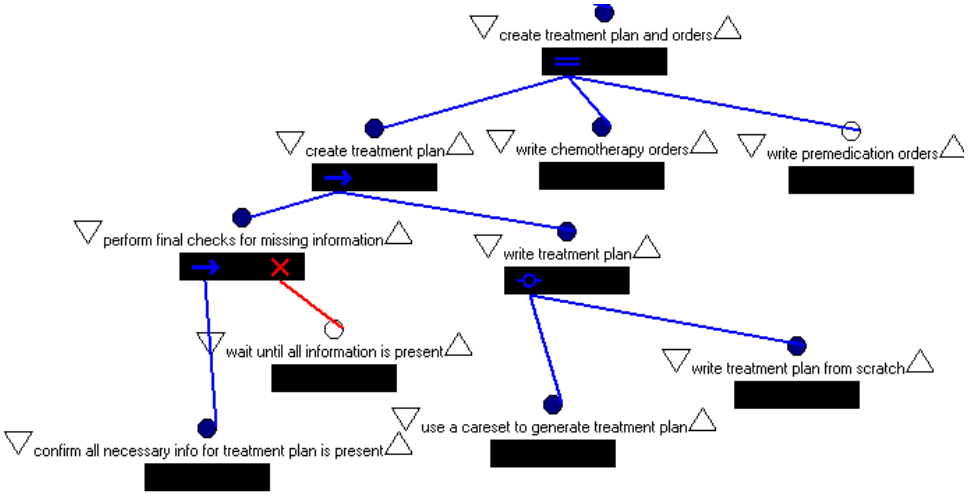

create treatment plan and orders
create treatment plan
write chemotherapy orders
write premedication orders
perform final checks for missing information
write treatment plan
wait until all information is present
write treatment plan from scratch
use a careset to generate treatment plan
confirm all necessary info for treatment plan is present

Figure H.4. Diagram "create treatment plan and orders"

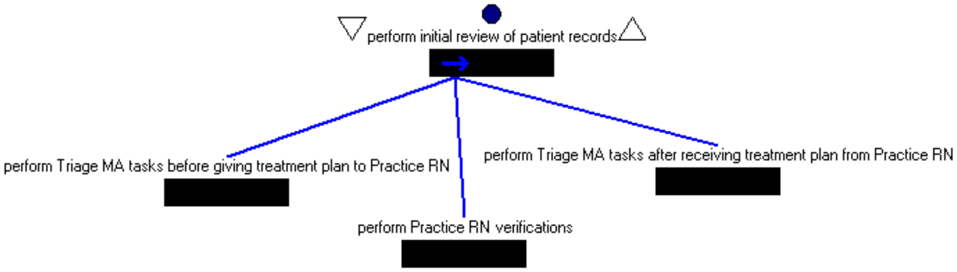

perform initial review of patient records
perform Triage MA tasks before giving treatment plan to Practice RN
perform Triage MA tasks after receiving treatment plan from Practice RN
perform Practice RN verifications

Figure H.5. Diagram "perform initial review of patient records"

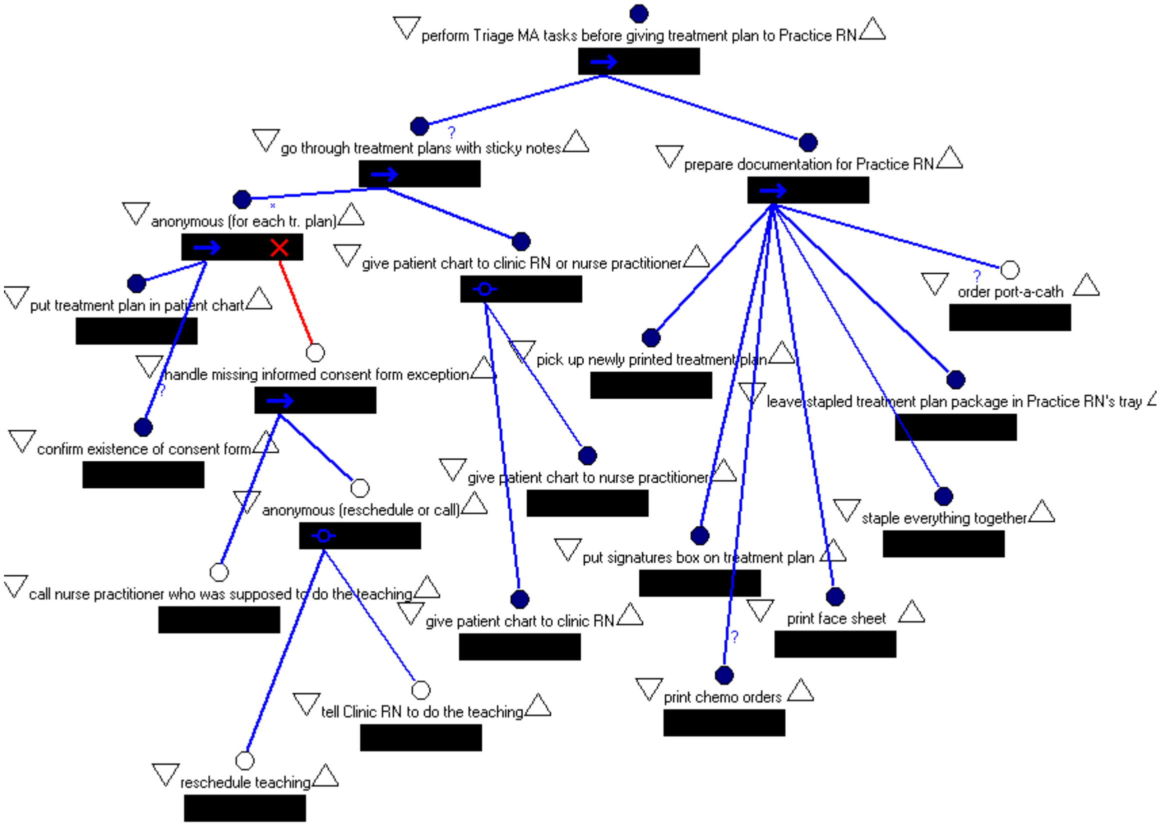

perform Triage MA tasks before giving treatment plan to Practice RN
?
go through treatment plans with sticky notes
prepare documentation for Practice RN
anonymous (for each tr. plan)
give patient chart to clinic RN or nurse practitioner
?
order port-a-cath
put treatment plan in patient chart
pick up newly printed treatment plan
handle missing informed consent form exception
leave stapled treatment plan package in Practice RN's tray
?
confirm existence of consent form
give patient chart to nurse practitioner
anonymous (reschedule or call)
staple everything together
put signatures box on treatment plan
call nurse practitioner who was supposed to do the teaching
give patient chart to clinic RN
print face sheet
?
print chemo orders
tell Clinic RN to do the teaching
reschedule teaching

Figure H.6. Diagram "perform Triage MA tasks before giving treatment plan to Practice RN"

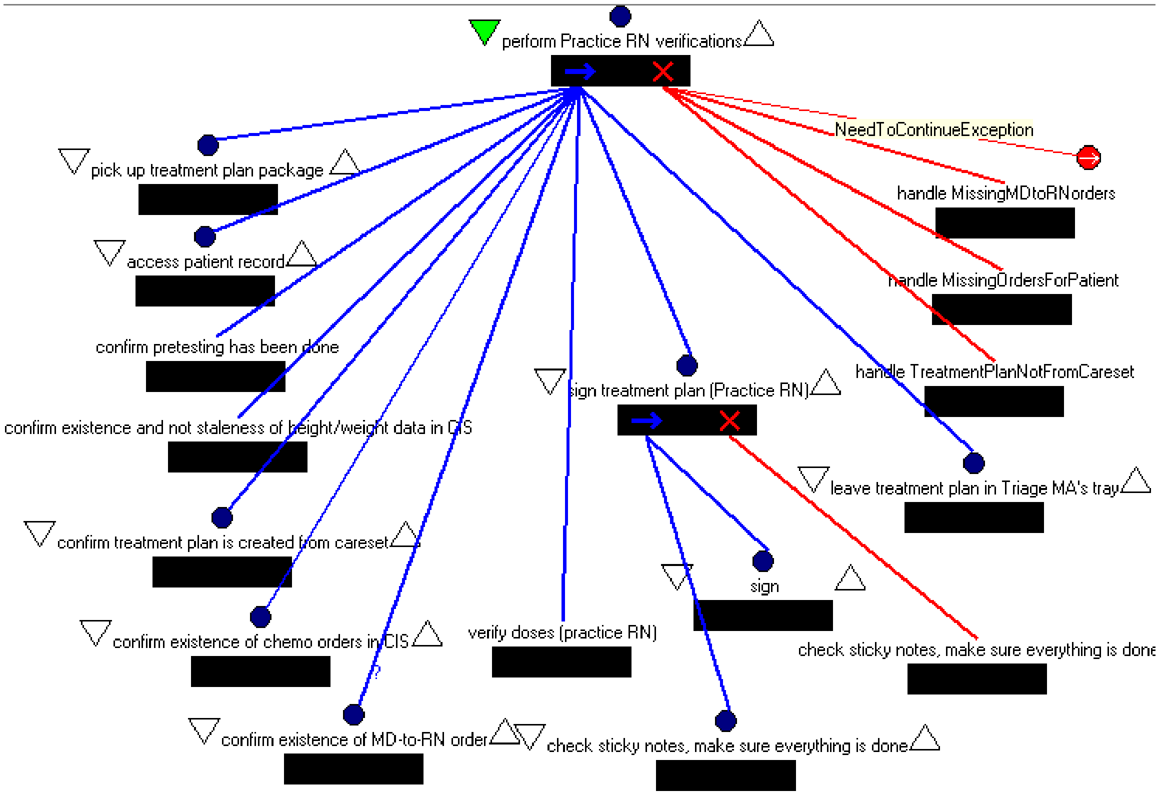

perform Practice RN verifications
NeedToContinueException
pick up treatment plan package
handle MissingMDtoRNorders
access patient record
handle MissingOrdersForPatient
confirm pretesting has been done
handle TreatmentPlanNotFromCareset
sign treatment plan (Practice RN)
confirm existence and not staleness of height/weight data in CIS
leave treatment plan in Triage MA's tray
confirm treatment plan is created from careset
sign
confirm existence of chemo orders in CIS
verify doses (practice RN)
check sticky notes, make sure everything is done
confirm existence of MD-to-RN order
check sticky notes, make sure everything is done

Figure H.7. Diagram "perform Practice RN verifications"

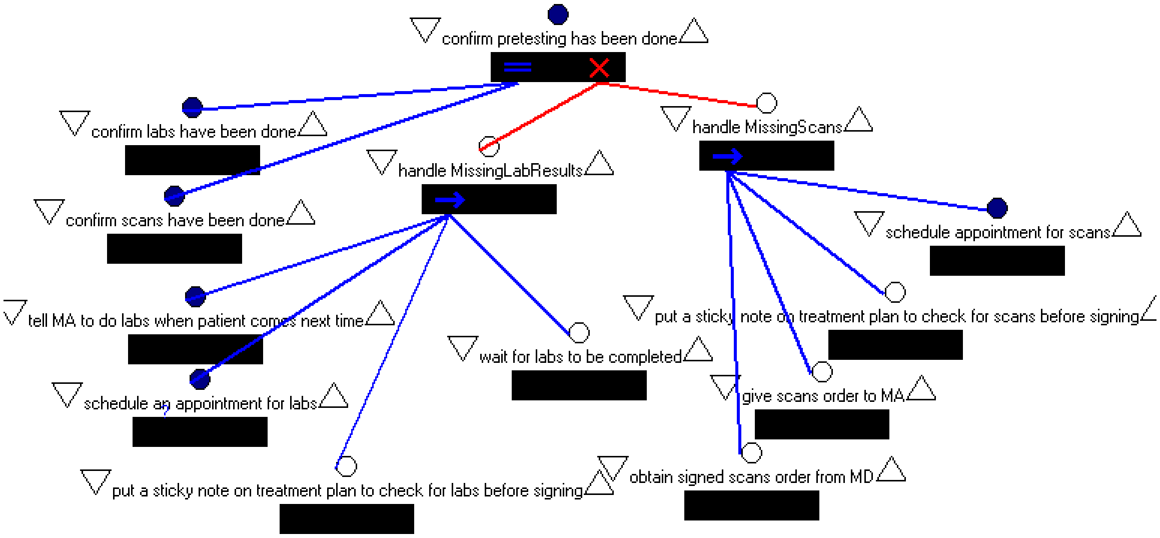

confirm pretesting has been done
confirm labs have been done
handle MissingScans
handle MissingLabResults
confirm scans have been done
schedule appointment for scans
tell MA to do labs when patient comes next time
put a sticky note on treatment plan to check for scans before signing
wait for labs to be completed
schedule an appointment for labs
give scans order to MA
put a sticky note on treatment plan to check for labs before signing
obtain signed scans order from MD

Figure H.8. Diagram "confirm pretesting has been done"

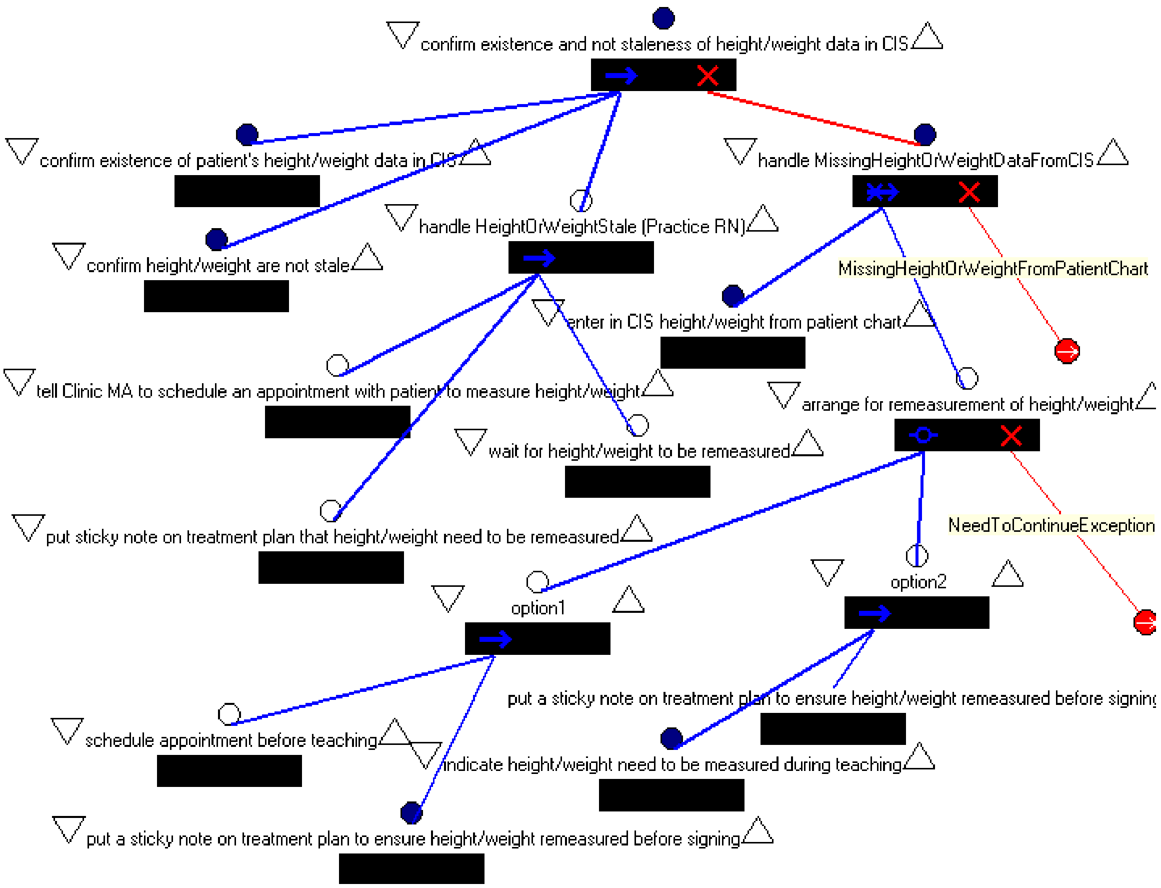

confirm existence and not staleness of height/weight data in CIS
confirm existence of patient's height/weight data in CIS
handle MissingHeightOrWeightDataFromCIS
handle HeightOrWeightStale (Practice RN)
confirm height/weight are not stale
MissingHeightOrWeightFromPatientChart
enter in CIS height/weight from patient chart
tell Clinic MA to schedule an appointment with patient to measure height/weight
arrange for remeasurement of height/weight
wait for height/weight to be remeasured
NeedToContinueException
put sticky note on treatment plan that height/weight need to be remeasured
option2
option1
put a sticky note on treatment plan to ensure height/weight remeasured before signing
schedule appointment before teaching
indicate height/weight need to be measured during teaching
put a sticky note on treatment plan to ensure height/weight remeasured before signing

Figure H.9. Diagram "confirm existence and not staleness of height/weight data in CIS"

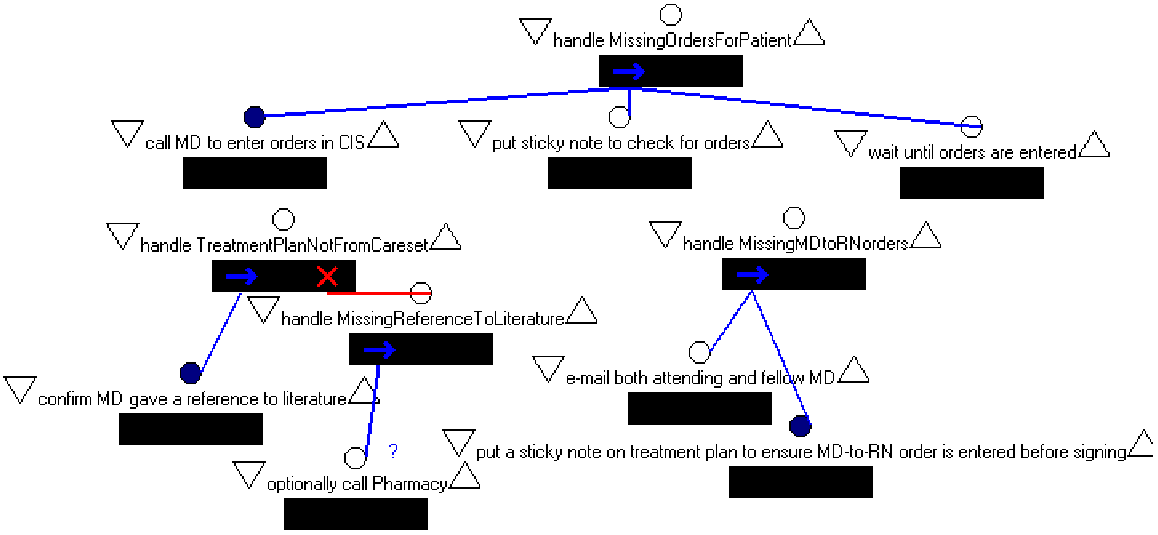

handle MissingOrdersForPatient
call MD to enter orders in CIS
put sticky note to check for orders
wait until orders are entered
handle TreatmentPlanNotFromCareset
handle MissingMDtoRNorders
handle MissingReferenceToLiterature
confirm MD gave a reference to literature
e-mail both attending and fellow MD
?
put a sticky note on treatment plan to ensure MD-to-RN order is entered before signing
optionally call Pharmacy

Figure H.10. Exception Handlers Used in Diagram "perform Practice RN verifications"

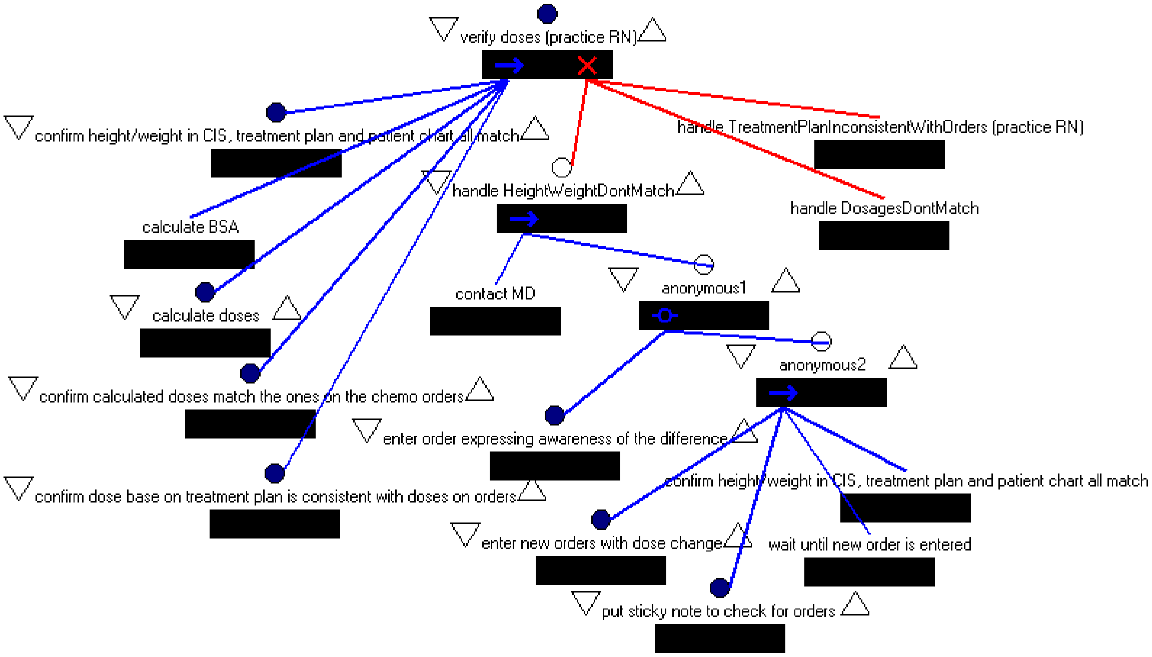

verify doses (practice RN)
confirm height/weight in CIS, treatment plan and patient chart all match
handle TreatmentPlanInconsistentWithOrders (practice RN)
handle HeightWeightDontMatch
handle DosagesDontMatch
calculate BSA
contact MD
anonymous1
calculate doses
anonymous2
confirm calculated doses match the ones on the chemo orders
enter order expressing awareness of the difference
confirm height/weight in CIS, treatment plan and patient chart all match
confirm dose base on treatment plan is consistent with doses on orders
enter new orders with dose change
wait until new order is entered
put sticky note to check for orders

Figure H.11. Diagram “verify doses (practice RN)”

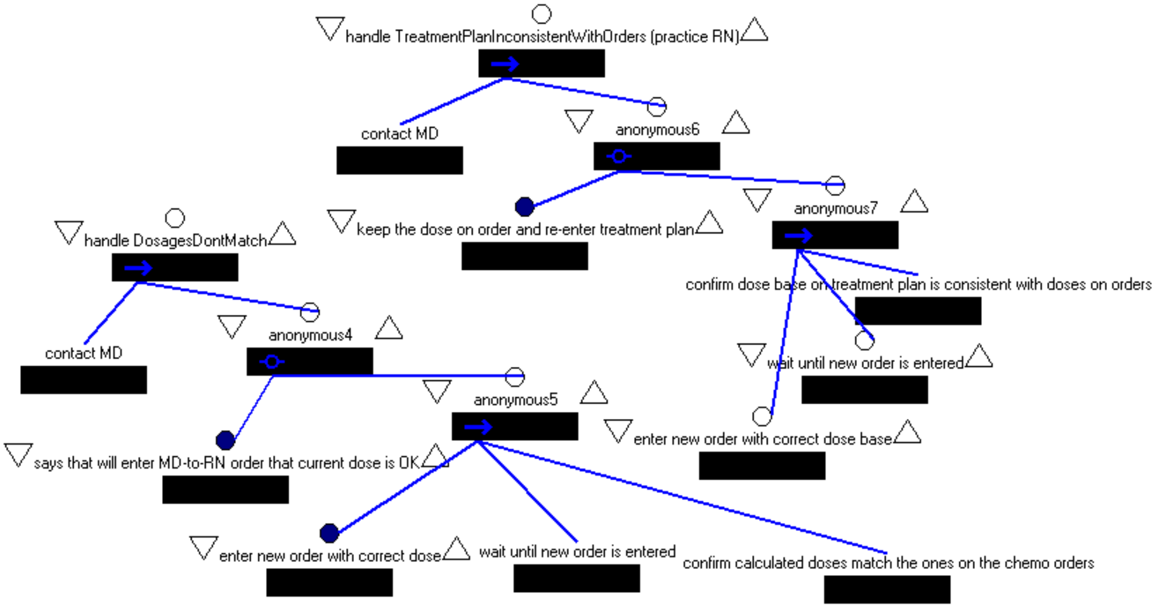

handle TreatmentPlanInconsistentWithOrders (practice RN)
contact MD
anonymous6
anonymous7
handle DosagesDontMatch
keep the dose on order and re-enter treatment plan
confirm dose base on treatment plan is consistent with doses on orders
contact MD
anonymous4
wait until new order is entered
anonymous5
enter new order with correct dose base
says that will enter MD-to-RN order that current dose is OK
enter new order with correct dose
wait until new order is entered
confirm calculated doses match the ones on the chemo orders

Figure H.12. Exception Handlers Used in Diagram “verify doses (practice RN)”

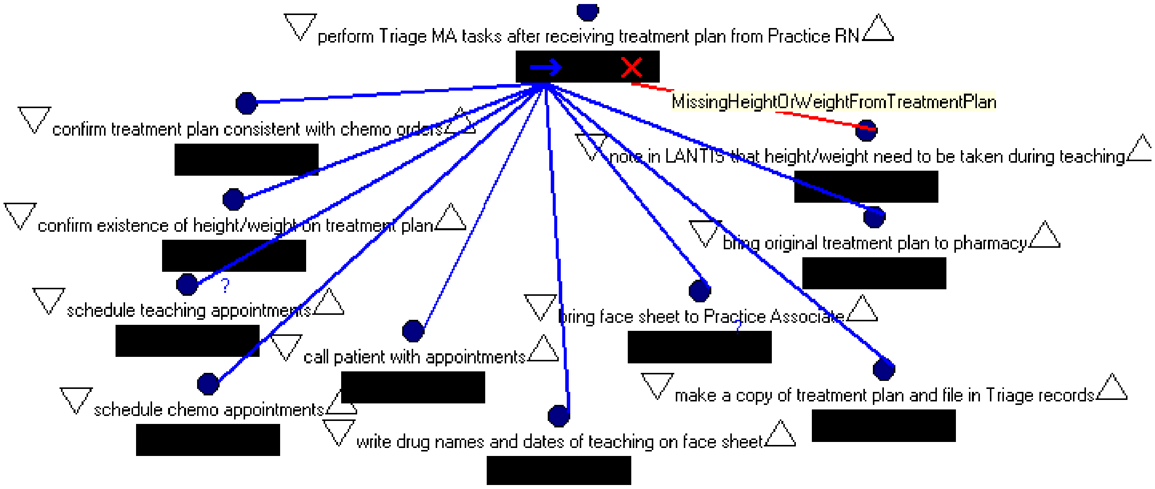

perform Triage MA tasks after receiving treatment plan from Practice RN
[redacted]
MissingHeightOrWeightFromTreatmentPlan
confirm treatment plan consistent with chemo orders
[redacted]
note in LANTIS that height/weight need to be taken during teaching
[redacted]
confirm existence of height/weight on treatment plan
[redacted]
bring original treatment plan to pharmacy
[redacted]
?
schedule teaching appointments
[redacted]
bring face sheet to Practice Associate
[redacted]
call patient with appointments
[redacted]
make a copy of treatment plan and file in Triage records
[redacted]
schedule chemo appointments
[redacted]
write drug names and dates of teaching on face sheet
[redacted]

Figure H.13. Diagram "perform Triage MA tasks after receiving treatment plan from Practice RN"

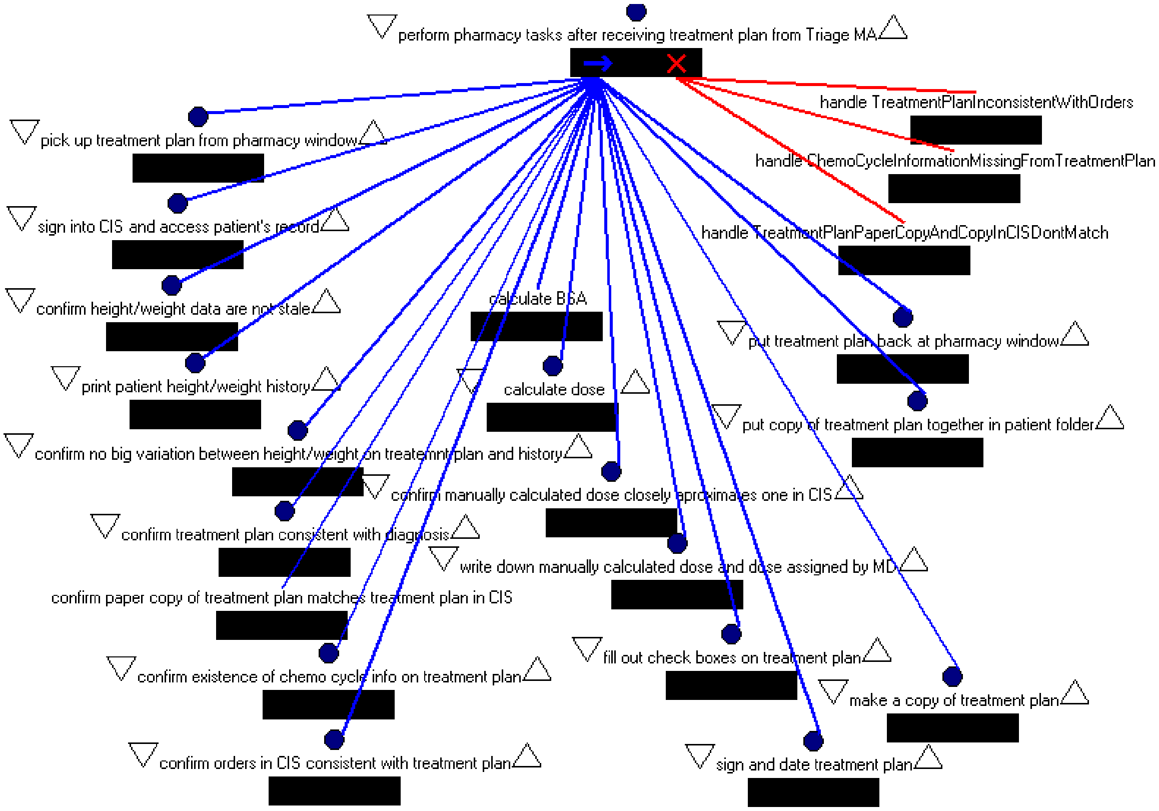

perform pharmacy tasks after receiving treatment plan from Triage MA
[redacted]
handle TreatmentPlanInconsistentWithOrders
[redacted]
pick up treatment plan from pharmacy window
[redacted]
handle ChemoCycleInformationMissingFromTreatmentPlan
[redacted]
sign into CIS and access patient's record
[redacted]
handle TreatmentPlanPaperCopyAndCopyInCISDontMatch
[redacted]
calculate BSA
[redacted]
confirm height/weight data are not stale
[redacted]
put treatment plan back at pharmacy window
[redacted]
print patient height/weight history
[redacted]
calculate dose
[redacted]
put copy of treatment plan together in patient folder
[redacted]
confirm no big variation between height/weight on treatemnt plan and history
[redacted]
confirm manually calculated dose closely aproximates one in CIS
[redacted]
confirm treatment plan consistent with diagnosis
[redacted]
write down manually calculated dose and dose assigned by MD
[redacted]
confirm paper copy of treatment plan matches treatment plan in CIS
[redacted]
fill out check boxes on treatment plan
[redacted]
confirm existence of chemo cycle info on treatment plan
[redacted]
make a copy of treatment plan
[redacted]
confirm orders in CIS consistent with treatment plan
[redacted]
sign and date treatment plan
[redacted]

Figure H.14. Diagram "perform pharmacy tasks after receiving treatment plan from Triage MA"

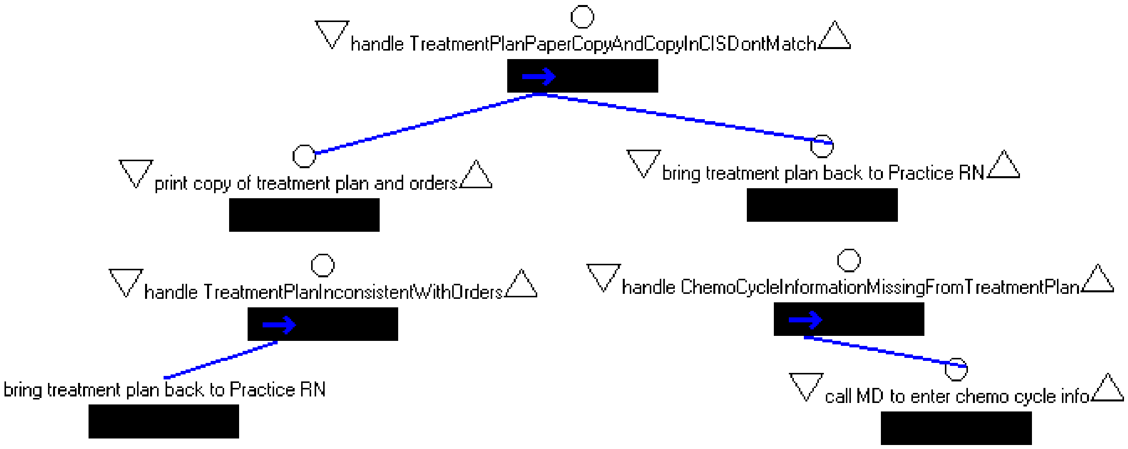

handle TreatmentPlanPaperCopyAndCopyInCISDontMatch
print copy of treatment plan and orders
bring treatment plan back to Practice RN
handle TreatmentPlanInconsistentWithOrders
handle ChemoCycleInformationMissingFromTreatmentPlan
bring treatment plan back to Practice RN
call MD to enter chemo cycle info

Figure H.15. Exception Handlers Used in Diagram "perform pharmacy tasks after receiving treatment plan from Triage MA"

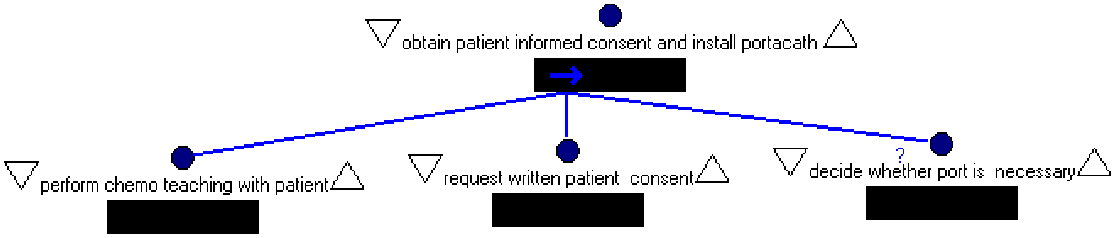

obtain patient informed consent and install portacath
perform chemo teaching with patient
request written patient consent
?
decide whether port is necessary

Figure H.16. Diagram "obtain patient informed consent and install portacath"

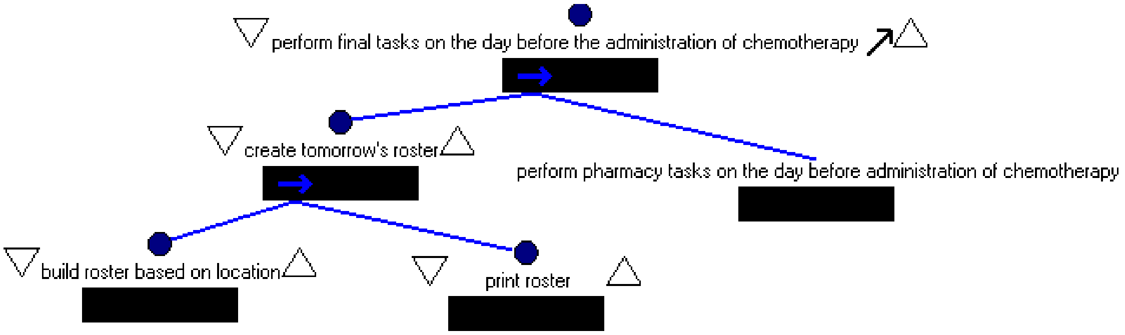

perform final tasks on the day before the administration of chemotherapy
create tomorrow's roster
perform pharmacy tasks on the day before administration of chemotherapy
build roster based on location
print roster

Figure H.17. Diagram "perform final tasks on the day before the administration of chemotherapy"

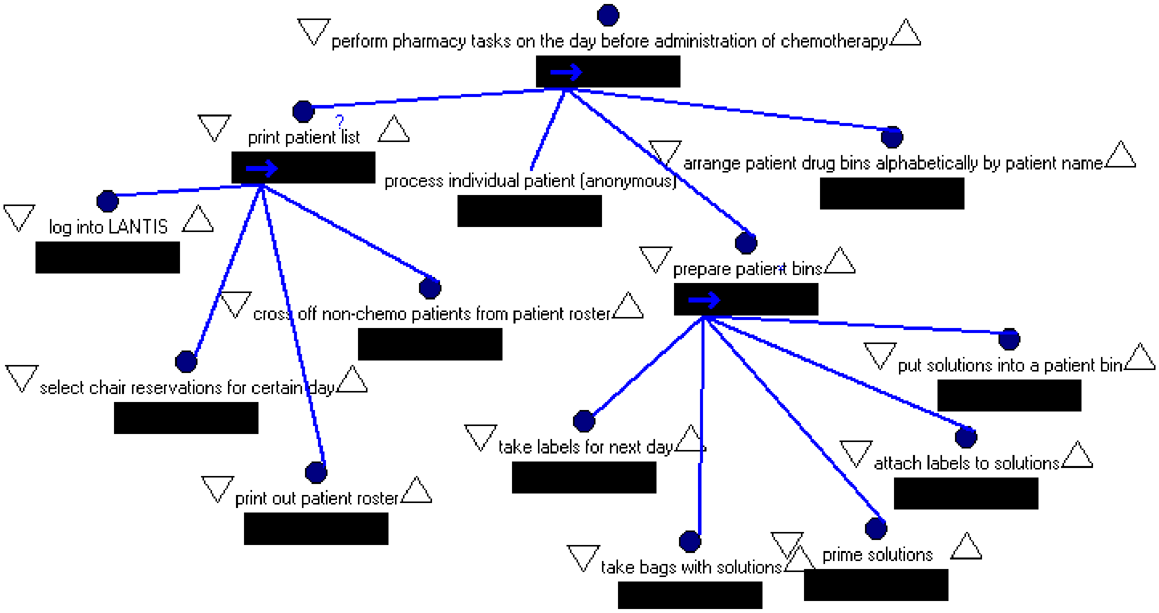

perform pharmacy tasks on the day before administration of chemotherapy
print patient list
?
process individual patient (anonymous)
arrange patient drug bins alphabetically by patient name
log into LANTIS
prepare patient bins
cross off non-chemo patients from patient roster
put solutions into a patient bin
select chair reservations for certain day
take labels for next day
attach labels to solutions
print out patient roster
take bags with solutions
prime solutions

Figure H.18. Diagram "perform pharmacy tasks on the day before administration of chemotherapy"

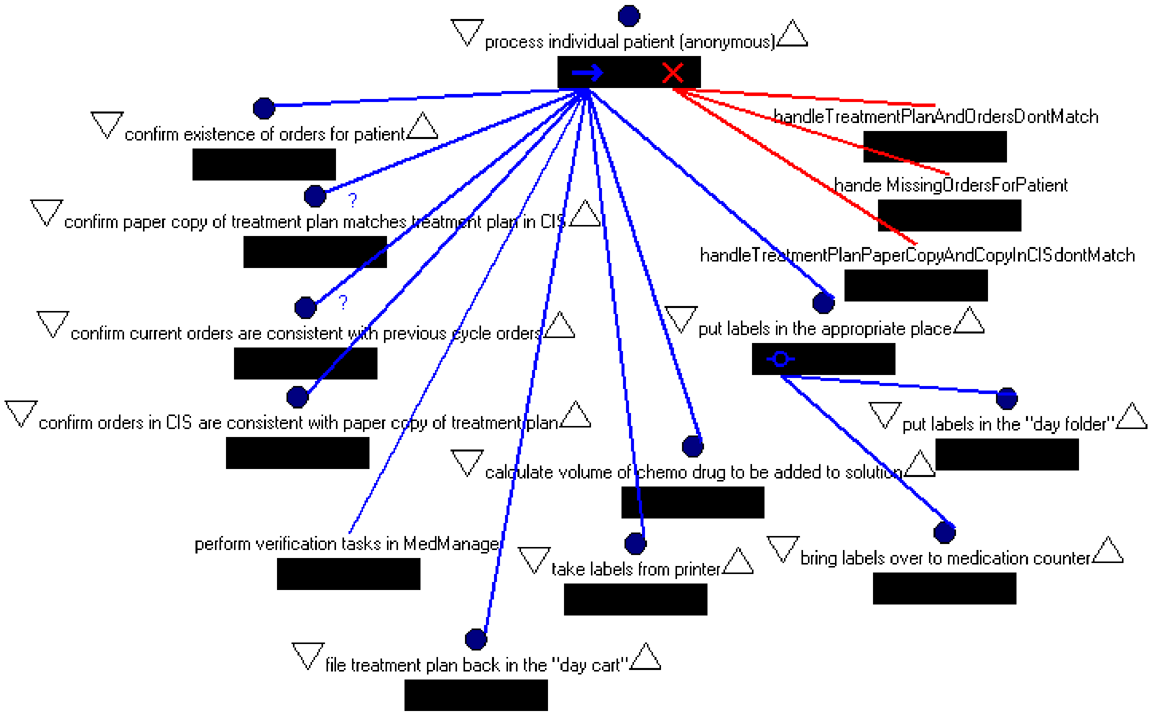

process individual patient (anonymous)
confirm existence of orders for patient
handleTreatmentPlanAndOrdersDontMatch
?
confirm paper copy of treatment plan matches treatment plan in CIS
hande MissingOrdersForPatient
handleTreatmentPlanPaperCopyAndCopyInCISdontMatch
?
confirm current orders are consistent with previous cycle orders
put labels in the appropriate place
confirm orders in CIS are consistent with paper copy of treatment plan
put labels in the "day folder"
calculate volume of chemo drug to be added to solution
perform verification tasks in MedManager
take labels from printer
bring labels over to medication counter
file treatment plan back in the "day cart"

Figure H.19. Diagram "process individual patient (anonymous)"

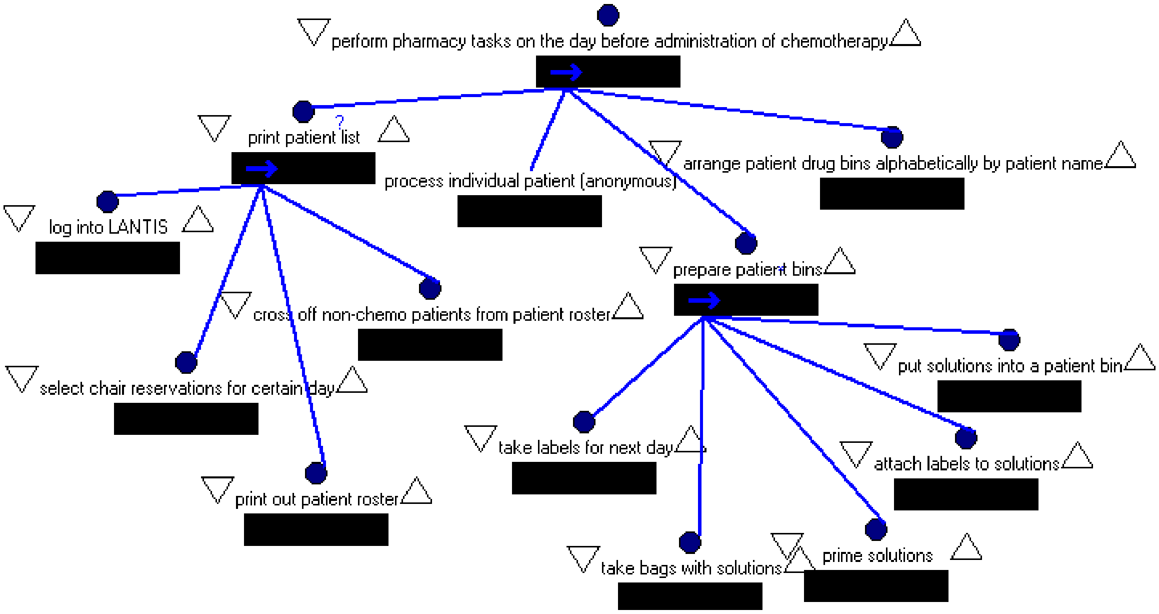

perform pharmacy tasks on the day before administration of chemotherapy
print patient list
?
process individual patient (anonymous)
arrange patient drug bins alphabetically by patient name
log into LANTIS
prepare patient bins
+
cross off non-chemo patients from patient roster
put solutions into a patient bin
select chair reservations for certain day
take labels for next day
attach labels to solutions
print out patient roster
take bags with solutions
prime solutions

Figure H.20. Exception Handlers Used in Diagram "process individual patient (anonymous)"

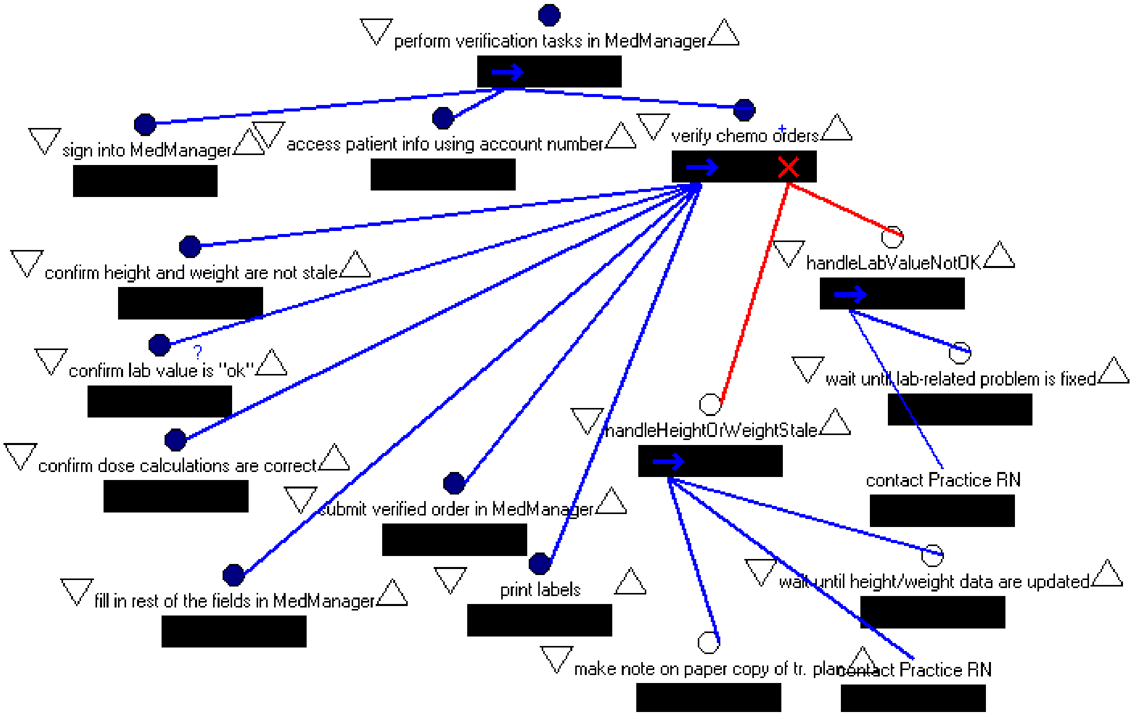

perform verification tasks in MedManager
sign into MedManager
access patient info using account number
verify chemo orders
+
confirm height and weight are not stale
handleLabValueNotOK
confirm lab value is "ok"
?
wait until lab-related problem is fixed
handleHeightOrWeightStale
confirm dose calculations are correct
contact Practice RN
submit verified order in MedManager
fill in rest of the fields in MedManager
print labels
wait until height/weight data are updated
make note on paper copy of tr. plan
contact Practice RN

Figure H.21. Diagram "perform verification tasks in MedManager"

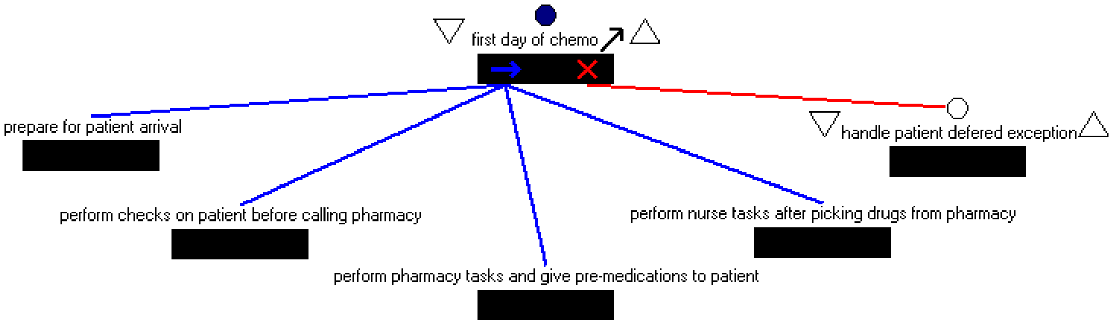

first day of chemo
prepare for patient arrival
handle patient defered exception
perform checks on patient before calling pharmacy
perform nurse tasks after picking drugs from pharmacy
perform pharmacy tasks and give pre-medications to patient

Figure H.22. Diagram "first day of chemo"

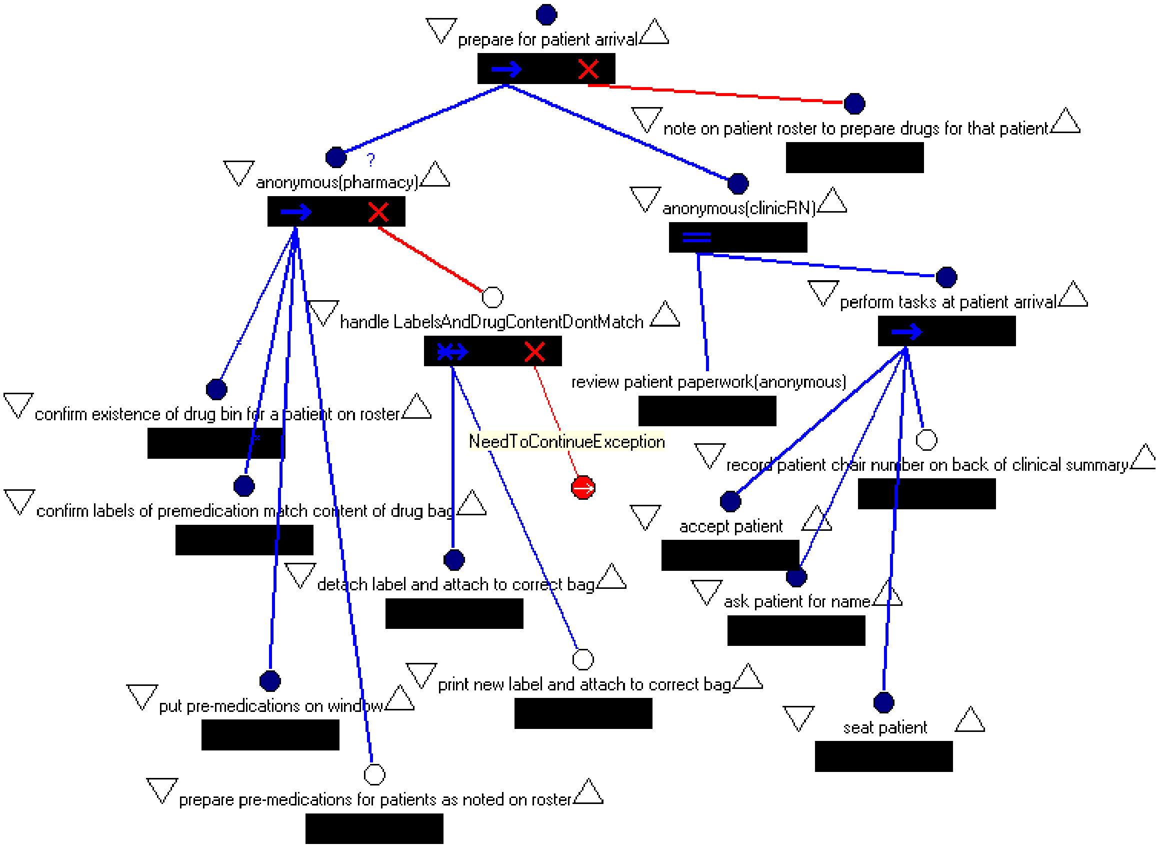

prepare for patient arrival
note on patient roster to prepare drugs for that patient
?
anonymous(pharmacy)
anonymous(clinicRN)
handle LabelsAndDrugContentDontMatch
perform tasks at patient arrival
review patient paperwork(anonymous)
confirm existence of drug bin for a patient on roster
NeedToContinueException
record patient chair number on back of clinical summary
confirm labels of premedication match content of drug bag
accept patient
detach label and attach to correct bag
ask patient for name
print new label and attach to correct bag
put pre-medications on window
seat patient
prepare pre-medications for patients as noted on roster

Figure H.23. Diagram "prepare for patient arrival"

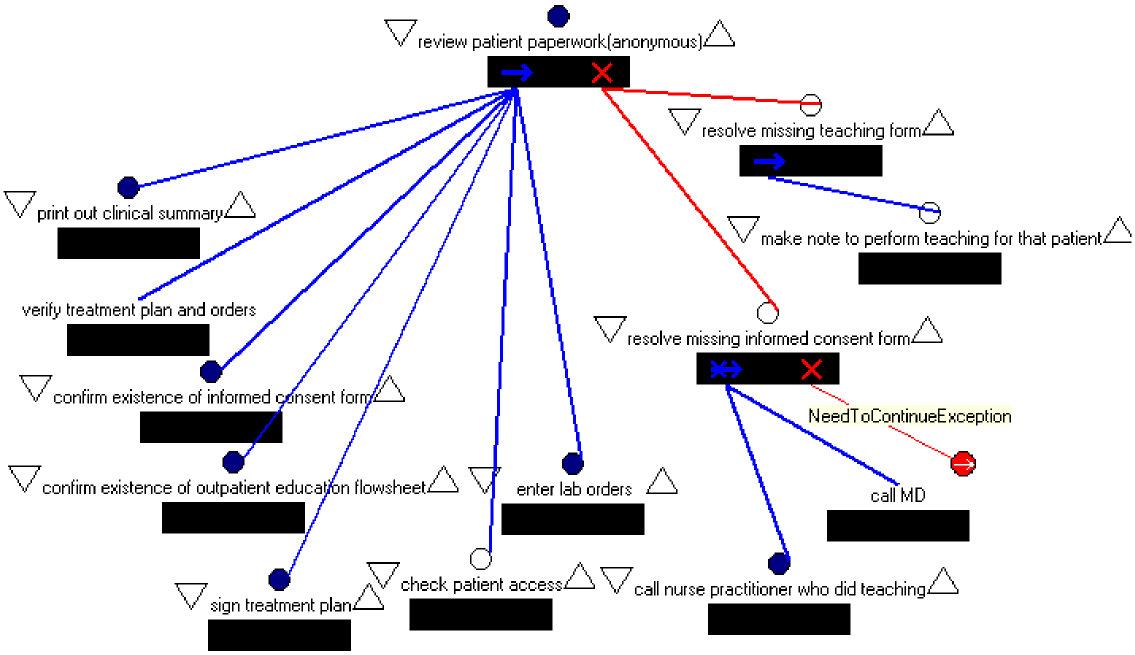

review patient paperwork(anonymous)
resolve missing teaching form
print out clinical summary
make note to perform teaching for that patient
verify treatment plan and orders
resolve missing informed consent form
confirm existence of informed consent form
NeedToContinueException
confirm existence of outpatient education flowsheet
enter lab orders
call MD
check patient access
call nurse practitioner who did teaching
sign treatment plan

Figure H.24. Diagram “review patient paperwork(anonymous)”

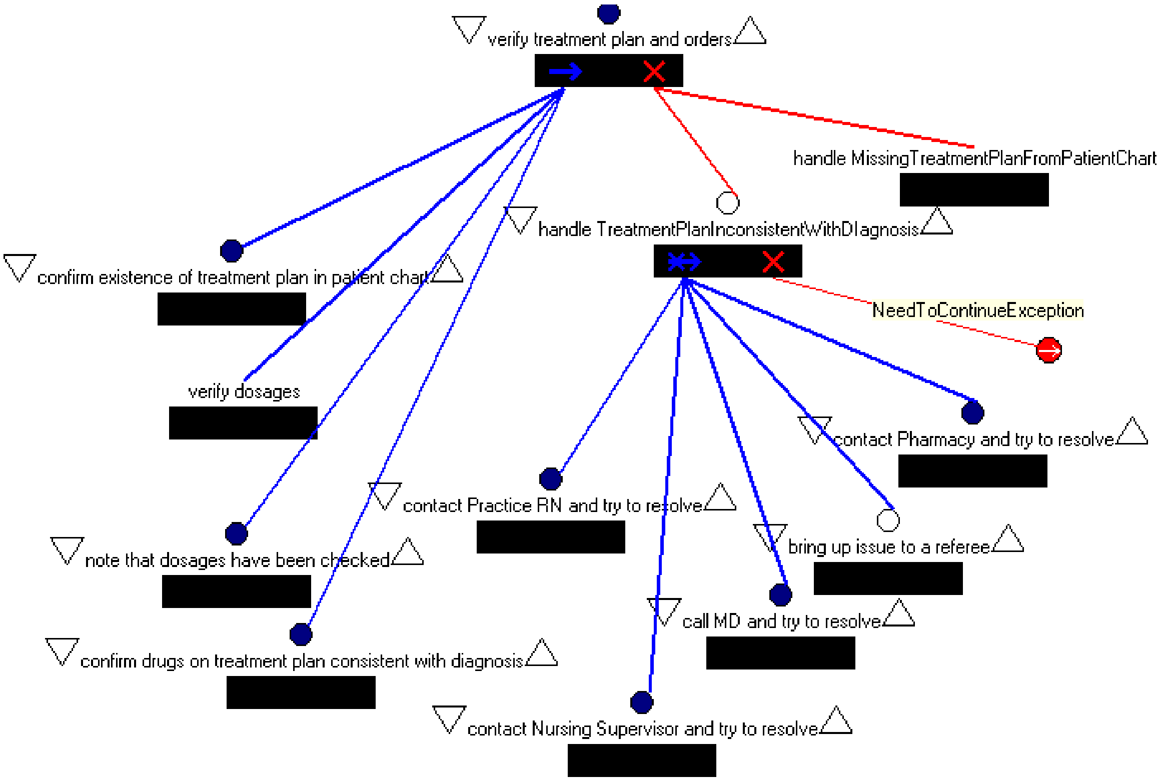

verify treatment plan and orders
handle MissingTreatmentPlanFromPatientChart
handle TreatmentPlanInconsistentWithDIagnosis
confirm existence of treatment plan in patient chart
NeedToContinueException
verify dosages
contact Pharmacy and try to resolve
contact Practice RN and try to resolve
note that dosages have been checked
bring up issue to a referee
call MD and try to resolve
confirm drugs on treatment plan consistent with diagnosis
contact Nursing Supervisor and try to resolve

Figure H.25. Diagram “verify treatment plan and orders”

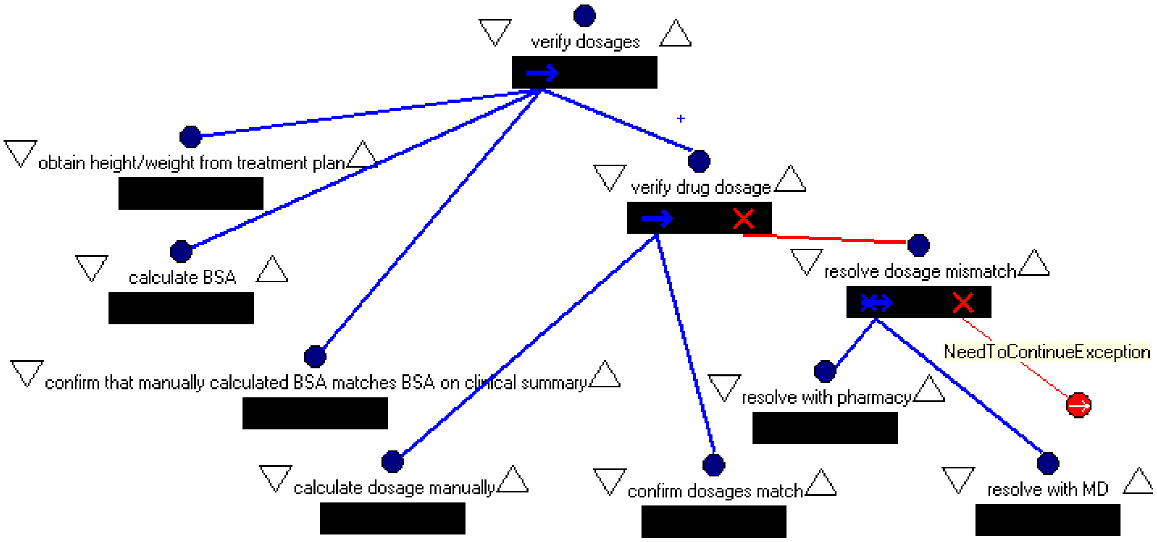


Figure H.26. Diagram "verify dosages"

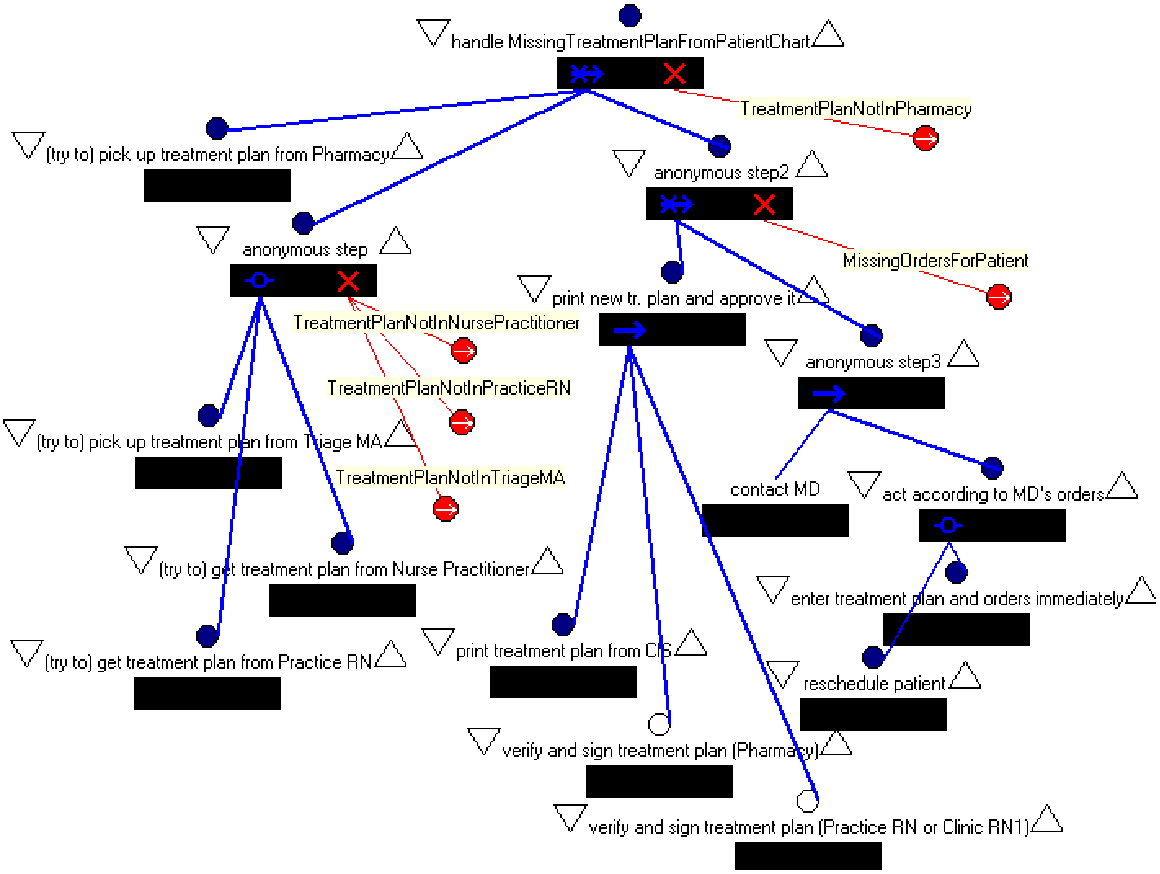

handle MissingTreatmentPlanFromPatientChart
TreatmentPlanNotInPharmacy
(try to) pick up treatment plan from Pharmacy
anonymous step2
anonymous step
MissingOrdersForPatient
print new tr. plan and approve it
TreatmentPlanNotInNursePractitioner
anonymous step3
TreatmentPlanNotInPracticeRN
(try to) pick up treatment plan from Triage MA
TreatmentPlanNotInTriageMA
contact MD
act according to MD's orders
(try to) get treatment plan from Nurse Practitioner
enter treatment plan and orders immediately
(try to) get treatment plan from Practice RN
print treatment plan from CIS
reschedule patient
verify and sign treatment plan (Pharmacy)
verify and sign treatment plan (Practice RN or Clinic RN1)

Figure H.27. Diagram "handle MissingTreatmentPlanFromPatientChart"

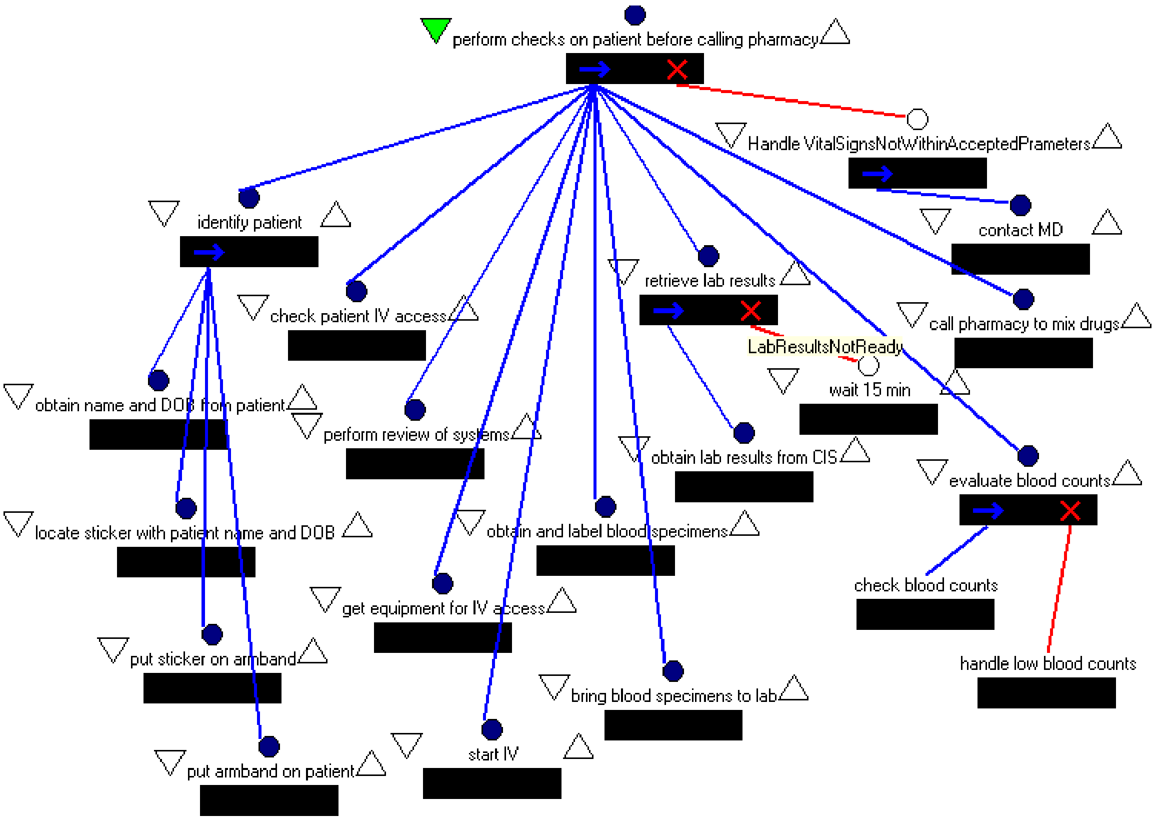

perform checks on patient before calling pharmacy
Handle VitalSignsNotWithinAcceptedPrameters
contact MD
identify patient
check patient IV access
retrieve lab results
LabResultsNotReady
call pharmacy to mix drugs
wait 15 min
obtain name and DOB from patient
perform review of systems
obtain lab results from CIS
evaluate blood counts
locate sticker with patient name and DOB
obtain and label blood specimens
check blood counts
get equipment for IV access
put sticker on armband
handle low blood counts
bring blood specimens to lab
start IV
put armband on patient

Figure H.28. Diagram “perform checks on patient before calling pharmacy”

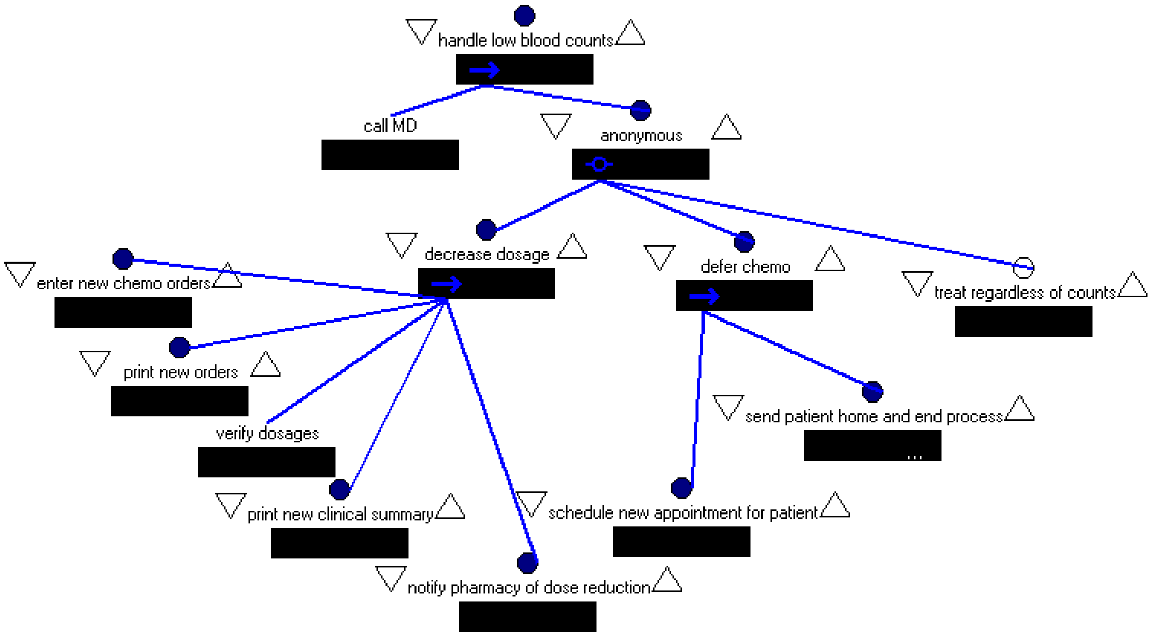

handle low blood counts
call MD
anonymous
decrease dosage
enter new chemo orders
defer chemo
treat regardless of counts
print new orders
send patient home and end process
verify dosages
print new clinical summary
schedule new appointment for patient
notify pharmacy of dose reduction

Figure H.29. Diagram “handle low blood counts”

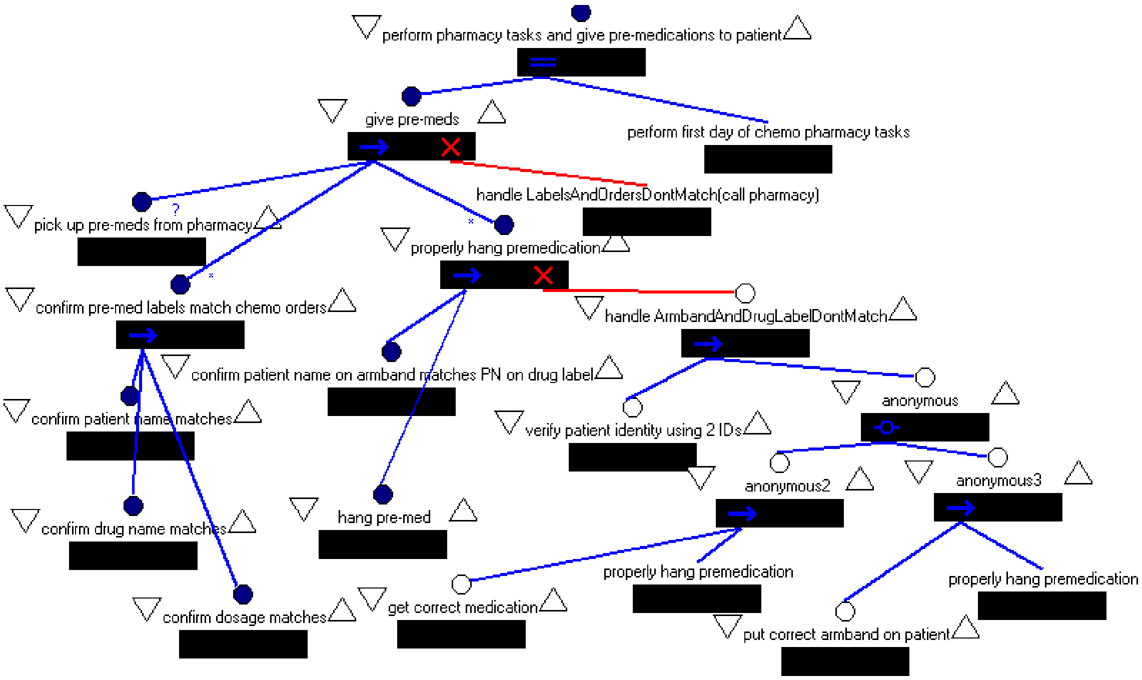

perform pharmacy tasks and give pre-medications to patient
give pre-meds
perform first day of chemo pharmacy tasks
handle LabelsAndOrdersDontMatch(call pharmacy)
?
pick up pre-meds from pharmacy
properly hang premedication
confirm pre-med labels match chemo orders
handle ArmbandAndDrugLabelDontMatch
confirm patient name on armband matches PN on drug label
confirm patient name matches
verify patient identity using 2 IDs
anonymous
anonymous2
anonymous3
hang pre-med
confirm drug name matches
properly hang premedication
properly hang premedication
confirm dosage matches
get correct medication
put correct armband on patient

Figure H.30. Diagram "perform pharmacy tasks and give pre-medications to patient"

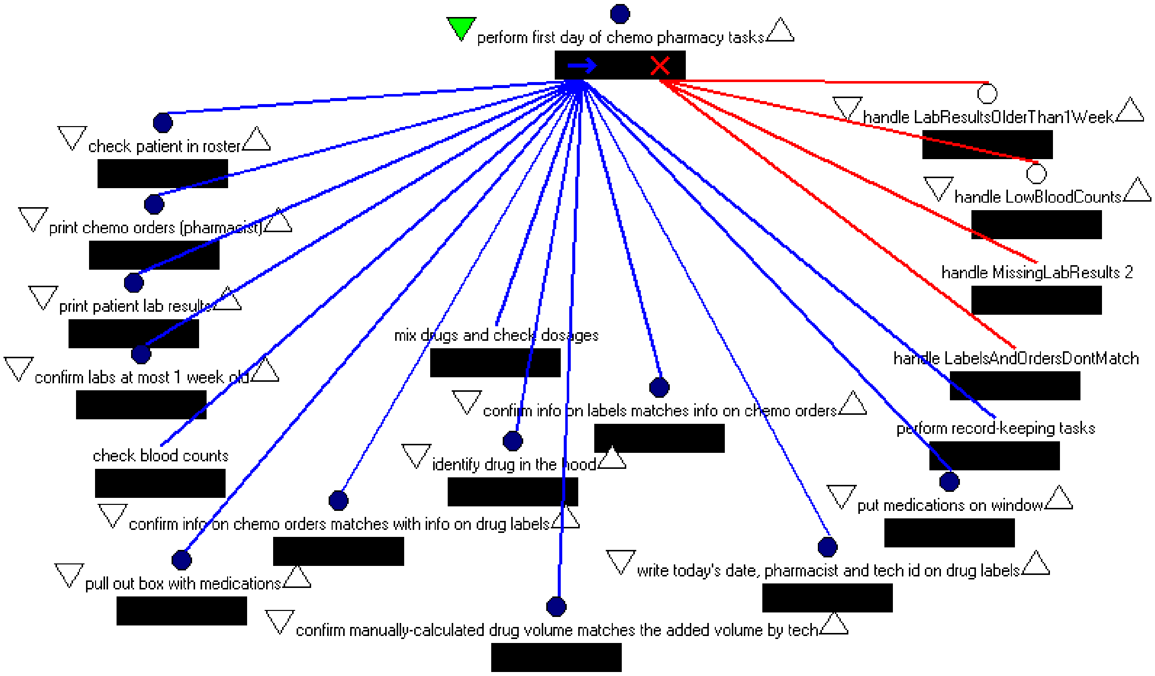

perform first day of chemo pharmacy tasks
handle LabResultsOlderThan1Week
check patient in roster
handle LowBloodCounts
print chemo orders (pharmacist)
handle MissingLabResults 2
print patient lab results
mix drugs and check dosages
handle LabelsAndOrdersDontMatch
confirm labs at most 1 week old
confirm info on labels matches info on chemo orders
perform record-keeping tasks
check blood counts
identify drug in the hood
put medications on window
confirm info on chemo orders matches with info on drug labels
write today's date, pharmacist and tech id on drug labels
pull out box with medications
confirm manually-calculated drug volume matches the added volume by tech

Figure H.31. Diagram "perform first day of chemo pharmacy tasks"

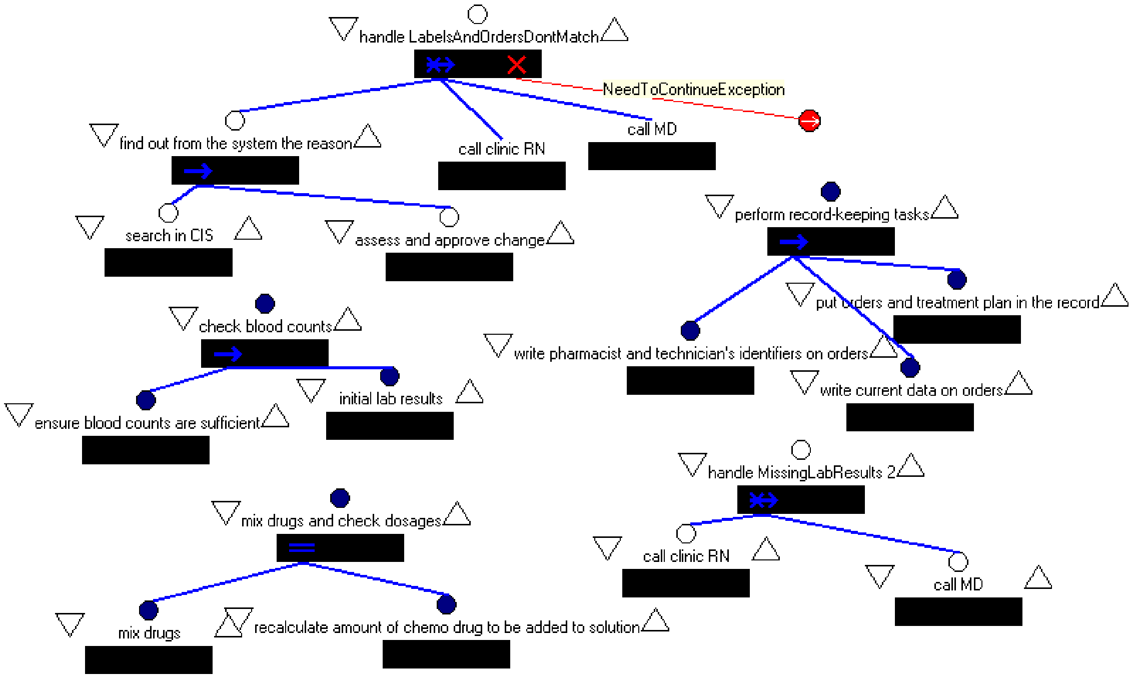

handle LabelsAndOrdersDontMatch
NeedToContinueException
find out from the system the reason
call clinic RN
call MD
perform record-keeping tasks
search in CIS
assess and approve change
put orders and treatment plan in the record
check blood counts
write pharmacist and technician's identifiers on orders
write current data on orders
ensure blood counts are sufficient
initial lab results
handle MissingLabResults 2
mix drugs and check dosages
call clinic RN
call MD
mix drugs
recalculate amount of chemo drug to be added to solution

Figure H.32. Definitions of Steps Used in Diagram "perform first day of chemo pharmacy tasks"

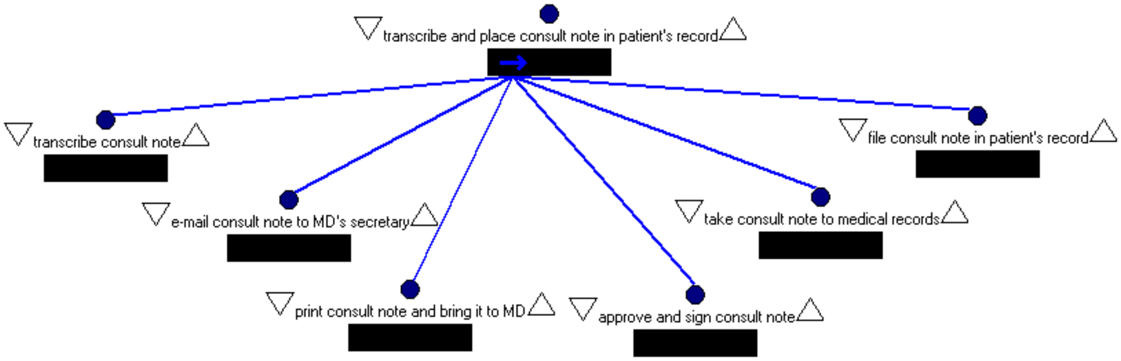

transcribe and place consult note in patient's record
transcribe consult note
file consult note in patient's record
e-mail consult note to MD's secretary
take consult note to medical records
print consult note and bring it to MD
approve and sign consult note

Figure H.33. Diagram "transcribe and place consult note in patient's record"

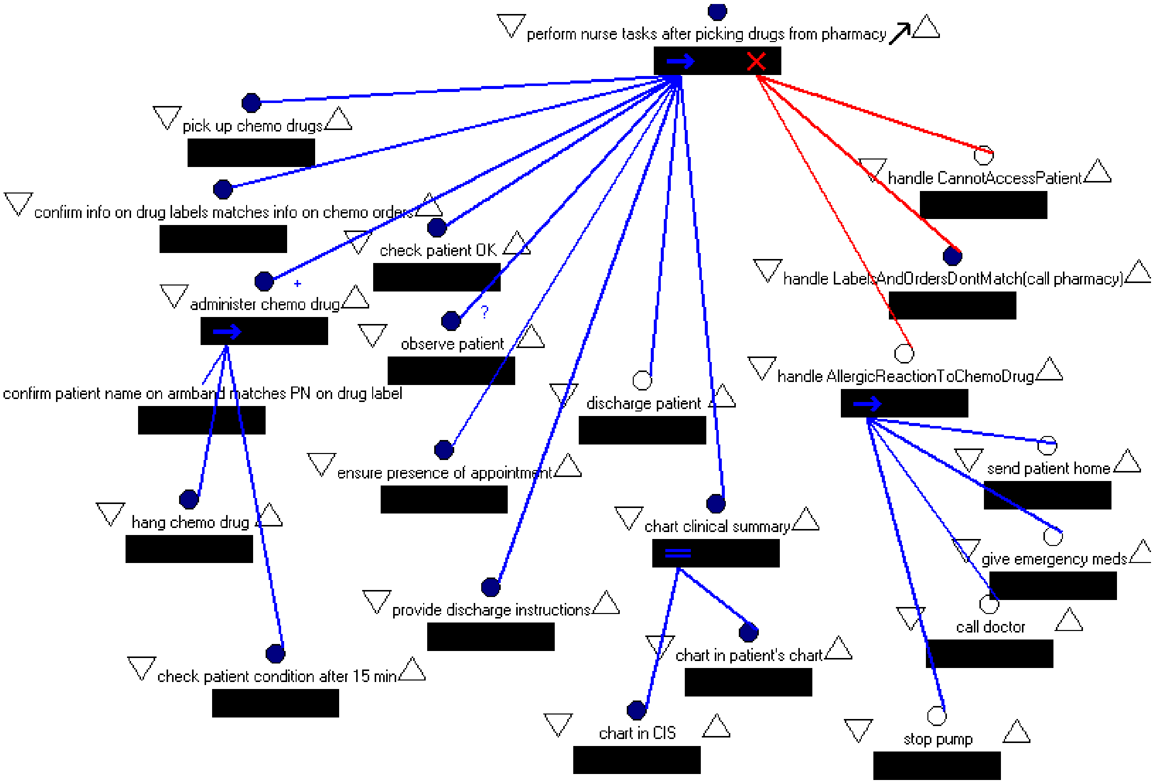

perform nurse tasks after picking drugs from pharmacy
pick up chemo drugs
confirm info on drug labels matches info on chemo orders
check patient OK
administer chemo drug
+
observe patient
?
confirm patient name on armband matches PN on drug label
discharge patient
ensure presence of appointment
hang chemo drug
chart clinical summary
provide discharge instructions
chart in patient's chart
check patient condition after 15 min
chart in CIS
handle CannotAccessPatient
handle LabelsAndOrdersDontMatch(call pharmacy)
handle AllergicReactionToChemoDrug
send patient home
give emergency meds
call doctor
stop pump

Figure H.34. Diagram "perform nurse tasks after picking drugs from pharmacy"

APPENDIX I

CHEMOTHERAPY PROCESS VERIFICATION REPORT

A Patient Eligibility

A.1 Pathologist must review patient pathology before chemotherapy can be administered

Iteration 1:

- Event Binding:

  administer chemotherapy→

  administer chemo drug STARTED

  pathologist reviews pathology→

  verify pathology report COMPLETED

- Result: the property does not hold. The DNL of the property says "*'administer chemotherapy' cannot occur unless 'pathologist reviews pathology' has already occurred. 'pathologist reviews pathology' is required to occur, but 'administer chemotherapy' is not required to occur. ...*". So the property does not allow chemotherapy to be administered without a review. However it also requires that every execution include an occurrence of the event "*pathologist reviews pathology*". The counter example trace shows that an exception thrown by an earlier step terminates the whole process. When this happens, the corresponding steps for "*pathologist reviews pathology*" and "*administer chemotherapy*" will not be executed. In the real world process, this kind of executions should be allowed to occur. Therefore the property is incorrect.

- Change: in the property, the option for "*is 'pathologist reviews pathology' required to occur at least once?*" is changed to "*No*". The DNL becomes "*'administer chemotherapy' cannot occur unless 'pathologist reviews pathology' has already occurred. 'pathologist reviews pathology' is not required to occur, and if it does not occur, administer chemotherapy can never occur. Even if 'pathologist reviews pathology' does occur, 'administer chemotherapy' is not require to occur*".

Iteration 2:

- Event Binding: the same as Iteration 1.
- Result: no violation found.

Note: in the process, there is not such a step as "*pathologist reviews pathology*". The pathology report is a parameter passed to the chemo process. The step "*verify pathology report*" verifies that the pathology report is from Baystate. If the report is not from Baystate, exception *PathologyReportNotFromBaystate* is thrown, and the handler "*send path report to baystate pathology for review*" will be executed. The handler has a *restart* continuation badge. Therefore when "*send path report to baystate pathology for review*" completes, "*verify pathology report*" will be executed again. In other words, the pathology report will be reviewed within the scope of the chemo process only if it is not from the Baystate. Depending on which step the event "*pathologist reviews pathology*" is bound to, the corrected property could either hold or not hold. If the event is bound to "*verify pathology report*" COMPLETED, the corrected property holds as shown in the Result section. If the event is bound to "*send path report to baystate pathology for review*" COMPLETED. The verifier produced a violation trace showing that if the pathology report is from Baystate, it will not be reviewed before chemotherapy is administered.

A.2 Patient must have consult with an Attending MD before chemotherapy can be administered

Iteration 1:

- Event Binding:

  administer chemotherapy→
  administer chemo drug STARTED
  patient has a consult with an attending MD→
  perform patient consultation COMPLETED

- Result: the property does not hold. Similar to A.1, this property requires that every execution include an occurrence of the event "*patient has a consult with an attending MD*". However, the event does not have to happen if any exception thrown earlier terminates the process.

- Change: in the property, the option for "*is 'patient has a consult with an attending MD' required to occur at least once?*" is changed to "*No*".

Iteration 2:

- Event Binding: the same as Iteration 1.

- Result: no violation found.

A.3 If the Attending MD decides no cancer diagnosis, chemotherapy cannot be administered

Iteration 1:

- Event Binding:

administer chemotherapy→

administer chemo drug STARTED

attending MD decides no cancer diagnosis→

verify diagnosis of cancer TERMINATED

- Result: no violation found.

Note: the elicitation of the sub-process containing "*verify diagnosis of cancer*" is not completed yet. The interface of step "*verify diagnosis of cancer*" declares an exception *java.lang.Exception* might be thrown by this step. This exception should be given a more specific name later.

B Legal Constraints

B.1 Consult note must be put into patient record before chemotherapy can be administered

Iteration 1:

- Event Binding:

administer chemotherapy→

administer chemo drug STARTED

consult note is put into patient's record→

file consult note in patient's record COMPLETED

- Result: violation found. The property requires that every execution include an occurrence of the event "*consult note is put into that patient's record*". However, the violation trace shows that this event does not happen if any exception thrown earlier terminates the process.

- Change: in the property, the option for "*is 'consult note is put into patient's record' is required to occur?*" to "*No. 'consult note is put into that patient's* *record' is not required to occur, if it does not occur, 'administer chemo drug' is never allowed to occur*".

Iteration 2:

- Event Binding: the same as Iteration 1.

- Result: violation found. The step "*file consult note in patient's record*" is executed in parallel with the step "*administer chemo drug*". Therefore, "*file consult note in patient's record*" can be executed after "*administer chemo drug*". This is an error in the real process.

B.2 Patient must sign consent form before chemotherapy can be administered

Iteration 1:

- Event Binding:

  administer chemotherapy→

  administer chemo drug STARTED

  patient signs consent form→

  request written patient consent COMPLETED

- Result: no violation found.

B.3 Treatment plan must be present before chemotherapy can be administered

Iteration 1:

- Event Binding:

administer chemotherapy→

administer chemo drug STARTED

treatment plan must is present→

confirm existence of treatment plan in patient chart COMPLETED

- Result: violation found. The step "*confirm existence of treatment plan in patient chart*" may throw exception *Missing Treatment Plan From Patient Chart*. The exception is handle by step "*handle Missing Treatment Plan From Patient Chart*". This handler step will create a new treatment plan. Therefore, the event "*treatment plan must is present*" should be considered to occur when either "*confirm existence of treatment plan in patient chart*" or "*handle Missing Treatment Plan From Patient Chart*" is completed.

Iteration 2:

- Event Binding:

administer chemotherapy→

administer chemo drug STARTED

treatment plan must is present→

confirm existence of treatment plan in patient chart COMPLETED |

handle Missing Treatment Plan ¿From Patient Chart COMPLETED

- Result: no violation found.

B.4 Before patient can sign consent form, patient must have consult with an Attending MD

Iteration 1:

- Event Binding:

patient has a consult with an attending MD→

perform patient consultation COMPLETED

patient signs consent form→

request written patient consent STARTED

- Result: no violation found.

B.5 Before patient can sign a new consent form, patient must have chemotherapy teaching

Iteration 1:

- Event Binding:

patient has chemotherapy teaching→

perform chemo teaching with patient COMPLETED

patient signs consent form→

request written patient consent STARTED

- Result: violation found. Step "*perform chemo teaching with patient*" and "*request written patient consent*" are sub-steps of a parallel step. Therefore, they can be executed in any order.

- Change: the parent step of "*perform chemo teaching with patient*" and "*request written patient consent*" is changed to a sequential step.

Iteration 2:

- Event Binding: the same as Iteration 1.

- Result: no violation found.

B.6 Patient must have consult with an Attending MD before patient can have chemotherapy teaching

Iteration 1:

- Event Binding:

  patient has chemotherapy teaching→

  perform chemo teaching with patient STARTED

  patient has a consult with an attending MD→

  perform patient consultation COMPLETED

- Result: no violation found.

C Development of Treatment Plan and Chemotherapy Orders

C.1a Treatment plan must be approved by Clinic RN before chemotherapy can be administered the first time

Iteration 1:

- Event Binding:

  administer chemotherapy→

  administer chemo drug STARTED

  clinic RN approves treatment plan→

  sign treatment plan COMPLETED

- Result: no violation found.

C.1b Treatment plan must be approved by Practice RN before chemotherapy can be administered the first time

Iteration 1:

- Event Binding:

  administer chemotherapy→

  administer chemo drug STARTED

  practice RN approves treatment plan→

  sign treatment plan (Practice RN) COMPLETED

- Result: no violation found.

C.1c Treatment plan must be approved by Pharmacy before chemotherapy can be administered the first time

Iteration 1:

- Event Binding:

  administer chemotherapy→

  administer chemo drug STARTED

  pharmacist approves treatment plan→

  sign and date treatment plan COMPLETED

- Result: no violation found.

C.2a Chemo order must be verified by Clinic RN before chemotherapy can be administered for the first time

Iteration 1:

- Event Binding:

  administer chemotherapy→

  administer chemo drug STARTED

  chemotherapy order is verified by clinic RN→

  verify treatment plan and orders COMPLETED

- Result: no violation found.

C.2b Chemo order must be verified by Practice RN before chemotherapy can be administered

Iteration 1:

- Event Binding:

  administer chemotherapy→
  administer chemo drug STARTED
  chemotherapy order is verified by practice RN→
  perform Practice RN verifications COMPLETED

- Result: no violation found.

C.2c Chemo order must be verified by Pharmacy before chemotherapy can be administered

Iteration 1:

- Event Binding:

  administer chemotherapy→
  administer chemo drug STARTED
  chemotherapy order is verified by pharmacy→
  verify chemo orders COMPLETED

- Result: violation found. Before step "*verify chemo orders*", There is a step "*confirm existence of orders for patient*" that might throw exception *Missing Order For Patient*. If this exception is thrown, "*verify chemo orders*" will not be executed. The exception handler "*hande Missing Order For Patient*" will fix the error. Therefore, we should also consider that the event "*chemotherapy order is verified by pharmacy*" occurs if "*hande Missing Order For Patient*" is completed.

Iteration 1:

- Event Binding:

  administer chemotherapy→

  administer chemo drug STARTED

  chemotherapy order is verified by pharmacy→

  verify chemo orders COMPLETED |

  hande Missing Order For Patient COMPLETED

- Result: no violation found.

C.6a Chemotherapy orders must be consistent with the treatment plan before PracticeRN can approve the treatment plan

Iteration 1:

- Event Binding:

  practice RN ensures that chemo order and treatment plan are consistent→

  verify doses (practice RN) COMPLETED

  practice RN approves the treatment plan→

  sign treatment plan (Practice RN) STARTED

- Result: no violation found.

C.6b Chemotherapy orders must be consistent with the treatment plan before Pharmacy can approve the treatment plan

Iteration 1:

- Event Binding:

pharmacy ensures that chemo order and treatment plan are consistent→

confirm orders in CIS consistent with treatment plan COMPLETED

pharmacy approves the treatment plan→

sign and date treatment plan STARTED

- Result: no violation found.

C.6c Chemotherapy orders must be consistent with the treatment plan before Clinic RN can approve the treatment plan

Iteration 1:

- Event Binding:

clinic RN ensures that chemo order and treatment plan are consistent→

verify dosages COMPLETED

clinic RN approves the treatment plan→

sign treatment plan STARTED

- Result: no violation found.

C.6d Chemotherapy orders must be consistent with the treatment plan before Pharmacy can verify the chemotherapy orders

Iteration 1:

- Event Binding:

pharmacy ensures that chemo order and treatment plan are consistent→

confirm orders in CIS are consistent with

paper copy of treatment plan COMPLETED |

handle TreatmentPlanAndOrdersDontMatch COMPLETED

chemotherapy order is verified by pharmacy→

verify chemo orders STARTED

- Result: no violation found.

C.6e Chemotherapy orders must be consistent with the treatment plan before Clinic RN can verify the chemotherapy orders

- Event Binding: for the clinic RN, checking the consistency of chemo order and treatment plan and verifying chemo order are performed at the step "*verify treatment plan and orders*". There are no different sub-steps that can be bind to these two events.

C.6f Chemotherapy orders must be consistent with the treatment plan before Practice RN can verify the chemotherapy orders

- Event Binding: for the practice RN, checking the consistency of chemo order and treatment plan and verifying chemo order are performed at the step "*perform Practice RN verifications*". There are no different sub-steps that can be bind to these two events.

C.7a Chemotherapy drugs must be consistent with cancer diagnosis before Practice RN can approve treatment plan

- Event Binding: no binding for event "*practice RN ensures that chemo drugs and cancer diagnosis are consistent*"

C.7b Chemotherapy drugs must be consistent with cancer diagnosis before Pharmacy can approve treatment plan

Iteration 1:

- Event Binding:

pharmacist ensures that chemo drugs and cancer diagnosis are consistent→
confirm treatment plan consistent with diagnosis COMPLETED

pharmacist approves treatment plan→

sign and date treatment plan STARTED

- Result: no violation found.

C.7c Chemotherapy drugs must be consistent with cancer diagnosis before Clinic RN can approve treatment plan

Iteration 1:

- Event Binding:

clinic RN ensures that chemo drugs and cancer diagnosis are consistent→

confirm drugs on treatment plan consistent

with diagnosis COMPLETED |

handle TreatmentPlanInconsistentWithDiagnosis COMPLETED

clinic RN approves treatment plan→

sign treatment plan STARTED

- Result: no violation found.

C.7d Chemotherapy drugs must be consistent with cancer diagnosis before Practice RN can verify chemotherapy orders

- Event Binding: no binding for event "*practice RN ensures that chemo drugs and cancer diagnosis are consistent*"

C.7fe Chemotherapy drugs must be consistent with cancer diagnosis before Pharmacy can verify chemotherapy orders

Iteration 1:

- Event Binding:

pharmacist ensures that chemo drugs and cancer diagnosis are consistent→

confirm treatment plan consistent with diagnosis COMPLETED

chemotherapy order is verified by pharmacy→

verify chemo orders STARTED

- Result: no violation found.

C.7f Chemotherapy drugs must be consistent with cancer diagnosis before clinic RN can verify chemotherapy orders

- Event Binding: for the Clinic RN, checking the consistency of chemo order and cancer diagnosis can be bind to step "*confirm drugs on treatment plan consistent with diagnosis*" and verifying chemo order can be bind to step "*verify treatment plan and orders*". However, "*confirm drugs on treatment plan consistent with diagnosis*" is a sub-step of "*verify treatment plan and orders*".

C.8a Chemotherapy drugs must be in doses that are consistent with patient data before Practice RN can approve treatment plan

Iteration 1:

- Event Binding:

practice RN checks that chemo drugs are in doses→

verify doses (practice RN) COMPLETED

practiceRN approves treatment plan→

sign treatment plan (Practice RN) STARTED

- Result: no violation found.

C.8b Chemotherapy drugs must be in doses that are consistent with patient data before Pharmacy can approve treatment plan

Iteration 1:

- Event Binding:

  pharmacy checks that chemo drugs are in doses→
  confirm manually calculated dose closely
  approximates one in CIS COMPLETED
  pharmacist approves treatment plan→
  sign and date treatment plan STARTED

- Result: no violation found.

C.8c Chemotherapy drugs must be in doses that are consistent with patient data before Clinic RN can approve treatment plan

Iteration 1:

- Event Binding:

  clinic RN checks that chemo drugs are in doses→
  verify drug dosage COMPLETED
  clinic RN approves treatment plan→
  sign treatment plan STARTED

- Result: no violation found.

C.8d Chemotherapy drugs must be in doses that are consistent with patient data before Practice RN can verify chemotherapy orders

- Event Binding: for the practice RN, checking chemo drugs in doses can be bind to step "*verify doses (practice RN)*" and verifying chemo order can be bind to step "*perform Practice RN verifications*". However, "*verify doses (practice RN)*" is a sub-step of "*perform Practice RN verifications*".

C.8e Chemotherapy drugs must be in doses that are consistent with patient data before Pharmacy can verify chemotherapy orders

Iteration 1:

- Event Binding:

pharmacy checks that chemo drugs are in doses→
confirm manually calculated dose closely
approximates one in CIS COMPLETED
chemotherapy order is verified by pharmacy→
verify chemo orders STARTED

- Result: no violation found.

C.8f Chemotherapy drugs must be in doses that are consistent with patient data before Clinic RN can verify chemotherapy orders

- Event Binding: for the Clinic RN, checking chemo drugs in doses can be bind to step "*verify drug dosage*" and verifying chemo order can be bind to step "*verify treatment plan and orders*". However, "*verify drug dosage*" is a sub-step of "*verify treatment plan and orders*".

C.9a Treatment plan cannot be approved by Practice RN if there is stale or disparate data

Iteration 1:

- Event Binding:

practice RN checks that data is not stale→
confirm existence and not staleness of
height/weight data in CIS COMPLETED

practice RN approves treatment plan→

sign treatment plan (Practice RN) STARTED

- Result: no violation found.

C.9b Treatment plan cannot be approved by Pharmacy if there is stale or disparate data

Iteration 1:

- Event Binding:

pharmacy checks that data is not stale→

confirm height and weight are not stale COMPLETED |

confirm height/weight data are not stale COMPLETED

pharmacist approves treatment plan→

sign and date treatment plan STARTED

- Result: no violation found.

C.9c Treatment plan cannot be approved by Clinic RN if there is stale or disparate data

- Event Binding: no binding for event "*clinic RN checks that data is not stale*".

C.9d Chemotherapy orders cannot be verified by second RN if there is stale or disparate data

- Event Binding: for the practice RN, checking data is not stale can be bind to step "*confirm existence and not staleness of height/weight data in CIS*" and verifying chemo order can be bind to step "*perform Practice RN verifications*". However, "*confirm existence and not staleness of height/weight data in CIS*" is a sub-step of "*perform Practice RN verifications*".

C.9e Chemotherapy orders cannot be verified by Pharmacy if there is stale or disparate data

Iteration 1:

- Event Binding:

  pharmacy checks that data is not stale→

  confirm height and weight are not stale COMPLETED |

  confirm height/weight data are not stale COMPLETED

  chemotherapy order is verified by pharmacy→

  verify chemo orders STARTED

- Result: no violation found.

C.9f Chemotherapy orders cannot be verified by Clinic RN if there is stale or disparate data

- Event Binding: no binding for event "*clinic RN checks that data is not stale*".

D Activities Required Right before Chemotherapy is Administered

Part 1. Before chemotherapy can be administered to a patient

D.1 That patient must be correctly identified (before chemotherapy can be administered to a patient)

Iteration 1:

- Event Binding:

  administer chemotherapy→

  administer chemo drug STARTED

correctly identify patient→

identify patient COMPLETED

clinic RN gets responsibility for patient→

first day of chemo POSTED

clinic RN ends responsibility for patient→

first day of chemo COMPLETED |

first day of chemo TERMINATED

- Result: the property does not hold. Similar to MP A.1, this property requires that the event "*correctly identify patient*" must happen at least once. However, if any exception is thrown by the step "*prepare for patient arrival*", the whole process will be terminated before "*correctly identify patient*" could happen.

- Change: in the property, the option for "*is 'correctly identify patient' required to occur at least once?*" is changed to "*No*".

Iteration 2:

- Event Binding: the same as Iteration 1.

- Result: no violation found.

Note: event "*incorrectly identify patient*", "*make sure that patient has appropriate I.V. access*" and "*make sure that patient is well enough to receive chemotherapy*" are in the alphabet of this property. However, the property doesn't restrict their occurrence in any way. During the verification, I removed these events from the alphabet. I have sent an email to Rachel about this problem.

D.2a That patient must be well enough to receive chemotherapy

Iteration 1:

- Event Binding:

  administer chemotherapy→
  administer chemo drug STARTED
  find that patient is not well enough to receive chemotherapy→
  Exception *VitalSignsNotWithinAcceptedPrameters* is thrown by
  perform review of systems
  make sure that patient is well enough to receive chemotherapy→
  perform review of systems COMPLETED |
  Handle VitalSignsNotWithinAcceptedPrameters COMPLETED

- Result: no violation found.

Iteration 2:

In fact, a patient is consider to be well enough only if all vital signs are within accepted parameters and the blood counts are sufficient. On the other hand, a patient is consider not to be well enough if erither all vital signs are not within accepted parameters or the blood counts are not sufficient. Therefore, two sets of bindings need to be verified.

- Event Binding a:

  administer chemotherapy→
  administer chemo drug STARTED
  find that patient is not well enough to receive chemotherapy→
  Exception *VitalSignsNotWithinAcceptedPrameters* is thrown by
  perform review of systems
  make sure that patient is well enough to receive chemotherapy→
  perform review of systems COMPLETED |
  Handle VitalSignsNotWithinAcceptedPrameters COMPLETED

- Event Binding b:

  administer chemotherapy→

  administer chemo drug STARTED

  find that patient is not well enough to receive chemotherapy→

  Exception *LowBloodCounts* is thrown by

  check blood counts

  make sure that patient is well enough to receive chemotherapy→

  perform review of systems COMPLETED |

  check blood counts COMPLETED |

  handle low blood count COMPLETED |

  handle LowBloodCount COMPLETED

- Result: the property holds with both sets of bindings.

Note: the step "*check blood counts*" appears in two different places in the process: one in the fragment "*perform checks on patient before calling pharmacy*" and the other in the fragment "*perform first day of chemo pharmacy tasks*". And it has two different definitions and different handlers ("*handle low blood count*" and "*handle LowBloodCounts*") in those two places.

D.2b If that patient is not well enough, must wait until that patient is well enough to receive chemotherapy

Iteration 1:

- Event Binding:

  administer chemotherapy→

  administer chemo drug STARTED

  find that patient is not well enough to receive chemotherapy→

Exception *VitalSignsNotWithinAcceptedPrameters* is thrown by

perform review of systems

make sure that patient is well enough to receive chemotherapy→

perform review of systems COMPLETED |

Handle VitalSignsNotWithinAcceptedPrameters COMPLETED

- Result: the property does not hold. The property says that "*make sure that patient is well enough to receive chemotherapy*" must occur between the scope: after "*find that patient is not well enough to receive chemotherapy*" and before "*administer chemotherapy*". In the scope question tree, it requires that "*find that patient is not well enough to receive chemotherapy*" is required to occur. However, the event does not have to happen if any exception thrown earlier terminates the process.

- Change: The option for "*is 'find that patient is not well enough to receive chemotherapy' required?*" is changed to "*No*". And the modified property holds as expected.

Iteration 2:

Similar to D.2a, two sets of bindings need to be verified.

- Event Binding a:

administer chemotherapy→

administer chemo drug STARTED

find that patient is not well enough to receive chemotherapy→

Exception *VitalSignsNotWithinAcceptedPrameters* is thrown by

perform review of systems

make sure that patient is well enough to receive chemotherapy→

perform review of systems COMPLETED |

Handle VitalSignsNotWithinAcceptedPrameters COMPLETED

- Event Binding b:

  administer chemotherapy→

  administer chemo drug STARTED

  find that patient is not well enough to receive chemotherapy→

  Exception *LowBloodCounts* is thrown by

  check blood counts

  make sure that patient is well enough to receive chemotherapy→

  perform review of systems COMPLETED |

  check blood counts COMPLETED |

  handle low blood count COMPLETED |

  handle LowBloodCount COMPLETED

- Result: the property holds with both sets of bindings.

D.3a That patient must have appropriate I.V. access

Iteration 1:

- Event Binding:

  administer chemotherapy→

  administer chemo drug STARTED

  make sure that patient has appropriate I.V. access→

  check patient IV access COMPLETED

  find that patient does not have appropriate I.V. access→

  check patient IV access TERMINATED

- Result: no violation found.

Note: in the process, the step "*check patient IV access*" throws an exceptions. But there is no exception handler for this exception. If this exception is thrown, the whole process will be terminated. If an exception handler step is added into the process later on, the binding for the event "*make sure that patient has appropriate I.V. access*" must be changed, and this property has to be verified again.

D.3b If that patient loses their appropriate I.V. access, new appropriate I.V. access must be found

Iteration 1:

- Event Binding:

  administer chemotherapy→
      administer chemo drug STARTED
  make sure that patient has appropriate I.V. access→
      check patient IV access COMPLETED
  find that patient does not have appropriate I.V. access→
      check patient IV access TERMINATED

- Result: no violation found.

Note: similar to D.3a, if an exception handler step is added into the process to handle the exception thrown by "*check patient IV access*", the binding for the event "*make sure that patient has appropriate I.V. access*" must be changed, and this property has to be verified again.

D.4a (D.1 → D.2) That patient must be correctly identified before making sure that patient is well enough to receive chemotherapy

Iteration 1:

- Event Binding:

administer chemotherapy→

administer chemo drug STARTED

correctly identify patient→

identify patient COMPLETED

make sure that patient has appropriate I.V. access→

check patient IV access COMPLETED

make sure that patient is well enough to receive chemotherapy→

perform review of systems COMPLETED |

check blood counts COMPLETED |

handle low blood count COMPLETED |

handle LowBloodCount COMPLETED |

Handle VitalSignsNotWithinAcceptedPrameters COMPLETED

patient arrives for administration of chemotherapy→

perform checks on patient before calling pharmacy STARTED

- Result: no violation found.

Note: unlike property MP D.2a and MP D.2b, only one set of bindings is created for this property. This property requires that patient must be correctly identified before making sure that patient is well enough to receive chemotherapy. Patient identification must occur before any "*make sure that patient is well enough to receive chemotherapy*" event. Therefore, one can simply associate the event "*make sure that patient is well enough to receive chemotherapy*" with disjunctive bindings.

D.4b (D.2 → D.3) Patient must be well enough to receive chemotherapy before making sure that patient has appropriate I.V. Access

Iteration 1:

- Event Binding:

administer chemotherapy→

administer chemo drug STARTED

correctly identify patient→

identify patient COMPLETED

make sure that patient has appropriate I.V. access→

check patient IV access COMPLETED

make sure that patient is well enough to receive chemotherapy→

perform review of systems COMPLETED |

check blood counts COMPLETED |

handle low blood count COMPLETED |

handle LowBloodCount COMPLETED |

Handle VitalSignsNotWithinAcceptedPrameters COMPLETED

patient arrives for administration of chemotherapy→

perform checks on patient before calling pharmacy STARTED

- Result: no violation found.

Part 2. Checks required for the chemotherapy drugs

D.7a All chemotherapy drugs must be physically suitable for administration before chemotherapy can be administered

- Event Binding: no binding for event "*make sure all chemotherapy drugs must be physically suitable for administration*".

E Activities Required While Chemotherapy is being Administered

E.1 If a patient has an adverse reaction to an administration of chemo, the administration must be stopped immediately

Iteration 1:

- Event Binding:

  attending MD or attending MD's delegate decides patient disposition→
  call doctor STARTED
  patient has adverse reaction to administration of chemotherapy→
  Exception AllergicReactionToChemoDrug is thrown
  stabilize patient condition→
  give emergency meds STARTED
  stop administration of chemotherapy→
  stop pump STARTED

- Result: no violation found.

E.2 If a patient has an adverse reaction to an administration of chemo, that patient's condition must be stabilized

Iteration 1:

- Event Binding:

  attending MD or attending MD's delegate decides patient disposition→
  call doctor STARTED
  patient has adverse reaction to administration of chemotherapy→
  Exception AllergicReactionToChemoDrug is thrown
  stabilize patient condition→
  give emergency meds STARTED

- Result: violation found. The property requires that no other events are allowed to occur after "*patient has adverse reaction to administration of chemotherapy*" and before "*stabilize patient condition*". In the process, the exception handler "*handle AllergicReactionToChemoDrug*" is sequential step that has four sub-steps:

"*stop pump*", "*call doctor*", "*give emergency meds*", and "*send patient home*". It is obvious that "*Attending MD or Attending MD's delegate decides patient disposition*" will occur between "*patient has adverse reaction to administration of chemotherapy*" and "*stabilize patient condition*".

E.3 If a patient has an adverse reaction to an administration of chemo, that patient's disposition must be decided

Iteration 1:

- Event Binding:

  attending MD or attending MD's delegate decides patient disposition→
  call doctor STARTED
  patient has adverse reaction to administration of chemotherapy→
  Exception AllergicReactionToChemoDrug is thrown

- Result: no violation found.

F Activities Required Right After Chemotherapy has been Administered

F.1a A patient must be well enough to be discharged

Iteration 1:

- Event Binding:

  discharge patient→
  discharge patient STARTED
  patient becomes too ill to be discharged→
  Exception AllergicReactionToChemoDrug is thrown
  patient becomes well enough to be discharged→

check patient OK COMPLETED |

handle AllergicReactionToChemoDrug COMPLETED

- Result: no violation found.

F.1b If a patient becomes too ill to be discharged, they must become well enough

Iteration 1:

- Event Binding:

discharge patient→

discharge patient STARTED

patient becomes too ill to be discharged→

Exception AllergicReactionToChemoDrug is thrown

patient becomes well enough to be discharged→

check patient OK COMPLETED |

handle AllergicReactionToChemoDrug COMPLETED

- Result: no violation found.

F.2 If a patient needs observation, they must be observed

- Event Binding: no binding for event "*patient needs observation*".

F.3 After chemotherapy is administered, all prescriptions for necessary post-chemotherapy supportive care medications must be given to patient

- Event Binding: no binding for event "*give all necessary prescriptions for post-chemotherapy supportive care medications to patient*".

F.4 After chemotherapy is administered, all post-chemotherapy instructions must be given to patient

Iteration 1:

- Event Binding:

  discharge patient→

  discharge patient STARTED

  administer chemotherapy→

  administer chemo drug COMPLETED

  give all post-chemotherapy instructions to patient→

  provide discharge instructions COMPLETED

- Result: no violation found.

F.5 After chemotherapy is administered, a follow-up appointment must be scheduled

Iteration 1:

- Event Binding:

  discharge patient→

  discharge patient STARTED

  administer chemotherapy→

  administer chemo drug COMPLETED

  schedule follow-up appointment→

  ensure presence of appointment COMPLETED

- Result: no violation found.

F.6 After chemotherapy is administered, all laboratory results and chemo administration data must be entered into that patient's record

Iteration 1:

- Event Binding:

  business day ends→
      first day of chemo COMPLETED
  administer chemotherapy→
      administer chemo drug STARTED
  enter all patient laboratory results and chemotherapy
          administration data into patient record→
      chart clinical summary COMPLETED

- Result: violation found. The violation trace shows that after the chemotherapy is administered, an exception *AllergicReactionToChemoDrug* may be thrown from the step "*check patient OK*". This exception will be handled by a handler with the step "*handle AllergicReactionToChemoDrug*". The handler has a COMPLETE badge, which means that the step "*chart clinical summary*" will not be executed.

APPENDIX J

CHEMOTHERAPY PROCESS FAULT TREE

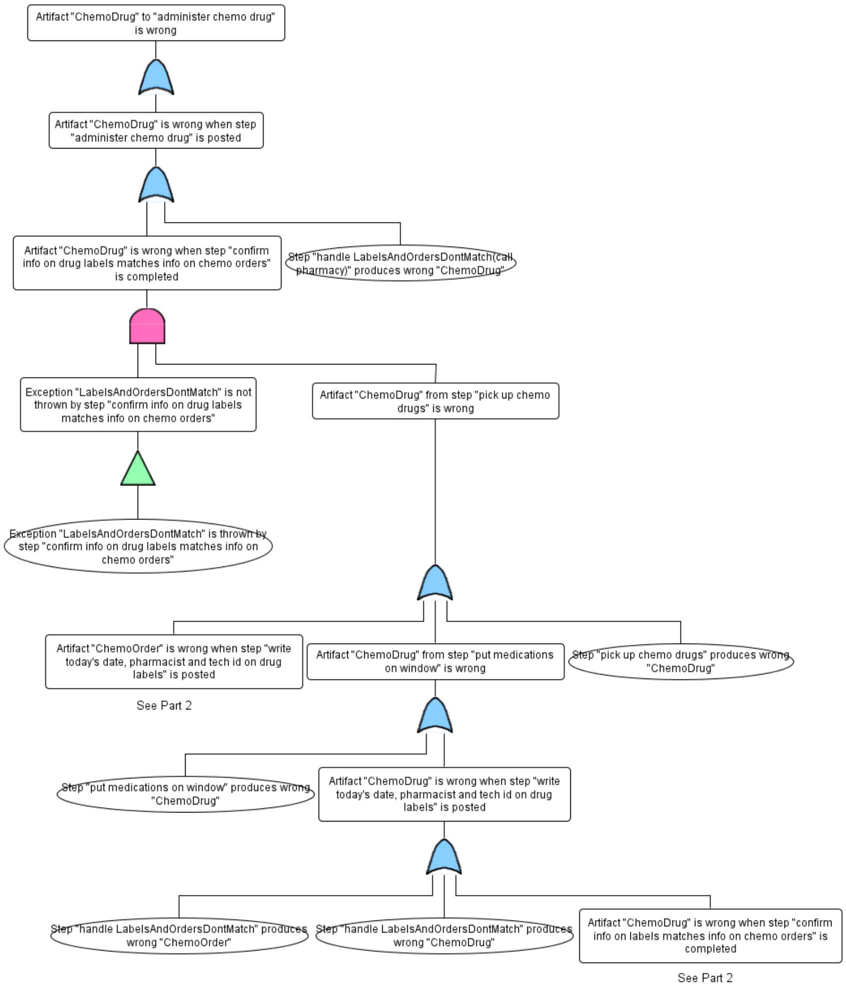

Artifact "ChemoDrug" to "administer chemo drug" is wrong
Artifact "ChemoDrug" is wrong when step "administer chemo drug" is posted
Artifact "ChemoDrug" is wrong when step "confirm info on drug labels matches info on chemo orders" is completed
Step "handle LabelsAndOrdersDontMatch(call pharmacy)" produces wrong "ChemoDrug"
Exception "LabelsAndOrdersDontMatch" is not thrown by step "confirm info on drug labels matches info on chemo orders"
Artifact "ChemoDrug" from step "pick up chemo drugs" is wrong
Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders"
Artifact "ChemoOrder" is wrong when step "write today's date, pharmacist and tech id on drug labels" is posted
See Part 2
Artifact "ChemoDrug" from step "put medications on window" is wrong
Step "pick up chemo drugs" produces wrong "ChemoDrug"
Step "put medications on window" produces wrong "ChemoDrug"
Artifact "ChemoDrug" is wrong when step "write today's date, pharmacist and tech id on drug labels" is posted
Step "handle LabelsAndOrdersDontMatch" produces wrong "ChemoOrder"
Step "handle LabelsAndOrdersDontMatch" produces wrong "ChemoDrug"
Artifact "ChemoDrug" is wrong when step "confirm info on labels matches info on chemo orders" is completed
See Part 2

Figure J.1. Chemotherapy Process Fault Tree Part 1

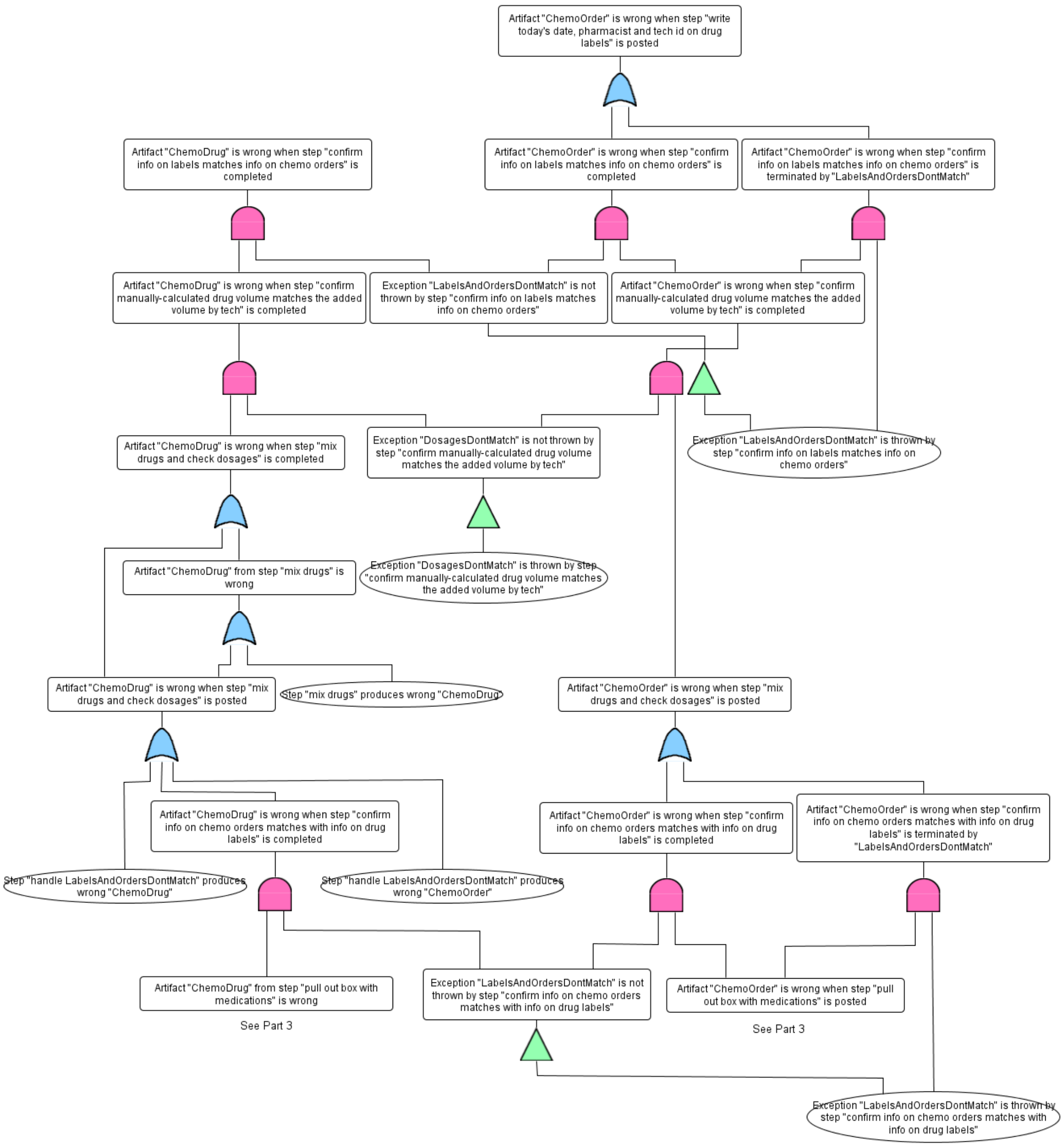

Artifact "ChemoOrder" is wrong when step "write today's date, pharmacist and tech id on drug labels" is posted
Artifact "ChemoDrug" is wrong when step "confirm info on labels matches info on chemo orders" is completed
Artifact "ChemoOrder" is wrong when step "confirm info on labels matches info on chemo orders" is completed
Artifact "ChemoOrder" is wrong when step "confirm info on labels matches info on chemo orders" is terminated by "LabelsAndOrdersDontMatch"
Artifact "ChemoDrug" is wrong when step "confirm manually-calculated drug volume matches the added volume by tech" is completed
Exception "LabelsAndOrdersDontMatch" is not thrown by step "confirm info on labels matches info on chemo orders"
Artifact "ChemoOrder" is wrong when step "confirm manually-calculated drug volume matches the added volume by tech" is completed
Artifact "ChemoDrug" is wrong when step "mix drugs and check dosages" is completed
Exception "DosagesDontMatch" is not thrown by step "confirm manually-calculated drug volume matches the added volume by tech"
Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on labels matches info on chemo orders"
Artifact "ChemoDrug" from step "mix drugs" is wrong
Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"
Artifact "ChemoDrug" is wrong when step "mix drugs and check dosages" is posted
Step "mix drugs" produces wrong "ChemoDrug"
Artifact "ChemoOrder" is wrong when step "mix drugs and check dosages" is posted
Artifact "ChemoDrug" is wrong when step "confirm info on chemo orders matches with info on drug labels" is completed
Artifact "ChemoOrder" is wrong when step "confirm info on chemo orders matches with info on drug labels" is completed
Artifact "ChemoOrder" is wrong when step "confirm info on chemo orders matches with info on drug labels" is terminated by "LabelsAndOrdersDontMatch"
Step "handle LabelsAndOrdersDontMatch" produces wrong "ChemoDrug"
Step "handle LabelsAndOrdersDontMatch" produces wrong "ChemoOrder"
Artifact "ChemoDrug" from step "pull out box with medications" is wrong
See Part 3
Exception "LabelsAndOrdersDontMatch" is not thrown by step "confirm info on chemo orders matches with info on drug labels"
Artifact "ChemoOrder" is wrong when step "pull out box with medications" is posted
See Part 3
Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on chemo orders matches with info on drug labels"

Figure J.2. Chemotherapy Process Fault Tree Part 2

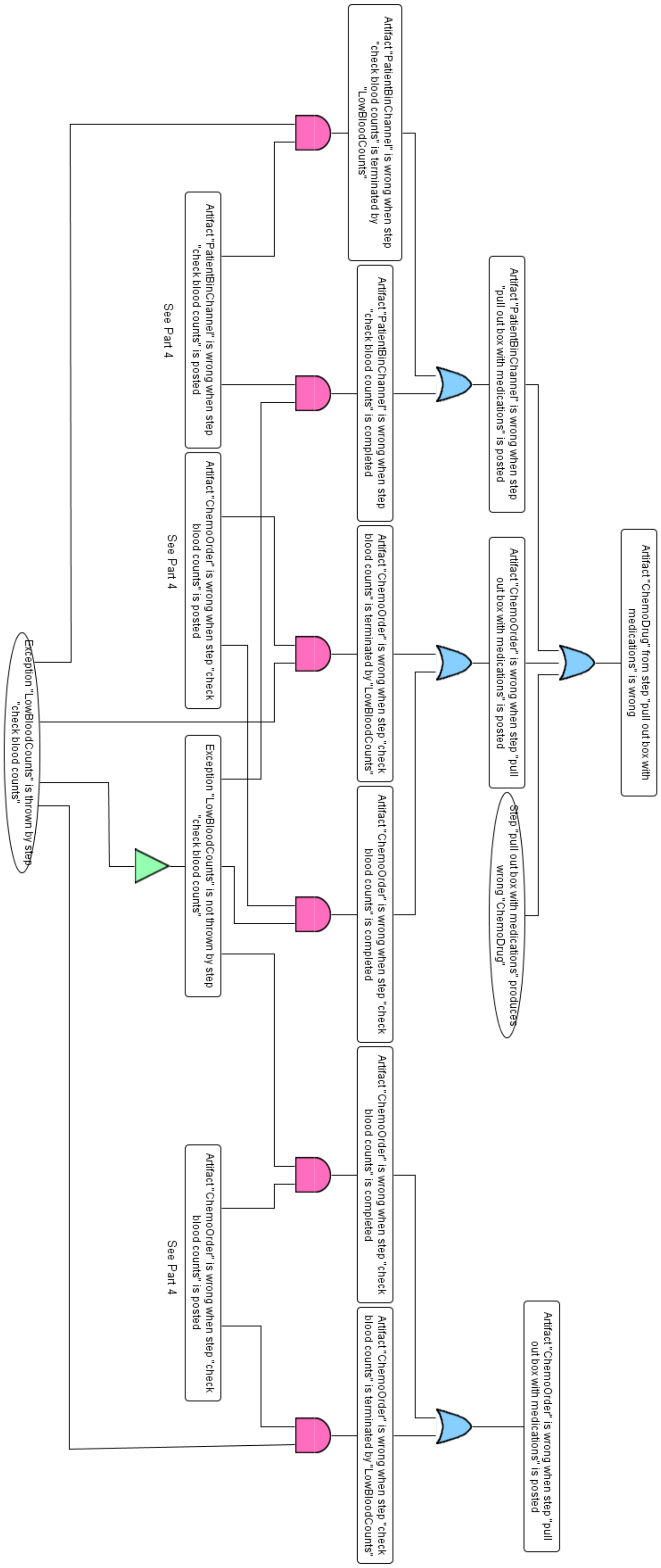

Artifact"PatientBinChannel" is wrong when step "check blood counts" is terminated by "LowBloodCounts"
Artifact"PatientBinChannel" is wrong when step "check blood counts" is posted
See Part 4
Artifact"PatientBinChannel" is wrong when step "check blood counts" is completed
Artifact"PatientBinChannel" is wrong when step "pull out box with medications" is posted
Artifact"ChemoDrug" from step "pull out box with medications" is wrong
Artifact"ChemoOrder" is wrong when step "check blood counts" is posted
See Part 4
Artifact"ChemoOrder" is wrong when step "check blood counts" is terminated by "LowBloodCounts"
Artifact"ChemoOrder" is wrong when step "pull out box with medications" is posted
Exception "LowBloodCounts" is thrown by step "check blood counts"
Exception "LowBloodCounts" is not thrown by step "check blood counts"
Artifact"ChemoOrder" is wrong when step "check blood counts" is completed
Step "pull out box with medications" produces wrong "ChemoDrug"
Artifact"ChemoOrder" is wrong when step "check blood counts" is completed
Artifact"ChemoOrder" is wrong when step "check blood counts" is posted
See Part 4
Artifact"ChemoOrder" is wrong when step "check blood counts" is terminated by "LowBloodCounts"
Artifact"ChemoOrder" is wrong when step "pull out box with medications" is posted

Figure J.3. Chemotherapy Process Fault Tree Part 3

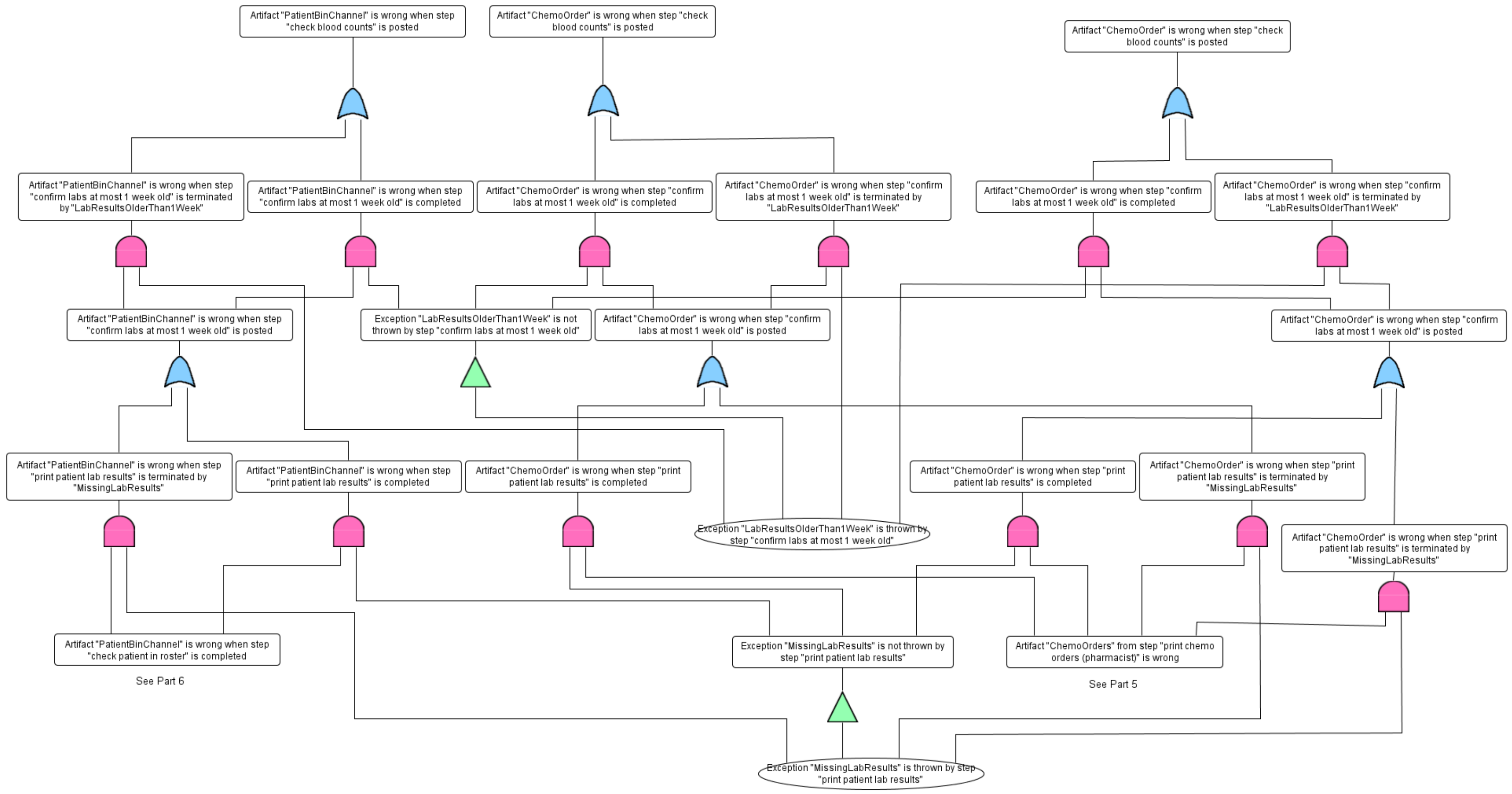

Artifact "PatientBinChannel" is wrong when step "check blood counts" is posted
Artifact "ChemoOrder" is wrong when step "check blood counts" is posted
Artifact "ChemoOrder" is wrong when step "check blood counts" is posted
Artifact "PatientBinChannel" is wrong when step "confirm labs at most 1 week old" is terminated by "LabResultsOlderThan1Week"
Artifact "PatientBinChannel" is wrong when step "confirm labs at most 1 week old" is completed
Artifact "ChemoOrder" is wrong when step "confirm labs at most 1 week old" is completed
Artifact "ChemoOrder" is wrong when step "confirm labs at most 1 week old" is terminated by "LabResultsOlderThan1Week"
Artifact "ChemoOrder" is wrong when step "confirm labs at most 1 week old" is completed
Artifact "ChemoOrder" is wrong when step "confirm labs at most 1 week old" is terminated by "LabResultsOlderThan1Week"
Artifact "PatientBinChannel" is wrong when step "confirm labs at most 1 week old" is posted
Exception "LabResultsOlderThan1Week" is not thrown by step "confirm labs at most 1 week old"
Artifact "ChemoOrder" is wrong when step "confirm labs at most 1 week old" is posted
Artifact "ChemoOrder" is wrong when step "confirm labs at most 1 week old" is posted
Artifact "PatientBinChannel" is wrong when step "print patient lab results" is terminated by "MissingLabResults"
Artifact "PatientBinChannel" is wrong when step "print patient lab results" is completed
Artifact "ChemoOrder" is wrong when step "print patient lab results" is completed
Exception "LabResultsOlderThan1Week" is thrown by step "confirm labs at most 1 week old"
Artifact "ChemoOrder" is wrong when step "print patient lab results" is completed
Artifact "ChemoOrder" is wrong when step "print patient lab results" is terminated by "MissingLabResults"
Artifact "ChemoOrder" is wrong when step "print patient lab results" is terminated by "MissingLabResults"
Artifact "PatientBinChannel" is wrong when step "check patient in roster" is completed
See Part 6
Exception "MissingLabResults" is not thrown by step "print patient lab results"
Artifact "ChemoOrders" from step "print chemo orders (pharmacist)" is wrong
See Part 5
Exception "MissingLabResults" is thrown by step "print patient lab results"

Figure J.4. Chemotherapy Process Fault Tree Part 4

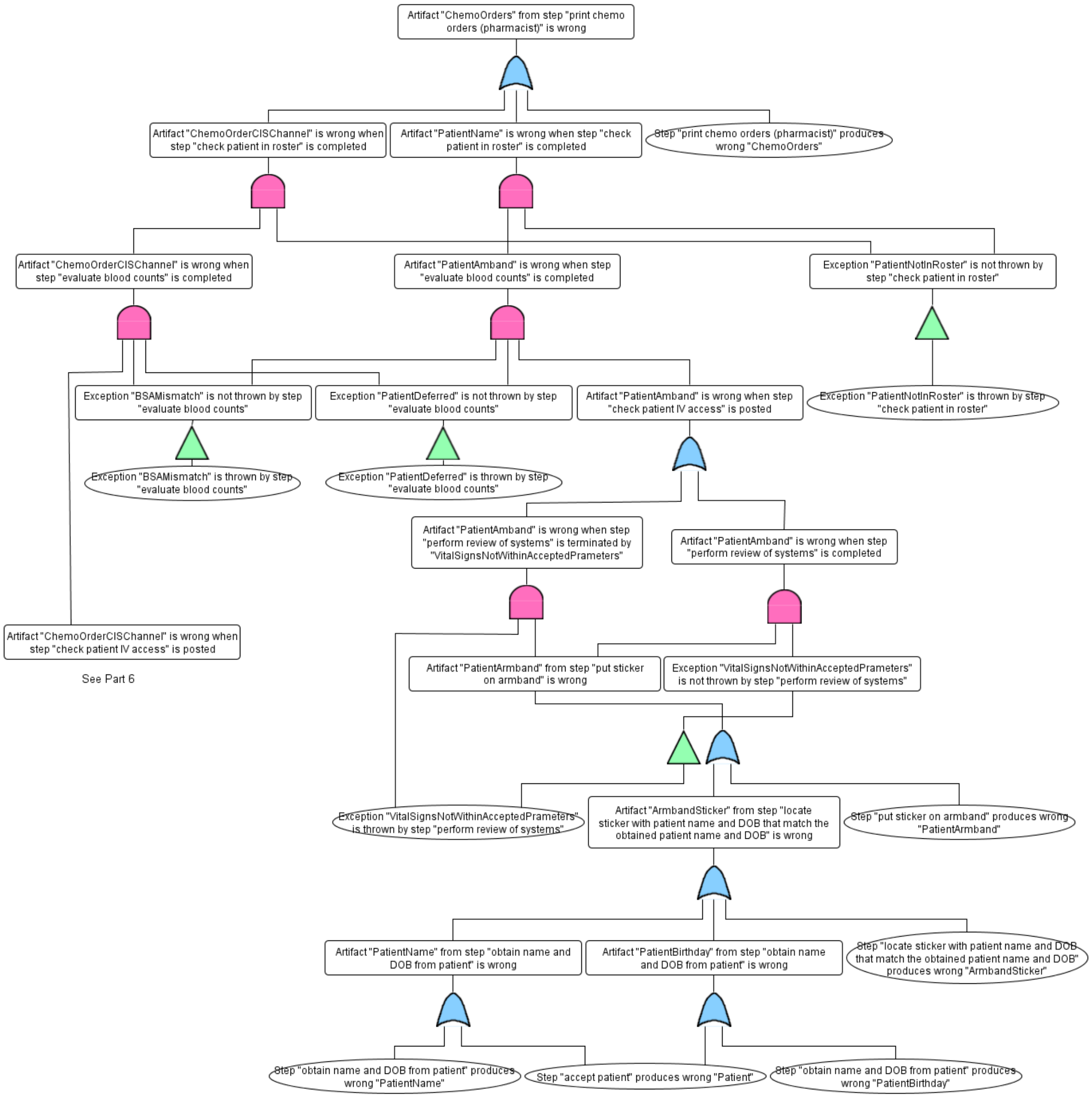

Artifact "ChemoOrders" from step "print chemo orders (pharmacist)" is wrong
Artifact "ChemoOrderCISChannel" is wrong when step "check patient in roster" is completed
Artifact "PatientName" is wrong when step "check patient in roster" is completed
Step "print chemo orders (pharmacist)" produces wrong "ChemoOrders"
Artifact "ChemoOrderCISChannel" is wrong when step "evaluate blood counts" is completed
Artifact "PatientAmband" is wrong when step "evaluate blood counts" is completed
Exception "PatientNotInRoster" is not thrown by step "check patient in roster"
Exception "BSAMismatch" is not thrown by step "evaluate blood counts"
Exception "PatientDeferred" is not thrown by step "evaluate blood counts"
Artifact "PatientAmband" is wrong when step "check patient IV access" is posted
Exception "PatientNotInRoster" is thrown by step "check patient in roster"
Exception "BSAMismatch" is thrown by step "evaluate blood counts"
Exception "PatientDeferred" is thrown by step "evaluate blood counts"
Artifact "PatientAmband" is wrong when step "perform review of systems" is terminated by "VitalSignsNotWithinAcceptedPrameters"
Artifact "PatientAmband" is wrong when step "perform review of systems" is completed
Artifact "ChemoOrderCISChannel" is wrong when step "check patient IV access" is posted
See Part 6
Artifact "PatientArmband" from step "put sticker on armband" is wrong
Exception "VitalSignsNotWithinAcceptedPrameters" is not thrown by step "perform review of systems"
Exception "VitalSignsNotWithinAcceptedPrameters" is thrown by step "perform review of systems"
Artifact "ArmbandSticker" from step "locate sticker with patient name and DOB that match the obtained patient name and DOB" is wrong
Step "put sticker on armband" produces wrong "PatientArmband"
Artifact "PatientName" from step "obtain name and DOB from patient" is wrong
Artifact "PatientBirthday" from step "obtain name and DOB from patient" is wrong
Step "locate sticker with patient name and DOB that match the obtained patient name and DOB" produces wrong "ArmbandSticker"
Step "obtain name and DOB from patient" produces wrong "PatientName"
Step "accept patient" produces wrong "Patient"
Step "obtain name and DOB from patient" produces wrong "PatientBirthday"

Figure J.5. Chemotherapy Process Fault Tree Part 5

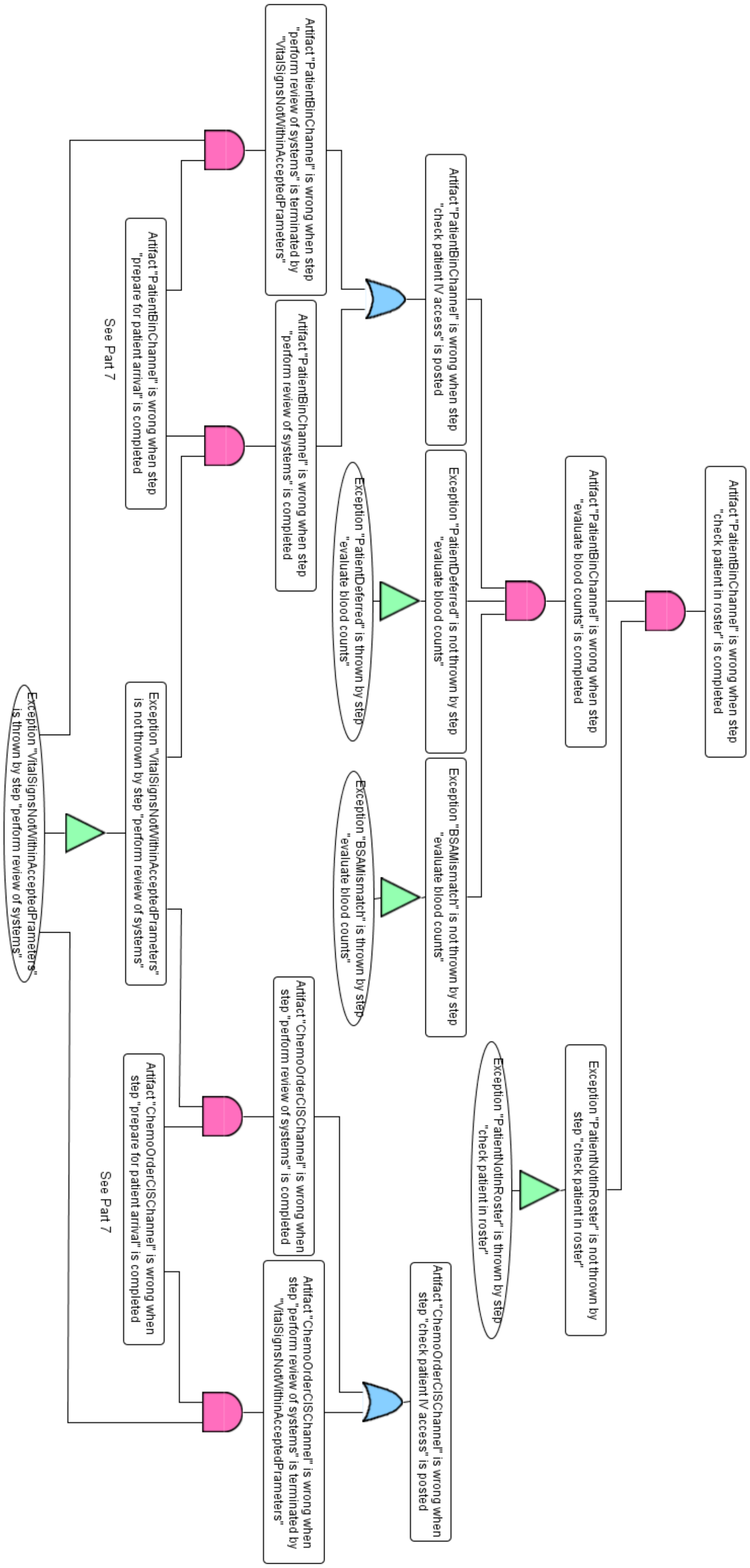

Artifact"PatientBinChannel" is wrong when step "perform review of systems" is terminated by "VitalSignsNotWithinAcceptedPrameters"
Artifact"PatientBinChannel" is wrong when step "prepare for patient arrival" is completed
See Part 7
Artifact"PatientBinChannel" is wrong when step "check patient IV access" is posted
Artifact"PatientBinChannel" is wrong when step "perform review of systems" is completed
Exception "PatientDeferred" is thrown by step "evaluate blood counts"
Exception "PatientDeferred" is not thrown by step "evaluate blood counts"
Artifact"PatientBinChannel" is wrong when step "evaluate blood counts" is completed
Artifact"PatientBinChannel" is wrong when step "check patient in roster" is completed
Exception "VitalSignsNotWithinAcceptedPrameters" is thrown by step "perform review of systems"
Exception "VitalSignsNotWithinAcceptedPrameters" is not thrown by step "perform review of systems"
Exception "BSAMismatch" is thrown by step "evaluate blood counts"
Exception "BSAMismatch" is not thrown by step "evaluate blood counts"
Artifact"ChemoOrderCISChannel" is wrong when step "perform review of systems" is completed
Exception "PatientNotInRoster" is thrown by step "check patient in roster"
Exception "PatientNotInRoster" is not thrown by step "check patient in roster"
Artifact"ChemoOrderCISChannel" is wrong when step "prepare for patient arrival" is completed
See Part 7
Artifact"ChemoOrderCISChannel" is wrong when step "perform review of systems" is terminated by "VitalSignsNotWithinAcceptedPrameters"
Artifact"ChemoOrderCISChannel" is wrong when step "check patient IV access" is posted

Figure J.6. Chemotherapy Process Fault Tree Part 6

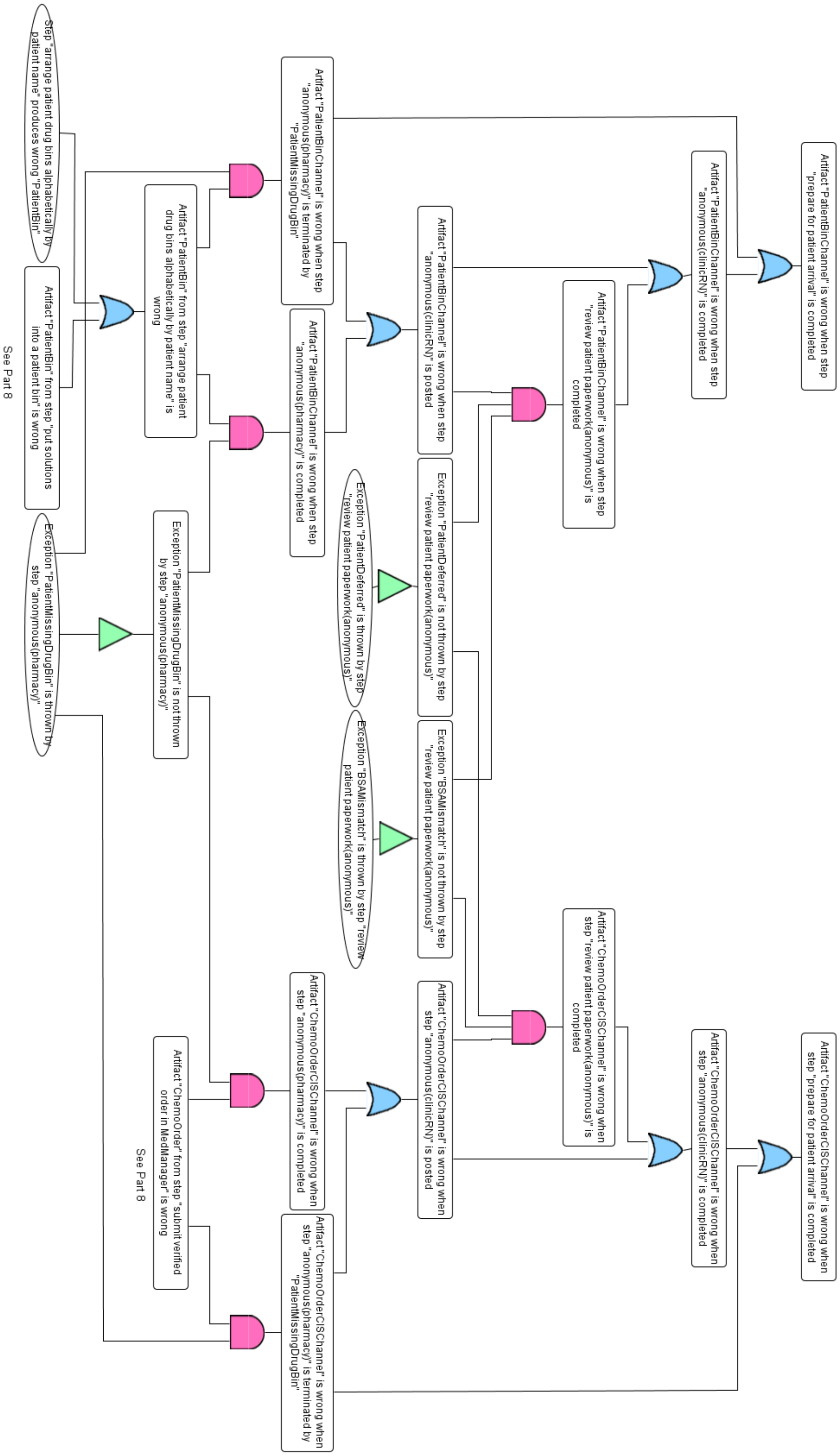

Artifact"PatientBinChannel" is wrong when step "prepare for patient arrival" is completed
Artifact"PatientBinChannel" is wrong when step "anonymous(clinicRN)" is completed
Artifact"PatientBinChannel" is wrong when step "review patient paperwork(anonymous)" is completed
Artifact"PatientBinChannel" is wrong when step "anonymous(clinicRN)" is posted
Exception "PatientDeferred" is not thrown by step "review patient paperwork(anonymous)"
Exception "BSAMismatch" is not thrown by step "review patient paperwork(anonymous)"
Exception "PatientDeferred" is thrown by step "review patient paperwork(anonymous)"
Exception "BSAMismatch" is thrown by step "review patient paperwork(anonymous)"
Artifact"PatientBinChannel" is wrong when step "anonymous(pharmacy)" is terminated by "PatientMissingDrugBin"
Artifact"PatientBinChannel" is wrong when step "anonymous(pharmacy)" is completed
Artifact"PatientBin" from step "arrange patient drug bins alphabetically by patient name" is wrong
Exception "PatientMissingDrugBin" is not thrown by step "anonymous(pharmacy)"
Step "arrange patient drug bins alphabetically by patient name" produces wrong "PatientBin"
Artifact"PatientBin" from step "put solutions into a patient bin" is wrong
Exception "PatientMissingDrugBin" is thrown by step "anonymous(pharmacy)"
See Part 8
Artifact"ChemoOrderCISChannel" is wrong when step "prepare for patient arrival" is completed
Artifact"ChemoOrderCISChannel" is wrong when step "anonymous(clinicRN)" is completed
Artifact"ChemoOrderCISChannel" is wrong when step "review patient paperwork(anonymous)" is completed
Artifact"ChemoOrderCISChannel" is wrong when step "anonymous(clinicRN)" is posted
Artifact"ChemoOrderCISChannel" is wrong when step "anonymous(pharmacy)" is completed
Artifact"ChemoOrderCISChannel" is wrong when step "anonymous(pharmacy)" is terminated by "PatientMissingDrugBin"
Artifact"ChemoOrder" from step "submit verified order in MedManager" is wrong
See Part 8

Figure J.7. Chemotherapy Process Fault Tree Part 7

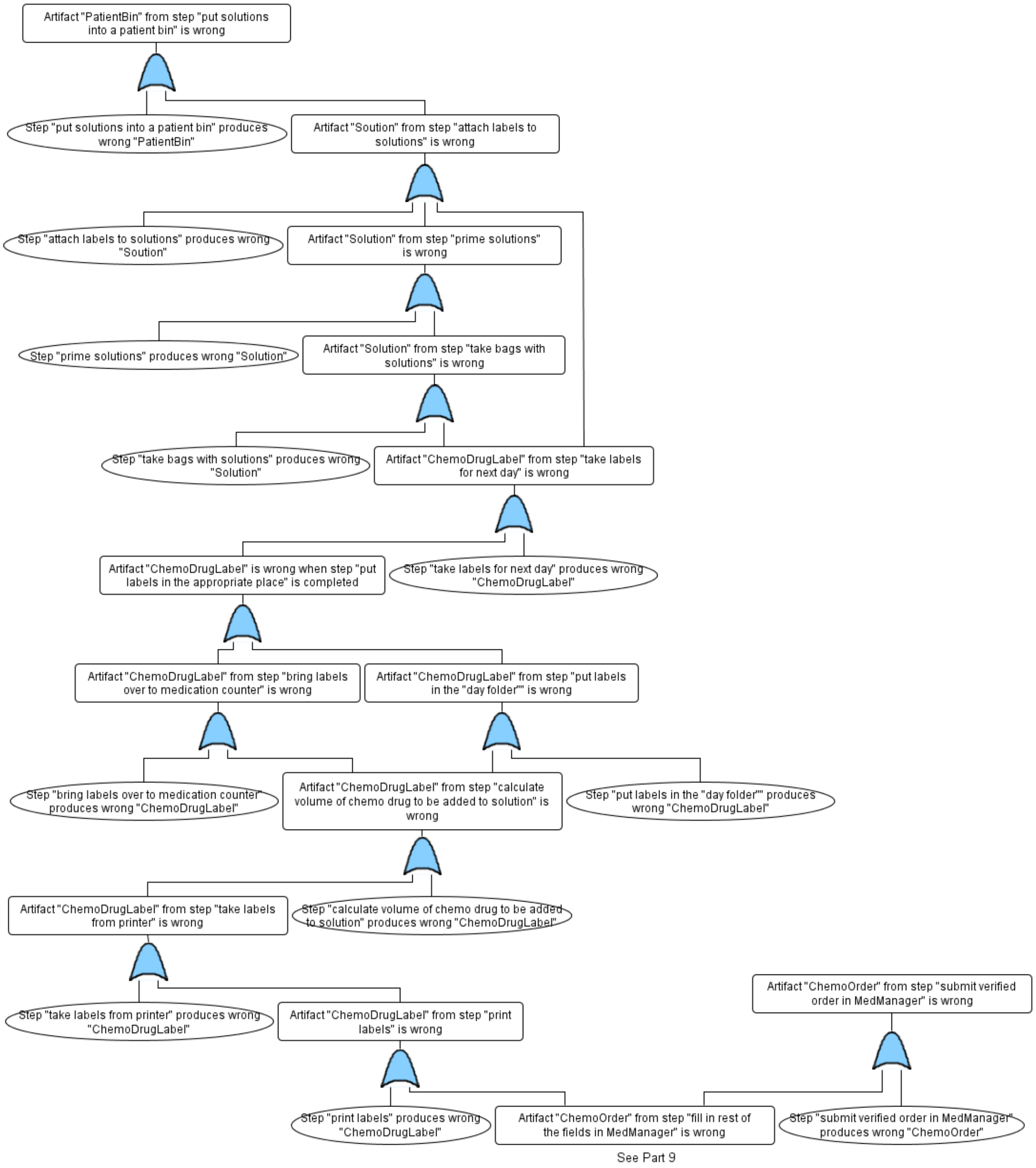

Artifact "PatientBin" from step "put solutions into a patient bin" is wrong
Step "put solutions into a patient bin" produces wrong "PatientBin"
Artifact "Soution" from step "attach labels to solutions" is wrong
Step "attach labels to solutions" produces wrong "Soution"
Artifact "Solution" from step "prime solutions" is wrong
Step "prime solutions" produces wrong "Solution"
Artifact "Solution" from step "take bags with solutions" is wrong
Step "take bags with solutions" produces wrong "Solution"
Artifact "ChemoDrugLabel" from step "take labels for next day" is wrong
Artifact "ChemoDrugLabel" is wrong when step "put labels in the appropriate place" is completed
Step "take labels for next day" produces wrong "ChemoDrugLabel"
Artifact "ChemoDrugLabel" from step "bring labels over to medication counter" is wrong
Artifact "ChemoDrugLabel" from step "put labels in the "day folder"" is wrong
Step "bring labels over to medication counter" produces wrong "ChemoDrugLabel"
Artifact "ChemoDrugLabel" from step "calculate volume of chemo drug to be added to solution" is wrong
Step "put labels in the "day folder"" produces wrong "ChemoDrugLabel"
Artifact "ChemoDrugLabel" from step "take labels from printer" is wrong
Step "calculate volume of chemo drug to be added to solution" produces wrong "ChemoDrugLabel"
Artifact "ChemoOrder" from step "submit verified order in MedManager" is wrong
Step "take labels from printer" produces wrong "ChemoDrugLabel"
Artifact "ChemoDrugLabel" from step "print labels" is wrong
Step "print labels" produces wrong "ChemoDrugLabel"
Artifact "ChemoOrder" from step "fill in rest of the fields in MedManager" is wrong
Step "submit verified order in MedManager" produces wrong "ChemoOrder"
See Part 9

Figure J.8. Chemotherapy Process Fault Tree Part 8

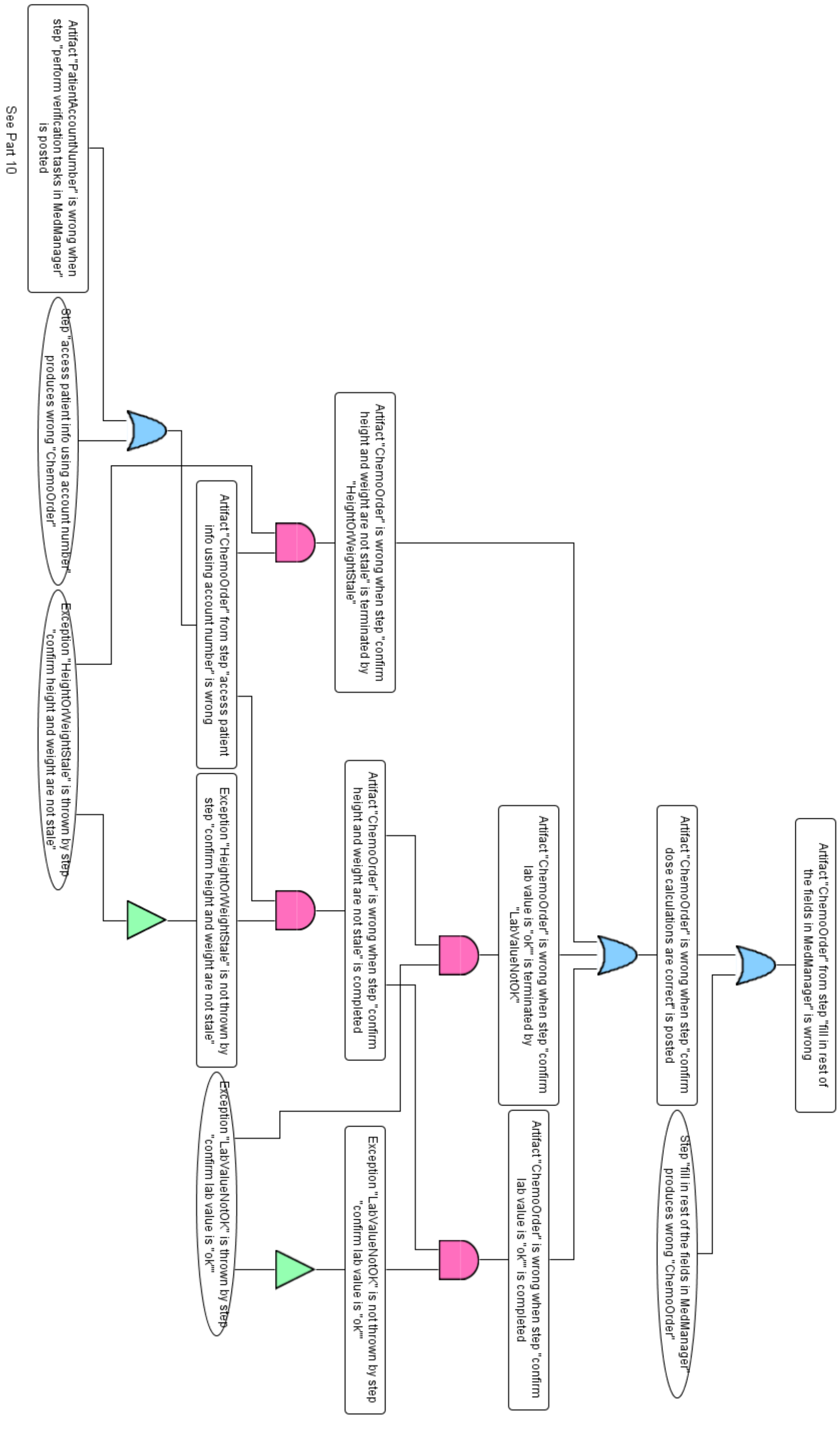

Artifact "ChemoOrder" from step "fill in rest of the fields in MedManager" is wrong
Artifact "ChemoOrder" is wrong when step "confirm dose calculations are correct" is posted
Step "fill in rest of the fields in MedManager" produces wrong "ChemoOrder"
Artifact "ChemoOrder" is wrong when step "confirm lab value is "ok"" is terminated by "LabValueNotOK"
Artifact "ChemoOrder" is wrong when step "confirm lab value is "ok"" is completed
Artifact "ChemoOrder" is wrong when step "confirm height and weight are not stale" is terminated by "HeightOrWeightStale"
Artifact "ChemoOrder" is wrong when step "confirm height and weight are not stale" is completed
Exception "LabValueNotOK" is not thrown by step "confirm lab value is "ok""
Artifact "ChemoOrder" from step "access patient info using account number" is wrong
Exception "HeightOrWeightStale" is not thrown by step "confirm height and weight are not stale"
Exception "LabValueNotOK" is thrown by step "confirm lab value is "ok""
Step "access patient info using account number" produces wrong "ChemoOrder"
Exception "HeightOrWeightStale" is thrown by step "confirm height and weight are not stale"
Artifact "PatientAccountNumber" is wrong when step "perform verification tasks in MedManager" is posted
See Part 10

Figure J.9. Chemotherapy Process Fault Tree Part 9

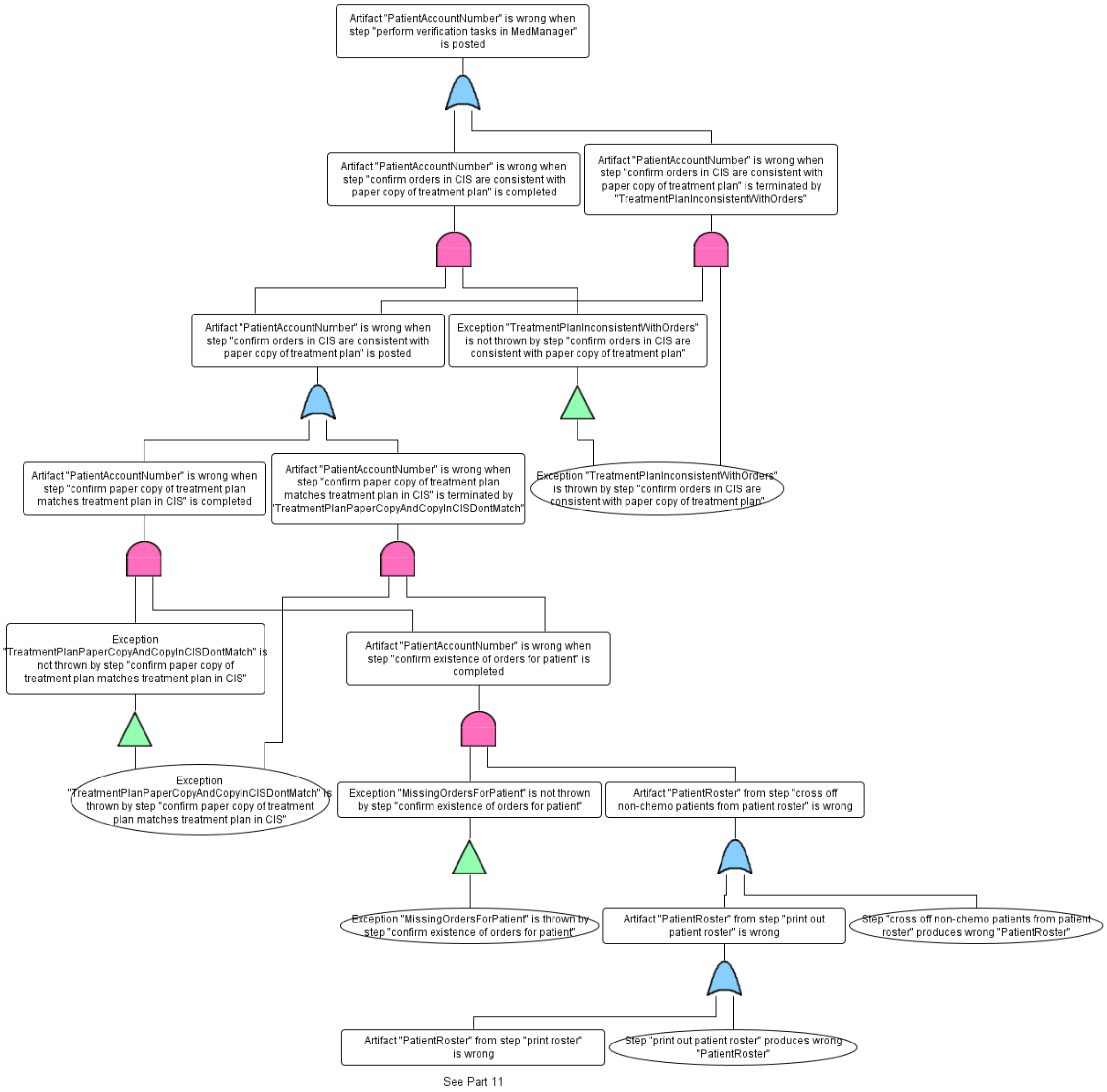

Artifact "PatientAccountNumber" is wrong when step "perform verification tasks in MedManager" is posted
Artifact "PatientAccountNumber" is wrong when step "confirm orders in CIS are consistent with paper copy of treatment plan" is completed
Artifact "PatientAccountNumber" is wrong when step "confirm orders in CIS are consistent with paper copy of treatment plan" is terminated by "TreatmentPlanInconsistentWithOrders"
Artifact "PatientAccountNumber" is wrong when step "confirm orders in CIS are consistent with paper copy of treatment plan" is posted
Exception "TreatmentPlanInconsistentWithOrders" is not thrown by step "confirm orders in CIS are consistent with paper copy of treatment plan"
Artifact "PatientAccountNumber" is wrong when step "confirm paper copy of treatment plan matches treatment plan in CIS" is completed
Artifact "PatientAccountNumber" is wrong when step "confirm paper copy of treatment plan matches treatment plan in CIS" is terminated by "TreatmentPlanPaperCopyAndCopyInCISDontMatch"
Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "confirm orders in CIS are consistent with paper copy of treatment plan"
Exception "TreatmentPlanPaperCopyAndCopyInCISDontMatch" is not thrown by step "confirm paper copy of treatment plan matches treatment plan in CIS"
Artifact "PatientAccountNumber" is wrong when step "confirm existence of orders for patient" is completed
Exception "TreatmentPlanPaperCopyAndCopyInCISDontMatch" is thrown by step "confirm paper copy of treatment plan matches treatment plan in CIS"
Exception "MissingOrdersForPatient" is not thrown by step "confirm existence of orders for patient"
Artifact "PatientRoster" from step "cross off non-chemo patients from patient roster" is wrong
Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"
Artifact "PatientRoster" from step "print out patient roster" is wrong
Step "cross off non-chemo patients from patient roster" produces wrong "PatientRoster"
Artifact "PatientRoster" from step "print roster" is wrong
Step "print out patient roster" produces wrong "PatientRoster"
See Part 11

Figure J.10. Chemotherapy Process Fault Tree Part 10

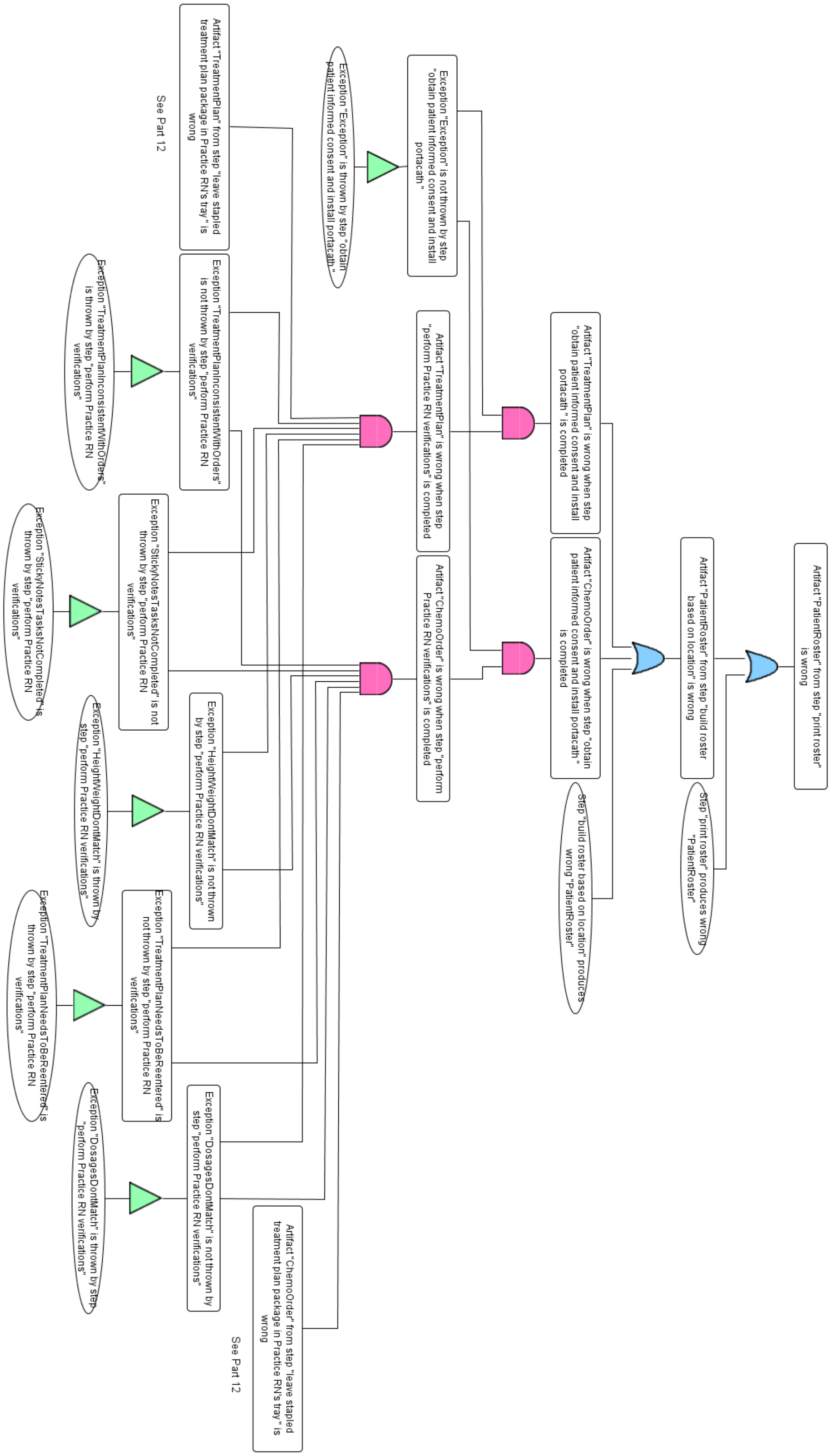

Artifact "PatientRoster" from step "print roster" is wrong
Artifact "PatientRoster" from step "build roster based on location" is wrong
Step "print roster" produces wrong "PatientRoster"
Artifact "TreatmentPlan" is wrong when step "obtain patient informed consent and install portacath" is completed
Artifact "ChemoOrder" is wrong when step "obtain patient informed consent and install portacath" is completed
Step "build roster based on location" produces wrong "PatientRoster"
Exception "Exception" is not thrown by step "obtain patient informed consent and install portacath"
Exception "Exception" is thrown by step "obtain patient informed consent and install portacath"
Artifact "TreatmentPlan" is wrong when step "perform Practice RN verifications" is completed
Artifact "ChemoOrder" is wrong when step "perform Practice RN verifications" is completed
Artifact "TreatmentPlan" from step "leave stapled treatment plan package in Practice RN's tray" is wrong
See Part 12
Exception "TreatmentPlanInconsistentWithOrders" is not thrown by step "perform Practice RN verifications"
Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"
Exception "StickyNotesTasksNotCompleted" is not thrown by step "perform Practice RN verifications"
Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"
Exception "HeightWeightDontMatch" is not thrown by step "perform Practice RN verifications"
Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"
Exception "TreatmentPlanNeedsToBeReentered" is not thrown by step "perform Practice RN verifications"
Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"
Exception "DosagesDontMatch" is not thrown by step "perform Practice RN verifications"
Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"
Artifact "ChemoOrder" from step "leave stapled treatment plan package in Practice RN's tray" is wrong
See Part 12

Figure J.11. Chemotherapy Process Fault Tree Part 11

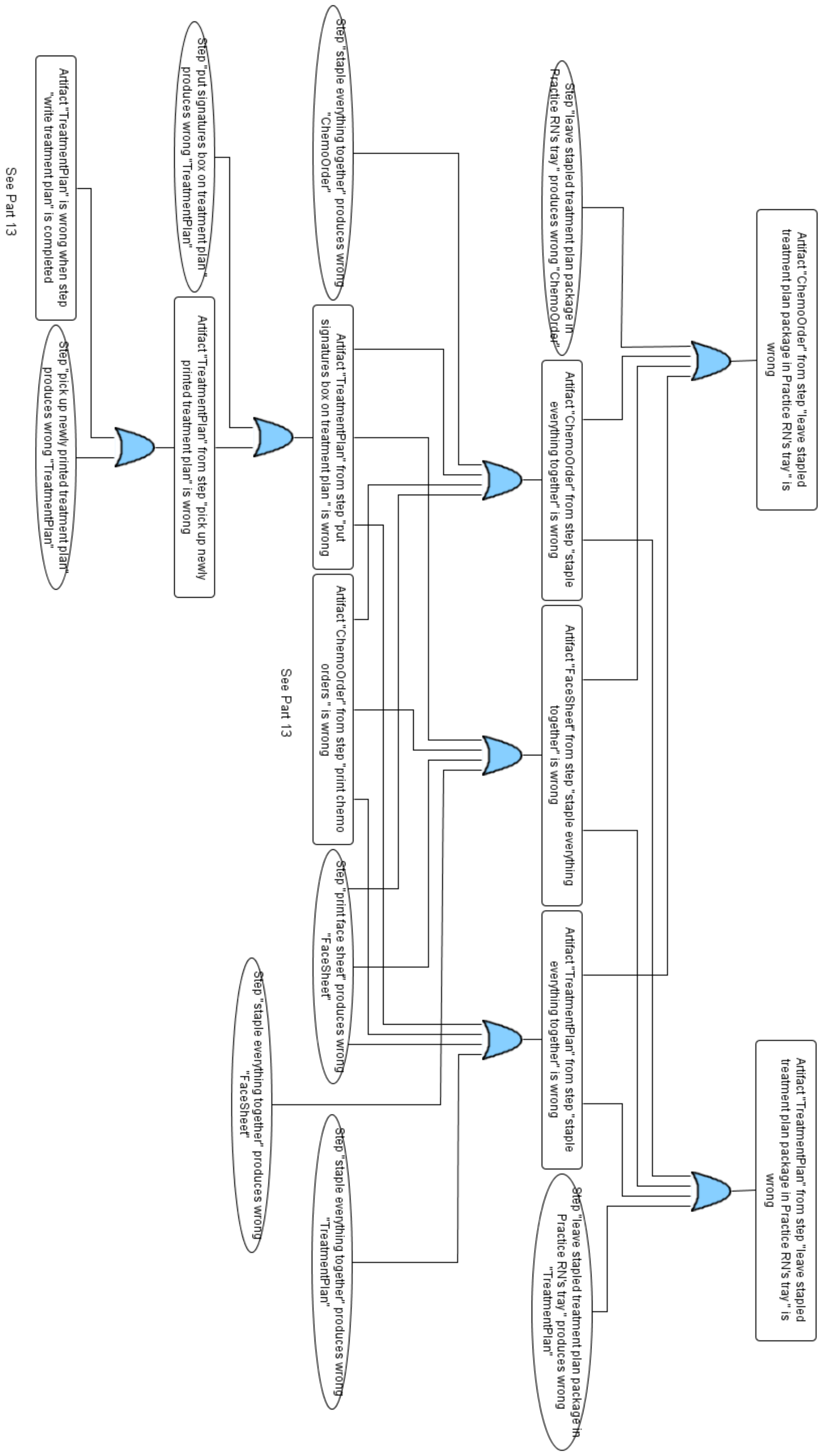

Artifact "ChemoOrder" from step "leave stapled treatment plan package in Practice RN's tray" is wrong
Artifact "TreatmentPlan" from step "leave stapled treatment plan package in Practice RN's tray" is wrong
Step "leave stapled treatment plan package in Practice RN's tray" produces wrong "ChemoOrder"
Artifact "ChemoOrder" from step "staple everything together" is wrong
Artifact "FaceSheet" from step "staple everything together" is wrong
Artifact "TreatmentPlan" from step "staple everything together" is wrong
Step "leave stapled treatment plan package in Practice RN's tray" produces wrong "TreatmentPlan"
Step "staple everything together" produces wrong "ChemoOrder"
Artifact "TreatmentPlan" from step "put signatures box on treatment plan" is wrong
Artifact "ChemoOrder" from step "print chemo orders" is wrong
See Part 13
Step "print face sheet" produces wrong "FaceSheet"
Step "staple everything together" produces wrong "TreatmentPlan"
Step "staple everything together" produces wrong "FaceSheet"
Step "put signatures box on treatment plan" produces wrong "TreatmentPlan"
Artifact "TreatmentPlan" from step "pick up newly printed treatment plan" is wrong
Artifact "TreatmentPlan" is wrong when step "write treatment plan" is completed
Step "pick up newly printed treatment plan" produces wrong "TreatmentPlan"
See Part 13

Figure J.12. Chemotherapy Process Fault Tree Part 12

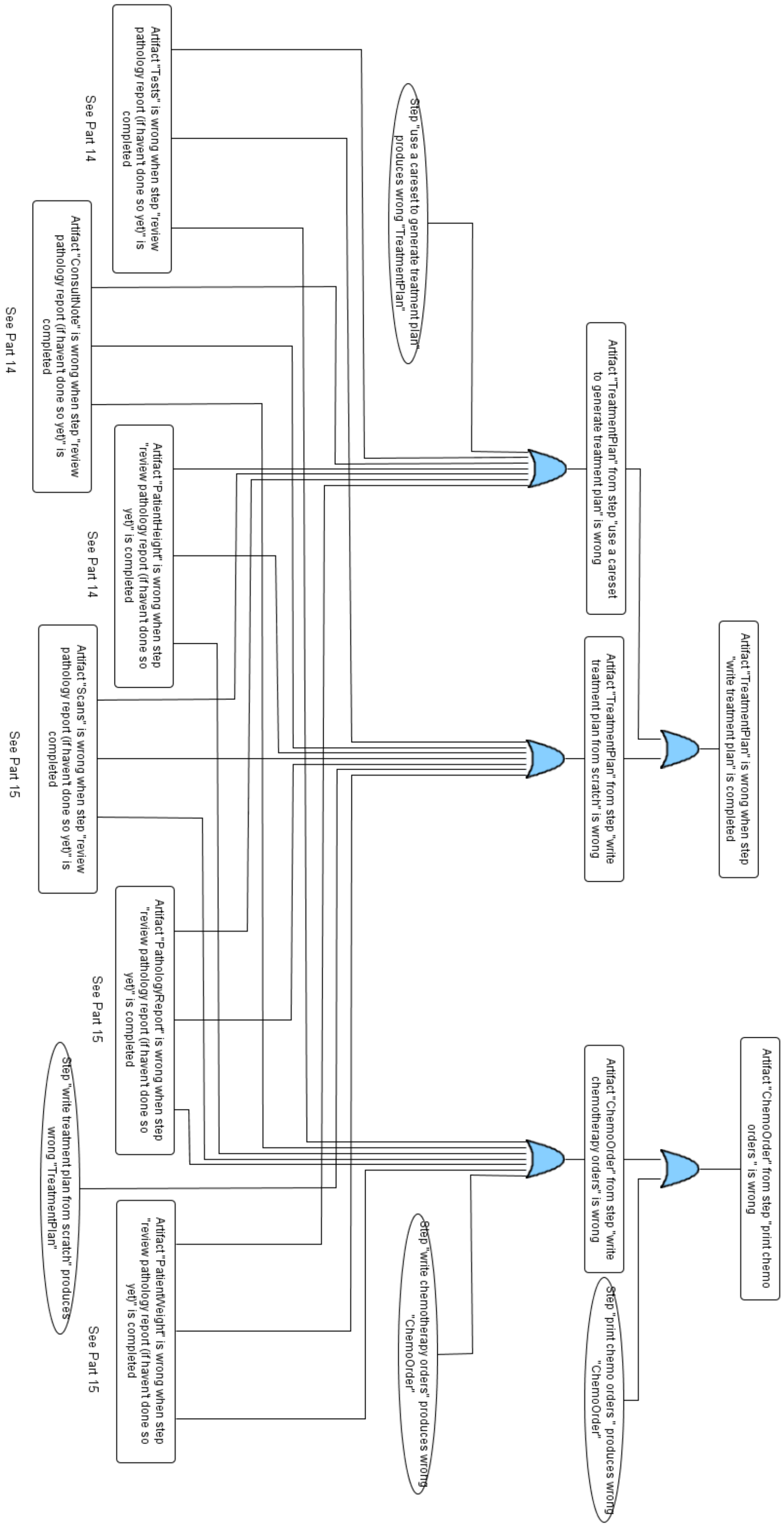

Artifact "Tests" is wrong when step "review pathology report (if haven't done so yet)" is completed
See Part 14
Step "use a careset to generate treatment plan" produces wrong "TreatmentPlan"
Artifact "ConsultNote" is wrong when step "review pathology report (if haven't done so yet)" is completed
See Part 14
Artifact "TreatmentPlan" from step "use a careset to generate treatment plan" is wrong
Artifact "PatientHeight" is wrong when step "review pathology report (if haven't done so yet)" is completed
See Part 14
Artifact "TreatmentPlan" from step "write treatment plan from scratch" is wrong
Artifact "TreatmentPlan" is wrong when step "write treatment plan" is completed
Artifact "Scans" is wrong when step "review pathology report (if haven't done so yet)" is completed
See Part 15
Artifact "PathologyReport" is wrong when step "review pathology report (if haven't done so yet)" is completed
See Part 15
Step "write treatment plan from scratch" produces wrong "TreatmentPlan"
Artifact "ChemoOrder" from step "write chemotherapy orders" is wrong
Artifact "ChemoOrder" from step "print chemo orders" is wrong
Step "write chemotherapy orders" produces wrong "ChemoOrder"
Artifact "PatientWeight" is wrong when step "review pathology report (if haven't done so yet)" is completed
See Part 15
Step "print chemo orders" produces wrong "ChemoOrder"

Figure J.13. Chemotherapy Process Fault Tree Part 13

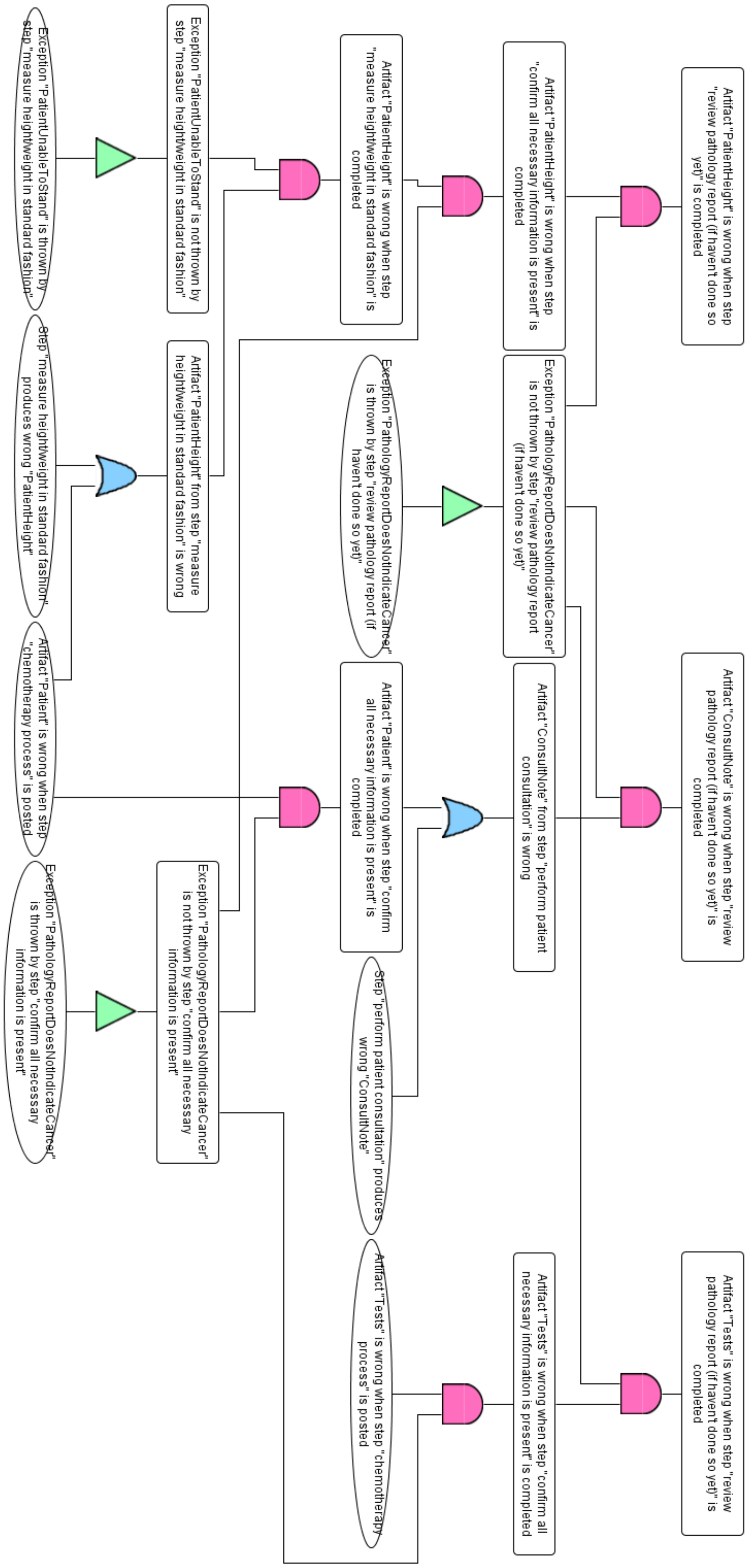

Artifact "PatientHeight" is wrong when step "review pathology report (if haven't done so yet)" is completed
Artifact "ConsultNote" is wrong when step "review pathology report (if havent done so yet)" is completed
Artifact "Tests" is wrong when step "review pathology report (if havent done so yet)" is completed
Artifact "PatientHeight" is wrong when step "confirm all necessary information is present" is completed
Exception "PathologyReportDoesNotIndicateCancer" is not thrown by step "review pathology report (if haven't done so yet)"
Artifact "ConsultNote" from step "perform patient consultation" is wrong
Artifact "Tests" is wrong when step "confirm all necessary information is present" is completed
Artifact "PatientHeight" is wrong when step "measure height/weight in standard fashion" is completed
Exception "PathologyReportDoesNotIndicateCancer" is thrown by step "review pathology report (if haven't done so yet)"
Artifact "Patient" is wrong when step "confirm all necessary information is present" is completed
Step "perform patient consultation" produces wrong "ConsultNote"
Artifact "Tests" is wrong when step "chemotherapy process" is posted
Exception "PatientUnableToStand" is not thrown by step "measure height/weight in standard fashion"
Artifact "PatientHeight" from step "measure height/weight in standard fashion" is wrong
Exception "PathologyReportDoesNotIndicateCancer" is not thrown by step "confirm all necessary information is present"
Exception "PatientUnableToStand" is thrown by step "measure height/weight in standard fashion"
Step "measure height/weight in standard fashion" produces wrong "PatientHeight"
Artifact "Patient" is wrong when step "chemotherapy process" is posted
Exception "PathologyReportDoesNotIndicateCancer" is thrown by step "confirm all necessary information is present"

Figure J.14. Chemotherapy Process Fault Tree Part 14

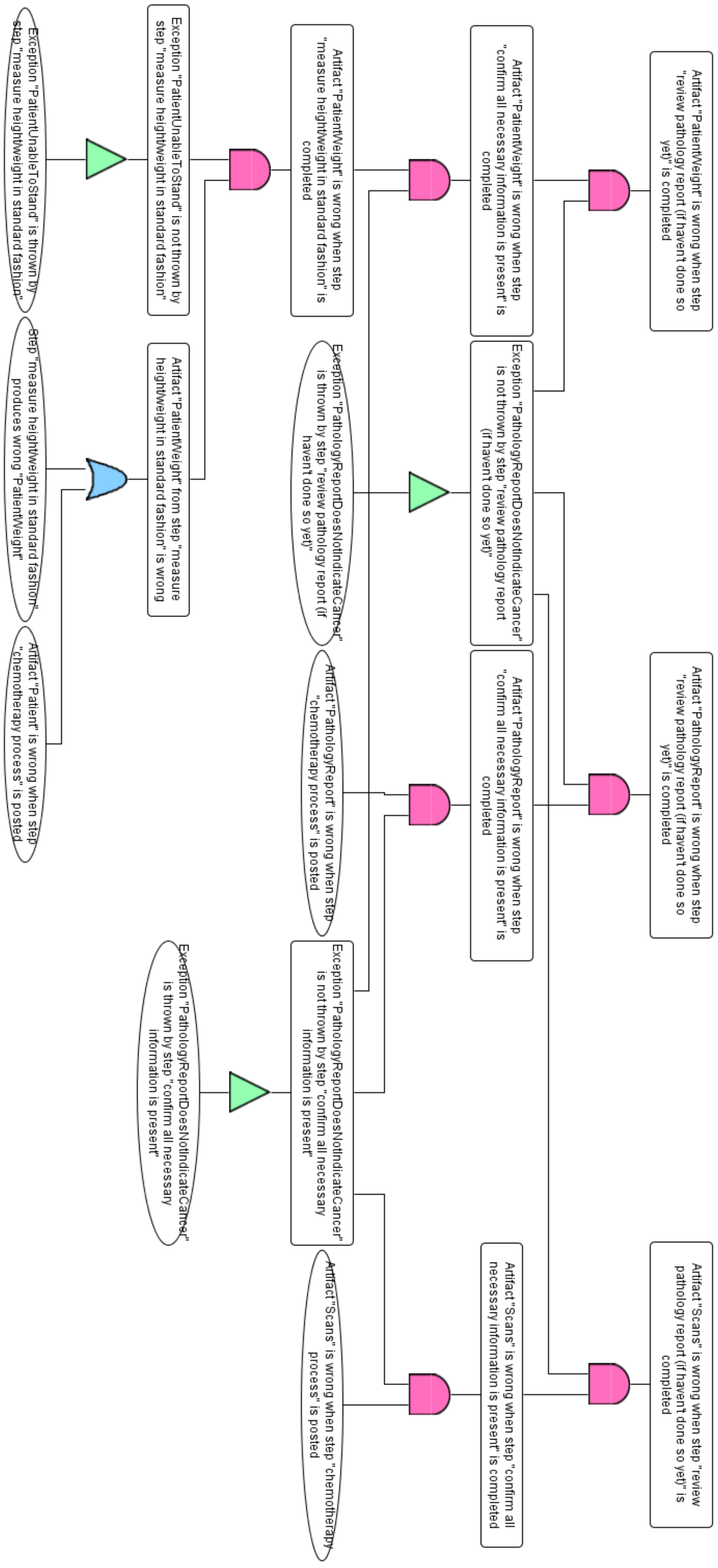

Artifact "PatientWeight" is wrong when step "review pathology report (if haven't done so yet)" is completed
Artifact "PathologyReport" is wrong when step "review pathology report (if haven't done so yet)" is completed
Artifact "Scans" is wrong when step "review pathology report (if haven't done so yet)" is completed
Artifact "PatientWeight" is wrong when step "confirm all necessary information is present" is completed
Exception "PathologyReportDoesNotIndicateCancer" is not thrown by step "review pathology report (if haven't done so yet)"
Artifact "PathologyReport" is wrong when step "confirm all necessary information is present" is completed
Artifact "Scans" is wrong when step "confirm all necessary information is present" is completed
Artifact "PatientWeight" is wrong when step "measure height/weight in standard fashion" is completed
Exception "PathologyReportDoesNotIndicateCancer" is thrown by step "review pathology report (if haven't done so yet)"
Artifact "PathologyReport" is wrong when step "chemotherapy process" is posted
Exception "PathologyReportDoesNotIndicateCancer" is not thrown by step "confirm all necessary information is present"
Artifact "Scans" is wrong when step "chemotherapy process" is posted
Exception "PatientUnableToStand" is not thrown by step "measure height/weight in standard fashion"
Artifact "PatientWeight" from step "measure height/weight in standard fashion" is wrong
Exception "PathologyReportDoesNotIndicateCancer" is thrown by step "confirm all necessary information is present"
Exception "PatientUnableToStand" is thrown by step "measure height/weight in standard fashion"
Step "measure height/weight in standard fashion" produces wrong "PatientWeight"
Artifact "Patient" is wrong when step "chemotherapy process" is posted

Figure J.15. Chemotherapy Process Fault Tree Part 15

APPENDIX K

CHEMOTHERAPY PROCESS MCSS

Total Number of MCSs: 52

MCS 1 (1 events):

{

Step “handle LabelsAndOrdersDontMatch(call pharmacy)” produces wrong “ChemoDrug”

}

MCS 2 (2 events):

{

Step “pick up chemo drugs” produces wrong “ChemoDrug”,

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 3 (2 events):

{

Step “put medications on window” produces wrong “ChemoDrug”,

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 4 (2 events):

{

Step "handle LabelsAndOrdersDontMatch" produces wrong "ChemoDrug",

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 5 (2 events):

{

Step "handle LabelsAndOrdersDontMatch" produces wrong "ChemoOrder",

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 6 (3 events):

{

Step "print chemo orders (pharmacist)" produces wrong "ChemoOrders",

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 7 (4 events):

{

Step "mix drugs" produces wrong "ChemoDrug",

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on labels matches info on chemo orders"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 8 (4 events):

{

Step "handle LabelsAndOrdersDontMatch" produces wrong "ChemoDrug",

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on labels matches info on chemo orders"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 9 (4 events):

{

Step "handle LabelsAndOrdersDontMatch" produces wrong "ChemoOrder",

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on labels matches info on chemo orders"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 10 (5 events):

{

Step "pull out box with medications" produces wrong "ChemoDrug",

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on chemo orders matches with info on drug labels"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on labels matches info on chemo orders"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 11 (6 events):

{

Step "submit verified order in MedManager" produces wrong "ChemoOrder",

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 12 (6 events):

{

Step “fill in rest of the fields in MedManager” produces wrong “ChemoOrder”,

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 13 (6 events):

{

Step “access patient info using account number” produces wrong “ChemoOrder”,

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 14 (6 events):

{

Step "put sticker on armband" produces wrong "PatientArmband",

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 15 (6 events):

{

Step "locate sticker with patient name and DOB that match the obtained patient name and DOB" produces wrong "ArmbandSticker",

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 16 (6 events):

{

Step "obtain name and DOB from patient" produces wrong "PatientBirthday",

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 17 (6 events):

{

Step "obtain name and DOB from patient" produces wrong "PatientName",

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 18 (6 events):

{

Step "accept patient" produces wrong "Patient",

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 19 (7 events):

{

Step “cross off non-chemo patients from patient roster” produces wrong “PatientRoster”,

!(Exception “MissingOrdersForPatient” is thrown by step “confirm existence of orders for patient”),

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 20 (7 events):

{

Step “print out patient roster” produces wrong “PatientRoster”,

!(Exception “MissingOrdersForPatient” is thrown by step “confirm existence of orders for patient”),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 21 (7 events):

{

Step "print roster" produces wrong "PatientRoster",

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 22 (7 events):

{

Step "build roster based on location" produces wrong "PatientRoster",

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence

of orders for patient”),

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 23 (8 events):

{

Step “take bags with solutions” produces wrong “Solution”,

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on chemo orders matches with info on drug labels”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on labels matches info on chemo orders”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 24 (8 events):

{

Step "put labels in the "day folder"" produces wrong "ChemoDrugLabel",

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on chemo orders matches with info on drug labels"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on labels matches info on chemo orders"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 25 (8 events):

{

Step "attach labels to solutions" produces wrong "Soution",

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on chemo orders matches with info on drug labels"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info

on labels matches info on chemo orders"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 26 (8 events):

{

Step "print labels" produces wrong "ChemoDrugLabel",

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on chemo orders matches with info on drug labels"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on labels matches info on chemo orders"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 27 (8 events):

{

Step "put solutions into a patient bin" produces wrong "PatientBin",

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on chemo orders matches with info on drug labels"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on labels matches info on chemo orders"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 28 (8 events):

{

Step "prime solutions" produces wrong "Solution",

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on chemo orders matches with info on drug labels"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on labels matches info on chemo orders"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 29 (8 events):

{

Step "take labels from printer" produces wrong "ChemoDrugLabel",

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on chemo orders matches with info on drug labels"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on labels matches info on chemo orders"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 30 (8 events):

{

Step "take labels for next day" produces wrong "ChemoDrugLabel",

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on chemo orders matches with info on drug labels"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info

on labels matches info on chemo orders”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 31 (8 events):

{

Step “bring labels over to medication counter” produces wrong “ChemoDrugLabel”,

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on chemo orders matches with info on drug labels”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on labels matches info on chemo orders”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 32 (8 events):

{

Step “calculate volume of chemo drug to be added to solution” produces wrong “ChemoDrugLabel”,

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on chemo orders matches with info on drug labels”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on labels matches info on chemo orders”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 33 (8 events):

{

Step “arrange patient drug bins alphabetically by patient name” produces wrong “PatientBin”,

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on chemo orders matches with info on drug labels”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on labels matches info on chemo orders”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 34 (13 events):

{

Step "leave stapled treatment plan package in Practice RN's tray " produces wrong "ChemoOrder",

!(Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception "Exception" is thrown by step "obtain patient informed consent and install portacath "),

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 35 (13 events):

{

Step "staple everything together" produces wrong "ChemoOrder",

!(Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception "Exception" is thrown by step "obtain patient informed consent and install portacath "),

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 36 (13 events):

{

Step "put signatures box on treatment plan " produces wrong "TreatmentPlan",

!(Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception "Exception" is thrown by step "obtain patient informed consent and install portacath "),

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 37 (13 events):

{

Step “print chemo orders ” produces wrong “ChemoOrder”,

!(Exception “TreatmentPlanInconsistentWithOrders” is thrown by step “perform Practice RN verifications”),

!(Exception “TreatmentPlanNeedsToBeReentered” is thrown by step “perform Practice RN verifications”),

!(Exception “HeightWeightDontMatch” is thrown by step “perform Practice RN verifications”),

!(Exception “DosagesDontMatch” is thrown by step “perform Practice RN verifications”),

!(Exception “StickyNotesTasksNotCompleted” is thrown by step “perform Practice RN verifications”),

!(Exception “Exception” is thrown by step “obtain patient informed consent and install portacath ”),

!(Exception “MissingOrdersForPatient” is thrown by step “confirm existence of orders for patient”),

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 38 (13 events):

{

Step "print face sheet" produces wrong "FaceSheet",

!(Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception "Exception" is thrown by step "obtain patient informed consent and install portacath "),

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 39 (13 events):

{

Step "write chemotherapy orders" produces wrong "ChemoOrder",

!(Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception "Exception" is thrown by step "obtain patient informed consent and install portacath "),

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 40 (13 events):

{

Step “use a careset to generate treatment plan” produces
wrong “TreatmentPlan”,

!(Exception “TreatmentPlanInconsistentWithOrders” is thrown by step
“perform Practice RN verifications”),

!(Exception “TreatmentPlanNeedsToBeReentered” is thrown by step
“perform Practice RN verifications”),

!(Exception “HeightWeightDontMatch” is thrown by step “perform Practice
RN verifications”),

!(Exception “DosagesDontMatch” is thrown by step “perform Practice
RN verifications”),

!(Exception “StickyNotesTasksNotCompleted” is thrown by step
“perform Practice RN verifications”),

!(Exception “Exception” is thrown by step “obtain patient informed
consent and install portacath ”),

!(Exception “MissingOrdersForPatient” is thrown by step “confirm existence
of orders for patient”),

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “DosagesDontMatch” is thrown by step “confirm
manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info
on drug labels matches info on chemo orders”)

}

MCS 41 (13 events):

{

Step "pick up newly printed treatment plan" produces wrong "TreatmentPlan",

!(Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception "Exception" is thrown by step "obtain patient informed consent and install portacath "),

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 42 (13 events):

{

Step "staple everything together" produces wrong "FaceSheet",

!(Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception "Exception" is thrown by step "obtain patient informed consent and install portacath "),

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 43 (13 events):

{

Step “leave stapled treatment plan package in Practice RN’s tray ” produces wrong “TreatmentPlan”,

!(Exception “TreatmentPlanInconsistentWithOrders” is thrown by step “perform Practice RN verifications”),

!(Exception “TreatmentPlanNeedsToBeReentered” is thrown by step “perform Practice RN verifications”),

!(Exception “HeightWeightDontMatch” is thrown by step “perform Practice RN verifications”),

!(Exception “DosagesDontMatch” is thrown by step “perform Practice RN verifications”),

!(Exception “StickyNotesTasksNotCompleted” is thrown by step “perform Practice RN verifications”),

!(Exception “Exception” is thrown by step “obtain patient informed consent and install portacath ”),

!(Exception “MissingOrdersForPatient” is thrown by step “confirm existence of orders for patient”),

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 44 (13 events):

{

Step "write treatment plan from scratch" produces wrong "TreatmentPlan",

!(Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception "Exception" is thrown by step "obtain patient informed consent and install portacath "),

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 45 (13 events):

{

Step "staple everything together" produces wrong "TreatmentPlan",

!(Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception "Exception" is thrown by step "obtain patient informed consent and install portacath "),

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 46 (14 events):

{

Step “perform patient consultation” produces wrong “ConsultNote”,

!(Exception “PathologyReportDoesNotIndicateCancer” is thrown by step “review pathology report (if haven’t done so yet)”),

!(Exception “TreatmentPlanInconsistentWithOrders” is thrown by step “perform Practice RN verifications”),

!(Exception “TreatmentPlanNeedsToBeReentered” is thrown by step “perform Practice RN verifications”),

!(Exception “HeightWeightDontMatch” is thrown by step “perform Practice RN verifications”),

!(Exception “DosagesDontMatch” is thrown by step “perform Practice RN verifications”),

!(Exception “StickyNotesTasksNotCompleted” is thrown by step “perform Practice RN verifications”),

!(Exception “Exception” is thrown by step “obtain patient informed consent and install portacath ”),

!(Exception “MissingOrdersForPatient” is thrown by step “confirm existence of orders for patient”),

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info

on drug labels matches info on chemo orders")

}

MCS 47 (15 events):

{

Artifact "Tests" is wrong when step "chemotherapy process" is posted,

!(Exception "PathologyReportDoesNotIndicateCancer" is thrown by step "confirm all necessary information is present"),

!(Exception "PathologyReportDoesNotIndicateCancer" is thrown by step "review pathology report (if haven't done so yet)"),

!(Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception "Exception" is thrown by step "obtain patient informed consent and install portacath "),

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 48 (15 events):

{

Artifact "Scans" is wrong when step "chemotherapy process" is posted,

!(Exception "PathologyReportDoesNotIndicateCancer" is thrown by step "confirm all necessary information is present"),

!(Exception "PathologyReportDoesNotIndicateCancer" is thrown by step "review pathology report (if haven't done so yet)"),

!(Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception "Exception" is thrown by step "obtain patient informed consent and install portacath "),

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 49 (15 events):

{

Artifact "PathologyReport" is wrong when step "chemotherapy process" is posted,

!(Exception "PathologyReportDoesNotIndicateCancer" is thrown by step "confirm all necessary information is present"),

!(Exception "PathologyReportDoesNotIndicateCancer" is thrown by step "review pathology report (if haven't done so yet)"),

!(Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception “Exception” is thrown by step “obtain patient informed consent and install portacath ”),

!(Exception “MissingOrdersForPatient” is thrown by step “confirm existence of orders for patient”),

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 50 (15 events):

{

Artifact “Patient” is wrong when step “chemotherapy process” is posted,

!(Exception “PathologyReportDoesNotIndicateCancer” is thrown by step “confirm all necessary information is present”),

!(Exception “PathologyReportDoesNotIndicateCancer” is thrown by step “review pathology report (if haven’t done so yet)”),

!(Exception “TreatmentPlanInconsistentWithOrders” is thrown by step “perform Practice RN verifications”),

!(Exception “TreatmentPlanNeedsToBeReentered” is thrown by step “perform Practice RN verifications”),

!(Exception “HeightWeightDontMatch” is thrown by step “perform Practice RN verifications”),

!(Exception “DosagesDontMatch” is thrown by step “perform Practice

RN verifications”),

!(Exception “StickyNotesTasksNotCompleted” is thrown by step “perform Practice RN verifications”),

!(Exception “Exception” is thrown by step “obtain patient informed consent and install portacath ”),

!(Exception “MissingOrdersForPatient” is thrown by step “confirm existence of orders for patient”),

!(Exception “BSAMismatch” is thrown by step “evaluate blood counts”),

!(Exception “PatientDeferred” is thrown by step “evaluate blood counts”),

!(Exception “PatientNotInRoster” is thrown by step “check patient in roster”),

!(Exception “DosagesDontMatch” is thrown by step “confirm manually-calculated drug volume matches the added volume by tech”),

!(Exception “LabelsAndOrdersDontMatch” is thrown by step “confirm info on drug labels matches info on chemo orders”)

}

MCS 51 (16 events):

{

Step “measure height/weight in standard fashion” produces wrong “PatientWeight”,

!(Exception “PathologyReportDoesNotIndicateCancer” is thrown by step “confirm all necessary information is present”),

!(Exception “PathologyReportDoesNotIndicateCancer” is thrown by step “review pathology report (if haven’t done so yet)”),

!(Exception “PatientUnableToStand” is thrown by step “measure height/weight in standard fashion”),

!(Exception “TreatmentPlanInconsistentWithOrders” is thrown by step

"perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception "Exception" is thrown by step "obtain patient informed consent and install portacath "),

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

}

MCS 52 (16 events):

{

Step "measure height/weight in standard fashion" produces wrong "PatientHeight",

!(Exception "PathologyReportDoesNotIndicateCancer" is thrown by step

"confirm all necessary information is present"),

!(Exception "PathologyReportDoesNotIndicateCancer" is thrown by step "review pathology report (if haven't done so yet)"),

!(Exception "PatientUnableToStand" is thrown by step "measure height/weight in standard fashion"),

!(Exception "TreatmentPlanInconsistentWithOrders" is thrown by step "perform Practice RN verifications"),

!(Exception "TreatmentPlanNeedsToBeReentered" is thrown by step "perform Practice RN verifications"),

!(Exception "HeightWeightDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "DosagesDontMatch" is thrown by step "perform Practice RN verifications"),

!(Exception "StickyNotesTasksNotCompleted" is thrown by step "perform Practice RN verifications"),

!(Exception "Exception" is thrown by step "obtain patient informed consent and install portacath "),

!(Exception "MissingOrdersForPatient" is thrown by step "confirm existence of orders for patient"),

!(Exception "BSAMismatch" is thrown by step "evaluate blood counts"),

!(Exception "PatientDeferred" is thrown by step "evaluate blood counts"),

!(Exception "PatientNotInRoster" is thrown by step "check patient in roster"),

!(Exception "DosagesDontMatch" is thrown by step "confirm manually-calculated drug volume matches the added volume by tech"),

!(Exception "LabelsAndOrdersDontMatch" is thrown by step "confirm info on drug labels matches info on chemo orders")

```
}
```

BIBLIOGRAPHY

[1] Bandera website.
URL `http://bandera.projects.cis.ksu.edu/`

[2] Cadence SMV website.
URL `http://www.kenmcmil.com/smv.html`

[3] Espresso website.
URL `http://embedded.eecs.berkeley.edu/pubs/downloads/espresso/index.htm`

[4] Galileo project website.
URL `http://www.cs.virginia.edu/∼ftree/`

[5] National guideline clearinghouse, U.S.A.
URL `http://www.guideline.gov/`

[6] National institute for health and clinical excellence, U.K.
URL `http://www.nice.org.uk/`

[7] OpenClinical website.
URL `http://www.openclinical.org/gmmintro.html`

[8] OpenFTA website.
URL `http://www.openfta.com/`

[9] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 1 edition, 2001.

[10] John D. Andrews. The use of NOT logic in fault tree analysis. *Quality and Reliability Engineering International*, 17(3):143–150, 2001.

[11] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services, version 1.1, May 2003.

[12] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986. ISSN 0020-0190.

[13] Electronics Industry Association. Systems engineering capability model. Technical Report EIA/IS 731, Washington DC, 1998.

[14] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*, pages 363–366. 2000. ISBN 3-540-67261-3.

[15] Sergio C. Bandinelli, Alfonso Fugetta, and Carlo Ghezzi. Software process model evolution in the SPADE environment. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, December 1993.

[16] Sergio C. Bandinelli, Alfonso Fuggetta, and Sandro Grigolli. Process Modeling in–the–large with SLANG. In Leon Osterweil, editor, *Proceedings of the Second International Conference on the Software Process*, pages 75–83. IEEE Computer Society Press, 1993.

[17] Simon Bäumler, Michael Balser, Andriy Dunets, Wolfgang Reif, and Jonathan Schmitt. Verification of medical guidelines by model checking - a case study. In A. Valmari, editor, *SPIN*, volume 3925 of *LNCS*, pages 219–233. 2006.

[18] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie(Translator). *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001.

[19] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[20] Grady Booch. *Object-oriented Analysis and Design with Applications*. Addison-Wesley Professional, 2 edition, September 1993.

[21] Emery R. Boose, Aaron M. Ellison, Leon J. Osterweil, Lori Clarke, Rodion Podorozhny, Julian L. Hadley, Alexander Wise, and David R. Foster. Ensuring reliable datasets for environmental models and forecasts. In *Ecological Informatics*. 2007. To appear.

[22] Aziz A. Boxwala, Mor Peleg, Samson Tu, Omolola Ogunyemi, Qing T. Zeng, Dongwen Wang, Vimla L. Patel, Robert A. Greenes, and Edward H. Shortliffe. Glif3: A representation format for sharable computer-interpretable clinical practice guidelines. *Journal of Biomedical Informatics*, 37(3):147–161, 2004. ISSN 1532-0464. doi:http://dx.doi.org/10.1016/j.jbi.2004.04.002.

[23] Preston Briggs. *Register Allocation via Graph Coloring*. Ph.D. thesis, Rice University, Houston, Texas, 1992.

[24] M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81(1):13–31, 1989.

[25] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[26] Aaron G. Cass, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, Jr. Stanley M. Sutton, and Alexander Wise. Little-jil/juliette: A process definition language and interpreter. In *22nd International Conference on Software Engineering*, pages 754–757. Limerick, Ireland, 2000.

[27] S. S. Cha, N. G. Leveson, and T. J. Shimeall. Safety verification in murphy using fault tree analysis. In *ICSE '88: Proceedings of the 10th International Conference on Software Engineering*, pages 377–386. 1988.

[28] Stefan Christov, Bin Chen, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. Rigorously defining and analyzing medical processes: An experience report. In *1st International Workshop on Model-Based Trustworthy Health Information Systems*. September 2007.

[29] Chrysler, Ford, and General Motors. *Potential Failure Mode and Effects Analysis in Design (Design fMEA) and Potential Failure Mode and Effects Analysis in Manufacturing and assembly Processes (Process FMEA) Reference Manual (SAE J-1739).*

[30] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *14th International Conference on Computer-Aided Verification*. July 2002.

[31] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(5):726–750, 1997.

[32] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, London, UK, 1982.

[33] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, Cambridge, MA, USA, 2001.

[34] Lori A. Clarke, Leon J. Osterweil, and George S. Avrunin. Supporting human-intensive systems. In *FSE/SDP Workshop on the Future of Software Engineering Research*. 2010.

[35] Rachel L. Cobleigh, George S. Avrunin, and Lori A. Clarke. User guidance for creating precise and accessible property specifications. In *14th ACM SIGSOFT Int. Symp. on Foundations of Software Eng.*, pages 208–218. Portland, OR, November 2006.

[36] James Corbett. Bandera intermediate representation(bir) specification version 0.6. 1999.

[37] James C. Corbett and George S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6(1):97–123, 1995.

[38] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448. ACM, New York, NY, USA, 2000.

[39] Olivier Coudert and Jean Christophe Madre. Fault tree analysis: 1020 prime implicants and beyond, 1993.

[40] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. pages 238–252, 1977.

[41] W. Edwards Deming. *Out Of The Crisis*. M.I.T. PRESS, August 2000.

[42] Joseph DeRosier, Erik Stalhandske, James P. Bagian, and Tina Nudell. Using healthcare failure modes and effects analysis: The va national center for patient safety's prospective risk analysis system. *The Joint Commission Journal on Quality Improvement*, 27(5):248–267, 2002.

[43] Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Transactions on Software Engineering and Methodology*, 13(4):359–430, October 2004. ISSN 1049-331X.

[44] Khaled El Emam, Jean-Normand Drouin, Walcelio Melo, and Alec Dorling (Foreword by), editors. *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*. Wiley-IEEE Computer Society Press, November.

[45] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *CADE-17: Proceedings of the 17th International Conference on Automated Deduction*, pages 236–254. Springer-Verlag, London, UK, 2000.

[46] E. Allen Emerson, Richard J. Trefler, and Thomas Wahl. Reducing model checking of the few to the one. In *8th International Conference on Formal Engineering Methods*, pages 94–113. 2006.

[47] W. Emmerich and V. Gruhn. FUNSOFT Nets: a Petri-Net based Software Process Modeling Language. In C. Ghezzi and GC. Roman, editors, *Proc. 6th ACM/IEEE Int. Workshop on Software Specification and Design (IWSSD)*, pages 175–184. IEEE Computer Society Press, Como, Italy, 1991.

[48] Clifton A. Ericson. Fault tree analysis – a history. In *Proceedings of the 17th International System Safety Conference*, pages 87–96. 1999.

[49] Hendrik Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. Ph.D. thesis, University of Twente, Enschede, The Netherlands, 2002.

[50] Javier Esparza and Mogens Nielsen. Decidability issues for petri nets – a survey. *Bulletin of the European Association for Theoretical Computer Science*, 52:245–262, 1994.

[51] Jesús Arias Fisteus, Luis Sánchez Fernández, and Carlos Delgado Kloos. Applying model checking to bpel4ws business collaborations. In *SAC '05: Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 826–830. 2005.

[52] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Ltsa-ws: A tool for model-based verification of web service compositions and choreography. In *ICSE '06: Proceeding of the 28th International Conference on Software Engineering*, pages 771–774. 2006.

[53] Michael A. Friedman. Automated software fault-tree analysis of pascal programs. In *Reliability and Maintainability Symposium*, pages 458–461. Atlanta, GA, January 1993.

[54] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting BPEL web services. In *3th International World Wide Web Conference (WWW)*. May 2004.

[55] Xiang Fu, Tevfik Bultan, and Jianwen Su. Model checking XML manipulating software. In *International Symposium on Software Testing and Analysis (ISSTA)*. 2004.

[56] Xiang Fu, Tevfik Bultan, and Jianwen Su. WSAT: A tool for formal analysis of web services. In *16th International Conference on Computer Aided Verification*. July 2004.

[57] Henry L. Gantt. *Work, Wages, and Profits*. Engineering Magazine Co., New York, 1916. Reprinted by Hive Publishing Company, Easton, Maryland, 1973.

[58] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A unified high-level petri net formalism for time-critical systems. *IEEE Transactions on Software Engineering*, 17(2):160–172, February 1991.

[59] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, New York, NY, USA, 1996.

[60] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. 1254:72–83, 1997.

[61] Object Management Group. Business process modeling notation (BPMN) specification, version 1.0, February 2006.

[62] Object Management Group. OMG systems modeling language (OMG SysML) specification, May 2006.

[63] Object Management Group. OMG unified modeling language (OMG UML), version 2.1.2, November 2007.

[64] Elizabeth A. Henneman, George S. Avrunin, Lori A. Clarke, Leon J. Osterweil, Chester Andrzejewski, Jr., Karen Merrigan, Rachel Cobleigh, Kimberly Frederick, Ethan Katz-Bassett, and Philip L. Henneman. Increasing patient safety and efficiency in transfusion therapy using formal process definitions. In *Transfusion Medicine Review*, volume 21, pages 49–57. January 2007.

[65] Elizabeth A. Henneman, George S. Avrunin, Lori A. Clarke, Leon J. Osterweil, Chester Andrzejewski Jr., Karen Merrigan, Rachel Cobleigh, Kimberly Frederick, Ethan Katz-Bassett, and Philip L. Henneman. Increasing patient safety and efficiency in transfusion therapy using formal process definitions. *Transfusion Medicine Review*, 21(1):49–57, January 2007.

[66] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri Nets. In Wil M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the Third International Conference on Business Process Management (BPM 2005)*, volume 3649 of *LNCS*, pages 220–235. Springer-Verlag, Nancy, France, 2005.

[67] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.

[68] Pavel Hruby. Structuring specification of business systems with UML. In *OOPSLA'98 Workshop on Behavioral Semantics of OO Business and System Specifications*. Vancouver, Canada, 1998.

[69] Software Engineering Institute. Integrated product development CMM (IPD-CMM). Technical report, Carnegie Mellon University. (Note: This model was never officially released and is no longer publicly available).

[70] Software Engineering Institute. Software CMM, version 2.0 (draft c). Technical report, Carnegie Mellon University, November 1997. (Note: This model was never officially released and is no longer publicly available).

[71] Radu Iosif, Matthew B. Dwyer, and John Hatcliff. Translating java for multiple model checkers: The bandera back-end. *Formal Methods in System Design*, 26(2):137–180, March 2005.

[72] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[73] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object Oriented Software Engineering: A Use Case Driven Approach.* (ACM Press) Addison-Wesley Professional, July 1992.

[74] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1, 2, 3. Springer, 2 edition, 1996.

[75] Peter D. Johnson, Samson Tu, Nick Booth, Bob Sugden, and Ian N. Purves. Using scenarios in chronic disease management guidelines for primary care. In *Proceedings of the AMIA Symposium*, pages 389–393. 2000.

[76] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Softw.*, 5(3):40–49, 1988. ISSN 0740-7459.

[77] Matthias Kloppmann, Dieter Koenig, Frank Leymann, Gerhard Pfau, Alan Rickayzen, Claus von Riegen, Patrick Schmidt, and Ivana Trickovic. WS-BPEL extension for people - BPEL4People, July 2005.

[78] T. Knape, L. Hedermana, V. P. Wade, M. Gargana, C. Harris, and Y. Rahman. A uml approach to process modelling of clinical practice guidelines for enactment. In *Medical Informatics Europe*. 2003.

[79] Linda T. Kohn, Janet M. Corrigan, and Molla S. Donaldson, editors. *To Err Is Human: Building a Safer Health System.* National Academy Press, Washington, D.C., 1999.

[80] Nancy G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley Professional, April 1995.

[81] Frank Leymann, editor. *Web Services Flow Language (WSFL 1.0).* IBM Software Group, May 2001.

[82] P. Liggesmeyer and M. Rothfelder. Improving system reliability with automatic fault tree generation. In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, page 90. IEEE Computer Society, Washington, DC, USA, 1998.

[83] Robyn R. Lutz and Robert M. Woodhouse. Requirements analysis using forward and backward search. *Annals of Software Engineering*, 3:459–475, 1997.

[84] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs.* Wiley, 2 edition, July 2006.

[85] Jr. Mark L. McKelvin, Gabriel Eirea, Claudio Pinello, Sri Kanajan, and Alberto L. Sangiovanni-Vincentelli. A formal approach to fault tree synthesis for the analysis of distributed fault tolerant systems. In *EMSOFT '05: Proceedings of the 5th ACM International Conference on Embedded Software*, pages 237–246. 2005.

[86] M.Bozzano and A.Villafiorita. Improving system reliability via model checking: the fsap / nusmv-sa safety analysis platform. In *The International Conference on Computer Safety, Reliability and Security*, number 2788 in LNCS, pages 49–62. Edimburgh, Scotland, United Kingdom, September 2003.

[87] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[88] Sang-Yoon Min, H.-D. Lee, and Doo-Hwan Bae. Softpm: A software process management system reconciling formalism with easiness. *Information & Software Technology*, 42(1):1–16, 2000.

[89] G. Molino, P. Terenziani, S. Montani, A.Bottrighi, and M. Torchio. GLARE: A domain-independent system for acquiring, representing and executing clinical guidelines. In *J. of the Amer. Medical Informatics Association (JAMIA) Symposium supplement*. 2006.

[90] Benjamin C. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985. ISSN 0018-9162.

[91] Mark A. Musen, Samson W. Tu, Amar K. Das, and Yuval Shahar. Eon: A component-based approach to automation of protocol-directed therapy. *Journal of the American Medical Informatics Association*, 3(6):367C–388, 1996.

[92] Rita Noumeir. Radiology interpretation process modeling. *J. of Biomedical Informatics*, 39(2):103–114, 2006. ISSN 1532-0464.

[93] Leon J. Osterweil, Norman K. Sondheimer, Lori A. Clarke, Ethan Katsh, and Daniel Rainey. Using process definitions to facilitate the specification of requirements. Technical report, Department of Computer Science, University of Massachusetts Amherst, 2006.

[94] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur ter Hofstede. WofBPEL: A tool for automated analysis of BPEL processes. In *LNCS 3826: Service-Oriented Computing - ICSOC 2005: Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005. / Boualem Benatallah, Fabio Casati, Paolo Traverso (Eds.)*, pages 484–489. Springer-Verlag, 2005.

[95] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Science of Computer Programming*, 67(2-3):162–198, 2007. ISSN 0167-6423.

[96] Ganesh J. Pai and Joanne Bechta Dugan. Automatic synthesis of dynamic fault trees from uml system models. In *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, page 243. Washington, DC, USA, 2002.

[97] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. Capability maturity model for software, version 1.1. Technical Report CMU/SEI-93-TR-024, Software Engineering Institute, Carnegie Mellon University, February 1993.

[98] Mark C. Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison-Wesley, 1995.

[99] M. Peleg, S. Tu, J. Bury, P. Ciccarese, J. Fox, R. Greenes, N. Johnson, A. Kumar, S. Miksch, S. Quaglini, A. Seyfang, and M. Shortliffe. Comparing computer-interpretable guideline models: A case-study approach. *Journal of the American Medical Informatics Association*, 10(1):52–68, Jan-Feb 2003.

[100] Mor Peleg, Aziz Boxwala, Samson Tu, Dongwen Wang, Omolola Ogunyemi, and Qing Zeng. *Guideline Interchange Format 3.5 Technical Specification.* InterMed Collaboratory, 2004.

[101] Haapanen Pentti and Helminen Atte. *Failure Mode and Effects Analysis of Software Based Automation Systems (STUK-YTO-TR 190).* VTT Industrial Systems, Helsinki, August 2002.

[102] Carl Adam Petri. *Kommunikation mit Automaten.* Bonn: Institut fűr Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[103] Carl Adam Petri. Kommunikation mit automaten. *New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377*, 1:1–Suppl.1, 1966. English translation.

[104] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[105] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[106] Thomas Pyzdek. *The Six Sigma Handbook: The Complete Guide for Greenbelts, Blackbelts, and Managers at All Levels.* McGraw-Hill, March 2003.

[107] S Quaglini, M Stefanelli, G Lanzola, V Caporusso, and S Panzarasa. Flexible guideline-based patient careflow systems. *Artificial Intelligence in Medicine*, 22(1):65–80, 2001.

[108] Vivek Ratan, Kurt Partridge, Jon Reese, and Nancy Leveson. Safety analysis tools for requirements specifications. Technical report, Safeware Engineering Corporation.

[109] Mohammad S. Raunak, Bin Chen, Amr Elssamadisy, Lori A. Clarke, and Leon J. Osterweil. Definition and analysis of election processes. In *SPW/ProSim 2006*, volume 3966 of *LNCS*, pages 178–185. Shanghai, May 2006.

[110] Proctor P. Reid, W. Dale Compton, Jerome H. Grossman, and Gary Fanjiang, editors. *Building a Better Delivery System: A New Engineering/Health Care Partnership.* National Academy Press, Washington, D.C., 2005.

[111] Donald J. Reifer. Software failure mode and effects analysis. *IEEE Transactions on Reliability*, 28(3), August 1979.

[112] Rasa Remenyte-Prescott and John D. Andrews. Analysis of non-coherent fault trees using ternary decision diagrams. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, 222(2):127–138, 2008.

[113] James R. Rumbaugh, Michael R. Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-Oriented Modeling and Design.* Prentice Hall, October 1990.

[114] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2 edition, 2002.

[115] Karsten Schmidt. Lola - a low level analyser. In *International Conference on Application and Theory of Petri Nets*, number 1825 in LNCS. Springer-Verlag, 2000.

[116] Yuval Shahar, Silvia Miksch, and Peter Johnson. The asgaard project: A task-specific framework for the application and critiquing of time-oriented clinical guidelines. *Artificial Intelligence in Medicine*, 14(1-2):29–51, 1998.
URL `citeseer.ist.psu.edu/shahar98asgaard.html`

[117] Walter A. Shewhart. *Economic Control of Quality of Manufactured Product.* D. Van Nostrand Co., New York, 1931.

[118] Nakajima Shin. Lightweight formal analysis of web service flows. *Progress in Informatics*, 2:57–76, November 2005.

[119] Christian Stahl. A Petri Net Semantics for BPEL. Informatik-Berichte 188, Humboldt-Universität zu Berlin, 2005.

[120] D. H. Stamatis. *Failure Mode and Effect Analysis: FMEA from Theory to Execution.* American Society for Quality, March 1995.

[121] J. B. Starren, G. Hripcsak, D. Jordan, B. Allen, C. Weissman, and P. D. Clayton. Encoding a post-operative coronary artery bypass surgery care plan in the arden syntax. *Computers in Biology and Medicine*, 24(5):411C–417, 1994.

[122] Harald Stőrrle and Jan Hendrik Hausmann. Towards a formal semantics of uml 2.0 activities. In *German Software Engineering Conference*, volume P-64 of LNI, pages 117–128. 2005.

[123] David R. Sutton and John Fox. The syntax and semantics of the PRO*forma* guideline modeling language. *Journal of the American Medical Informatics Association*, 10(5):433C–443, 2003.

[124] CMMI Product Team. CMMI for development, version 1.2. Technical Report CMU/SEI-2006-TR-008, Software Engineering Institute, Carnegie Mellon University, August 2006.

[125] CMMI Product Team. CMMI for acquisition, version 1.2. Technical Report CMU/SEI-2007-TR-017, Software Engineering Institute, Carnegie Mellon University, November 2007.

[126] Annette ten Teije, Mar Marcos, Michel Balser, Joyce van Croonenborg, Christoph Duelli, Frank van Harmelen, Peter Lucas, Silvia Miksch, Wolfgang Reif, Kitty Rosenbrand, and Andreas Seyfang. Improving medical protocols by formal methods. *Artificial Intell. in Medicine*, 36(3):193–209, 2006.

[127] P. Terenziani, L. Giordano, A. Bottrighi, S. Montani, and L. Donzella. SPIN model checking for the verification of clinical guidelines. In *Workshop on AI Techniques in Healthcare: Evidence-based Guidelines and Protocols*. August 2006.

[128] Paolo Terenziani, Gianpaolo Molino, and Mauro Torchio. A modular approach for representing and executing clinical guidelines. *Artificial Intelligence in Medicine*, 23(3):249–276, 2001.

[129] Satish Thatte. *XLANG: Web Services for Business Process Design*. Microsoft Corporation, June 2001.

[130] U.S. Military. *Procedures for Performing a Failure Mode, Effects and Criticality Analysis (MIL-P-1629)*, 1949.

[131] W.E. Vesely, F.F. Goldberg, N.H. Roberts, and D.F. Haasl. *Fault Tree Handbook (NUREG-0492)*. U.S. Nuclear Regulatory Commission, Washington, D.C., January 1981.

[132] Danhua Wang, Jingui Pan, George Avrunin, Lori A. Clarke, Leon J. Osterweil, and Bin Chen. An automatic failure mode and effect analysis technique for processes defined in the little-jil process definition language. In *The 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE 2010)*. San Francisco Bay, USA, 2010.

[133] Michael Weber and Ekkart Kindler. The petri net markup language. *Petri Net Technology for Communication-Based Systems*, 2472:124–144, February 2004.

[134] Judith M. Wilkinson and Karen Van Leuven. *Fundamentals of Nursing*. F. A. Davis Company, June 2007.

[135] Judith M. Wilkinson and Karen Van Leuven. Procedure checklist for administering a blood transfusion. `http://davisplus.fadavis.com/wilkinson/PDFs/Procedure_Checklists/PC_Ch36-01.pdf`, 2007.

[136] Eran Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. *SIGPLAN Notices*, 36(3):27–40, 2001.

[137] YanPing Yang, QingPing Tan, and Yong Xiao. Verifying web services composition based on hierarchical colored petri nets. In *IHIS '05: The First International Workshop on Interoperability of Heterogeneous Information Systems*, pages 47–54. 2005.