

**MODEL-BASED GUIDANCE
FOR HUMAN-INTENSIVE PROCESSES**

A Dissertation Presented

by

STEFAN C. CHRISTOV

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 2015

Computer Science

© Copyright by Stefan C. Christov 2015

All Rights Reserved

MODEL-BASED GUIDANCE FOR HUMAN-INTENSIVE PROCESSES

A Dissertation Presented

by

STEFAN C. CHRISTOV

Approved as to style and content by:

George S. Avrunin, Co-chair

Lori A. Clarke, Co-chair

Leon J. Osterweil, Member

Jenna L. Marquard, Member

Elizabeth A. Henneman, Member

Lori A. Clarke, Chair
Computer Science

To my parents, my brother, and my grandparents.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to Lori Clarke and George Avrunin, my Ph.D. advisers, for holding me to the highest standards, for their support and guidance during the years, and for believing that I could reach this point. This dissertation is as much their achievement as it is mine.

I am thankful to Lee Osterweil for the interesting research discussions and his thoughtful feedback on this work.

I would like to thank Jenna Marquard and Elizabeth Henneman for their assistance with the non-computer science aspects of this work and for their help with conducting human subject studies.

I am grateful to the students of the LASER lab at UMass Amherst for creating a friendly and an enjoyable work environment, and especially to Heather Conboy for sharing her expertise on the lab's code base and for helping me build on that code base.

I would like to thank Barbara Lerner and Philip Henneman for the opportunities to hone my research skills via several research collaborations.

I am deeply grateful to my family for raising me in an intellectually stimulating environment, for encouraging me to pursue my studies, for their patience while I was working on this dissertation far away from them, and for their endless love.

I am thankful to Francesca Colantuoni for making my life happier once she entered it and for her tremendous support during the final stage of my dissertation. I am thankful to Sophal Khun for showing me on multiple occasions what it means to truly care for a friend. I would like to thank Borislava Simidchieva for her help during the

years and for overcoming many obstacles together. I am grateful to all my friends who were with me in moments of joy and in moments of sadness, with whom I shared my journey in graduate school, and who contributed to making my time in Amherst an unforgettable and an exciting experience.

ABSTRACT

MODEL-BASED GUIDANCE FOR HUMAN-INTENSIVE PROCESSES

FEBRUARY 2015

STEFAN C. CHRISTOV

B.Sc., STATE UNIVERSITY OF NEW YORK, COLLEGE AT BROCKPORT

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

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

Human-intensive processes (HIPs), such as medical processes involving coordination among doctors, nurses, and other medical staff, often play a critical role in society. We use the word “process” to refer to the coordination of activities to achieve a task or a goal, where the activities may be performed by humans, devices, or software systems. We say that a process is “human-intensive” if the contributions of human process performers have a significant impact on the process outcomes and require substantial domain expertise and insight. Despite considerable work and progress in error reduction, human errors are still a major concern for many HIPs. For example, human errors in medical HIPs constitute a large part of the preventable medical errors estimated to cause the death of between 98,000 and 400,000 people each year in the U.S.

To address this problem of human errors in HIPs, this thesis investigates two approaches for online process guidance, i.e., for guiding process performers while a

process is being executed. Both approaches rely on monitoring a process execution and base the guidance they provide on a detailed formal process model that captures the recommended ways to perform the corresponding HIP. The first approach, which we call deviation detection and explanation, automatically detects when an executing HIP deviates from a set of recommended executions of that HIP, as specified by the process model. Such deviations could represent errors and, thus, detecting and reporting deviations as they occur could help catch errors before something bad happens. The approach also provides information to help explain a detected deviation to assist process performers with identifying potential errors and with planning recovery from these errors. The second approach, which we call process state visualization, proactively guides process performers by showing them information relevant to the current process execution, such as the activities that need to be performed at each point of that process execution. The goal of the process state visualization approach is to reduce the number of human errors.

The success of the online process guidance approaches presented in this work depends on the correctness and the degree of completeness of the underlying process model. If the process model is incorrect with respect to some set of requirements or it is incomplete by not representing all the process executions that domain experts have agreed should be captured in the model, then the online guidance may be incorrect or there might not be enough information in the process model to provide guidance in certain scenarios. Given the complexity of some HIPs, creating a correct and sufficiently complete process model is challenging. This thesis investigates process elicitation techniques to help create such process models and discusses an evaluation of these techniques based on their application to real-world HIPs.

The major contributions of this work can be summarized as follows:

- Compared the relative strengths and weaknesses of several techniques for process elicitation and process model validation to help create correct and suffi-

ciently complete process models needed for the proposed online process guidance approaches.

- Developed an approach for deviation detection and explanation and evaluated it with realistic process models and synthetic process executions with seeded errors.
 - Recognized delayed deviation detection as a potential obstacle for the approach and investigated its frequency and consequences.
- Developed an initial approach for visualization of process execution state and demonstrated it on a medical case study.

TABLE OF CONTENTS

	Page
ACKNOWLEDGMENTS	v
ABSTRACT	vii
LIST OF TABLES	xv
LIST OF FIGURES	xvi
 CHAPTER	
1. INTRODUCTION	1
2. BACKGROUND	7
2.1 Process Improvement Environment	7
2.1.1 Offline Analyses	9
2.1.2 Online Analyses	12
2.1.3 Thesis Focus	13
2.2 Prerequisites for Online Process Guidance	14
2.2.1 Precise, Correct, and Sufficiently Complete Process Model	14
2.2.2 Process Execution Monitor	15
2.3 The Little-JIL Process Modeling Language	16
2.3.1 The Little-JIL Diagrammatic Representation	17
2.3.1.1 Example Little-JIL Process Model	19
2.3.2 The Little-JIL Narrative Representation	21
2.3.2.1 Design of Little-JIL Narrator	24

3. PROCESS ELICITATION AND MODELING	28
3.1 Process Elicitation	29
3.1.1 Case Study	30
3.1.1.1 Elicited Process	30
3.1.1.2 Elicitation Methods	31
3.1.1.3 Results	35
3.1.1.4 Discussion	41
3.1.1.5 Threats to Validity	43
3.1.1.6 Summary	44
3.2 Process Modeling	45
3.2.1 Exception Handling Patterns	45
3.2.1.1 Selected Patterns	48
3.2.1.2 Evaluation of Exception Handling Patterns	55
3.2.2 Comparing the Little-JIL Diagrammatic and Narrative Representations	60
3.2.2.1 User Study	61
3.2.3 Overall Experience with Process Modeling	67
3.2.3.1 Experience with Using Little-JIL	67
3.2.3.2 Process Model Validation	70
4. DEVIATION DETECTION AND EXPLANATION	73
4.1 Overview	73
4.1.1 Example of Applying the Deviation Detection and Explanation Approach to a Medical Process	78
4.1.2 Issues.....	81
4.1.2.1 Delayed Deviation Detection and Potential Harm Due to Delay	81
4.1.2.2 Potential Harm When Deviations Are Immediately Detected	84
4.1.2.3 Performance of the Deviation Detector	85
4.2 Deviation Detection Framework	86
4.3 Experimental Evaluation	88

4.3.1	Experimental Design	88
4.3.1.1	Process Models	90
4.3.1.2	Synthetic Sequences with Errors	92
4.3.1.3	Experiments	96
4.3.1.4	Potentially Harmful Steps	96
4.3.2	Results	97
4.3.3	Discussion	97
4.3.3.1	Delayed Deviation Detection and Potential Harm Due to Delay	99
4.3.3.2	Potential Harm When Deviations Are Immediately Detected	101
4.3.3.3	Performance of the Deviation Detector	103
4.3.4	Threats to Validity	103
4.4	Limitations of the Deviation Detection Approach	104
4.5	Deviation Explanation	107
4.5.1	Error Localization	107
4.5.1.1	Legal Sequence Selection	108
4.5.1.2	Alignment Computation	111
4.5.1.3	Potential Error Index Identification	112
4.5.2	Evaluation of Error Localization Approach	114
4.5.2.1	Experimental Design	116
4.5.2.2	Results	117
4.5.2.3	Discussion	117
4.5.2.4	Threats to Validity	118
4.5.3	Limitations of the Error Localization Approach	119
5.	VISUALIZATION OF PROCESS EXECUTION STATE	120
5.1	Overview	120
5.2	The Smart Checklist Metaphor	124
5.3	The Smart Checklist Prototype	126
5.3.1	Back-end Implementation	126
5.3.2	Visualization	126
5.4	Preliminary Evaluation	133

6. RELATED WORK	142
6.1 Process Elicitation and Modeling	142
6.1.1 Process Elicitation	142
6.1.2 Process Modeling	143
6.1.2.1 Exception Handling in HIPs	143
6.1.2.2 Process Model Representations	145
6.2 Online Process Guidance for HIPs	147
6.2.1 Visualization of the Execution State of HIPs	147
6.2.1.1 Trauma Center Process Guidance System	147
6.2.1.2 Visualization of Patient Flow	149
6.2.1.3 Visualization of Patient Location During Perioperative Clinical Processes	150
6.2.2 Traditional Checklists and Process Guides	151
6.2.2.1 Checklists	151
6.2.2.2 Process Guides	154
6.2.3 Approaches for Dealing with Process Deviation	155
6.2.3.1 Software Process Validation	155
6.2.3.2 Dealing with Deviations in Software Design Processes	156
6.2.3.3 A Framework for Formalizing Inconsistencies and Deviations in HIPs	158
6.2.3.4 Conformance between Process Executions and Process Models	160
6.2.3.5 Adherence to Medical Guidelines	161
6.2.4 Error Localization	161
6.2.4.1 Fault Localization	161
6.2.4.2 Anomaly Detection	163
6.2.4.3 Plan and Policy Recognition Approaches	164
6.3 Human Errors	168
6.3.1 “Human Error” by James Reason	169
6.3.2 The Eindhoven Classification Model	170
6.3.3 Error Causes vs. Error Manifestations	171

7. CONCLUSION AND FUTURE WORK	172
 APPENDICES	
A. ARTIFACTS USED IN PROCESS ELICITATION STUDY	179
A.1 Open-ended Prompts	179
A.2 Complete Process Traces	180
A.2.1 Trace 1	180
A.2.2 Trace 2	181
A.2.3 Trace 3	182
A.3 Complete Process Model (Textual Description)	184
B. ARTIFACTS USED IN STUDY OF PROCESS REPRESENTATIONS	190
C. LOW-LEVEL PROCESS MODEL REPRESENTATION	210
 BIBLIOGRAPHY	 225

LIST OF TABLES

Table		Page
2.1	Example phrasing templates.	25
3.1	Normal flow process steps.	38
3.2	Exceptional situations.	39
3.3	Responses to exceptional situations.	39

LIST OF FIGURES

Figure	Page
2.1 Process Improvement Environment.	8
2.2 Little-JIL Step (figure adapted from [29])	19
2.3 Little-JIL model of a simplified blood transfusion process.	20
2.4 The narrative corresponding to the Little-JIL process model in Figure 2.3.	23
2.5 Architecture of the Little-JIL Narrator.	24
3.1 Sample sequence of observed steps	33
3.2 Sample open-ended scenarios.	34
3.3 Sample process trace.	36
3.4 Number of new events identified using each elicitation method.	36
3.5 Normal flow process steps identified and refuted via each method.	37
3.6 Exceptional situations identified and refuted via each method.	37
3.7 Responses to exceptional situations identified and refuted via each method.	38
3.8 Structure of the Ordered Alternatives Pattern in Little-JIL.	49
3.9 Using the Ordered and Unordered Alternative Patterns when Planning a Trip.	50
3.10 The Structure of the Deferred Fixing Pattern in Little-JIL.	52
3.11 Using the Deferred Fixing Pattern to Complete Seat Selection at a Later Time.	53

3.12	The Structure of the Compensate Pattern in Little-JIL.....	54
3.13	Using the Compensate Pattern to Cancel a Trip.....	54
3.14	The number of occurrences of each pattern in the chemotherapy and ODR process models.	56
3.15	Study results—computer science students.	63
3.16	Study results—nursing students.	63
4.1	Deviation detection and explanation approach.	74
4.2	Simplified blood transfusion process.	79
4.3	An Example Extended Control Flow Graph.	81
4.4	Deviation detection experimental framework.	87
4.5	Applying the deviation detection approach to blood transfusion and chemotherapy process models.	98
4.6	Example application of the error localization approach given the sequence of performed steps <i>psu</i> and the process model in Figure 4.2.....	110
4.7	Example application of the error localization approach given the sequence of performed steps <i>acde</i> and the process model in Figure 4.3.....	111
4.8	Deviation detection and error localization experimental framework.	114
4.9	Applying the error localization approach to a blood transfusion process model.	117
5.1	The smart checklist at the beginning of executing the simplified blood transfusion process.	128
5.2	The smart checklist after the nurse has performed several steps from the simplified blood transfusion process.	129
5.3	The smart checklist after the nurse has successfully completed the simplified blood transfusion process.....	130

5.4	The smart checklist at a point of the executions of the simplified blood transfusion process where the nurse is about to indicate that problem has arisen.	134
5.5	The smart checklist after the nurse has performed several steps from the simplified blood transfusion process and is about to indicate that another problem has arisen.	135
5.6	The smart checklist after the simplified blood transfusion process was not successfully completed due to problems during the execution of the process.	136
6.1	Visualization of the process guidance system used during trauma resuscitation at the Alfred Trauma Center, Melbourne, Victoria, Australia (2006 – 2008).	148
6.2	Visualization of cardiac patient flow.	150
C.1	The stages of translating a Little-JIL process model into a low-level process model.	211
C.2	Trace flow graph derived from the Little-JIL model in Figure C.3.	212
C.3	Little-JIL model of a simplified final stage of a chemotherapy process.	213
C.4	The task automaton constraint for Task 1 of the TFG in Figure C.2.	215

CHAPTER 1

INTRODUCTION

Human-intensive processes (HIPs), such as medical processes involving coordination among doctors, nurses, and other medical staff, often play a critical role in society. We use the word *process* to refer to the coordination of activities to achieve a task or a goal, where the activities may be performed by humans, devices, or software systems. We say that a process is *human-intensive* if the contributions of human process performers have a significant impact on the process outcomes and require substantial domain expertise and insight. Despite considerable work and progress in error reduction, human errors are still a major concern for many HIPs. For example, human errors in medical HIPs constitute a large part of the preventable medical errors estimated to cause the death of 98,000 people each year in the U.S. [82]. More than a decade after this estimate, a 2009 US National Research Council report [122] indicates that the problem with errors still persists and that “it is widely recognized that today’s health care . . . suffers substantially as a result of medical errors”. A recent study [78] from 2013 reports an even higher estimate of deaths per year in the U.S., between 210,000 and 400,000, due to medical errors.

To address this problem of human errors in HIPs, this thesis investigates two approaches for *online process guidance*, i.e., for guiding process performers while a process is being executed. Both approaches rely on monitoring a process execution and base the guidance they provide on a detailed formal process model that captures the recommended ways to perform the corresponding HIP. The first approach, which we call *deviation detection and explanation*, automatically detects when an execut-

ing HIP deviates from a set of recommended executions of that HIP, as specified by the process model. Such deviations could represent errors and, thus, detecting and reporting deviations as they occur could help catch errors before something bad happens. The approach also provides information to help explain a detected deviation to assist process performers with identifying potential errors and with planning recovery from these errors. The second approach, which we call *process state visualization*, proactively guides process performers by showing them information relevant to the current process execution, such as the activities that need to be performed at each point of that process execution. The goal of the process state visualization approach is to reduce the number of human errors.

We use the widely-adopted definition of *error*: “failure of a planned action to be completed as intended (i.e., error of execution) or the use of a wrong plan to achieve an aim (i.e., error of planning)” [82, 111]. The online process guidance approaches presented in this work focus on planning errors, and in particular on planning errors that correspond to problems with the sequence of activities performed to achieve an aim, e.g., omitting an activity that should have been done (error of omission) or performing an activity that should not have been done (error of commission). Planning errors that are related to violation of real-time constraints, such as “an activity should be performed within five minutes after another activity has been performed”, are outside the scope of this work.

The two online process guidance approaches discussed in this thesis are part of an overall process improvement environment (PIE). This environment supports offline analyses for finding defects, inefficiencies, and vulnerabilities in HIPs, and it also supports online analyses to provide guidance to process performers while a process is being executed. A key component of the PIE is a *detailed and formal process model*. The offline analyses are applied to this model to reason about the corresponding real-world HIP, to improve the model, and to potentially suggest improvements for

the modeled HIP. A carefully analyzed and validated process model that captures the recommended ways to perform a HIP is then used as the basis for online process guidance.

The success of the online process guidance approaches presented in this work depends on the correctness and the completeness of the underlying process model. If the process model is incorrect with respect to some set of requirements or it is incomplete by not representing all the process executions that domain experts have agreed should be captured in the model, then the online guidance may be incorrect or there might not be enough information in the process model to provide guidance in certain scenarios. Given the complexity of some HIPs, creating a correct and sufficiently complete process model is challenging. This thesis investigates process elicitation techniques to help create such process models and discusses an evaluation of these techniques based on their application to real-world HIPs [31,36,93]. We believe the criticality of certain HIPs, such as medical processes, warrants the time and effort needed to create high-quality process models that can in turn be leveraged to support continuous process improvement [53,119] via various static analyses (e.g., model checking [37], fault-tree analysis [128], and failure-mode and effects analysis [121]) and to support various aspects of online process guidance, such as the approaches discussed in this work.

There have been considerable efforts to reduce the number and impact of errors in HIPs. One line of work has focused on improving the design of HIPs to make them less prone to human errors [14,50,93] and then train people to follow the redesigned processes. Improvements in the design of HIPs include changing the order of tasks, adding redundancy checks, and reducing the workload and/or working hours of process performers. Despite such efforts, however, human process performers still make errors while performing some HIPs, due to various reasons such as cognitive overload, distraction, and fatigue.

To further reduce the occurrence and impact of human errors, approaches have been investigated for supporting process performers while executing a process by encouraging conformance with some specification of the recommended ways to perform a process (e.g., process aids such as checklists [67, 130, 136] and care sets [92]). Such process aids, however, tend to specify only the major steps during nominal flow, omitting important details such as exceptional scenarios and concurrent process execution [31, 70]. Some of these process aids focus on training, but provide no support during the actual performance of the process. The ones that could be used while a process is being performed, such as checklists, in addition to often omitting important details, also add to the workload of already heavily-burdened process performers. The use of checklists, for example, often requires process performers to check what needs to be done, to remember what activities they completed, and to decide what the appropriate checklist is to use in a given context.

To remove some of the burdens that process aids, such as checklists, place on process performers, there have been attempts to create systems that automatically check the compliance of a process that is being performed with a specification of that process. For example, Fitzgerald et al. designed and deployed a process guidance system in a trauma center to guide medical professionals during the first 30 minutes of trauma resuscitation [57]. This system increased compliance with the underlying medical algorithms and reduced error rates, but these algorithms did not support complex process behaviors, such as concurrency and exception handling.

The online process guidance approaches investigated in this thesis are intended to go beyond these limitations. The approach for process state visualization provides dynamic, context-sensitive guidance and can automatically generate a checklist tailored to the current process execution. The deviation detection and explanation approach complements the process state visualization approach and could be particularly useful in situations where providing proactive guidance via the process state visualization

approach is not feasible or process performers have elected to not use such proactive guidance. By automatically checking conformance with the recommended ways to perform a process, the deviation detection and explanation approach reduces the burden on process performers to themselves ensure this conformance. Because both the deviation detection and explanation approach and the process state visualization approach are based on a detailed process model, these approaches can provide online guidance in a wider range of process execution scenarios than existing approaches do, including exceptional scenarios and concurrent execution, which is where human errors often occur [88].

The major contributions of this work can be summarized as follows:

- Compared the relative strengths and weaknesses of several techniques for process elicitation to help create correct and sufficiently complete process models needed for the proposed online process guidance approaches.
- Developed an approach for deviation detection and explanation and evaluated it with realistic process models and synthetic process executions with seeded errors.
 - Recognized delayed deviation detection as a potential obstacle for the approach and investigated its frequency and consequences.
- Developed an initial approach for visualization of process execution state and demonstrated it on a medical case study.

Chapter 2 introduces the overall process improvement environment that includes the online process guidance approaches investigated in this work and discusses prerequisites for these approaches. Chapter 2 also describes the Little-JIL process modeling language, which we use in the evaluation of the proposed online guidance approaches. Chapter 3 presents several process elicitation approaches to help create such process

models. It also discusses other approaches for facilitating the modeling of complex HIPs. Chapter 4 discusses the deviation detection and explanation approach and Chapter 5 discusses the process state visualization approach. Related work is covered in Chapter 6. Chapter 7 concludes this thesis and discusses future research directions.

CHAPTER 2

BACKGROUND

The proposed online process guidance approaches are part of a larger Process Improvement Environment (PIE). This chapter provides an overview of the PIE, discusses which components of the PIE are the focus of this work, and discusses prerequisites for our online process guidance approaches. The chapter also describes the Little-JIL process modeling language, which we use in the evaluation of the proposed online process guidance approaches.

2.1 Process Improvement Environment

The PIE is shown in Figure 2.1 and described in more detail in [14] and [15]. The goal of the PIE is to improve HIPs via various *offline* and *online analyses*. Offline analyses are applied to a model (the *Process Model* in Figure 2.1) of a real-world process to infer characteristics of that process and suggest improvements for future executions of that process. Offline analysis use the Process Model and various specifications, such as properties and hazards (discussed in section 2.1.1). Online analyses are applied *while* the real-world HIP is being executed to improve an ongoing process execution. Online analyses use the Process Model and real-time events captured during the ongoing process execution.

The offline analyses shown in Figure 2.1 have been implemented and evaluated [14, 30–32, 109, 120, 129]. Editors exist for creating scenarios, properties, hazards, and failure modes, which are inputs to the various offline analyses. An editor for creating process models in a formal process modeling language [27] and an interpreter to

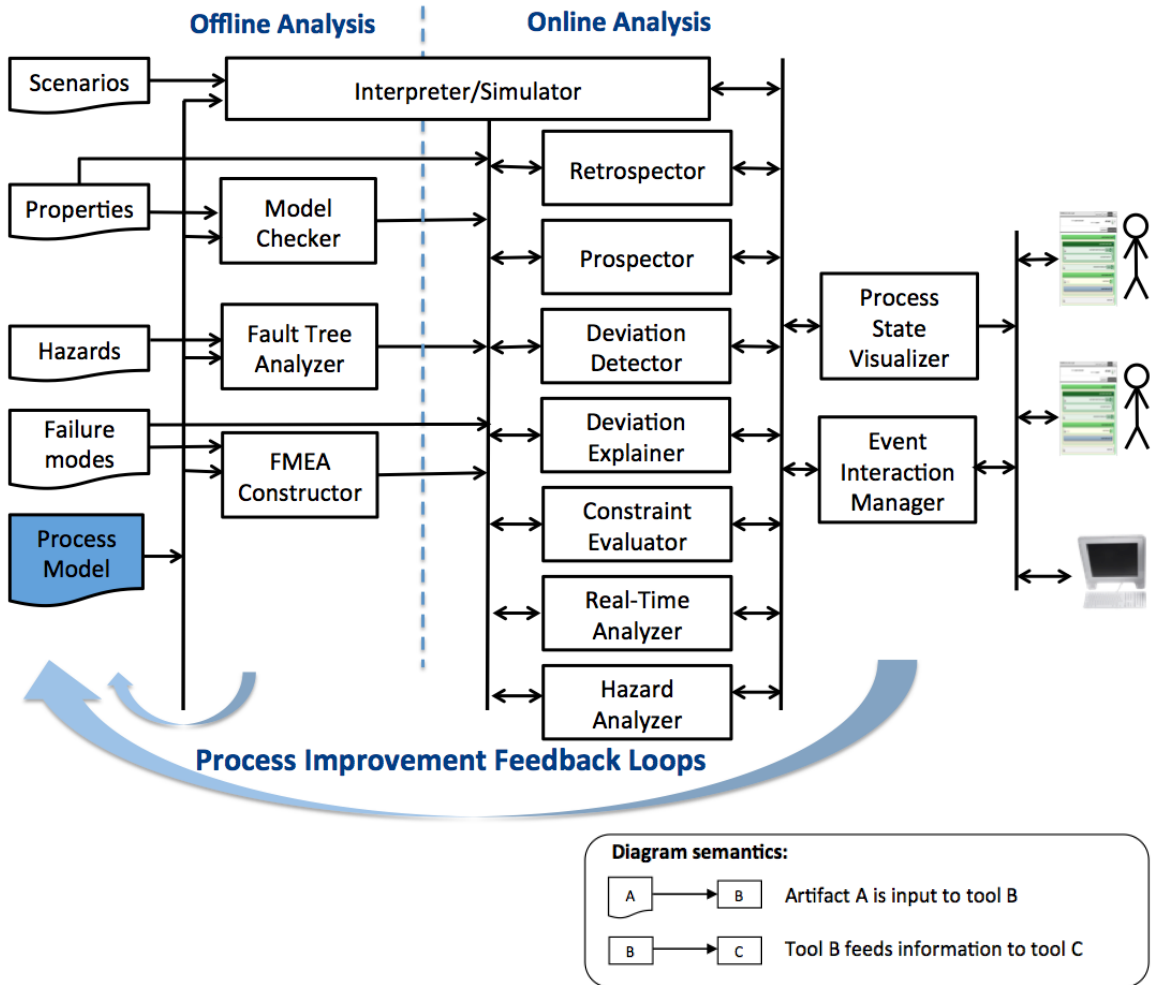


Figure 2.1: Process Improvement Environment.

execute such process models also exist as part of the PIE. At the time of writing this thesis, the online analyses are currently being investigated [15]. There are initial prototypes of the Deviation Detector [34], of the Deviation Explainer [33], and of the Process State Visualizer [40]. The rest of the online analysis components are yet to be implemented.

The key component of the PIE is the *Process Model*, which is a representation of a real-world HIP. This model captures detailed information about the corresponding real-world HIP, such as the activities in the HIP, who performs them and in what order(s), what artifacts are used or produced in the process, what resources are utilized, what problems might arise while performing the process and how such problems are handled. Chapter 3 discusses our work on creating such detailed process models and presents the process modeling notation used in the PIE.

2.1.1 Offline Analyses

As mentioned earlier, offline analysis are applied to the Process Model to infer characteristics of the real-world process and suggest improvements for future executions of the process. *Model Checking* [37], an analysis approach used to evaluate hardware and software systems, can be applied to the Process Model to determine whether all process executions captured by the model satisfy a given set of properties. These properties are typically requirements for the correct sequencing of process steps. An example property for a blood transfusion medical process is “The patient’s identity should be verified before infusing a unit of blood product into that patient.” Properties are typically expressed as finite-state automata or formulas in a suitable temporal logic. If the Model Checker finds that a property is violated, a counterexample—a process execution that demonstrates the violation—is provided. By examining the counterexamples, analysts can usually determine the cause of the

problem, which could be an inaccuracy in the property, an inaccuracy in the process model, or an actual defect in the modeled real-world process.

Another offline analyses technique part of the PIE is *Fault Tree Analysis (FTA)* [128]. FTA is a hazard analysis technique used to systematically identify and evaluate all possible causes of a given hazard. A hazard in a safety critical system 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” [88]. An example hazard in a blood transfusion process is “the patient received the wrong blood”.

Given a potential hazard in a process, FTA identifies events (component failures, human errors, etc.) in the process that could lead to the hazard and produces a fault tree, which provides a graphical depiction of all possible combinations of those events. Once a fault tree has been derived, qualitative and quantitative analysis can be applied to provide information, such as *minimal cut sets*, where a minimal cut set is a set of events whose occurrence is sufficient to cause a hazard. This information can then be used as guidance for reducing or removing vulnerabilities in the process.

Failure Mode and Effects Analysis (FMEA) [121], also referred to as Failure Mode Effect and Criticality Analysis (FMECA), is another safety analysis technique that can be applied to the Process Model. FMEA can be used to evaluate the impact of individual failures on the overall process. A failure mode represents a specific case in which some part of the process fails to meet its intent or requirements. Given a process model, potential failure modes are identified first. Then the process model is explored to determine all possible hazards that could eventually be caused by each failure mode. Failure modes then can be prioritized by their risks. The risk of a failure mode can be calculated based on the probability and detectability of that failure mode as well as the severities of hazards caused by that failure mode. For failure modes with high priorities, modifications to the process model (and sometimes

to the corresponding real-world HIP) are proposed to eliminate such failure modes or at least reduce their probability of occurring.

In the PIE, the Process Model can also be used to drive *Discrete-Event Simulation* to reason about efficiency of the modeled process. The Simulator in Figure 2.1 takes as input the Process Model along with scenario specifications, such as different combinations of resources to be used in the process. The Simulator can then execute the Process Model and determine which resource mixes optimize certain quantities of interest. For example, discrete event simulation can be used to study how different combinations of resources (e.g., the doctors, nurses, and beds available for treating patients) and allocation strategies affect patient length of stay in a hospital emergency department [41, 109].

The application of the offline analyses described above often results in finding problems with the Process Model or with the various specifications (scenarios, properties, hazards, failure models). Using the feedback from the offline analysis (indicated by the smaller semi-circular arrow going from the Offline Analysis area to the Process Model at the bottom of Figure 2.1), the Process Model and the specification are improved and the analyses are iteratively applied.

Sometimes, the problems discovered by the offline analyses are problems associated with the real-world HIP represented by the Process Model. In such cases, the analysis results can be used to find defects in the real-world HIP and suggest process changes that eliminate these defects, forming the basis for continuous process improvement [119]. The process changes can be first applied to the Process Model and the offline analyses can be used again to check whether the proposed changes indeed fix the identified defects and whether the proposed changes introduce different defects in the process. After the proposed process changes are determined to be safe on the process model, they can then be implemented in the real-world HIP.

2.1.2 Online Analyses

We envision that a carefully analyzed and validated process model would be used to drive online analyses *while* the real-world HIP is being executed to improve an ongoing process execution. The *Retrospector* in Figure 2.1 would keep track of process execution history, provide capabilities for searching this history, and proactively select information from the history that might be relevant to the current activities of the process performers. The *Prospector* would provide possible future process execution information, such as tasks to be performed, resources to be required, and upcoming decisions to be made to help process performers plan their work.

The *Deviation Detector* would monitor an ongoing process execution to determine if an executing HIP deviates from a set of recommended ways to perform that HIP, as specified by the Process Model. Such deviations could represent errors and, thus, detecting and reporting deviations as they occur could help catch errors before harm is done. In complex and time-sensitive HIPs, simply informing process performers that deviations have occurred might not be sufficient for identifying errors in a timely manner and for deciding how to recover from them before harm is done. The *Deviation Explainer* would provide information that could be useful for identifying errors, which in turn can help process performers with planning how to recover from these errors.

The *Constraint Evaluator* would notify human performers when process execution has, or will imminently, violate a specified constraint. The *Real-Time Analyzer* would keep performers informed of looming deadlines increasing the urgency of warnings as deadlines approach. The *Hazard Analyzer* would combine results from the offline FMEA and FTA with events from an ongoing process execution to warn process performers of potential risks for hazards.

The online analyzers would interact with performers of HIPs via the *Event Interaction Manager* and the *Process State Visualizer*. The Event Interaction Manager would handle the flow of events between the online analysis components and process

performers to ensure that the right events and information are delivered to the right performers and that the right information from the ongoing process is delivered to the online analyzers. The Event Interaction Manager will communicate directly with automated process performers and it will funnel events meant for human process performers through the Process State Visualizer, which would be responsible for creating a visualization of the process execution state. The Event Interaction Manager consists of two components (not shown in Figure 2.1 to reduce visual clutter)—a mechanism for capturing events from an executing process and feeding these events to the online analyzers and mechanisms for capturing events from online analyzers and redirecting them to the appropriate process performers. We call the former mechanism a *Process Execution Monitor* and the latter mechanism a *Process Performers Notifier*. The online analyzers, which would utilize the process model and information from the offline analyses, together with the Event Interaction Manager and the Process State Visualizer would provide *online process guidance* to human process performers to assist them with ongoing tasks.

The PIE would also accumulate process execution information, such as sequences of performed steps on different process executions and problems that have arisen, and that information could be used to refine offline and online analyses, improve the process model (represented by the rounded arrow pointing from the online analysis area to the Process Model at the bottom of Figure 2.1), and the actual modeled HIP. The ultimate goal of the PIE is to support *continuous process improvement* as introduced by Shewhart [119] and effectively applied by Deming [53].

2.1.3 Thesis Focus

This thesis focuses on three of the online analyzers—the *Deviation Detector* (discussed in Chapter 4), the *Deviation Explainer* (also discussed in Chapter 4), and the *Process State Visualizer* (discussed in Chapter 5). This thesis also explores tech-

niques for eliciting HIPs and for creating corresponding process models (discussed in Chapter 3), as detailed and accurate process models are essential for the proposed online guidance approaches.

2.2 Prerequisites for Online Process Guidance

2.2.1 Precise, Correct, and Sufficiently Complete Process Model

A Process Model is essential for the proposed PIE as that model drives both the offline and the online analyzers. A Process Model used for online guidance is a representation of the process that captures the recommended ways to perform that process. Thus, such a process model is the “gold standard” for performing the process it represents. To be useful for online guidance in complex HIPs a process model needs to satisfy certain criteria:

- **Precision.** The process model needs to be precise, i.e., written in a notation with well-defined semantics, so that the model can unambiguously guide process performers and the various online analyses could be automatically applied. For example, the Deviation Detector needs to be able to algorithmically explore the process model to determine whether an ongoing process execution deviates from that model.
- **Correctness.** The process executions captured by the model need to be correct with respect to a set of requirements identified by domain experts, so that the model can provide correct guidance to process performers. If the model contains incorrect executions, then they could result in providing wrong guidance that could ultimately lead to undesired process outcomes.
- **Sufficient completeness.** To support an adequate level of online guidance during an ongoing HIP, a process model needs to be *sufficiently complete*, i.e., it needs to capture all the executions that process stakeholders have agreed the

model should capture. The model should adequately capture the complexities of the HIP it represents, such as the activities that need to be performed, complex and realistic control flow (e.g., exception handling and concurrency), types of human and non-human performers, and artifacts that are used or produced. Such a process model would be more realistic compared to checklists and simple flow charts, which in turn should allow for guiding process performers in a wider range of scenarios.

Creating process models that meet these criteria could be challenging, especially for complex HIPs. We expect process models to be incrementally improved as problems are found and/or the HIPs they represent change. Section 3.1 discusses some elicitation techniques for obtaining detailed process information needed to create process models that can be used for online process guidance. Section 3.2 describes our experience with process modeling, including: creating such process models and using Little-JIL, a semantically rich and formal process modeling language that can support the kinds of process models needed for online process guidance (section 2.3); exception handling patterns for facilitating the modeling of exceptional situations (section 3.2.1); and a study that compares the effectiveness of a diagrammatic and a textual process representation in terms of their ability to facilitate process understanding (section 3.2.2).

2.2.2 Process Execution Monitor

Some of the online analyzers in the PIE rely on a mechanism for monitoring an executing process and for capturing events of interest. For example, the Deviation Detector needs to know the sequence of performed activities to determine whether process performers have deviated from the recommended ways to perform the process, as specified by the Process Model. As previously discussed, this mechanism, which

we call the Process Execution Monitor, is one of the two components of the Event Interaction Manager in Figure 2.1.

The details of such a mechanism for monitoring an executing process and associated issues are outside the scope of this work. Recent developments in HIPs should facilitate monitoring of executing processes. For example, electronic medical records are being introduced in more medical HIPs and data entries in such records could be used to infer what activities are being/have been performed. Medical scribes are increasingly being used in medical HIPs to document the executing process [65] and thus could capture the activities as they are being performed. Computer vision techniques could also be utilized to recognize process events in a live video stream of cameras installed in the environment where a process is taking place.

2.3 The Little-JIL Process Modeling Language

To create models that satisfy the criteria discussed in section 2.2.1 we chose the Little-JIL process modeling language [27], because it has the capabilities to create such models. Little-JIL's rich semantics allow creating realistic models of complex HIPs. Little-JIL supports modeling of process activities, complex control flow (such as exception handling and concurrency), specification of process performers responsible for the various activities and of the artifacts used or produced during these activities. Little-JIL's semantics are formally defined, allowing for Little-JIL process models to be automatically interpreted and analyzed by the various online analyzers in the PIE in Figure 2.1. Little-JIL's compact visual, diagrammatic representation supports scalability of process models, allowing us to capture large and complex HIPs. Little-JIL also has a hyperlinked textual representation, *the Little-JIL narrative*, that is automatically kept synchronized with the diagrammatic notation. The availability of these two representations facilitates communication with domain experts during process elicitation and validation of the resulting models.

A Little-JIL process model consists of three main specifications—a resource specification, an artifact specification, and a coordination specification. The resource specification defines the process performers (called *agents*) and resources (human and non-human) needed to perform process activities. The artifact specification defines the products of the process activities. The coordination specification brings these two together by defining which agents, using which resources, perform which activities on which artifacts at which times. The main building blocks of a Little-JIL process model are the *steps*. A step corresponds to an activity performed by a human or non-human agent. A Little-JIL process model is a hierarchical decomposition of steps where each step can be decomposed into substeps to an arbitrary level of detail.

We first describe the Little-JIL diagrammatic representation and also introduce some of Little-JIL’s semantics. Section 2.3.2 describes the Little-JIL narrative representation.

2.3.1 The Little-JIL Diagrammatic Representation

Figure 2.2 illustrates the diagrammatic representation of a Little-JIL step. A step in Little-JIL’s diagrammatic notation is iconically represented by a black bar. Right above the black bar is the *step name*, which is the name of the process activity the step represents. A step can be decomposed into substeps, Figure 2.2 shows one such substep (we also refer to the decomposed step as the *parent step*). The *sequencing badge* of a step determines the order in which substeps need to be completed and how many of them need to be completed, so that the step is considered completed. There are four sequencing badges in Little-JIL. A *sequential* badge (represented by an arrow pointing to the right) means that all substeps need to be completed in left-to-right order for the step to be considered completed. A *parallel* badge (represented by an equal sign) means that all substeps need to be completed, but they can be done in any order, including in parallel. A *try* badge (represented by an arrow crossing an

X) means that substeps should be tried in left-to-right order until one of them is successfully completed. At that point, the parent step is considered completed. A *choice* badge (represented by a horizontal line segment crossing a circle) means that any substep can be chosen to be performed. If that substep is successfully completed, the parent step is considered completed; otherwise one of the remaining substeps can be chosen to be performed next, until one is successfully completed.

In Little-JIL, it can be specified how many times a step should be performed. This is done via the *cardinality* symbol. The plus symbol in Figure 2.2 means that the *Substep* should be performed one or more times. Little-JIL allows for specifying an arbitrary cardinality, including upper and lower bounds.

Steps in Little-JIL can *throw exceptions* to represent abnormal situations in a process execution. When a step throws an exception, that exception propagates up the step tree until a matching exception handler is found. At that point, the exception handler is executed. For example, if the *Substep* in Figure 2.2 throws an exception and the *Handlerstep* is a matching handler for that exception, the *Handlerstep* will be executed next. Exception handlers are regular steps and, thus, they can be further decomposed to any desired level of detail. Exception handlers are attached to the X badge on the left side of the parent step bar.

After an exception handler is completed, the *continuation badge* connecting the handlers to its parent step determines where in the process model control is returned. There are four continuation badges in Little-JIL. A *continue* badge (represented by an arrow pointing to the right) means that control should return at the step after the step that threw the exception (what that step is depends on the sequencing badge of the parent step). A *rethrow* badge (represented by an arrow pointing upwards) means that the exception should be re-thrown and matching handlers searched up the step tree. A *restart* badge means that the parent step of the one that threw the

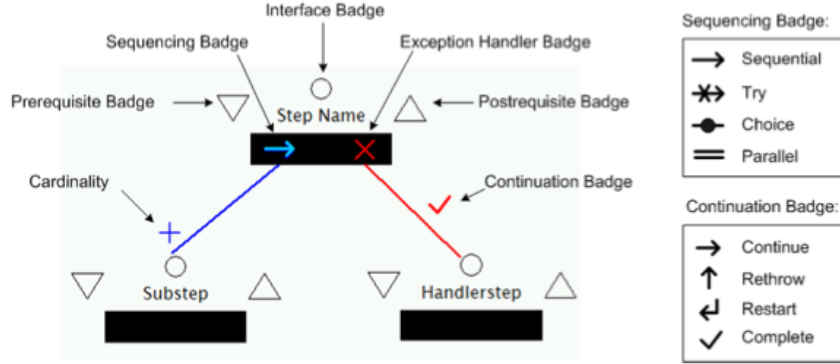


Figure 2.2: Little-JIL Step (figure adapted from [29])

exception should be restarted. A *complete* badge means that the parent step of the one that threw the exception is considered completed.

A Little-JIL step can also have pre- and postrequisites, which are specified via the pre- and post requisite badges (represented by triangles pointing downwards and upwards respectively). Prerequisites are conditions that need to be true or other steps that need to be performed before the step is allowed to start execution. Postrequisites are conditions that need to be true or other steps that need to be performed after a step has finished execution.

The circle on top of a step’s black bar represents the step’s interface. The step’s interface is the place where the agent responsible for executing the step, the resources utilized by the step, the artifacts produced by the step, and potential exceptions that can be thrown by the step are specified.

2.3.1.1 Example Little-JIL Process Model

Figure 2.3 shows a Little-JIL process model of a simplified blood transfusion process. This process is decomposed into six substeps: *obtain patient’s blood type*, *order blood from blood bank*, *prepare blood*, *pick up blood from blood bank*, *perform bedside checks*, and *infuse blood*. These six substeps need to be performed in left-to-

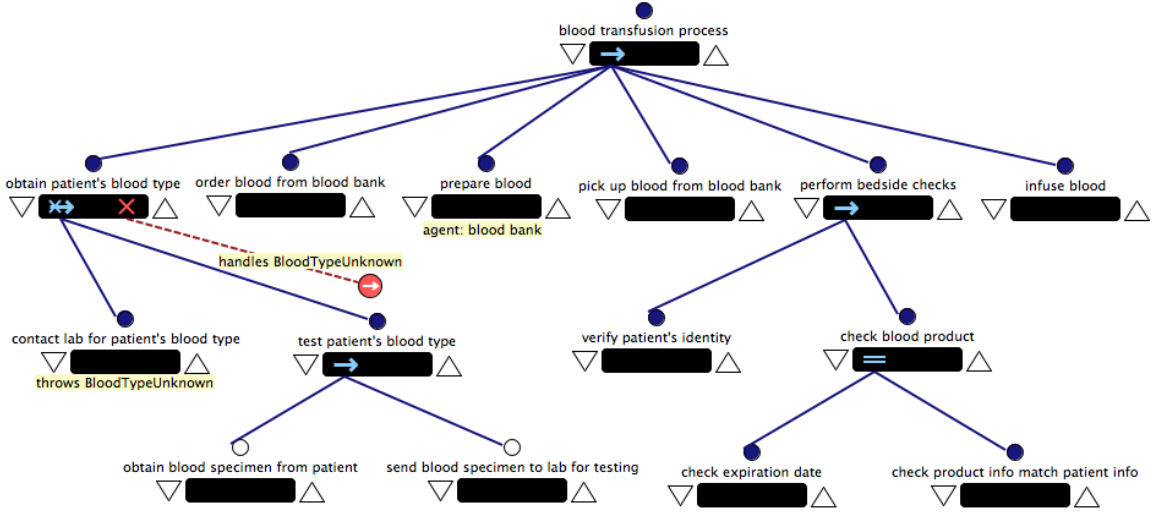


Figure 2.3: Little-JIL model of a simplified blood transfusion process.

right order, indicated by the sequencing badge (right arrow) of the parent step *blood transfusion process*. Some of these six substeps are further decomposed (e.g., *obtain patient's blood type*), whereas others are not, i.e., they are *leaf steps* (e.g., *order blood from blood bank*).

Obtain patient's blood type is a *try step* (indicated by the *try* sequencing badge on the left part of the step bar), which means that to perform that step, the nurse needs to first contact the lab for the patient's blood type, and if the blood type is unknown, the nurse needs to test the patient's blood type. The possibility that the blood type might be unknown is represented by specifying that the step *contact lab for patient's blood type* can throw the exception *BloodTypeUnknown*. If this exception is thrown, it is propagated up the step tree and the matching exception handler (attached to the exception handling badge of the step *obtain patient's blood type*) is executed next. This exception handler is a *simple handler*, meaning that it just specifies where control flows after the exception is thrown. It has the *continue* continuation badge, which means that the step to be performed next is *test patient's blood type*.

Once the nurse has obtained the patient’s blood type, the nurse needs to order the blood from the blood bank. At that point, the blood bank can prepare the blood¹. Once the blood has been prepared, the nurse needs to pick it up and then perform the bedside checks on the patient. These checks consist of verifying the patient’s identity and checking the blood product information. The step *check blood product information* is a parallel step (indicated by the equal sign step badge), which represents the fact that its substeps *check expiration date* and *check product info matches patient info* can be performed in any order². Finally, after the nurse has performed the bedside checks, the blood can be infused into the patient.

2.3.2 The Little-JIL Narrative Representation

The Little-JIL narrative representation is a textual, hyper-linked, description of the process model. In addition to a natural-language description of the process model, this textual representation includes a table of contents and an index of process steps. The Little-JIL narrative is generated by the Little-JIL Narrator. The Narrator takes as input a Little-JIL process model, a set of templates of English phrases that correspond to the different semantic features of the process modeling language, and customization rules, and weaves together the narrative representation.

Figure 2.4 shows the narrative corresponding to part of the Little-JIL model of the simplified blood transfusion process in Figure 2.3. The narrative consists of two main parts: a table of contents on the left and the descriptive part on the right. The table of contents lists the names of the steps from the process model and uses

¹The note under the step *prepare blood* indicates that the agent for that step is the blood bank. All other steps in this process model are performed by the nurse. To reduce visual clutter on the diagram, we elide the agent specification for these steps.

²Different problems could arise while performing the substeps of *check blood product* (e.g., the nurse might find that the blood has expired). Thus, the substeps should throw various exceptions to represent these possible problems. These exceptions and their corresponding handlers are elided from Figure 2.3 to reduce visual clutter and because they are not essential for this discussion.

the same icons used in the diagrammatic representation to represent the step kinds (sequential, parallel, choice, try step). The parent/child relationship from the Little-JIL process model tree is captured by the indentation in the table of contents. For example, the steps at one level of indentation under *blood transfusion process*, namely *obtain patient's blood type*, *order blood from blood bank*, *prepare blood*, and so on, are the substeps of *blood transfusion process*. Each step in the table of contents is also a hyperlink and clicking on it will bring up a more detailed description of that step in the descriptive part of the narrative.

The descriptive part of the narrative (the right part of Figure 2.4) contains a section for each step in the process model. This step section consists of several subsections that present various attributes of the given step, such as name, pre/post requisites, substep sequencing information, exceptions, and required resources. For instance, the step section for *obtain patient's blood type* contains subsections about the step's outputs, the resources needed to perform the step, the step's substeps and the order in which they have to be executed. The step section for *contact lab for patient's blood type* contains similar information and also the exception that the step throws and how that exception is handled.

Unlike the diagrammatic representation of the process model in Figure 2.3 where familiarity with the notation semantics is assumed, the descriptive part of the narrative representation provides sentences to explain the process. For example, the step section for *obtain patient's blood type* in Figure 2.4 explains what it means for the step to be a try step, namely that its first substep needs to be tried first, and if it fails, the second substep should be tried and so on.

The descriptive part of the narrative also uses hyperlinks to facilitate navigation. When the substep *test patient's blood type* (in the step section for *obtain patient's blood type*) is clicked, for example, its step section will be displayed and the user will see the detailed information associated with that step. Another facility to help

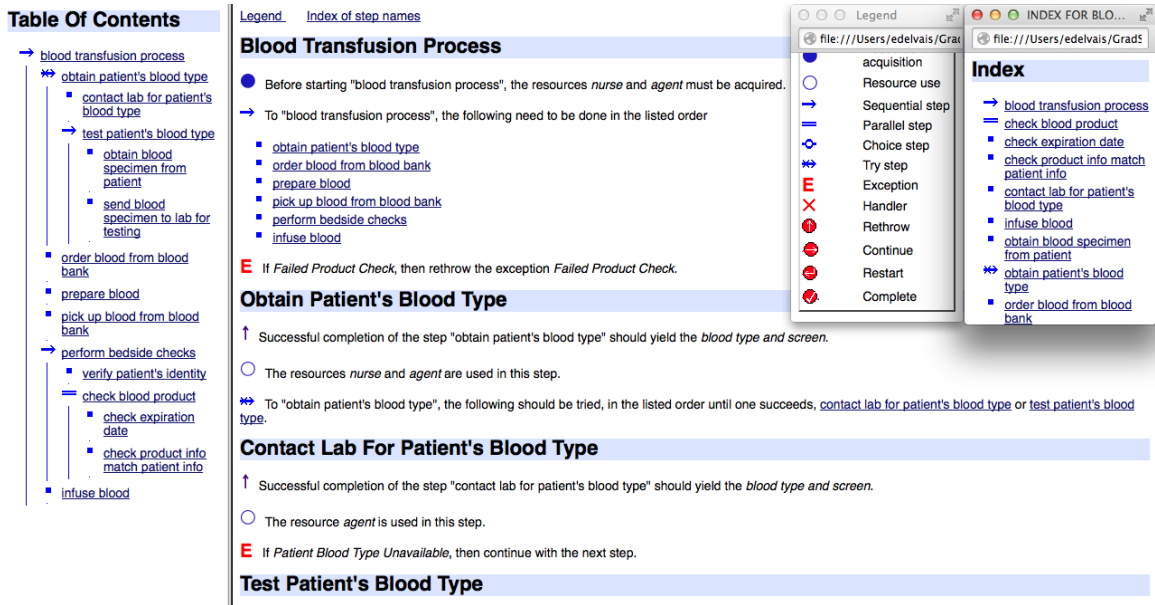


Figure 2.4: The narrative corresponding to the Little-JIL process model in Figure 2.3.

with navigation is an alphabetized index of step names (shown in the top right of Figure 2.4), where each step is represented as a hyperlink that can be clicked to display the description for that step. The index can be opened at any time by clicking on *Index of step names* on the top of the main part of the narrative and closed when not needed.

The narrative uses the same icons as the diagrammatic representation of the process model. Although these icons are not necessary to understand the narrative view, they might be helpful for users who would like to work with both views at the same time. They also provide some visual grouping of sentences based on the icon the sentences are associated with. The meaning of the icons can be seen in a legend (shown to the left of the index of step names in Figure 2.4), which can be opened and closed the same way as the index of step names.

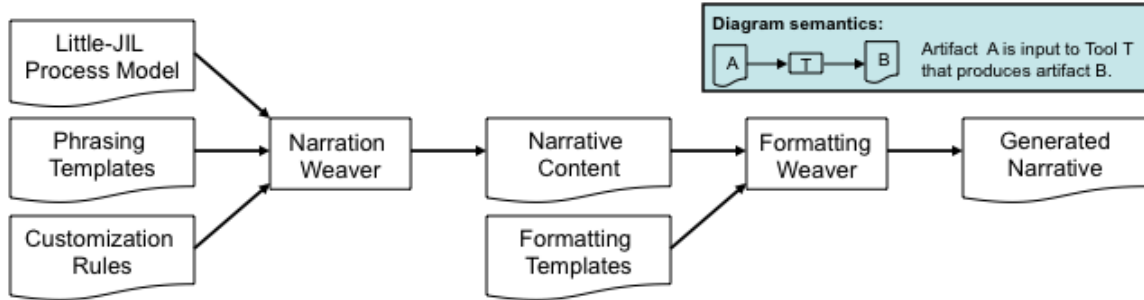


Figure 2.5: Architecture of the Little-JIL Narrator.

2.3.2.1 Design of Little-JIL Narrator

Figure 2.5 shows the high-level architecture of the Narrator. The *Little-JIL Process Model*, the *Phrasing Templates*, and the *Customization Rules* are used by the *Narration Weaver* to produce the *Narrative Content*, which contains just the content and the structure without any formatting of the natural-language document to be generated. The *Formatting Weaver* then combines the *Narrative Content* together with the *Formatting Templates* to produce the final *Generated Narrative*.

2.3.2.1.1 Phrasing Templates. The Phrasing Templates are parameterized, natural-language phrases that correspond to the different semantic features of the Little-JIL process language (e.g., what it means for a step to be sequential), where the parameters represent information that is specific to a given process model. Table 2.1 shows three example phrasing templates. The first phrasing template is used with sequential process steps to generate a sentence explaining the order of execution of their substeps. This template was used to generate the sentence in Figure 2.4 explaining the order in which the substeps of *blood transfusion process* should be performed.

The second phrasing template in Table 2.1 is used to generate a sentence that explains what it means for a step to be a try step. For instance, applying this phrasing template to the step *obtain patient’s blood type* (in Figure 2.3) results in the sentence *To “obtain patient’s blood type”, the following should be tried, in the listed*

To [stepName], the following need to be done in the listed order [substepsList].
To [stepName], the following should be tried, in the listed order, until one succeeds [substepsList].
Successful completion of the [activity] [step name] should yield the [parameter name].

Table 2.1: Example phrasing templates.

order, until one succeeds: “contact lab for patient’s blood type” or “test patient’s blood type”. This exact sentence can be seen in the step section for *obtain patient’s blood type* in Figure 2.4.

The third phrasing template in Figure 2.1 is used to generate a sentence explaining what it means for a step to have an output parameter. For instance, applying this phrasing template to the step *obtain patient’s blood type*, which outputs the patient’s blood type and screen, results in the sentence *Successful completion of the step “obtain patient’s blood type” should yield the blood type and screen.*

2.3.2.1.2 Customization Rules. The Customization Rules in the Narrator architecture in Figure 2.5 represent a set of user preferences to customize the content and the structure of the generated natural-language narrative. The Narrator supports the use of synonyms. For example, different words can be used to refer to a process activity. The parameter *[activity]* in the third phrasing template in Figure 2.1 is a placeholder for such a synonym. The word “step” was used in place of *[activity]* when this phrasing template was instantiated to describe the output parameter of the step *obtain patient’s blood type* in Figure 2.4.

Another kind of customization supported by the Little-JIL Narrator deals with the ability to hide or show certain kinds of process information. For example, the user can select to hide or show sentences that present information about the resources in a process model. Before the narrative shown in Figure 2.4 was generated, the option to show resources was selected. Thus, resource information, such as the human agents

responsible for executing the process steps, is included in the narrative. The user can choose to hide or show this information or information about other aspects of the process, such as parameters and exceptions. The Narrator also provides the flexibility to choose what kinds of process steps to associate certain information with. For example, the user can choose to show resource information only for leaf steps but not intermediate steps. This is sometimes useful as intermediate steps in Little-JIL are often used for coordination purposes, whereas the actual work performed by agents is modeled by leaf steps.

The Little-JIL Narrator also provides facilities to customize the sentence structure of the generated narrative. For instance, the user can define when the substeps of a step should be enumerated as a comma-separated phrase or when they should be shown as a list.

2.3.2.1.3 Formatting Templates. The Narrative Content artifact produced by the Narration Weaver (as shown in Figure 2.5) contains the raw content of the generated narrative, but does not have any formatting information. It is the job of the Formatting Templates to define the presentation style of the generated narrative. This design essentially follows the well-established recommendations from the web application domain to separate content from presentation.

For Figure 2.4, for example, the Formatting Templates were responsible for defining text font, text color, text size, text style (e.g., bold vs. non-bold), spacing information and background color for the table of contents, the main part of the narrative, the index and the legend. The Formatting Templates were also responsible for associating images (e.g, arrow, filled circle, check mark, etc.) with the different sections of the narrative and for the visualization (indentation and vertical lines that help keeping track of the hierarchical decomposition) of the table of contents. The set of Formatting Templates used by the Formatting Weaver in this example resulted in an HTML-based generated narrative. A different set of Formatting Templates could be

used to produce a plain text (not hyperlinked) narrative or a narrative in some other document format.

The Phrasing Templates, the Customization Rules, and the Narrative Content artifacts are currently XML documents following a schema we defined. The Narration Weaver is a Java system. The formatting templates are expressed as XSLT [7] templates and the Formatting Weaver is therefore an XSLT processor.

CHAPTER 3

PROCESS ELICITATION AND MODELING

It is difficult to create process models for online process guidance that satisfy the requirements discussed in section 2.2.1. One reason for this difficulty is that it is time-consuming and challenging to elicit accurate and sufficiently complete process information from process stakeholders. A single stakeholder often does not know how the entire process works, or should work, necessitating elicitation of the process from several stakeholders. Different stakeholders, however, often have different understanding of the process, use different terminology, and could even provide conflicting information. Furthermore, for large and complex HIPs, it is often impractical for all stakeholders to take part in the elicitation. Thus, processes are often elicited from a subset of all process performers, which makes difficult capturing process nuances that depend on personal style and preferences.

It is also difficult to formally represent an elicited process, so that the resulting model is precise enough to support automated online process guidance and yet rich enough to support process performers in various scenarios and circumstances. Many process modeling notations lack the semantics to express complex process behavior, such as exception handling and concurrent execution. Some notations that do have such semantics have awkward syntax, increasing the probability making an error while creating the process model. Regardless of what process modeling notation is used, intricate process details, such as resumption semantics after exception handling or synchronization in the presence of concurrency, are difficult to capture correctly.

To address the process elicitation challenges, we investigated several process elicitation methods and evaluated their effectiveness in terms of their ability to reveal several kinds of important process information. Section 3.1 describes this investigation. To address some of the challenges in representing elicited process information via a process model that satisfies the desiderata from section 2.2.1, we took several steps:

- For our online process guidance approach, we used the Little-JIL process modeling language [27], whose rich semantics allow for capturing complex real-world processes. Little-JIL’s explicit support of exception handling and concurrency and its diagrammatic notation facilitated accurately modeling complex process behaviors.
- We identified common ways in which people handle exceptions in processes and codified these ways into a catalog of exception handling patterns to facilitate modeling of exceptional behaviors.
- To facilitate the understanding of process models, we developed a natural language process representation that can be automatically generated from a formal process model. We conducted a user study to compare this natural language representation with the existing Little-JIL diagrammatic representation.
- We also employed formal analysis techniques to investigate their ability to validate process models.

Our experience with process modeling is described in Section 3.2.

3.1 Process Elicitation

The process elicitation methods we studied fall under the general approach of *task analysis* [44, 81]. The goal of task analysis is to describe the manual and mental activities involved in accomplishing a task. We focused on observations of and

interviews with process performers, which are common task elicitation methods for evaluating how performers complete a certain task. In particular, we investigated the ability of these methods to reveal specific kinds of process information—process steps that occur on the normal process flow; exceptional situations that might arise during the performance of a process; and process performers’ responses to such exceptional situations. The following section describes the case study on which we applied these methods, the methods themselves, and our observations about these methods.

3.1.1 Case Study

To evaluate the elicitation methods discussed above, we used them to elicit a part of a critical medical process and then compared them in terms of their ability to elicit certain process information. The kinds of process information that we focused on were steps on the *normal process flow* (i.e., activities that are performed on process executions when no problems arise), *exceptional situations* (i.e., problems that might arise during a process execution), and *recognition of and responses to exceptional situations*. We based our decision to elicit these kinds of process information on frameworks for understanding and eliciting processes from other high-risk domains [103, 133, 134, 137].

3.1.1.1 Elicited Process

We elicited the chemotherapy treatment plan review process performed by a Practice Registered Nurse (RN)¹ as a critical part of a larger process for outpatient breast cancer chemotherapy administration at the D’Amour Center for Cancer Care in Western Massachusetts. We chose the RNs’ treatment plan review process because it is

¹The term “Practice RN” is specific to the D’Amour Center for Cancer Care. It refers to RNs responsible for verification of chemotherapy treatment plan and orders and who participate in the preparation for chemotherapy administration. The RNs who are primarily in charge of the actual chemotherapy administration are called “Clinic RNs” in this cancer center.

a safety-critical process whose complexity makes it an appropriate benchmark for evaluating different process elicitation methods.

Chemotherapy treatment plans differ by cancer type, the stage or extent of the disease, the goal of therapy, and a patient’s tolerance for therapy with specific agents. Therapeutic options may change rapidly as new research findings are released, thus requiring ongoing diligence and review to ensure that a chemotherapy treatment plan is appropriate for the patient and is safely executed by the team of care providers [11,61]. During the treatment plan review process, RNs perform essential coordinating functions with physicians (to ensure the treatment plan is accurate), schedulers (to ensure chemotherapy appointments on the treatment plan are scheduled correctly), and Clinic Registered Nurses (to ensure chemotherapy medications are administered as directed in the treatment plan). The RNs also use several data sources when reviewing treatment plans (e.g. clinical notes, electronic medical records (EMRs), the paper chart, *careset*² information, reference books, and online resources).

3.1.1.2 Elicitation Methods

We used five elicitation methods to detail the process by which RNs review treatment plans. We used unstructured interviews, direct observations, and three types of semi-structured interviews to elicit normal flow steps, exceptional situations, and RNs’ recognition of and responses to exceptional situations . We obtained protocol approval from the Cancer Center Clinical Research Review Committee and from the Institutional Review Boards of the University of Massachusetts and Baystate Medical Center; we obtained informed consent from all participants.

3.1.1.2.1 Unstructured Interviews. Over the course of six months, we conducted six unstructured interviews with a senior RN experienced in reviewing treat-

²A *careset* is a standardized treatment (based on best practices) for a given diagnosis.

ment plans. These unstructured interviews were open and conversational in nature. These interviews were iterative in that we elicited information about the treatment plan review process, created a precise and detailed natural language description of the process, and then presented that description to the RN for further refinement. In parallel, we created a formal and detailed model of the treatment plan review process in the Little-JIL process modeling language [27] (Little-JIL is presented in section 2.3). We performed several iterations of the unstructured interviews—guided by the semantic features of Little-JIL³—until the RN expressed satisfaction that the natural language description of the process accurately represented the treatment plan review process as performed at the Cancer Center. These unstructured interviews occurred in parallel with interview sessions with other clinicians involved in the chemotherapy process, as one of our goals was to create a process model for the full outpatient breast cancer chemotherapy preparation and administration process.

3.1.1.2.2 Direct Observations. After completing the unstructured interviews with the senior RN, we observed three other experienced RNs conduct two treatment plan reviews each. We stopped collecting data after six observations because research using qualitative data collection approaches suggests that prominent themes tend to emerge after six data collection points [64]. Additionally, during our observations, we observed the same normal flow steps and exceptional situations across multiple sessions—a sign of data saturation. Our goal in conducting these observations was to improve our understanding of the process captured during the unstructured interviews by observing the process as it happened in the real world. Because cognitive tasks are often difficult or impossible to observe, the observed RNs used a think-aloud protocol [44] to verbally describe their cognitive tasks (such as verifying that information

³Little-JIL’s explicit support for exception handling, for example, prompted us to ask about information related to exceptional situations that might arise during the performance of the process and what the response(s) should be. Little-JIL’s semantics are discussed in detail in section 2.3.

1. Look up the patient's record in the electronic medical record (EMR) (e.g. using account number)
2. Confirm presence of several height/weight records (in EMR) and that they are consistent with each other
3. Confirm patient's height/weight readings are not out of date
4. Note that there are not enough entries for patient's height/weight to judge stability of height/weight
5. Note that the patient's height/weight are out of date
6. Retrieve patient chart from medical records
7. Confirm presence of several height/weight records and that they are consistent with each other
8. Note that there are not enough entries for patient's height/weight to judge stability of height/weight
9. Confirm medication name, dose base and cycle info on treatment plan match doctor's clinical note
10. Tell Clinic MA/Triage MA (Scheduler)/Clinic RN to measure height/weight on next patient visit
11. Hold treatment plan

Figure 3.1: Sample sequence of observed steps

on two artifacts matched) as they completed the process. Two researchers were present for each observation and, using the audio recordings from the sessions, they reconciled the differences in the observation notes after all observations were complete. Figure 3.1 shows a sample sequence of observed steps. Throughout this chapter, we refer to such a sequence of process steps as a *process trace*.

3.1.1.2.3 Semi-structured interviews. We conducted three kinds of semi-structured interviews with each of the three observed RNs. In the semi-structured interviews, we asked each RN the same questions in the same order, but asked clarifying questions as needed. As often occurs in qualitative research, the interview materials for the semi-structured interviews were guided by the findings from the unstructured interviews, as these two methods complement one another [24]. We conducted the semi-structured interviews after the observations to avoid affecting the process by which the RNs completed their treatment plan reviews. As in the observations, two researchers were present for each interview and, using the audio recordings from the

1. A Triage MA leaves a treatment plan and orders for a patient in your tray. You confirm that pretesting has been done. What do you do next?
2. When you go to check that a patient's height and weight have been entered in the CIS (*the Cancer Center's EMR system*), you notice they are missing. How do you proceed?
3. When you go to check that a patient's height and weight have been entered in the CIS, you notice a patient's height and weight measurements are stale. How do you proceed?
4. You receive new height and weight measurements for a patient. There is a 6% change in the dose based on these new values. How do you proceed?
5. While reviewing a patient's treatment plan, you notice that the treatment plan was not created from a careset. How do you proceed?
6. While reviewing a patient's treatment plan and orders, you notice the orders were entered by a Fellow. How do you proceed?
7. While verifying doses for a patient, you notice that the height and weight in the treatment plan doesn't match the height and weight in the CIS or in the patient chart. How do you proceed?

Figure 3.2: Sample open-ended scenarios.

sessions, they reconciled the differences in the interview notes after all interviews were completed.

3.1.1.2.3.1 Open-ended prompts. In the first kind of the semi-structured interviews, we constructed fifteen plausible scenarios from the process model created during the unstructured interviews, where each scenario represented one partial process trace. These scenarios were open-ended: we asked the RNs how they would continue the treatment plan review process given the scenario. Some scenarios prompted the RNs with only normal flow steps, while others included steps performed in response to exceptional situations as well. Seven sample scenarios are shown in Figure 3.2.

3.1.1.2.3.2 Complete process traces. In the second kind of the semi-structured interviews, we presented each of the RNs with three complete treatment plan review process traces in free-text form, based on the process model created during the unstructured interviews. The first process trace represented a normal flow process execution; the second and third process traces included exceptional situations and possible responses to those exceptional situations. The third process trace is shown

in Figure 3.3. We asked each RN whether each process trace was feasible and, if not, what steps should be added, removed, or reordered.

3.1.1.2.3.3 Full process model. In the third part of the semi-structured interviews, we presented each RN with a free-text representation of the full process model created during the unstructured interviews. This full process model included the normal process flow, all exceptional situations and all possible responses to exceptional situations defined during the unstructured interviews. We asked each RN whether the full process model captured the process accurately and, if not, what steps should be added, removed, or reordered.

The fifteen open-ended prompts, the three complete process traces, and the full process model used in the semi-structured interviews are included in Appendix A.

3.1.1.3 Results

Figure 3.4 shows the results for all the elicitation methods based on the number of new normal flow steps, exceptional situations, and responses to exceptional situations (hereafter referred to as events) that were identified via each method. We elicited 35 unique normal flow process steps, 16 unique exceptional situations, and 31 unique steps as part of the RNs' responses to exceptional situations, or 82 total unique events. We identified 52 of the 82 (63%) events through the unstructured interviews, 22 new events through the observations (15 solely elicited during the observations), 7 new events through the semi-structured interviews using open-ended prompts (5 solely elicited using this method), and 1 new event through the semi-structured interviews using the full process description.

Figures 3.5, 3.6, and 3.7, show by which method we elicited each event. The integers on the horizontal axis represent events, where the names of the corresponding events are provided in Tables 3.1, 3.2, and 3.3 respectively. The elicitation methods are listed on the vertical axis in Figures 3.5, 3.6, and 3.7, and each pair (event E,

1. You pick up the treatment plan and the orders that the Triage MA has left.
2. You confirm that labs have been done.
3. You discover that a lab result is missing and the drugs are not platinum based.
4. You tell an MA to draw the labs next time the patient comes.
5. You find out that the patient does not have a scheduled appointment and you tell the MA to schedule one.
6. You put a sticky note on the treatment plan to check for the labs before signing the plan.
7. You confirm that the scans have been done.
8. You confirm the existence of patient height/weight data in the CIS.
9. You confirm that the patient's height/weight are not stale (i.e more than 2 weeks old).
10. You confirm that the treatment plan is created from a careset.
11. You confirm the existence of chemo orders for the patient but you find out that they are missing.
12. You call the MD to enter the orders in the system.
13. You put a sticky note on the treatment plan to check for the orders.
14. You stop your work on the treatment plan for this patient and wait until the MD enters the orders.
15.
16. (in 2 days) You find out that the MD has entered the orders for that patient.
17. You confirm that the orders have been entered by an Attending.
18. You verify the doses:
 - a. You confirm that the height/weight on treatment plan, in CIS, and in the patient chart all match.
 - b. You calculate the patient's BSA using height/weight from CIS.
 - c. You calculate doses using the BSA just calculated and the information from the treatment plan.
 - d. You confirm that the calculated doses match the ones on the chemo orders.
 - e. You confirm that the dose base on treatment plan is consistent with the doses on orders.
19. You check all sticky notes to make sure that everything is completed and you confirm that the labs have been done.
20. You sign the treatment plan.
21. You leave the treatment plan in Triage MA's tray.

Figure 3.3: Sample process trace.

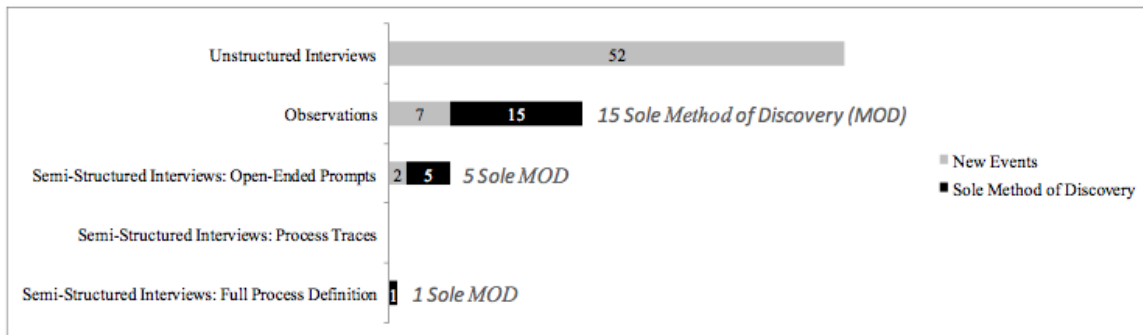


Figure 3.4: Number of new events identified using each elicitation method.

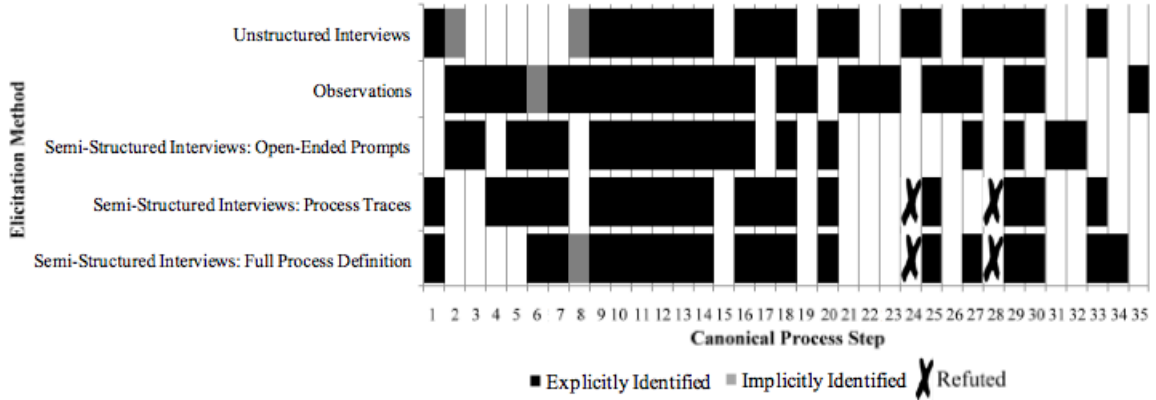


Figure 3.5: Normal flow process steps identified and refuted via each method.

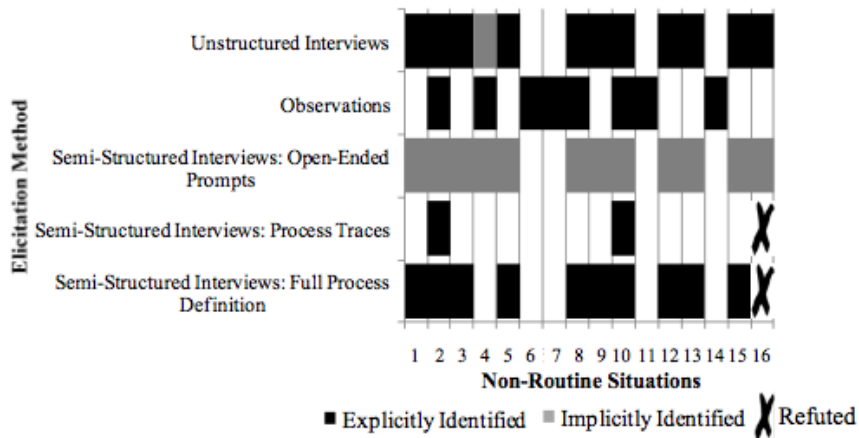


Figure 3.6: Exceptional situations identified and refuted via each method.

elicitation method M) is interpreted as the contribution of method M for eliciting event E . In section 3.1.1.4, we elaborate on how we reconciled differences in the RNs' responses.

Events shown in black in Figures 3.5, 3.6, and 3.7 signify that the event was *explicitly identified* using a specific elicitation method. An event was *explicitly identified* if it was mentioned during the unstructured interviews, occurred during the observations, or was mentioned during the semi-structured interviews.

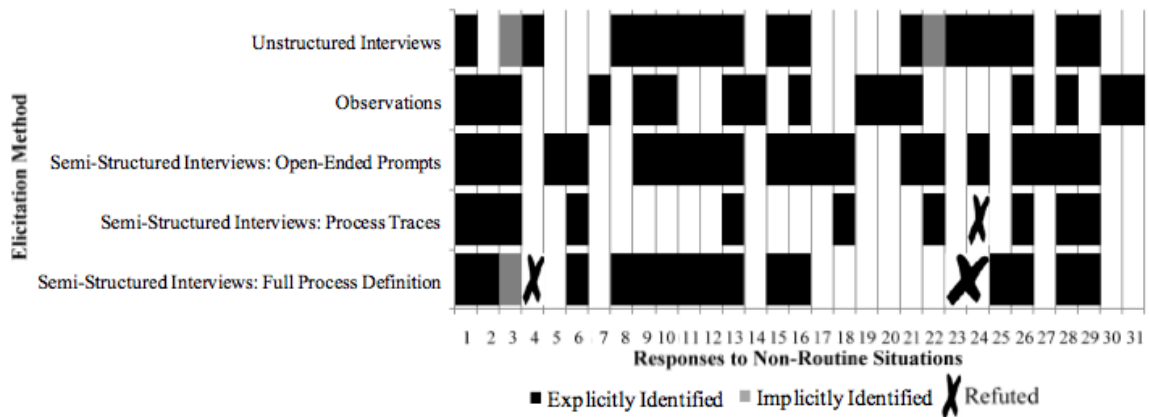


Figure 3.7: Responses to exceptional situations identified and refuted via each method.

Step Name	
1	Pick up treatment plan and orders
2	Look up the patient's record in the EMR (e.g. using account number)
3	Confirm correct MD on treatment plan
4	Confirm diagnosis on treatment plan matches diagnosis on MD clinical note
5	Confirm medication name, dose base and/or cycle info on treatment plan match MD clinical note
6	Confirm correct pre-medications for diagnosis (via experience)
7	Confirm correct medications for diagnosis (via experience)
8	Locate careset corresponding to treatment plan
9	Confirm correct pre-medications for diagnosis (via careset)
10	Confirm correct medications for diagnosis (via careset)
11	Confirm patient's height/weight readings are not out-of-date
12	Confirm presence of several height/weight records (in the EMR) and that they are consistent with one another
13	Confirm height/weight on treatment plan, EMR, and/or patient chart all match
14	Calculate body surface area (BSA) manually using height/weight from treatment plan or EMR
15	Use BSA calculated by computer
16	Calculate dose(s) using dose base(s) on treatment plan or on careset and patient's BSA
17	Confirm existence of chemo orders (for medications and pre-medications) in the EMR
18	Confirm manually calculated dose matches dose on orders in the EMR
19	Enter manually calculated BSA in appropriate box in treatment plan
20	Confirm orders are created by, or approved by, an attending MD (via an MD-to-RN order in EMR system)
21	Confirm cycle info on treatment plan matches cycle info in careset and/or in orders in EMR
22	Confirm the patient is not having concurrent radiation to that specified on the treatment plan (in the EMR)
23	Confirm that adequate time has passed since previous chemotherapy treatment plan was completed
24	Confirm labs have been done
25	Confirm scans have been done
26	Confirm exams have been done (if any are needed)
27	Confirm specified dose base on treatment plan matches dose base in careset
28	Check sticky notes
29	Sign treatment plan
30	Give treatment plan to scheduler
31	Confirm pretesting results are within normal limits
32	Confirm there are medication orders for every cycle of the treatment
33	Confirm dose base on treatment plan is consistent with doses on orders
34	Confirm scans have been scheduled
35	Confirm patient has a scheduled appointment

Table 3.1: Normal flow process steps.

<i>Non-routine Situation Name</i>	
1	Patient's height/weight have never been measured in the building (Cancer Center)
2	Patient's height/weight are out-of-date
3	Patient's height/weight are not entered in EMR
4	There are not enough entries for patient's height/weight to judge stability of height/weight
5	Height/weight in the treatment plan, in the EMR, and/or in the patient's chart do not match
6	A diagnosis(es) on the clinical note is(are) missing from the treatment plan
7	A medication from the clinical note is not on the treatment plan
8	The treatment plan is not from a careset
9	MD has not provided reference supporting the non-standard careset
10	Orders (for meds or pre-meds) are missing from the EMR
11	Old orders are still in the EMR
12	There is no MD-to-RN note approving orders entered by a fellow MD
13	There is more than a 5% discrepancy between calculated doses and doses on order
14	Clinical notes in the EMR are missing
15	Scans have not been done
16	Labs have not been done

Table 3.2: Exceptional situations.

<i>Response to Non-routine Situation Name</i>	
1	Tell Medical Assistant (MA) or Clinic RN to measure height/weight on next patient visit
2	Call another facility (Breast Center) to inquire about patient's height/weight
3	Retrieve patient chart from medical records
4	Enter height/weight from patient chart into the EMR
5	Ask the person who entered the height/weight in the patient's chart to enter the height/weight into the EMR
6	Write a nursing order to measure height/weight on day of treatment
7	Write missing diagnosis information on the paper copy of treatment plan
8	Check if the MD provided reference literature for the non-standard treatment plan
9	Look for a literature reference for the non-standard treatment plan (drugs, cycle info)
10	Ask the pharmacist to look for a literature reference for the non-standard treatment plan (drugs, cycle info)
11	Contact MD to provide a literature reference for the treatment plan
12	Contact MD to resolve differences in dose base between the treatment plan and the orders
13	Contact MD to enter missing orders
14	Contact MD to discontinue orders
15	Contact MD about discrepancy in dose on orders
16	Contact MD (or MD and fellow) to review and/or revise fellow's order
17	Contact MD to notify about discrepancy in height/iweight between treatment plan and EMR
18	Contact Nurse Practitioner to enter missing orders
19	Check if clinical note explains different cycle on treatment plan
20	Check if clinical note explains different drug on treatment plan
21	Schedule an additional patient appointment
22	Check for an existing patient appointment that is before the next day of treatment
23	Obtain signed order from MD for scans
24	Tell MA that the patient needs labs during next visit (only if platinum based)
25	Tell MA that patient needs scans during next visit
26	Place reminder to self that plan is incomplete (make copy of treatment plan and note on copy, turn folder backwards in box, put note on electronic calendar, put a sticky note on the treatment plan)
27	Attach literature reference to chart
28	Hold treatment plan
29	Wait for information
30	Delete out-of-date pre-medication orders in the EMR
31	Change timing of pre-medications on orders in EMR (based on wrong timing)

Table 3.3: Responses to exceptional situations.

Events in gray signify that the event was *implicitly identified* using the elicitation method. An event was *implicitly identified* if it was not mentioned during the interviews, but was implied via another step that was mentioned during the interviews. For example, the step “retrieve patient chart from medical records” was not explicitly mentioned during the unstructured interviews, but the step “enter height/weight from patient chart into EMR” implied that the RN retrieved the patient chart from the patient’s medical record. Some normal flow steps and exceptional situations identified during the unstructured interviews were used in the questions we asked during the semi-structured interviews and so were considered to be implicitly identified in the semi-structured interviews, an exception being when the RN explicitly objected to the step or situation. During the open-ended prompt interviews, we prompted the RNs with exceptional situations specifically, so all exceptional situations identified using this method were implicitly identified.

Events shown with an “X” were *refuted* using an elicitation method. An event was *refuted* if during the semi-structured interviews, at least one RN stated that a given event elicited during the unstructured interviews is not performed at any time during the treatment plan review process. For example, the step “confirm labs have been done” was explicitly identified as a normal flow process step during the unstructured interviews by the senior RN, but was refuted by some of the RNs during the semi-structured interviews. In the next section, we discuss instances where the RNs disagreed with one another and instances where an RN’s responses were inconsistent across interviews and observations.

Events shown in white were *not addressed* via the corresponding elicitation method, meaning that they were not observed or were not mentioned during one of the interviews. For example, the exceptional situation “Old orders still in the EMR” was not addressed during the unstructured interviews but was explicitly identified during the observations. Having old orders in the EMR is clinically significant as a Clinic Reg-

istered Nurse may administer the wrong dose of a given medication or give another medication instead of or in addition to the intended medication.

3.1.1.4 Discussion

Unstructured interviews with an expert RN were a useful way to glean detailed knowledge about the structure of the process, providing information about 51 events. The unstructured interviews were labor-intensive as they included several iterations with the RN over the course of 6 months. Between the meetings, we had to carefully examine our meeting notes and recordings to extract all relevant information and to create a precise model of the treatment plan review process.

The observation and semi-structured interview data showed that even after the significant amount of time and effort spent during the unstructured interviews, the resulting elicited process information was still problematic. We discovered that the process model created from the unstructured interviews omitted certain events and included events later refuted by the RNs. Observations were a useful way to discover additional normal flow steps (10), exceptional situations (5), and responses to exceptional situations (7). Semi-structured interviews with prompts were a useful means to find additional normal flow steps that RNs perform (3) as well as steps that RNs perform in response to exceptional situations (5). Finally, semi-structured interviews using full process traces and the full process model did not provide information about new events, but were an essential means for the RNs to refute events defined via other elicitation methods.

The white spaces in Figures 3.5, 3.6, and 3.7 signify that a specific elicitation method did not identify a specific event. An elicitation method may not address a specific event or series of events for a variety of reasons. As the unstructured interviews occurred first, they did not address any events identified for the first time during the observations and semi-structured interviews. The observations only ad-

dressed events that explicitly occurred during the observations; situations that did not arise could not be observed. The open-ended prompt portion of the semi-structured interviews focused on sub-sections of the process, so was not designed to cover the entire process. Similarly, the complete process traces presented to the RNs during the semi-structured interviews were only a sample of all possible traces. Finally, the full process model presented to the RNs during the semi-structured interviews, while exhaustive, did not include additional events elicited via the observations or via the semi-structured interviews using open-ended prompts and full process traces.

Some of the inconsistencies between the unstructured interviews and the other methods likely occurred because we initially interviewed only one RN, but we subsequently observed and conducted semi-structured interviews with three RNs. A single RN may conduct the process differently than the other RNs, or may not mention process events that seem obvious to her. Yet, holding time-intensive interviews with more than one individual may not yield enough additional information to justify the extra time requirements for the interviewers and participants.

In a few instances concerning the confirmation or refutation of a particular event, individual RNs were inconsistent within their individual responses across the observations and interviews or the RNs disagreed with one another about a particular event. For instance, step 24 on the normal flow (*confirm labs have been done*) was refuted by RN 1 and RN 2, but RN 3 noted that this step is done if the medications are platinum-based. In step 4 of the responses to exceptional situations (*enter height/weight from the patient chart into the EMR*), RN 1 and RN 2 stated that this step either did not happen or that the original individual who entered the height/weight in the paper chart—typically the medical assistant—would transcribe the information into the EMR. In general, we noted a step as refuted if at least one RN refuted the step, knowing that the refuted steps may require further scrutiny.

Based on our findings, it appears to be important to use multiple methods when understanding how individuals complete a complex process, such as chemotherapy treatment plan review; each subsequent process elicitation method provided new information about the process. Observations added 22 new events not discovered via the unstructured interviews. Semi-structured interviews using open-ended prompts added 7 new events not discovered via the unstructured interviews or observations, but did not add new exceptional situations. Semi-structured interviews using the full process model produced only 1 new event. Finally, the semi-structured interviews using the full process traces and full process model identified 6 refuted events.

3.1.1.5 Threats to Validity

There are several limitations inherent in this study, which can guide the design of future studies aiming to identify how best to elicit complex processes. We purposefully ordered our elicitation methods to glean the most new information from each subsequent elicitation method and to minimize the impact of a given elicitation method on the results of subsequent elicitation methods. We first conducted unstructured interviews so as not to prompt the senior RN with any preconceived notions about the process. With the second group of RNs, we conducted observations followed by semi-structured interviews to not influence how the RNs acted during the observations. Additionally, we developed the semi-structured interview guides solely from the unstructured interview findings, and these were not updated based on the observational data or between interviews. We did so to allow the RNs freedom to refute events without being biased as to what behaviors other RNs exhibited during the observations. These methods could easily be reordered, or the materials could be continuously updated based on findings to-date. Each of these changes may produce different or additional events, or may omit steps discovered using our order.

In this study, we focused on the viability of using the proposed elicitation methods to identify normal flow process steps, exceptional situations, and the identification of and responses to the exceptional situations. We did not explicitly analyze how well each elicitation method helped to understand the order in which events happen during the process. This study addressed one type of worker completing a single type of process. Additionally, one RN participated in the unstructured interviews, and three different RNs all completed the observations and semi-structured interviews. The relative strengths and weaknesses of the elicitation methods may differ when applied to a different process, depending on whether all RNs participate in all of the elicitation methods, or different RNs participate in one elicitation method each.

3.1.1.6 Summary

Through this work, we describe our application of five process elicitation methods to the complex, safety-critical chemotherapy treatment plan review process: unstructured interviews, observations, and three types of semi-structured interviews. We also detail our quantitative and qualitative evaluation of the relative strengths and weaknesses of these elicitation methods. By using the aforementioned five process elicitation methods, we identified a large number of events (82) involved in the process by which RNs review chemotherapy treatment plans. These elicitation methods allowed us to determine how many and which process components were associated with the normal process flow (35), which were exceptional situations (16), and which were identification of and responses to exceptional situations (31). Each of the five elicitation methods contributed uniquely to our understanding of the process by which RNs review chemotherapy treatment plans. The research detailed in this section indicates that to create accurate descriptions of complex processes, a combination of elicitation methods should be used.

3.2 Process Modeling

This section describes work related facilitating process modeling and also discusses our overall experience with creating models of complex HIPs. The section starts with a discussion of exception handling patterns that we identified and codified to facilitate the modeling of exceptional situations. Then follows a study that compares two process model representations—a diagrammatic and a textual representation—in terms of their ability to facilitate process understanding. The section concludes with a brief discussion of our experience with creating process models with Little-JIL and with validating these models.

3.2.1 Exception Handling Patterns

Exceptional situations occur in any but the most trivial HIPs. People or other resources might be unavailable when they are needed, the activities of process performers might be incorrect or inappropriate, or deadlines might not be met. In each of these cases, additional action is required beyond the nominal process execution. We refer to the recognition that a problem has occurred during a process execution as an *exception* and to the non-nominal activities that are taken to address such a problem as *exception handling*. In some cases, exceptions might not be particularly unusual or surprising, such as when trying to book a flight the day before a trip and finding out that there are no seats on a flight or hotel rooms available.

In real-world HIPs, the number and complexity of such exceptional situations is typically large and the need to assure that they are appropriately handled may be quite important. Furthermore, exceptional situations are often the source of errors [88], perhaps due to the complexity of such situations and/or the lack of appropriate training, as training often focuses mostly on nominal process executions. To enable the process guidance approaches discussed in this work to reduce the number of errors or catch errors before harm is done in exceptional situations, a process model

needs to specify in detail the recommended exception handling behaviors of real-world HIPs. Modeling exception handling, however, is difficult. A process modeler needs to consider the exceptions that might arise and during which activities they might arise, how these exceptions should be handled, and how the process should proceed afterwards. Because of the difficulty of modeling exceptional situations, such situations are often modeled incorrectly or even not modeled at all.

To facilitate the modeling of exceptional situations that could arise in HIPs, we propose to use a set of *exception handling patterns* [86]. We believe that exception handling patterns can improve the writing of models of HIPs the same way object-oriented design patterns have improved the writing of object-oriented software. Exception handling patterns can raise the level of abstraction at which process modelers think about and discuss exceptional situations. If such patterns are appropriately codified, process modelers can reuse modeling idioms when they recognize that a HIPs exhibits a certain exception handling pattern, thus avoiding to need to model the exceptional behavior from first principles, which could be a time-consuming and error-prone task.

The exception handling patterns we identified are based on significant experience in modeling HIPs from a variety of domains, including health care [32, 109], labor-management dispute resolution [38], software development [28], and elections [99, 110, 120]. Members of our research team have defined processes in each of these domains, and some of the processes have been non-trivial in size, consisting of hundreds of steps. In the course of defining these processes, our team members have recognized strong similarities among the ways in which the domain experts have described how they deal with exceptional situations. This led to careful attempts to characterize and categorize these different approaches to exception handling.

We grouped the exception handling patterns we identified into three high-level categories:

- *Trying other alternatives.* This category contains patterns for presenting alternative means to perform the same activity. Thus, for example, if one hotel has no vacancies, we will try other hotels, or stay at the home of a friend or relative.
- *Inserting behavior.* This category contains patterns for inserting additional activities after an exceptional situation has been recognized and before returning to the normative process. Thus, for example, if the passenger's name is wrong on the itinerary after purchasing a plane ticket, then the passenger may need to perform the extra work of contacting the airline to fix that problem.
- *Canceling behavior.* This category contains patterns for aborting the current processing when the process is not allowed to continue after an exceptional situation has been recognized. If there is no date and time at which all key people can be present at a meeting, then the whole trip might be cancelled, possibly requiring the cancellation of various travel arrangements.

Following the style introduced in the classic book on design patterns for object-oriented software [60], for each pattern, we provide:

- Name—the name of the pattern
- Intent—what recurring behavior the pattern captures
- Applicability—in what situations the pattern should be used
- Structure—the general structure of the pattern expressed in two different process definition formalisms
- Participants—the roles played by different parts of the process that contribute to the pattern
- Example—an example from a real-world process that exhibits the pattern

- Variations—small changes that can be made in the application of a pattern to get slightly different effects

In the remainder of this section, we present one pattern from each of the above three categories and use the Little-JIL process modeling language [27] to describe pattern structure and example usage. The rest of the exception handling patterns are described in detail in [86], where also examples of each pattern are shown in two other process modeling notations—UML Activity Diagrams [97] and BPMN [96]—and Little-JIL, UML Activity Diagrams, and BPMN are compared in terms of their ability to support the identified exception handling patterns.

After presenting one pattern from each of the three categories, we describe an evaluation where we studied the occurrence of all the patterns from [86] in two process models from the healthcare and the online dispute resolution domains.

3.2.1.1 Selected Patterns

3.2.1.1.1 Ordered Alternatives

Pattern Name: Ordered Alternatives

Intent: There are multiple ways to accomplish a task and there is a fixed order in which the alternatives should be tried. Provision must be made for the possibility that no alternatives will be successful.

Applicability: This pattern is applicable when there is a preferred order among the alternatives that should be tried in order to execute a task.

Structure: The Little-JIL diagram in Figure 3.8 depicts the structure of the Ordered Alternatives pattern. As discussed in section 2.3, processes are represented in Little-JIL as hierarchical decompositions of steps. Here, we see the step named *Task* with three substeps, each defining one way to complete the task. The icon at the left end of the black step bar of *Task* indicates that this is a *Try* step. The semantics of the Little-JIL Try step match the definition of this pattern quite closely, as the

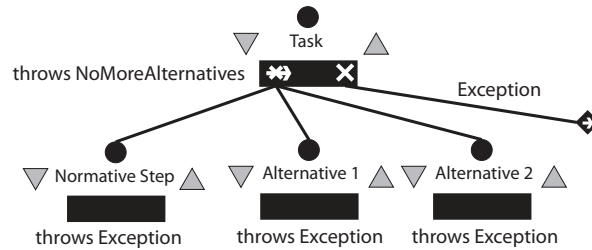


Figure 3.8: Structure of the Ordered Alternatives Pattern in Little-JIL.

Try step semantics specify that the step’s children represent alternatives that are to be tried in order from left to right. If an alternative succeeds, the parent step is completed and no more alternatives are offered. If execution of an alternative throws an exception, the exception is handled by the handler attached to the Try step by the rightmost edge. The icon associated with the exception handler indicates that the Try step should continue with the next alternative. This continues until one of the alternative substeps succeeds. If none of the substeps succeeds, a special exception, called *NoMoreAlternatives*, is thrown. This exception must be handled by an ancestor of the Try step. Indicating that all alternatives have failed is part of the pattern, but the handling of that exception must take place in the context in which the pattern is used rather than as part of the pattern.

Participants: This pattern has three types of participants: the menu, the alternatives, and the continuer. The **menu** is the portion of the process that organizes the alternatives into an order. The **alternatives** are the various ways in which the desired task can be carried out. While the figures show three alternatives, there is no limit to the number of alternatives that could be used in this pattern. Each alternative, except possibly the last, must have the potential to throw an exception that causes consideration of the next alternative. The **continuer** is the exception handler that indicates the process should continue to the next alternative.

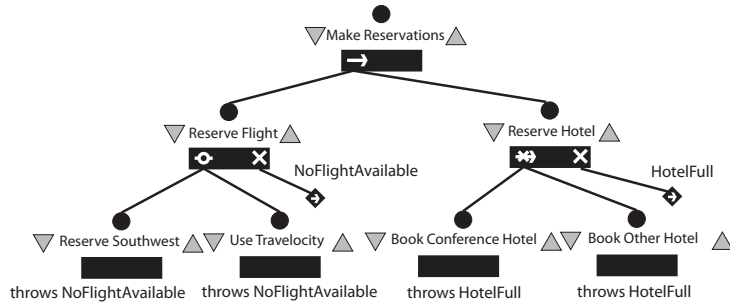


Figure 3.9: Using the Ordered and Unordered Alternative Patterns when Planning a Trip.

Sample Code and Usage: Figure 3.9 shows the use of the Ordered Alternatives pattern in a Little-JIL process to plan travel to attend a conference. This pattern can be seen in the *Reserve hotel* step. Here, the process requires first trying to get a reservation at the conference hotel before considering other hotels. If the conference hotel is full, the *HotelFull* exception is thrown. This is handled by causing the *Book other hotel* step to be attempted next.

Variations: One variation of this pattern uses boolean conditions after an alternative is tried, rather than expecting the alternative to throw an exception. If the condition evaluates to true, it means the alternative has succeeded. If the condition evaluates to false it means the alternative failed and the process should proceed to the next alternative. The tradeoff here is essentially the same as we see in procedural programming when deciding whether a function should return a status value to indicate whether it has succeeded or it should throw an exception.

If the conditions under which an alternative will succeed are known in advance, the alternatives are better represented with a construct similar to an if-else construct in a traditional programming language. This allows the orders to be specified while avoiding the need for exception handling. This is the Exclusive Choice pattern presented as a control flow pattern by van der Aalst, et al. [125].

3.2.1.1.2 Deferred Fixing

Pattern Name: Deferred Fixing

Intent: When an exceptional situation arises during a process execution, action must be taken to record that situation and to possibly address the situation either partially or temporarily, because addressing the situation fully is either not immediately possible or not necessary. Later in the process, an additional action needs to be taken to complete the recovery from the condition that caused the occurrence of the exceptional situation.

Applicability: This pattern is useful in preventing the process from coming to a halt even though the potentially disruptive effects of an unusual, yet predictable, situation cannot be addressed completely. The pattern is useful in those cases where addressing the problem definitively is possible only when more time or information becomes available, where the need for further work to complete the handling of the exception can be captured in the state of the process, and where temporary measures can enable the process to proceed to the point where such additional time and information have become available.

Structure: Figure 3.10 is a Little-JIL depiction of the structure of this pattern. In Figure 3.10 an exception is thrown during the execution of *Substep 1*. The exception is handled by *Do temporary fix*, an exception handler that makes some expedient temporary adjustment, records the need for a more complete fix, and then returns to the nominal process flow, as indicated by the **continue** handler. However, at some later stage of the process, an additional step (or collection of steps), represented by the step *Some step*, must be executed to either complete the handling of the exceptional condition or check that the exceptional condition has been already handled. This check is made by an **edge predicate**, denoted by the parenthetical condition, prior to executing *Some step*. The edge predicate checks the process state to determine if the fix is required. Note that the dotted line notation is not Little-JIL syntax, but is

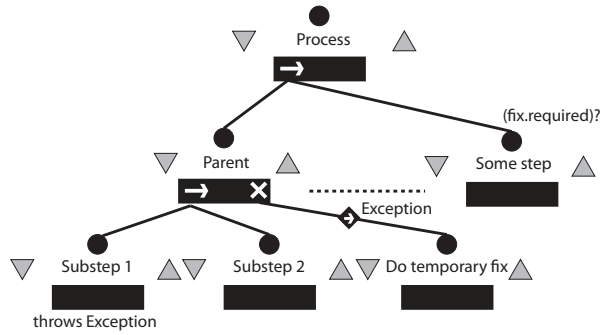


Figure 3.10: The Structure of the Deferred Fixing Pattern in Little-JIL.

intended just to note that an arbitrary amount of work may occur between when the temporary fix takes place and the fix is completed.

Participants: There are three participants in the Deferred Fixing pattern: the detector of the anomaly, the logger/patcher, and the fixer. The **anomaly detector** is the portion of the process that recognizes that a problem has arisen and throws an exception to represent the recognition of that problem. The **logger/patcher** is responsible for recording the anomaly and possibly doing a temporary fix. In the Deferred Fixing pattern, the logger/patcher is the exception handler. The **fixer** is the later step that examines the log and completes the handling of the exceptional situation. Notice that the fixer does not use an exception handling mechanism, yet is a key participant in resolving the anomaly.

Sample Code and Usage: Figure 3.11 shows an example of the deferred fixing pattern. Here a traveler has successfully reserved a flight but the website that is used to select a seat is unavailable. *Reserve flight* throws the *SeatSelectionWebsiteIsDown* exception. This is handled by making a note to select seats later and then continuing with reserving the hotel and car. At some later point in the process, a test is made to see if the seats have been selected. If not, the *Select plane seats* step is executed.

3.2.1.1.3 Compensate

Pattern Name: Compensate

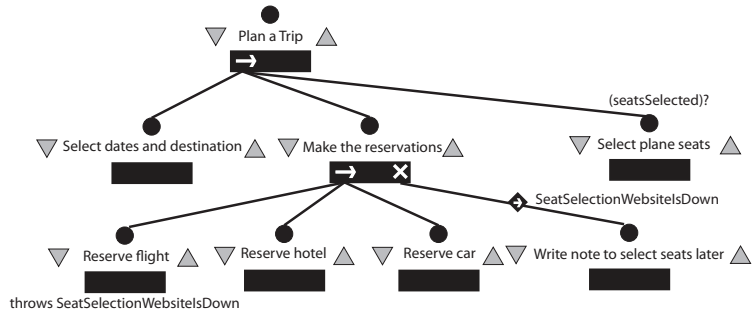


Figure 3.11: Using the Deferred Fixing Pattern to Complete Seat Selection at a Later Time.

Intent: When canceling an activity, it is often necessary to undo work that has already been completed. This pattern addresses the need to determine what work must be undone and to then execute the compensating action(s) needed to undo that work.

Applicability: This pattern is particularly useful in contexts in which it is not possible to know at the outset that a task will succeed, or the results produced by the task will prove ultimately to be acceptable. Because of this, the process must incorporate mechanisms for undoing the part(s) of the task that did complete and/or replacing the outputs that proved to be unacceptable. In some cases, the state of the process after compensation may appear the same as if the failed activities never occurred. Often, however, there will be a record that the activity occurred but the compensating activity nullifies the effect of the original activity, as when a credit card credit compensates for a credit card charge.

Structure: Figure 3.12 shows the structure of the Compensate pattern in Little-JIL. *Step 1* and *Step 2* can be done in any order, including in parallel. If *Step 2* fails but *Step 1* completes, an exception handler is used to compensate for the effects of *Step 1*. Notice that as part of the exception handling, it needs to explicitly checked if *Step 1* has been completed. This is done because Little-JIL does not have a

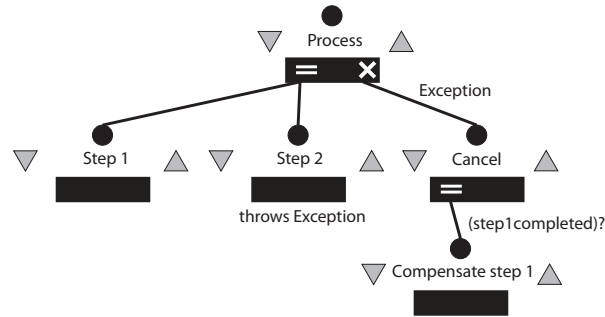


Figure 3.12: The Structure of the Compensate Pattern in Little-JIL.

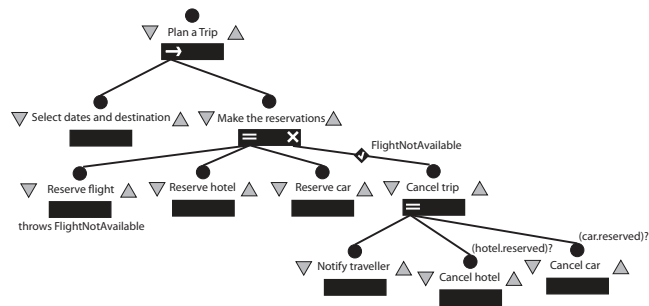


Figure 3.13: Using the Compensate Pattern to Cancel a Trip.

compensation construct, unlike BPMN, for example. This makes the Compensation pattern somewhat awkward to express in Little-JIL.

Participants: The participants in this pattern are the Actor, the Canceler and the Compensator. The **Actor** performs some task that the **Canceler** later wants to undo. The undo is accomplished by the **Compensator**, which understands the work that was completed and how to undo it.

Sample Code and Usage: Figure 3.13 show a variation of the Little-JIL process of planning a trip. In this example, the reservations can be made in any order. If we fail to get a flight, we cancel the trip. This will require canceling hotel and car reservations if they have already been made.

Variations: Compensation can be combined with other patterns. In particular, any time that an activity fails with an exception, it may be necessary to undo some

work that has been completed. Thus, compensation could form part of the exception handling used in any of the preceding patterns.

Another variation is that it is not always necessary to include in the process the tests to determine what work needs to be compensated, even in the absence of a compensation construct like BPMN has. This is the case if the location of the exception handler is sufficient to determine what work has been completed, as would be the case if the compensation was in the context of sequential tasks rather than concurrent tasks.

3.2.1.2 Evaluation of Exception Handling Patterns

To evaluate the catalogue of exception handling patterns, we examined several existing models of real-world processes. The main question in which we were interested was how well the exceptional situations encountered in the real world can be specified by the exception handling patterns from the catalogue. Associated goals were to determine the relative frequencies of the occurrence of the various patterns in the models of the real world processes and to gain some intuition about the amount of effort entailed in representing the handling of exceptions in real-world processes. Here, we present results obtained from studying Little-JIL models of two processes from different domains—the medical and the digital government domains.

The chemotherapy process model. The Little-JIL model of a chemotherapy process was the largest model of a real-world process that we had access to. At the time this evaluating was performed, the chemotherapy process model consisted of 467 Little-JIL steps, 283 of which were Little-JIL leaf steps. The model captures the process of preparation for and administration of outpatient breast cancer chemotherapy. In a period of approximately a year, computer scientists elicited the process from medical professionals working at the a cancer center in Western Massachusetts and created a Little-JIL model of that process. Since many medical errors that lead to

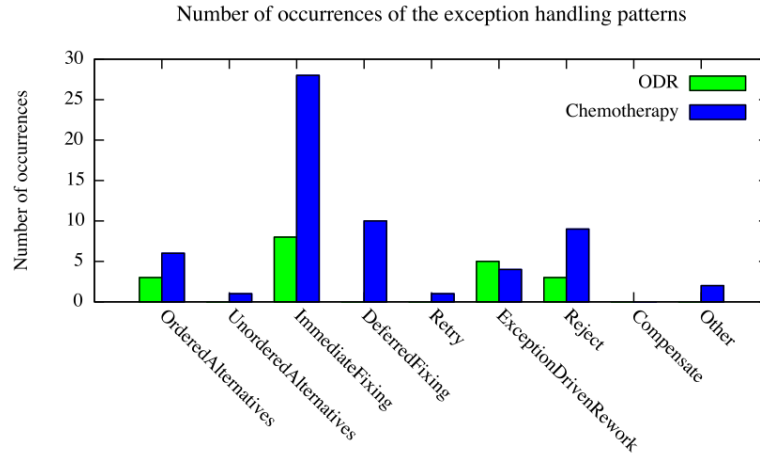


Figure 3.14: The number of occurrences of each pattern in the chemotherapy and ODR process models.

patient safety hazards occur in atypical, exceptional situations, a great deal of effort was spent in identifying such exceptional situations and specifying them precisely in Little-JIL. In particular, 207 of all 467 steps (or about 44%) were used to represent these exceptional situations and their handling in this process model.

The resulting process model captured 59 exceptional situations specifying precisely the types of the exceptions, where in the process they occurred, the actions undertaken to remedy the exceptions and the resumption of nominal flow after handling the exception. We studied these 59 exceptional situations and tried to determine if and how they related to the patterns in the catalogue. Figure 3.14 presents the results of this investigation. Most of the exceptional situations in the chemotherapy process that were captured by the Little-JIL model seemed to be instances of the patterns in the catalogue. The two exceptional situations that did not easily match any pattern in the catalogue actually seemed to combine elements of both the inserting and the canceling behaviors.

The ODR process model. The second process model that we examined was a Little-JIL model of an online dispute resolution (ODR) process that was used to drive a web-based application for dispute resolution. The model captures a dispute

resolution process used by the National Mediation Board (NMB) to resolve conflicts between two different parties. It was elicited from a mediator working for the NMB. At the time this evaluation was performed, the ODR process model consisted of 209 Little-JIL steps (69 leaf steps), 108 of which (or about 52%) were part of the exceptional flow.

The ODR Little-JIL process model specified 19 exceptional situations. Figure 3.14 shows how many of them are instances of each of the different exception handling patterns in the catalogue.

Observations. Almost all of the exception handling situations specified in the process models we studied (76 out of the 78, or 97.5%) were instances of the patterns from the catalogue. This was encouraging, suggesting that the identified patterns cover a significant portion of the exceptional situations that arise in real-world HIPs, and underscoring the potential value for enunciating a set of patterns that could be useful in guiding the efforts of people who are attempting to define real-world processes. We hope that defining these process patterns might cause process definers to be more aware of the presence of exceptions, and more comfortable in incorporating them into process models.

The variation in levels of usage of the different patterns was also interesting. We note that the Immediate Fixing pattern occurred most frequently in these examples, with 36 uses. This seems to be reasonable as agents are perhaps most inclined to try to remedy exceptional situations immediately in the real world. We are less certain of the reasons for the relative scarcity of instances of some of the other patterns. It seems indeed possible that Unordered Alternatives are less prevalent because people innately have preconceived preferences among alternatives, but this is not immediately obvious to us. It is far less obvious why there are so few instances of the Compensation pattern, as this seems to be a relatively common reaction to exceptional circumstances. Here we are concerned that the relative scarcity of this pattern in our examples might

be due to possible bias in the examples themselves. We note that these examples were written in Little-JIL, a notation in which compensation is relatively difficult to specify, suggesting that the facilities of a language might have a noticeable effect upon the process features that are incorporated into a process model. Furthermore, in Little-JIL, it is possible to represent the idea of compensation by not explicitly fleshing out the compensation activities in the Little-JIL step tree, but relying on the agent(s) responsible for the handling of an exception to perform the corresponding compensation. If a compensation is expressed this way in a process model, then the compensate structure in Figure 3.12 will not be observed in the corresponding process model.

Another interesting observation concerns the nesting of exception handling patterns within each other. We said that an instance of pattern A is nested within an instance of pattern B, if the instance of A occurs entirely “inside” the exception handler of the instance of B. 10 of the pattern instances (7 from the chemotherapy and 3 from the ODR case studies) were nested within other patterns. All the nested patterns, except one, were instances of the Ordered Alternatives pattern. This seems to suggest that when people are dealing with an exceptional situation, they often have a prioritized list of tasks to try to fix the problem.

Threats to validity. The results presented above may have been affected by unintentional personal bias in two ways. First, the decision about whether a certain behavior is exceptional or not and consequently whether this behavior was considered in the evaluation of the patterns, may have been biased. The chemotherapy process model was created by the author of this thesis and deciding which behaviors were part of the nominal flow and which behaviors were part of the exceptional flow may have been influenced by the author’s awareness of the exception handling patterns work. To attempt to reduce possible bias in categorization of the exceptions in the chemotherapy process, we also consulted the medical professionals from whom the

process was elicited. We asked which behaviors they considered nominal and which behaviors they considered exceptional. Their responses did confirm our selection and categorization of the exceptions in that process.

The second process model (the ODR process model) was created by a process model developer who was unaware of the exception handling patterns when creating the model. Thus, the frequency and distribution of instances of the process patterns should not have been influenced by the work on the patterns. This may have affected the distribution of types of exception patterns that were observed, however.

The second source of bias is associated with classifying the exceptional behaviors from the process models as instances of the patterns from the catalogue. The precise structure of the patterns was an extremely useful guide in making these classification decisions, but on a small number of occasions personal judgment was involved in deciding which pattern a particular exceptional situation used.

Evaluation of processes in other notations. We made an attempt to examine process models in UML and BPMN. The process models that we had access to, however, were focused primarily on nominal flow and lacked thorough specification of exceptional flow. For example, we looked at the model-based Simulation, Verification & Testing (SV&T) process repository created as a part of the MODELPLEX project [4]. The processes were formalized in SPEM2.0 [98] using the EPF Tool [3]. The SV&T process repository contained 47 process diagrams, some of which were at a higher level of abstraction and others were lower level decompositions of the higher level diagrams. There were only a few occasions on which some process behavior modeled with decision nodes could be thought of as exceptional behavior. There seemed to be six instances of the Rework, one instance of Reject, and one instance of Immediate Fixing in the processes in this repository. The process models were, however, specified at a very high level (to make them general) and omitted detail that could allow us to claim with certainty that specified behavior is an instance of

a given exception handling pattern. We were unable to gain access to any sizable BPMN process repository for this work. Thus, our evaluation focused on two Little-JIL process models that specified a significant amount of exceptional behaviors and at the same time were detailed and precise enough to allow us to categorize these exceptional behaviors as instances of the presented patterns.

3.2.2 Comparing the Little-JIL Diagrammatic and Narrative Representations

To develop accurate models of a human-intensive process, it is usually important that various stakeholders carefully review, evaluate, correct, and propose improvements to these models. Some stakeholders (e.g., domain experts, process performers, user interface designers, even some programmers), however, might not be experts in process modeling. Consequently, such stakeholders may not have the skills to understand the process models except at a relatively superficial level. We have seen this problem in our own work on modeling medical procedures. Medical professionals may be able to point out glaring misrepresentations, but are not sufficiently versed in modeling to fully understand the implications, for example, of complex control flow such as the handling of exceptional situations and concurrent execution.

To help stakeholders with diverse backgrounds understand complex processes, we have used two different process representations in our work—the Little-JIL visual, *diagrammatic* representation (described in section 2.3.1 and the Little-JIL textual, *narrative* representation (described in section 2.3.2). Each of these representations, however, has its own strengths and weaknesses to facilitate process understanding. Knowing these strengths and weaknesses could help process modelers what notation to use when interacting with other process stakeholders, depending on the kinds of process information to be discussed and the background of the stakeholders who will participate in such discussions. To explore the strengths and weaknesses of the

Little-JIL diagrammatic and narrative representations in terms of facilitating process understanding, we conducted a user study.

3.2.2.1 User Study

3.2.2.1.1 Study Design. We created two process models and for each we created the corresponding Little-JIL diagrammatic and narrative representations. The two process models were of equal complexity, in terms of their size and the language features they employed. In fact, one model was created by essentially “reshuffling” the steps in the other one. We used colors for step names (e.g., *perform blue*) as opposed to names of real activities (e.g., *drive to work*) to not bias the results based on a subject’s experience with a domain. The process model representations are included in Appendix B.2.

For each process model, we also created a questionnaire consisting of nine questions testing the understanding of the process. The two questionnaires were of equal complexity since they were almost identical except for variations in step names. We also included a final questionnaire with five open-ended questions asking for general feedback about the two process representations. The questionnaires are shown in Appendix B.2.

Half of the subjects were presented with the narrative representation of process model 1 followed by the diagrammatic representation of process model 2; the other half of the subjects were presented with the diagrammatic representation of process model 1 followed by the narrative representation of process model 2. We presented the two process representations in different orders to the two groups of subjects, because even though the understanding of process model 1 should not have helped with understanding process model 2, the subjects might have improved their understanding of fundamental process concepts (such as various kinds of control flow, exception handling and artifact use). The subjects were assigned randomly to one of these two

groups. Subjects were provided with a 10-15 minutes training in their first process model representations (the narrative or the diagrammatic representation). Then, the subjects were presented with a process model in that representation and with the questionnaire for that process model. After the subjects answered the questionnaire (there was no time limit imposed), that subjects were provided with a 10-15 minutes training in the other process model representation. Then, the subjects were presented with the process model for their second process represented in their second representation and with the questionnaire for that process model. After the subjects answered the second questionnaire (again, no time limit was imposed), the subjects were presented with the final questionnaire. Subjects were given laptops and they viewed the narrative representation in a browser to be able to follow the hyperlinks in the narrative.

The training materials used in the study are shown in Appendix B.1. During a training period, the subjects were given an example process model in either the diagrammatic or the narrative process representation and that representation was explained to them using the corresponding training script. The training script was given to the study subjects at the beginning of each training and they were allowed to keep that script and use it as a reference during the entire study session. The subjects were allowed and encouraged to ask any questions that they might have during the training. After the training, the subjects were allowed to ask only about clarifications on the questionnaires.

We performed the above study with two sets of subjects—computer science students and nursing students. We decided to study two different sets to see whether the subject’s background is related to their ability to understand a process via each of the two representations. 16 computer science students participated in the study: 6 graduate and 10 undergraduate. 17 undergraduate nursing students participated in the study. We tried to recruit as many subjects as possible and to keep the two

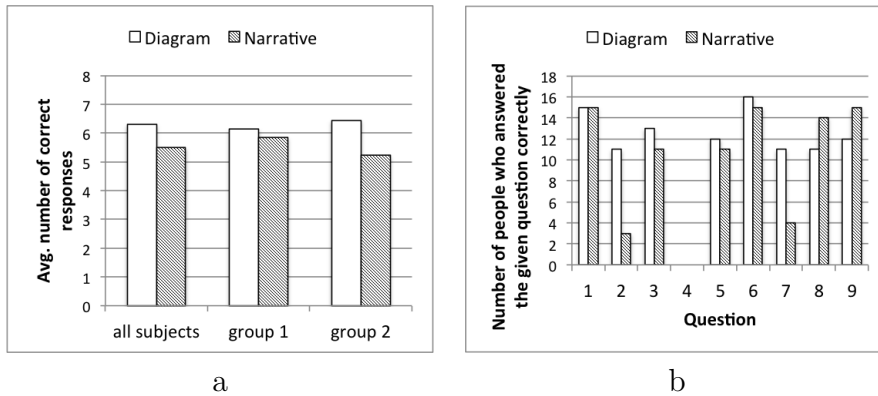


Figure 3.15: Study results—computer science students.

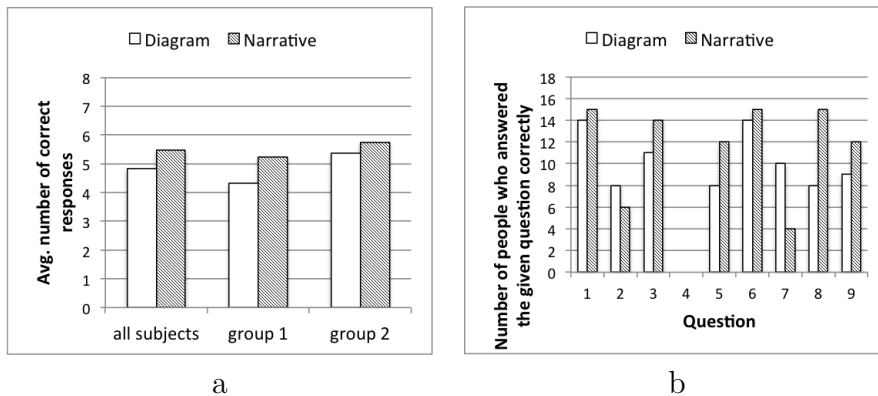


Figure 3.16: Study results—nursing students.

sets of subjects of similar sizes. Students who had experience with the Little-JIL diagrammatic or narrative representations were not allowed to participate.

3.2.2.1.2 Results and Discussion. On average, computer science students answered correctly a higher number of questions—6.3 vs. 5.5 (out of 8⁴)—when they were using the diagrams (Figure 3.15a), and this difference is statistically significant ($p = 0.037$)⁵. The same tendency was observed when examining the two groups of

⁴There were 9 questions in the original questionnaire, but question 4 was excluded from our analysis because it covered the difficult concept of non-deterministic choice and the training we provided might not have been sufficient to answer this question. It was answered correctly by only one person when they were using the narrative.

⁵A paired t-test was used to determine statistical significance.

computer science students separately (Figure 3.15a), where the difference for group 2 was statistically significant ($p = 0.047$) too. Graduate computer science students answered more questions correctly than undergraduate computer science students for both of the notations (on average, graduate students answered 6.5 and 5.8 questions correctly using the diagrams and the narrative respectively, whereas undergraduates answered on average 6.2 and 5.3 questions correctly), but, again, the same tendency was observed—a higher number of questions was answered correctly when the diagrams were used (the differences were not statistically significant). These results might suggest that computer science students are more comfortable with the diagrammatic notation, perhaps because of their experience with programming and familiarity with control flow and data flow constructs.

Nursing students, on the other hand, answered a higher number of questions—5.47 vs. 4.82 (out of 8)—when they were using the narrative representation (Figure 3.16a), but this difference is not statistically significant ($p = 0.238$). The same tendency was observed when examining the two groups of nursing students separately (Figure 3.16a), but again the difference was not statistically significant. The nursing students who saw their first process model in the narrative representation (students in group 2) answered a higher number of questions on their first model than the nursing students who saw their first process model in the diagrammatic notation (students in group 1)—5.75 vs. 4.33—and the statistical significance of that difference ($p = 0.066^6$) was between the significance levels of $p = 0.05$ and $p = 0.10$. This might suggest that if only one process representation is to be used, then nursing students (or perhaps even professionals without computer science background, in general) should be presented with a narrative process representation.

⁶A paired t-test could not be used in this case, so a t-test that assumes the two samples have unequal variance had to be used.

Figures 3.15b and 3.16b show the break down of correct answers per question for computer science and nursing students respectively⁷. Computer science students gave statistically significantly more correct answers for questions 2 and 7 when using the diagrammatic representation than when using the narrative representation ($p = 0.002$ and $p = 0.004$ respectively). Computer science students gave more correct answers to question 9 using the narrative representation than when using the diagrammatic representation ($p = 0.083$, between the significance levels of $p = 0.05$ and $p = 0.10$). The results for questions 2, 7, and 9 were similar when computer science undergraduate and graduate students were considered separately, but only the difference in correct answers for question 2 was statistically significant for both undergraduates and graduates and the difference for question 7 for undergraduates.

Nursing students gave statistically significantly more correct answers to question 7 using the diagrammatic representation ($p = 0.009$) and statistically significantly more correct answers to question 8 using the narrative representation ($p = 0.014$). Nursing students also gave more correct answers to question 3 using the narrative than when using the diagrammatic representation ($p = 0.083$, between the significance levels of $p = 0.05$ and $p = 0.10$).

Question 2 was about prerequisites, 3 about parallel execution and 7 about the number of times a step can be executed. Questions 8 and 9 were about artifacts. The above results suggest that the two process representations complement each other in terms of facilitating the understanding of different kinds of process information and also that each representation makes different kinds of process information more accessible to people with different backgrounds.

⁷Question 4 was excluded from our analysis because it covered the difficult concept of non-deterministic choice and the training we provided might not have been sufficient to answer this question. It was answered correctly by only one person when they were using the narrative.

10 computer science students (37.5%) found the diagrams easier to understand, the other 6 preferred the narrative. 8 nursing students found the diagrams easier to understand, the other 9 preferred the narrative. The answers to the open-ended, qualitative questions also indicated that the order of step execution and artifacts were easier to understand with the narrative, whereas the number of times a step needs to be performed was easier to understand with the diagrams. Several people mentioned that the narrative was easier to understand at first with little training.

3.2.2.1.3 Threats to validity. The sample size was small and subjects from only two areas of expertise participated in the study. Thus, the results may not apply to stakeholders with different backgrounds and education, or even stakeholders with backgrounds and education similar to the ones of the study subjects.

The quality of the training might have affected the results of the study. For instance, if more time is spent on training, the quality of the training and the resulting understanding of the corresponding process representation could be improved. Subjects could be given multiple training examples or the subjects could be event be tested and the test results discussed to reaffirm the understanding of the process representations and correct any misunderstandings.

On the other hand, during the study, the subjects received focused training about how each of the two process representations expressed process information that the subsequent questionnaires asked about. In a real-world situation, the amount of time between training and a subsequent use of one of the process representations might be much longer and the training would most likely be more general.

Due to time constraints on the study sessions, the two process models used were relatively small (19 steps each). Realistic process models, especially models of complex human-intensive processes, however, could be larger. It is not clear whether the results obtained in the study will apply to such larger models.

3.2.3 Overall Experience with Process Modeling

3.2.3.1 Experience with Using Little-JIL

The well-defined formal semantics of Little-JIL and its diagrammatic and narrative notations have been useful in our work on eliciting and modeling processes from several domains, and in particular processes from the highly-complex medical domain. During process elicitation, Little-JIL’s rich semantic features, such as support for exception handling and concurrency, encouraged process modelers to ask questions about corresponding aspects of processes [14]. This helped elucidate process aspects that are important but are often overlooked during open, unstructured interviews. In fact, many of the difficulties we encountered in creating accurate process models arose because initial versions based on open, unstructured interviews were too restrictive or did not adequately capture all important process behaviors. Initial versions did not contain adequate description of exception handling as domain experts often focus on normal process executions during elicitation interviews. Prompted by Little-JIL’s support for exception handling, process modelers asked questions such as “What do you do if this check fails?” and “How do you resume the normal execution of the process (if at all), after you have dealt with the failure of the check?”.

When asked to describe a complex procedure, domain experts often presented a long list of activities, saying that the activities have to be done in sequence. It was often the case, however, that sequential ordering was too restrictive and subsequent questions by process modelers (stimulated by Little-JIL’s support for various activity orderings) revealed that some subsets of activities can be done in any order, including in parallel (especially when multiple process performers were involved). In other cases, domain experts would say that some activities could be done in any order (including in parallel), but subsequent questions (stimulated by Little-JIL’s support for synchronization) indicated that there is some partial order between the activities.

The need for precision in Little-JIL process models also proved to be helpful during process elicitation and modeling. Having to precisely decompose a step into its substeps, for example, led to identifying problems with existing terminology and clarifying this terminology, resulting in better understanding of the process, more accurate models, and even improvements to the process. In a chemotherapy case study, we found that terms like “verify”, “confirm”, “check”, “match”, and “consistent” were used loosely [32]. The same word used at different times or in different contexts often had different meanings, even when used by the same individual. Since many of the critical errors that may occur in a process like chemotherapy may arise from neglecting small details (like not checking to see if the patient height or weight measurements on which the chemotherapy dose is based are sufficiently up-to-date while “verifying” the doses), we had to develop a precise naming template and a glossary that disambiguated the use of different terms. This led to not only a better understanding of the process and more accurate process models, but also the clarified terminology was rapidly adopted for clinical use and contributed to improved accuracy of staff communication [93].

The diagrammatic notation of Little-JIL also seemed useful for process elicitation and modeling. The medical professionals who participated in several medical process case studies [14] did not have a computer science background and were initially leery about having to learn a process modeling language. Having a diagrammatic representation that is relatively easy to understand seemed to help considerably. The ability of that notation to also provide restricted views of the process model also seemed to help, since low-level details about the process model, such as the artifacts that needed to be provided to process steps, could be suppressed until the higher-level views of the major steps and their decompositions were agreed upon.

Having a natural language representation of the process was also helpful, since it further lowered the technology hurdle for the medical professionals. The Little-JIL

narrative (described in more detail in section 2.3.2) is a hyperlinked natural language representation of a process and it is automatically generated from a Little-JIL process model. Unlike manually created natural language representation, which are often inconsistent and ambiguous, the Little-JIL narrative uses carefully defined natural language templates to consistently and precisely describe the process information captured by the formal Little-JIL process model. Even some of the computer scientists found it useful to read through the narrative to find inconsistencies or to convince themselves that the model was accurate⁸. Real processes are large and complex, however, so both visual and textual representations can quickly become ungainly, suggesting the need to develop effective process model summarization technologies.

Process elicitation guided by developing in parallel a Little-JIL process model has had positive impact on the actual elicited processes. The very act of trying to precisely model a process often led to discovering process defects and to identifying possible process improvements. Sometimes during elicitation, misunderstandings between process stakeholders arose. In one situation it was determined that an artifact that was being created was not being used subsequently. In another situation, a deadlock that involved two different medical professionals was discovered while eliciting a chemotherapy process. A nurse needed to obtain the height and weight of a patient to be able to verify drug dosages and to subsequently sign the patient's treatment plan. There was a hospital rule, however, that an appointment scheduler must wait for a signed treatment plan before scheduling a visit at which the patient's height and weight could be verified. Thus, having both the nurse and the scheduler follow the rules creates a deadlock situation, which was actually being observed in clinical practice. To break the deadlock, some nurses were signing an unverified treatment plan so that the scheduler could schedule a visit at which height and weight could

⁸We describe a user study that compares the Little-JIL diagrammatic and narrative process representations in section 3.2.2.

be verified. The signing of an unverified and potentially incorrect treatment plan is dangerous, however, because it may mislead other medical professionals and perhaps result in the administration of the wrong dosage. The discovery of this deadlock led to a modification in the actual clinical process.

Little-JIL process models, being rich and precise, have also proved useful for training purposes. The senior oncology pharmacist who participated in the elicitation of a chemotherapy process printed the natural language process description corresponding to a Little-JIL process model and posted it as a training guide and reference for other pharmacists working in the chemotherapy center [14]. After having participated in the elicitation and modeling with Little-JIL of a blood transfusion process and becoming more aware of different exceptional situations that might arise and inconsistent use of terminology, a nursing professor modified the way she taught students how to perform that process.

3.2.3.2 Process Model Validation

As previously discussed, creating process models could be difficult. To determine how accurately a model represents what it models, a model is usually validated. *Validation*, as defined by the U.S. Department of Defense, is “the process of determining the degree to which a model is an accurate representation of the real-world from the perspective of the intended uses of the model” [9]. Since in this work models are used for online process guidance, we adapt this definition of validation to insist that models be an accurate representation of the the set of process executions as agreed by domain experts and process designers. We consider model accuracy to consist of two components—model correctness and model completeness. A model is correct when all the process executions it captures are correct with respect to a set of requirements. A model is sufficiently complete when it does not omit any of the process executions that should be part of that model as agreed by domain experts and process designers.

A process model is rarely accurate the first time it is created and it often takes several iterations discussing the model (or the information it represents, if the domain experts are not versed in the modeling notation) with domain experts to improve the accuracy of that model. During such discussions, process modelers, and sometimes even domain experts, obtain new knowledge about the process, meaning that there is a fine line between process elicitation and process model validation. Thus, the process elicitation techniques described in section 3.1 could be used for process model validation. In fact, as discussed in section 3.1, a process model was created in parallel while the unstructured interviews were performed to elicit the chemotherapy treatment plan review process. The subsequent elicitation methods, namely structured interviews and direct observations, revealed inaccuracies in the model and led to modifications in that model. These inaccuracies correspond to the new process information that was obtained via the application of these subsequent elicitation methods and included omitted steps from the normal process flow specified by the model, missing exceptional situations and the responses to these situations.

Another technique to validate process models is to apply static analyses approaches to such models. We have applied model checking [37], fault-tree analysis [128], and failure mode and effects analysis [121] to several models of HIPs from different domains to find defects and vulnerabilities in the modeled HIPs ([14,35,110,120]). We have observed that before any problems can be found in the real-world HIPs, multiple inaccuracies in the models of these HIPs are uncovered via these static analysis approaches. For example, we have found that important steps were missing from the process models and that control flow among steps in these models was incorrectly specified, especially control flow related to exceptional situations where various aspects need to be considered, such as how is an exception handled, what happens to other potentially concurrently executing process activities while the exception is being handled, and how the process resume nominal flow after the exception has

been handled. We have found such static analyses approaches to be very useful for validating process models and we believe they should be used in conjunction with more traditional model validation approaches such as inspections, observations, and interviews.

CHAPTER 4

DEVIATION DETECTION AND EXPLANATION

4.1 Overview

The deviation detection and explanation approach presented in this chapter is a component of the process improvement environment discussed in Chapter 2. Figure 4.1 shows the high-level architecture of the deviation detection and explanation approach. As a process is being performed, the Process Execution Monitor captures events associated with the performed steps and incrementally creates the *sequence of performed step events*. We use the term *step* to refer to an activity of interest performed by humans, software or hardware devices as part of a process. To simplify the discussion, hereafter we shall refer to the sequence of step events associated with the sequence of actual performed steps as the *sequence of performed steps*. We make the simplifying assumption that the sequence of performed steps is accurate, meaning that the sequence contains all the events in the appropriate order for all steps of interest that were performed, and there are no “noise” events (events that do not correspond to a performed step of interest). We briefly discuss several possible approaches for monitoring an executing process in section 4.4, but the details of these approaches are outside the scope of this work.

Every time the Deviation Detector receives a step event from the Process Execution Monitor, it checks whether the corresponding sequence of steps performed so far is one of the recommended sequences as specified by the Process Model. If it is not, a *deviation* is detected, where we define a deviation as a situation where the sequence of performed steps is not a prefix of one of the recommended sequences as specified

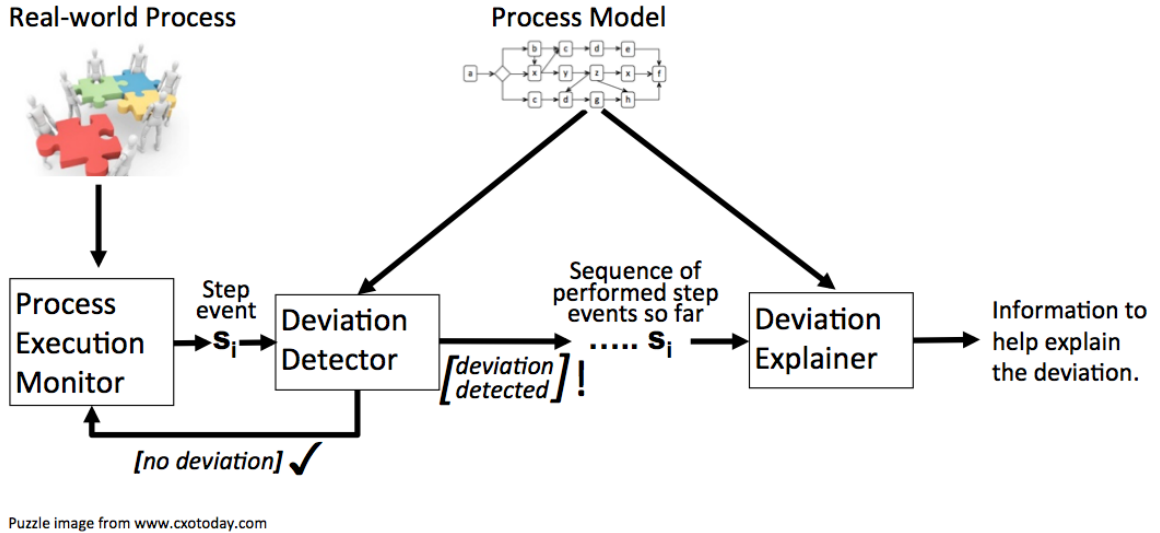


Figure 4.1: Deviation detection and explanation approach.

by the Process Model. Upon deviation detection, process performers are warned that a potential error might have occurred and the Deviation Explainer provides process performers with information that could be useful for identifying potential errors and for determining how to recover from such errors.

The deviation detection and explanation approach focuses on *planning errors*, where a planning error is the use of a wrong plan to achieve an aim [82]. We use a standard definition of *plan*, which is “a sequence of activities”, and adapt it to be a “sequence of steps”, because in this work we use the term *step* to represent a process activity. Process performers can use the wrong sequences of steps to achieve an aim either because they choose a wrong sequence to achieve that aim or because they choose a correct sequence, but as they carry out that sequence they perform (or do not perform) a step, which causes the sequence of performed steps to not be one of the possible correct sequences for achieving the aim. The deviation detection and explanation approach targets the latter kind of planning errors. Thus, we assume that process performers intend to perform a correct sequence of steps to achieve an aim (we call that sequence the *intended sequence of steps*) and we restrict the

definition of planning error to “performing (or not performing) one or more steps, which causes the sequence of performed steps to not be a prefix of the intended sequence of steps”. Hereafter, we will use this meaning of planning error and we will use the terms *planning error* and *error* interchangeably, unless explicitly indicated otherwise. We also define the term *first erroneous step* as the first step in the sequence of performed steps where the sequence of performed steps and the intended sequence of steps differ from each other. In general, detecting a deviation does not necessarily mean that process performers have made an error. If an intended sequence of steps is not captured by a process model, because the process model is incomplete, then if that sequence is performed, a deviation will be detected, but there would be no error. In this work, we assume that the process model used for deviation detection is complete and, thus, when a deviation is detected it means that an error has been made. In practice, it is difficult to create complete process models, especially models of complex HIPs, such as medical processes. We believe, however, that process models could capture a significant portion of the recommended ways to perform a process to the extent that such models could be useful for the deviation detection approach to help catch errors before harm is done without at the same time generating too many false positive deviation warnings due to the incompleteness of the model. Chapter 3 discusses approaches for eliciting processes and creating corresponding models.

It is also interesting to note that making an error during a process execution does not necessarily mean that a deviation will be detected (immediately, or even eventually). Even if we assume that the process model is correct (i.e., all the sequences of steps it captures are recommended sequences to perform a process), it is still possible for an error to occur without the Deviation Detector being able to detect a deviation. Section 4.1.2.1 discusses this phenomenon in further detail. In this work, we assume that the process model used for deviation detection is correct. In practice it is difficult to create correct process models, but there are approaches for analyzing

process models with respect to a set of correctness properties with the goal of finding and removing defects from the models. Model checking [37] is one such approach and it is discussed in section 2.1.1.

In the rest of this section, we describe in more detail the components of the deviation detection and explanation approach shown in Figure 4.1.

Process Model. As discussed in chapter 2.2, in our experience, to support on-line guidance for complex HIPs, the process model needs to be written in a notation with rich and well-defined semantics. Specifically, such a notation should provide support for modeling human choice, exception handling, concurrency, and synchronization. For our work, we have chosen the Little-JIL process modeling language [27], because it satisfies these requirements (Little-JIL is presented in Chapter 3). Little-JIL’s compact visual notation and the corresponding automatically-generated precise natural language representation facilitate communication with domain experts and, hence, the elicitation and validation of process models. The proposed deviation detection and explanation approach, however, is independent of the process modeling language; it can be used with any language that has well-defined semantics that are also sufficiently rich to capture complex real-world processes.

To make the implementation of the deviation detection experimental framework (described in section 4.2) independent of a particular process modeling language, we used a low-level process model representation into which various high-level process modeling languages can be translated. This translation and the resulting low-level representation are explained in detail in Appendix C.

To keep the discussion at a higher level and to better illustrate some of the issues related to deviation detection and explanation, in this chapter we chose to use a conceptual, simplified process representation—an extended control flow graph (ECFG). An ECFG process model consists of nodes and edges. Activity nodes (graphically illustrated as rounded rectangles) represent steps from the real world process. Each

activity node is labeled with the name of the step it represents. There is a directed edge from activity node A to activity node B, if the process step represented by node A can immediately precede the process step represented by node B. An edge can optionally have guards (shown as text in square brackets), which can specify the conditions under which control can flow over that edge.

Fork and join nodes (graphically illustrated as black bars with the words “fork” and “join” next to them respectively) are used to represent concurrent execution and synchronization. The destination nodes of all edges leaving a fork node are the beginning of node sequences whose corresponding step sequences from the real process can be performed in parallel with each other. The source nodes of all edges going to a join node are nodes whose corresponding process steps need to all be performed before control can flow through the join node.

The nodes and the edges in an ECFG form a weakly connected directed graph. Nodes that do not have any incoming edges are start nodes (i.e., control can start at such nodes); nodes that do not have any outgoing edges are final nodes (i.e., control ends at such nodes).

An ECFG representing a simplified blood transfusion process is shown in Figure 4.2. In that example, each activity node is also labeled with a single letter for convenience in the discussion in section 4.5.

Deviation Detector. The Deviation Detector traverses the process model, starting from every start “location” of that model¹, to determine whether the sequence of performed steps so far is a legal sequence of steps through the model. For efficiency, this traversal is done incrementally, breadth-first, maintaining a frontier of possible locations² in the process model that could correspond to the last performed

¹What “start location” means, depends on the kind of process model used. In the low-level representation of a Little-JIL model that we use in our implementation, the start location is a node in a graph.

²Again, what “location” means depends on the particular process model used.

step. When this frontier becomes empty, i.e., when there is no sequence of locations through the process model that corresponds to the sequence of performed steps, a deviation is detected.

Deviation Explainer. When a deviation is detected, the Deviation Explainer compares the sequence of performed steps against sequences of steps from the process model. Given the sequence of performed steps, it uses a similarity measure based on string comparison techniques to identify sequences of steps from the process model that are likely to have been intended by the process performers. The differences between these likely intended sequences and the sequence of performed steps are then interpreted to obtain information that could be used to explain the detected deviation. The deviation explanation approach is discussed in more detail in section 4.5.

4.1.1 Example of Applying the Deviation Detection and Explanation Approach to a Medical Process

Figure 4.2 shows a model of a simplified blood transfusion process in the ECFG notation. Figure 4.2 focuses on some of the steps performed by the nurse and, to reduce clutter, leaves out information that is not relevant to this discussion, such as the tasks of others (e.g., blood bank, physician) as well as most of the exceptional situations that could arise (e.g., the patient does not speak the language or is unconscious). According to this model, to carry out a physician’s order for blood transfusion, the nurse needs to first contact the laboratory to check whether the patient’s blood type is known. If the blood type is not known, the nurse needs to obtain a blood specimen and send it to the blood bank for testing. Once the patient’s blood type is known and the blood bank has prepared the blood product, the nurse can pick up the blood from the blood bank.

After picking up the blood and before infusing it into the patient, the nurse needs to first verify the patient’s identity and then verify the blood product to ensure that

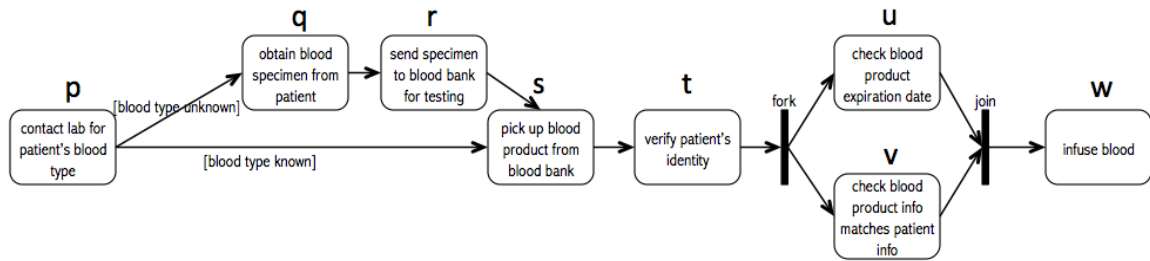


Figure 4.2: Simplified blood transfusion process.

the correct blood product will be given to the correct patient. Verifying the patient's identity involves asking the patient for identification information, such as name and date of birth, and making sure that this information matches the patient's ID band and the patient chart. Verifying the blood product involves checking its expiration date and checking that the information on the product (patient name, date of birth, and blood type) matches the same information on the patient ID band and in the patient chart.

A common error reported in the medical literature, and one that can cause severe harm to patients, is not fully following the procedure for verifying the patient's identity [70]. A possible instance of this error is omitting the step *verify patient's identity* altogether. Consider the situation where in a busy emergency department patient A is wearing the incorrect ID band—that of patient B. Perhaps a registration clerk had to place ID bands on several patients and inadvertently switched the ID bands; or there was a shift change and due to a miscommunication the new clerk placed the ID band for patient B on patient A. Suppose that patient B is the one for whom a blood transfusion was ordered. Since patient A is wearing patient B's ID band, if the nurse does not check the identity of patient A prior to infusing the blood, patient A might receive the blood ordered for patient B. Note that the nurse might still have successfully performed the blood product checks since they are done against the ID band and not against the patient's real identity.

Potential harm as a result of this error might be avoided if the nurse is warned that the process is being performed incorrectly before the infusion is started. One way to achieve this is by comparing the sequence of steps the nurse has performed against the process model. A possible sequence of steps when the nurse forgets to verify the patient's identity is *contact lab for patient's blood type, pick up blood from blood bank, check blood product expiration date, check blood product info matches patient info, infuse blood*. As soon as the nurse starts the step *check blood product expiration date*, the sequence of steps is no longer a sequence allowed by the model specified in Figure 4.2³. Informing the nurse about such a deviation might help the nurse recover from the error before harm is done (i.e., infusing blood into the wrong patient).

Depending on the level of expertise of the process performer and the complexity of the error, just a warning that an error might have been committed might be sufficient to identify the error and to recover from it. In the example above, it might be fairly easy for an experienced nurse to determine what went wrong. In more complicated situations, however, perhaps involving a less experienced process performer or involving multiple process performers working concurrently and dealing with exceptional cases, additional information might help determine what the errors were and how to recover from them.

For instance, a hypothesis about the location(s) in the sequence of performed steps where the error was committed could be presented. In the current example, the nurse could be told that an error might have occurred when the third step in the sequence, *check blood product expiration date*, was performed. In a more complex situation, the actual error might have occurred earlier than when it was detected (an example is

³Note that checking the blood product expiration date before verifying the patient's identity might not be problematic by itself. The hospital might have designed the process the way it is shown in Figure 4.2, however, based on experience that when the blood product checks are done before verifying the patient's identity, the verification of the patient's identity is more likely to be omitted, or for efficiency reasons.

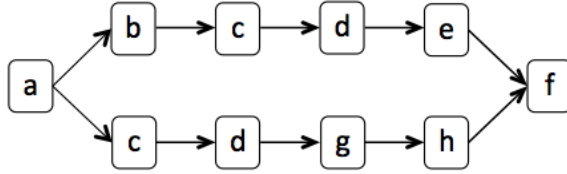


Figure 4.3: An Example Extended Control Flow Graph.

discussed in section 4.1.2). Pointing the process performers to that earlier location in the performed sequence of steps could provide them with the necessary context to determine what the error is and how to recover from it.

4.1.2 Issues

There are several interesting issues related to the proposed deviation detection approach. These issues are discussed next.

4.1.2.1 Delayed Deviation Detection and Potential Harm Due to Delay

It is sometimes possible that process performers make an error, but a deviation cannot be immediately detected. For example, consider the example ECFG process model in Figure 4.3. The two legal sequences of steps allowed by this process model are $abcdef$ and $acdghf$. Suppose that during a particular execution of the process represented in Figure 4.3, the process performers intend to carry out the sequence $abcdef$, but forgot to perform step b . In this situation, the first erroneous step is c since it was performed after a instead of the omitted b . A deviation cannot be detected, however, until e is performed (resulting in $acde$ being the sequence of performed steps), because acd is a legal sequence through the process model. Thus, there is a *deviation detection delay* with respect to when an error is committed and we call this situation *delayed deviation detection*.

Given that the intent of process performers is often unknown, there are situations, like the one in the above example, when a deviation detector cannot determine which branch(es) should be taken at a branch point during a particular process execution.

Such situations might arise in real HIPs when, for example, decisions need to be made based on subjective, and sometimes error-prone, human judgement, perhaps under uncertainty or in the absence of evidence-based guidelines. In some cases, additional information could be collected in such situations to determine which branch should be taken. For example, process performers could be asked to declare their intent at such branch points. In other situations, however, collecting such additional information may be impractical and, thus, we assume that there will be branch points at which a deviation detector will not know which branch should be taken.

Such branch points can cause deviation detection delays. For instance, if at such a branch point a human makes a commission error consisting of performing steps from a non-intended branch and then performs the steps from the intended branch, a deviation detector will not be able to detect the commission error until the human performs the first step of the intended branch, or possibly until even later, if the two branches have common steps. Similarly, if a process performer omits a step from the beginning of the intended branch (as is the situation in the above example based on the model in Figure 4.3), a delayed deviation detection could also arise.

To understand and characterize the issue of delayed deviation detection, it is useful to have a way to measure a deviation detection delay. A deviation detection delay could be measured in terms of real time, e.g., number of seconds between the moment when the first erroneous step was performed and the moment when a deviation is detected. As discussed in Chapter 1, however, real-time issues are outside the scope of the deviation detection and explanation approach and, thus, we decided to not use this kind of measurement of deviation detection delay. Instead, we are interested in deviation detection delay in terms of the number of steps between the first erroneous step in the sequence of performed steps and the step where the deviation is detected. For instance, in the example based on the model in Figure 4.3 discussed above, the

deviation detection delay is 2 steps, given that the first erroneous step in the sequence of performed steps *acde* is *c*, but a deviation is not detected until *e* is performed.

It is important to note that even though delayed deviation detection can arise in real process executions, the existence of delayed deviation detection and the delay itself cannot be determined without knowing the intent of process performers. Under some assumptions about the errors that can occur during the execution of a process, for example the number and kind of errors, it might be possible to determine what the maximum delay might be upon the detection of a deviation. For instance, if we assume that process performers can make at most one error per execution of the process modeled in Figure 4.3 and this error is omission of a single step, then upon detecting a deviation at the end of the sequence of performed steps *acde*, it can be determined that the maximum delay is 2 steps. It cannot be determined, however, whether the deviation detection delay is indeed 2 steps without knowing whether process performers intended to perform the *abcd...* or the *acdg...* sequence of steps through the process model.

The issue of delayed deviation detection leads to situations where the deviation detection approach might not be able to detect deviations before harm is done as a result of an error. In principle, deviation detection delay can be arbitrarily long (if the two branches in Figure 4.3 had a longer sequence of identical substeps—in this case this sequence is *cd*—the deviation detection delay could be even longer) and serious harm could potentially occur before the deviation is detected. It is important to study the issue of delayed deviation detection to understand to what extent it constitutes an obstacle for the proposed deviation detection approach (e.g., how often delayed deviation detection arises and what the severity of the consequences from the delay is) and to also identify strategies for dealing with delayed deviation detection.

Some important research questions related to delayed deviation detection are:

- What are the characteristics of deviation detection delays in complex HIPs?

- How often might deviation detection delays occur in complex HIPs?
- How long could such deviation detection delays be?
- How critical/harmful could such deviation detection delays be?
- What are the causes for potential deviation detection delays?
- How can deviation detection delays be reduced or avoided?
 - How can a process model be statically analyzed to determine potential deviation detection delays?
 - How can a process be changed to reduce the potential for deviation detection delays?
 - What can be done during an execution of a process to avoid deviation detection delays?

In this work, we focus on the first set of research questions. The second set is discussed under future work.

4.1.2.2 Potential Harm When Deviations Are Immediately Detected

Even when deviations are immediately detected as the errors that cause them are committed, harm could potentially still occur. Such a situation can arise when an omission error is made before a potentially harmful step. For example, in the simplified blood transfusion process shown in Figure 4.2, one of the checks that need to be performed before infusing the blood is to ensure that the blood product has not expired. The process model allows this check to occur immediately before infusing the blood. If the nurse forgets to check whether the blood product has expired, the deviation will be detected when the nurse starts the infusion. Even though there is no deviation detection delay (based on the above definition of deviation detection delay), harm could still occur as the patient might receive expired blood. This is an example

of a vulnerability in the process model or in the process itself (assuming the model is accurate) with respect to the proposed deviation detection approach—a deviation cannot be detected until a potentially harmful step is already started.

Another situation where harm could occur, even if the deviation is immediately detected, is when the error is a commission of a harmful step (e.g., *administer a drug*). In this case, a deviation cannot be detected before the harmful step is started because the sequence of steps up to that point would be a legal sequence through the process model.

In this work, we explore the following research questions related to the above issue:

- How often do situations where harm could occur, even when deviations are immediately detected, arise in some complex HIPs?
- What can be done to reduce or avoid such situations?

4.1.2.3 Performance of the Deviation Detector

It is important that the Deviation Detector does not take too long to compute whether a deviation has occurred after a new step is added to the sequence of performed steps. Otherwise, if a deviation occurs, the warning about that deviation might get issued after harm is already done as a result of the deviation.

Models of realistic HIPs and the corresponding search space that the Deviation Detector needs to explore while traversing a process model could be of significant size, possibly requiring more information to be stored at runtime than the available computer memory allows.

We explore the following research questions related to the performance of the Deviation Detector:

- What is the size and the complexity of HIPs that can be handled by the Deviation Detector?

- What are the running time and the space requirements of the Deviation Detector when applied to some realistic HIPs?

4.2 Deviation Detection Framework

To study the above issues, ideally, we would need (1) a realistic model of a HIP and (2) a set of step sequences with errors that correspond to erroneous real executions of the modeled HIP. We were able to create several realistic models of HIPs by using the process elicitation and modeling techniques discussed in Chapter 3. We were not able to obtain, however, step sequences that correspond to real process executions with errors. Obtaining such sequences for HIPs is challenging as some important process steps, such as cognitive activities (e.g., verifying the patient stated name matches the name on the patient’s ID band), are difficult to automatically capture and often require time-consuming, expensive, and potentially error-prone manual observations of an executing process. Furthermore, obtaining real process executions that contain errors might increase the number of executions that need to be observed as it is unethical to encourage process performers to commit errors. The recognition of potential errors in real process executions could be a challenging task on its own.

By interviewing domain experts and surveying domain literature, we identified several plausible errors that could occur in HIPs and roughly where in a process execution such errors might occur. We used this set of errors in our evaluation of the deviation detection approach, but we did not deem this set to be comprehensive enough by itself to serve as the basis for this evaluation.

To overcome the above obstacles in obtaining realistic process executions with errors, we created a deviation detection experimental framework (shown in Figure 4.4) that supports the generation of synthetic sequences of steps based on the model of a HIP and the seeding of errors in such generated sequences. The characteristics of the generated sequences and the number and kinds of seeded errors can be controlled,

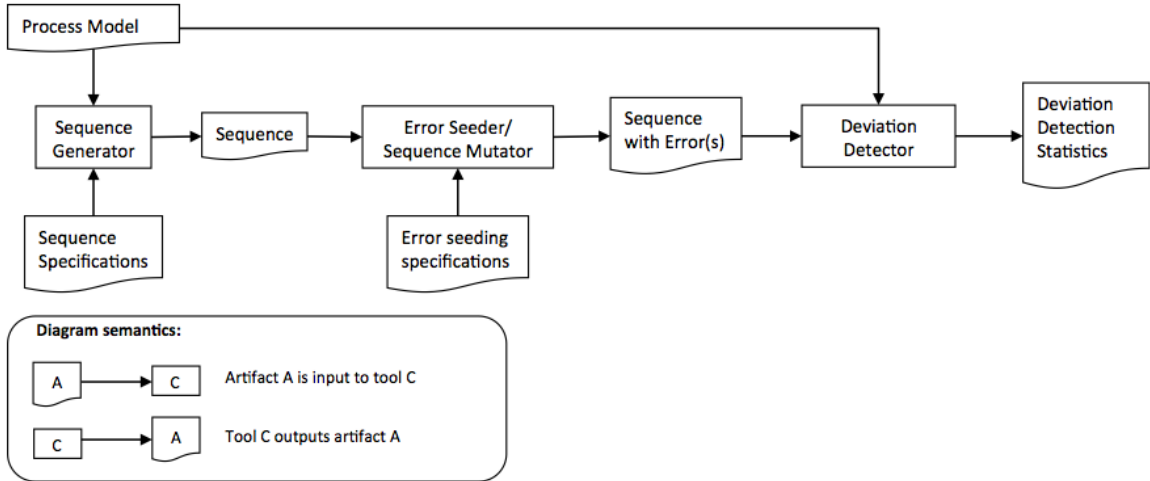


Figure 4.4: Deviation detection experimental framework.

so that the previously discussed deviation detection issues can be studied in various circumstances.

In this framework, a *Sequence Generator* takes as input the Process Model and a set of *Sequence Specifications* and then produces a sequence of steps from the Process Model that satisfies the given Sequence Specifications. The Process Model is a low-level process model translated from a Little-JIL process model. The translation procedure and the low-level model are described in Appendix C.

The Sequence Specifications are used to control the characteristics of a generated sequence and include the minimum/maximum length of that sequence, whether the sequence should be generated by a random walk through the process model or by a non-random walk favoring or avoiding certain steps. For example, if the goal is to study deviation detection issues on the nominal flow of a given process, it can be specified that a sequence corresponding to a process execution with no exceptions should be generated.

A synthetically generated sequence corresponds to a legal process execution. We will refer to such a sequence as a *legal sequence*. To obtain a sequence that corresponds to a process execution with error(s), a legal sequence is processed by the Error Seeder.

The Error Seeder also takes as input Error Seeding Specifications, which are used to control the kinds and number of errors that get seeded into the given legal sequence. Examples of the kinds of errors that could be seeded are omission of a step, commission of a step, or omission of several steps that pertain to a given subprocess.

A sequence with seeded errors (also referred to as a *mutant*) represents a process execution with errors. The Deviation Detector takes a sequence of steps as input and determines whether that sequence is a legal sequence. The Deviation Detector can also be instrumented to collect various Deviation Detection Statistics, such as whether there was a deviation detection delay, how long that delay was, and how much time it took to detect the deviation.

In this experimental framework, a generated legal sequence can be considered to be an intended process execution, a mutated sequence can be considered to be a sequence of performed steps, a mutation can be considered to be an error, and the index of the (first) mutation can be considered to be the index of the first erroneous step. These interpretations allow us to experimentally evaluate the deviation detection approach with respect to the issues described in section 4.1.2, and in particular with respect to delayed deviation detection where the delay can only be measured if the intended sequence of steps is known.

4.3 Experimental Evaluation

In this section we describe how we utilized the deviation detection framework to experimentally study the deviation detection issues discussed in section 4.1.2.

4.3.1 Experimental Design

To perform an initial evaluation of the proposed deviation detection approach, we used the experimental framework to apply the approach to models of chemotherapy and blood transfusion processes elicited from and validated by domain experts in-

volved in these processes [14,31,35,70]. We generated various sequences of steps from these models and then mutated these sequences to represent process executions with errors. The kinds of mutations we used are described below.

Our evaluation method is somewhat related to mutation testing [52] and fuzz testing [94]. Mutation testing is a common software engineering technique where a computer program is systematically mutated (usually by making a predefined set of simple changes to the source code) to evaluate software test suites and software testing and analysis approaches. All mutated versions of a program are executed with the inputs from a test suite and the smaller the number of mutated versions that pass the test suite, the higher quality the test suite is considered to be. This reasoning can be extended to assess the quality of software testing approaches for assisting with the creation of test suites.

Fuzz testing is a form of random testing for evaluating the robustness of a program. Inputs for a program are usually randomly generated and can include values outside the program’s set of legal inputs. The goal is to find inputs that can “break” the program, e.g., cause the program to crash or halt, and then determine the vulnerabilities in the program code that are responsible for the breaking of the program given the randomly generated inputs.

Our evaluation method is similar to mutation testing in that we apply a predefined set of mutations. Unlike mutation testing, however, we do not mutate the process model (which is the analog of a computer program), but we mutate sequences of steps from that model (which are the analog of computer program execution traces). Furthermore, we do not evaluate the collection of inputs to a program (which is what a test suite is), but we evaluate the deviation detection approach that takes as input a program (the process model) and an execution of that program (the sequence of performed steps).

If we consider the Deviation Detector (which implements the proposed deviation detection approach) and not the process model to be the analog of a computer program and the sequence of performed steps and the process model to be the analog of input/test data for that program, then our evaluation method shares some similarities with fuzz testing—we “stress-test” the Deviation Detector by giving it as inputs a large number of synthetically generated sequences of steps to evaluate the Deviation Detector (and the approach it implements). We are interested, for example, in finding sequences of steps that represent process executions with errors, such that the Deviation Detector cannot detect a deviation at the index of the sequence where the error is seeded. We are then interested in finding vulnerabilities in the process model (but not the deviation detector, which is the analog of the program in this case) that cause such delayed deviation detections.

The following sections describe in more detail the artifacts we used in the experimental evaluation—the process models and the generated synthetic sequences with errors—and the experiments we performed.

4.3.1.1 Process Models

We used two realistic models of a chemotherapy and a blood transfusion processes to evaluate the deviation detection approach. The chemotherapy process model was elicited from medical professionals participating in a chemotherapy ordering and administration process in a regional cancer center in Western Massachusetts. The elicitation took place over the course of a year and multiple interviews with domain experts were conducted until the domain experts expressed satisfaction with the accuracy and the completeness of the process model⁴. For additional validation of

⁴The process model was created in Little-JIL. Some domain experts, who felt comfortable with the notation, reviewed parts of the Little-JIL model in addition to the corresponding natural language description of that model; others preferred to review only the natural language description.

the process model, some domain experts were also observed while performing some activities from the process.

The blood transfusion process model was part of a benchmark for studying the applicability of software engineering techniques to improving medical processes [35]. We developed that benchmark with a nursing faculty member working on patient safety [70]. This elicitation also took place over a year and multiple interviews with the nursing faculty member were performed until she expressed satisfaction with the accuracy and the completeness of the process model.

The two selected process models are of significant size and complexity. At the time of writing this dissertation, the chemotherapy process model covers multiple phases of the chemotherapy process, including diagnosing the patient and ordering chemotherapy; thorough review of the treatment plan and medication orders by a medical assistant, a nurse, and a pharmacist; conducting an informational/teaching session with the patient and obtaining informed consent form; preparing chemotherapy drugs, performed by a pharmacist and pharmacy technicians; assessing the patient and administering the drugs, performed by a clinical nurse. The chemotherapy Little-JIL process model includes 283 steps performed by human process performers and specifies 59 exception handling situations⁵. The corresponding low-level representation (discussed in Appendix C) has 2,358 nodes and 701,887 edges.

The blood transfusion process model is based on a standard blood transfusion checklist from the medical literature [130] and includes additional information about problems that might arise during the process and their handling. This model covers the process activities starting from receiving a physician order for transfusion to discharging the patient. It specifies multiple phases of the process, including verifying

⁵An exception handling situation usually involves multiple steps to deal with the exception. A step (e.g., *assess patient*), can occur both on a nominal process execution and on an execution where an exceptional situation arises (e.g., patient develops an allergic reaction). Furthermore, a step can also be part of different exception handling situations.

that the patient blood type and screen are available (and if they are not, performing the subprocess of obtaining and labeling a blood specimen); ordering and obtaining a blood product from the blood bank; performing various verifications on the patient and on the blood product before starting the transfusion; monitoring the patient during the infusion and appropriately reacting if a transfusion reaction is suspected. The blood transfusion Little-JIL process model includes 102 steps (including 63 exception handling situations) and the corresponding low-level representation has 97,237 nodes and 242,442,845 edges. Even though the blood transfusion model has fewer steps than the model of the chemotherapy process, the exception handling behavior in the blood transfusion process is more complex, requiring a large number of nodes and edges in the low-level representation to express this behavior.

4.3.1.2 Synthetic Sequences with Errors

To evaluate the deviation detection approach, we applied it to synthetic sequences of steps containing typical planning errors. Although there is a disagreement in the human error literature about a standardized taxonomy of human errors [71, 82], two error kinds appear in the intersection of most of the proposed planning error taxonomies [77, 78, 82]—omission and commission errors. An omission error occurs when a step(s) that should be performed is not performed; a commission (planning) error occurs when the wrong step(s) is performed.

In our initial evaluation of the deviation detection approach, we focused on several kinds of planning errors that involve a single step and on the error of omission of a single subprocess. This decision was based on blood transfusion errors identified in the medical literature [70] and on conversations with blood transfusion and chemotherapy domain experts.

4.3.1.2.1 Single-Step Errors. From each of the two process models, we generated two kinds of legal sequences of steps—sequences that represented randomly se-

lected process executions and sequences that represented nominal process executions, i.e., executions where no exceptions occur. We generated 50 different sequences of each kind for a total of 100 legal sequences per process model. There were no common sequences between the 50 sequences that represented randomly selected process executions and the 50 sequences that represented nominal process executions—the sequences that represented randomly selected process executions all contained at least one exceptional situation, due to the high number of exceptional situations in both process models. All generated sequences represented *full* process executions. We consider a process execution to be full if after the last step in that execution, the process model does not allow any additional steps be performed. Statistics about the lengths of the generated sequences are shown in Figure 4.5.

The number 50 for generated sequences of each kind was arbitrary, but considered to be large enough based on an exploratory analysis we performed before the experimental evaluation. In this analysis, we generated sequences from the process models, seeded errors in them, applied the Deviation Detector to these sequences with seeded errors, and analyzed the results. We were observing whether there was a deviation detection delay, and if there was, we noted the length of the delay, whether harm could occur as a result of the delay, and the causes for that delay. We were also observing how situations could arise where harm could potentially occur even when there is no deviation detection delay and what the performance of the deviation detector was in terms of execution time and memory used. After having applied the deviation detector to about 30-40 sequences with seeded errors from each process model, we stopped obtaining any new findings as part of the above observations—a sign of data saturation. Thus, for the experimental evaluation of the deviation detection approach, we decided to generate 50 different sequences of each kind (described above) for each model. After systematic mutations, each set of 50 sequences resulted in a much larger set of sequences with seeded errors—between 1000 and almost 8000

sequences in a set—which is a much larger number of sequences with seeded errors than the 30-40 after which we reached data saturation in our exploratory analysis.

For the experimental evaluation of the deviation detection approach, three different single-step mutations—deletion, insertion, and substitution of a step—were applied to each generated sequence to represent three different kinds of single-step errors—an error of omission, commission, and substitution respectively. Each mutation kind was applied at each index of a sequence to represent situations where errors occur at different points of a process execution. Thus, for each of the two sets of 50 randomly selected sequences from the two process models, we created three sets of mutated sequences (mutants), one set for each mutation kind, resulting in one set of *deletion mutants*, one set of *insertion mutants*, and one set of *substitution mutants*. Similarly, for each of the two sets of 50 sequences that represented nominal process flow through the two process models, we created three corresponding sets of deletion, insertion, and substitution mutants. These mutations resulted in six sets of mutants for each process model and the number of mutants in each of these sets is shown in Figure 4.5 in the columns labeled “Experiment 1” through “Experiment 6”.

A deletion mutant was created by deleting a step from a sequence. A step from every sequence index (except the last index) was deleted from each of the original sequences. A deletion of the last step in a sequence was not performed because that would leave the sequence a legal process execution. An insertion mutant was created by inserting a step, chosen uniformly and at random from all process steps, into a sequence. A step was inserted between every two steps (including before the first step, but not after the last step) in each of the original sequences. If the inserted step was the same as the subsequent step in the sequence, this would always result in a delayed deviation detection; so, in these cases another step instead was randomly chosen for insertion. A substitution mutant was created by substituting a step in a sequence with a different step chosen uniformly and at random from all process steps.

A substitution mutation was done at every index of each of the original sequences. For all three mutation kinds, mutants that remained legal sequences were discarded.

4.3.1.2.2 Subprocess Errors. In addition to studying the Deviation Detection approach on sequences that represent process executions with single-step errors, we were also interested in studying the approach on sequences that represent process executions with more complex errors. One such kind of errors reported in the literature [70] and identified in discussions with domain experts is the omission of an entire subprocess.

We identified 5 blood transfusion and 5 chemotherapy subprocesses deemed most likely to be omitted, based on common blood transfusion and chemotherapy errors reported in the medical literature and on our interaction with domain experts involved in these processes. The selected blood transfusion subprocesses were: (1) *ensure correct patient is present* (performed by a nurse before notifying the blood bank to prepare the blood to prevent the possibility the blood product to expire because the patient is not available for transfusion or the wrong patient is in the room); (2) *verify patient ID band* and (3) *verify blood product information* (part of the bedside checks performed by the nurse before beginning the infusion); (4) *assess patient* and (5) *evaluate patient clinically* (part of the clinical evaluation prior to the infusion, performed again by the nurse). The selected chemotherapy subprocesses were: (1) *record height and weight* (performed during patient registration by a clerk); (2) *confirm all necessary information is present* (part of the consultation and assessment, performed by an oncologist before creating the treatment plan and chemotherapy orders); (3) *confirm pretesting has been done* and (4) *confirm existence and not staleness of height/weight data* (part of the treatment plan and orders verifications performed by a Practice Registered Nurse (RN)); (5) *obtain patient informed consent* (performed by a Nurse Practitioner or a Clinic Nurse prior to chemotherapy administration).

For each identified subprocess, we generated 50 different sequences (allowing exceptions) from the corresponding process model, such that these sequences contained the steps from the subprocess. We then mutated each generated sequence by deleting all steps pertaining to the specific subprocess selected to be omitted. Thus, we created 250 mutants for each of the two process models. We call these mutants *subprocess omission mutants*.

4.3.1.3 Experiments

Using the deviation detection framework, the deviation detection approach was applied to each mutant and various statistics were collected. We performed seven experiments with the mutants from each process model. In experiments 1 through 3, we applied the deviation detection approach to the deletion, insertion, and substitution mutants generated from the 50 random sequences. In experiments 4 through 6, we applied the approach to the deletion, insertion, and substitution mutants generated from the 50 nominal flow sequences. In experiment 7, we applied the approach to the subprocess omission mutants.

4.3.1.4 Potentially Harmful Steps

To address the question of whether/how often harm might occur due to delayed deviation detection, we identified a set of potentially harmful steps from the blood transfusion and chemotherapy processes. These are steps, such that if an error has occurred prior to their performance, performing them could potentially result in immediate harm. One such step from the blood transfusion process is *begin infusion of blood product*. If, for example, the nurse has forgotten to verify the patient's identity prior to infusing the blood or forgotten to verify that the blood product matches the one listed in the blood transfusion order, the nurse can potentially infuse the wrong blood into the patient. An example of a potentially harmful step from the chemotherapy process is *administer chemotherapy drug*.

The set of potentially harmful steps for the blood transfusion process we used in this analysis is: $\{begin\ infusion\ of\ blood\ product, administer\ pre-transfusion\ medications, administer\ medications\ (during\ the\ transfusion), draw\ blood\ specimen\}$. The set of potentially harmful steps for the chemotherapy processes is: $\{administer\ chemotherapy\ drug, administer\ chemotherapy\ pre-medications, administer\ emergency\ medications, start\ IV, draw\ blood\ specimen\}$. The steps *draw blood specimen* and *start IV* are not as dangerous as the other ones in the two sets above, but could still cause significant inconvenience to the patient, if performed when they should not be performed.

Having identified the sets of potentially harmful steps for the blood transfusion and chemotherapy processes, we then inspected the mutated sequences described above to determine whether a potentially harmful step occurs between the mutation index and the index of deviation detection. For sequences for which this was the case, we manually analyzed whether the error (the mutation) could affect the potentially harmful step. If harm could occur as a result of the error, we counted the mutated sequence as one for which deviation detection delay could be potentially harmful.

4.3.2 Results

Figure 4.5 shows the results of applying the deviation detection approach to the blood transfusion and chemotherapy processes. The definition of deviation detection delay from section 4.1.2 is used, where the first erroneous step is the step at the *mutation index* (the index in a sequence of steps where the mutation was done for single-step errors or the index of the first mutation for subprocess errors).

4.3.3 Discussion

In this section, we discuss the results from the experimental evaluation with respect to the deviation detection issues presented in section 4.1.2.

		Experiment 1	Experiment 2	Experiment 3	Experiment 4	Experiment 5	Experiment 6	Experiment 7
Process definition					Blood transfusion process			
Set-up	Original traces	50 random			50 random, no exceptions			50 random sequences for each of 5 subprocesses
	Avg. trace length (number of steps)	21.52			70.62			70.54
	Min. trace length (number of steps)	4			69			68
	Max. trace length (number of steps)	114			73			73
	Mutation kind	Deletion at every position (except last)	Insertion before every position	Substitution at every position	Deletion at every position (except last)	Insertion before every position	Substitution at every position	Deletion of all steps belonging to subprocess of interest
Results	Number of mutants	1026	1076	1076	3481	3531	3531	250
	Number of mutants where deviation is detected after the mutation index	0 (0.00%) [27] *	6 (0.56%) [1] ±	5 (0.47%) [1] ›	6 (0.17%) [294] *	25 (0.7%) [7] ±	19 (0.54%) [1] ›	0 (0.00%) [23] *
	Avg. deviation detection delay (number of steps), for mutants with detection delay	0.00	1.00	1.00	1.00	1.08	1.00	0.00
	Min. deviation detection delay (number of steps), for mutants with detection delay	0	1	1	1	1	1	0
	Max. deviation detection delay (number of steps), for mutants with detection delay	0	1	1	1	2	1	0
	Number of mutants for which delay could be "potentially harmful"	0	0	0	0	0	0	0
	Number of mutants without deviation detection delay for which harm could potentially occur due to the deviation	1 (0.01%)	24 (2.2%)	35 (3.3%)	50 (1.4%)	96 (2.72%)	90 (2.52%)	50 (20.00%)
	Avg. time per mutant (sec.)	5.98	5.88	5.74	13.01	13.12	13.09	11.17
	Avg. time per step (sec.)	0.28	0.27	0.27	0.18	0.19	0.19	0.16

		Experiment 1	Experiment 2	Experiment 3	Experiment 4	Experiment 5	Experiment 6	Experiment 7
Process definition					Chemotherapy process			
Set-up	Original traces	50 random			50 random, no exceptions			50 random sequences for each of 5 subprocesses
	Avg. trace length (number of steps)	55.84			158.36			159.27
	Min. trace length (number of steps)	13			147			147
	Max. trace length (number of steps)	215			171			172
	Mutation kind	Deletion at every position (except last)	Insertion before every position	Substitution at every position	Deletion at every position (except last)	Insertion before every position	Substitution at every position	Deletion of all steps belonging to subprocess of interest
Results	Number of mutants	2742	2792	2792	7868	7918	7918	250
	Number of mutants where deviation is detected after the mutation index	0 (0.00%) [233] *	4 (0.14%) [2] ±	3 (0.11%) [4]	0 (0.00%) [795] *	13 (0.16%) [2] ±	11 (0.14%) [4] ±	0 (0.00%) [0] *
	Avg. deviation detection delay (number of steps), for mutants with detection delay	0.00	2.25	1.00	0.00	2.31	1.00	0.00
	Min. deviation detection delay (number of steps), for mutants with detection delay	0	1	1	0	1	1	0
	Max. deviation detection delay (number of steps), for mutants with detection delay	0	6	1	0	11	1	0
	Number of mutants for which delay could be "potentially harmful"	0	0	0	0	0	0	0
	Number of mutants without deviation detection delay for which harm could potentially occur due to the deviation	1 (0.00%)	33 (1.18%)	29 (1.04%)	20 (0.25%)	98 (1.24%)	109 (1.38%)	0 (0.00%)
	Avg. time per mutant (sec.)	27.42	28.16	28.43	35.32	34.55	34.49	16.37
	Avg. time per step (sec.)	0.49	0.50	0.51	0.22	0.22	0.22	0.10

* Number of mutants for which delay was due to a deletion in a shuffle region and deviation was detected right after the shuffle region.
± Number of mutants for which the delay was due to insertion of a step from a shuffle region in the model before the step has occurred in the corresponding shuffle region in the original sequence.
› Number of mutants for which delay was due to substituting a step from a shuffle region in the model before the step has occurred in the corresponding shuffle region in the original sequence.
For the mutants in square brackets we used the "minimum interpretation" for the deviation detection delay (see the Discussion section for more detail).

Figure 4.5: Applying the deviation detection approach to blood transfusion and chemotherapy process models.

4.3.3.1 Delayed Deviation Detection and Potential Harm Due to Delay

Delayed deviation detection occurred infrequently—in less than 1% of the mutants from all experiments. We analyzed each mutated sequence with deviation detection delay by tracing that sequence through the corresponding process model to determine the reasons for the delay. There were three main reasons: exception handling, optional steps, and shuffled steps. All of these three process structures were a source of branching similar to the one in Figure 4.3 that could potentially cause a deviation detection delay. For example, consider exception handling. Suppose a process can be performed by executing step sequence $abcd$, when there are no exceptional situations. If an exceptional situation arises while performing b , however, then the sequence of steps xyz should be performed to address this situation before continuing with steps c and d , resulting in sequence $abxyzcd$. If the legal sequence $abcd$ is mutated by inserting step x after step b , the deviation cannot be detected until step c is performed (which is one step after the mutation index) because abx is a prefix of a legal sequence. Optional steps caused deviation detection delay in an analogous way.

The third reason for deviation detection delays was shuffled steps. *Shuffled steps* are steps that should be done sequentially but are allowed to occur in any order. Sequences of steps that are generated from a process model and that contain shuffled steps have subsequences, which we call *shuffle subsequences*, such that the steps in these subsequences are allowed to occur in any order. For example, suppose that a process is performed by doing step a , followed by steps b , c , and d in any order, and then step e . There are six allowable sequences of steps to execute that process, corresponding to the six permutations of steps b , c , and d . One such allowable sequence is $abcde$. The subsequence bcd is a shuffle subsequence⁶. If the step sequence

⁶A shuffle subsequence does not have to be a contiguous subsequence of the sequence of performed steps. For example, if other steps can be performed in parallel with the shuffled steps, these other steps might be interleaved with the shuffled steps in the sequence of performed steps. For instance,

abcde is mutated by deleting step *b*, a deviation will not get detected until step *e* is performed because *acd* is a prefix of a legal sequence. This results in a deviation detection delay of 2, given that in our experimental evaluation we consider *abcde* to be the intended sequence, *acde* the sequence of performed steps, and *c* the first erroneous step in the sequence of performed steps.

Shuffled steps are different from other branch points in a process where there are more than one steps to perform next, because any of the shuffled steps is acceptable to be performed next during an execution of the process on which the shuffled steps need to be performed. In other branching situations when shuffled steps are not involved, for example when one branch is nominal flow and the other exceptional flow, one branch needs to be performed on certain process executions and the other branch needs to be performed on other process executions, but a deviation detector might not be able to determine which branch should be performed on the particular execution that is being monitored⁷.

Given that shuffled steps are common in the processes we studied (e.g., a nurse needs to perform several checks, but the order in which these checks are performed does not matter), we decided that deviation detection delays due to shuffled steps should be measured in a special way; otherwise the results related to delayed deviation detection would be distorted. If a mutation deletes a step from a shuffle subsequence, we measure the delay as the number of steps between last step in the shuffle subsequence and the step where a deviation is detected. If a mutation inserts or substitutes into a shuffle subsequence a step that is one of the steps in that shuffle subsequence, we measure the delay as the number of steps between the last occurrence of the inserted step in the shuffle subsequence and the step where a deviation is

if *a* and *b* are shuffled steps and *x* can occur in parallel with *a* and *b*, the sequence of performed steps might be *axb*, where *ab* is a shuffle subsequence.

⁷Section 4.1.2.1 discusses how this might happen.

detected. The results in Figure 4.5 reflect this modified way of measuring deviation detection delay when a mutation was performed in a shuffle subsequence.

In our analysis of the experimental results, we defined templates that could automatically recognize situations where a mutation was performed in a shuffle subsequence. Shuffled steps were explicitly annotated in the process models from which sequences of steps were generated and, thus, we knew which steps in a mutant correspond to a shuffle subsequence.

For the mutants where there was a deviation detection delay, the length of that delay was small—at most 2.31 steps after the mutation index on average and in most cases close to 1. In critical processes, such as medical procedures, however, even a delay of 1 could be harmful, if some potentially dangerous step, such as *administer chemotherapy medications*, is performed before detecting the deviation. In the performed experiments, however, there were no cases where the delay could result in harm (fourth row from the bottom in both tables in Figure 4.5).

4.3.3.2 Potential Harm When Deviations Are Immediately Detected

We found some mutated sequences for which the deviation is detected at the index of mutation, i.e., the deviation is immediately detected, but harm could potentially still be done (third row from the bottom in both tables in Figure 4.5). In experiments 2, 3, 5, and 6 for the blood transfusion and chemotherapy processes, this was due to mutating a sequence by inserting or substituting in a harmful step.

In experiments 1, 4, and 7 for the blood transfusion and chemotherapy processes, the cases where potential harm could occur even when the deviation is immediately detected were due to omitting a step or a subprocess that can immediately precede a harmful step and can affect that harmful step. For example, in the blood transfusion process model, the subprocess of verifying the blood product immediately precedes the step *begin infusion of blood product*. Thus, if steps from this subprocess (such as

ensure that blood product has not expired) are omitted, or the subprocess is omitted altogether, the deviation cannot be detected until the step *begin infusion of blood product* is started. For the blood transfusion process, the 50 cases in experiment 7 (the experiment where subprocess omission errors were seeded) of mutants in which harm could occur even when the deviation is immediately detected are due to omitting the subprocess *verify blood product information*. In the chemotherapy process model, we encountered similar structure, where the step *confirm that the patient ID band matches the drug label* can immediately precede the step *administer chemotherapy drug*.

Such structures represent real process vulnerabilities as there is little opportunity for process performers to realize that an error is made before they start a potentially harmful step. A possible strategy to deal with such process vulnerabilities is to introduce a non-harmful step before the potentially harmful one. This could be a step that requires verifying that all the preconditions for starting the potentially harmful step are met. In fact, such steps are already in place in some medical procedures, such as surgeries where the different surgical teams are required to stop at certain points of the procedure and confirm that every team has performed the necessary steps and is aware of the relevant information before proceeding further. The presence of such verification steps would allow the proposed deviation detection approach to detect deviations when such verification steps are performed and before potentially harmful steps are started.

Current process vulnerability analysis approaches, such as FTA and FMEA, do not take into account the vulnerabilities discussed above. It would be interesting, however, to explore how such approaches could be extended to identify such vulnerabilities and potentially even automate the identification. For example, if domain experts list potentially harmful steps in a process and the dependencies of these steps on other steps that affect the harmful steps, then a process model could be algorithmically

explored to discover situations where a step that affects a potentially harmful step can immediately precede or occur “shortly before” (for a domain-specific definition of “shortly-before”) that harmful step.

4.3.3.3 Performance of the Deviation Detector

The experiments were performed on a MacBook laptop with 2.4 GHz Intel Core 2 Duo processor. The deviation detection experimental framework is implemented in Java and the experiments were run with maximum heap size of 2.5 GB.

The running time results are shown in the last two rows of the tables in Figure 4.5. For the blood transfusion process, it took less than 6 seconds to determine whether a sequence of about 21.5 steps, on average, is a legal sequence; for the mutated sequences where exceptions were disallowed (these were longer sequences), it took around 13 seconds to determine whether a sequence of about 70.5 steps, on average, is a legal sequence. This amounts to less than a third of a second per step in a sequence. The running time of the deviation detector was similarly low for the chemotherapy process. Given that humans usually take more than a third of a second to perform a process activity, the running time results indicate that the deviation detector could detect deviations in real time and before harm is done as a result of a deviation for non-trivial processes of size and complexity similar to the ones we studied.

4.3.4 Threats to Validity

There are several threats to the validity of the obtained experimental results. The experimental evaluation was synthetic—we did not monitor a real executing process and did not apply the deviation detection approach to real executions of such a process. Despite our best efforts to create realistic process models using the process elicitation and modeling methods described in Chapter 3, it is likely that the models used in the experimental evaluation still contain some inaccuracies or miss some information potentially relevant to deviation detection. This would mean that

some of the synthetically generated sequences might be inaccurate representations of real process executions.

The experimental evaluation was based on only two models from a single domain. Even though these models were relatively large and complex in terms of covering a large set of process executions (including exceptional executions and concurrency within a single process execution), the results would change if the size and complexity of the models increases or if other models from the same domain or from different domains are used.

We “stress tested” the deviation detection approach by applying various kinds of mutations at every index of the synthetically generated sequences to represent various errors at various points of a process execution. The applied mutations, however, do not all correspond to realistic errors. For example, to create an insertion mutant we selected uniformly at random a step from all process steps and inserted it at a given index of a synthetically generated sequence. In some cases, however, such an insertion might not correspond to a realistic error (e.g., inserting the step *order blood from blood bank* in a part of the process where the blood transfusion has already been started and no additional transfusions have been ordered).

We mutated the generated sequences by performing single-step deletions, insertions, and substitutions and deletions of a subprocesses to represent errors of omission, commission, substitutions of a single step and omission of an entire subprocess. Other kinds of errors, however, can also occur during the execution of HIPs and we did not study the deviation detection approach in the presence of such errors.

4.4 Limitations of the Deviation Detection Approach

Even though the preliminary investigation of the proposed deviation detection approach is promising, there are several research challenges that need to be tackled before the approach can be applied in practice. The deviation detection approach

relies on receiving an accurate sequence of performed steps of interest from the Process Execution Monitor. Capturing accurately and in a timely manner what human process performers do, however, is difficult. We expect the introduction of more electronic devices in processes would facilitate process execution monitoring. For example, in a medical process, starting to receive infusion data from an infusion pump could be automatically interpreted as having started the step *begin infusion of blood product*; similarly, when patient height and weight data appear in an electronic medical record, this could be an indication that the step *measure height and weight* has been performed.

While the use of electronic devices in processes increases the opportunities for monitoring process executions, events from electronic devices could be misinterpreted. Furthermore, electronic devices cannot capture certain steps in processes, such as cognitive steps (e.g., a doctor making sure the patient information on two pieces of paper matches). A promising approach to capturing process steps more accurately than electronic devices and capturing steps that electronic devices cannot capture is the use of human scribes, which seems to be increasing in popularity [65] in medical processes.

Even in processes where human scribes are used, capturing the performance of cognitive steps remains a significant challenge. For example, suppose a human scribe is observing a nurse performing the simplified blood transfusion process depicted in Figure 4.2. To perform the step *check blood product expiration date* the nurse might look at the expiration date and time on the label of the blood product container and then reason whether the current time is before or after the expiration time (the nurse might obtain the current time by checking a watch, a wall clock, the clock on a computer screen, or perhaps rely on memory of a recent time check). A human scribe observing the nurse perform this step might notice that the nurse looked at the label on the blood product and at the patient orders, but would not be able to discern

exactly what information the nurse looked at and what kind of mental processing the nurse did with this information. Thus, the scribe might not be able to determine that the nurse checked the blood product expiration date. Such cognitive steps could be critical for the successful completion of a process and for ensuring that no harm is caused and, thus, often need to be included in models used for deviation detection and their performance needs to be captured as a process is being executed.

Several approaches for capturing such cognitive steps could be used. One is to have a device, instead of a human, perform such a step. For example, in some medical processes, certain verifications, such as checking that the blood has not expired or making sure that the patient information on the ID band matches the patient information on the drug label, are done by bar code scanners. Such scanners can send an event once the scan has been performed. Another approach is to have process performers use a think-aloud protocol, i.e., announce out loud, when performing critical cognitive steps. A human scribe or a voice recognition system can then hear such an announcement and then issue a corresponding event to the deviation detection system.

The success of the proposed deviation detection approach depends on the accuracy and the completeness of the process model as well. If the process model captures incorrect sequences of steps or misses sequences of steps that human performers are allowed to perform, then the deviation detection approach may suffer from false positives and false negatives. Given the complexity of processes, creating a high-quality process model is challenging. As discussed in Chapter 3, we have been investigating techniques for eliciting and validating process models and applied them successfully to several real-world processes [31, 36, 93]. We believe the criticality of certain processes, such as medical processes, warrants the time and effort needed to create high-quality process models that can in turn be leveraged to support continuous process improvement via various static analyses (e.g., model checking [37], fault-tree analysis [128],

and failure-mode and effects analysis [121]) and to support deviation detection and other aspects of online process guidance (e.g., smart checklist [15]).

4.5 Deviation Explanation

Detecting a deviation and issuing a warning to process performers that an error might have occurred could be useful to prevent harm resulting from an error. Upon receiving such a warning, process performers can try to recover from the potential error before performing potentially harmful activities that they would have otherwise performed, if a deviation had not been detected. As discussed in section 4.1.1, however, depending on the level of expertise of the process performers and the complexity of the error and of the process, just a warning that an error might have been occurred might be insufficient to identify an error and recover from it. This section discusses approaches for providing additional information upon deviation detection to assist process performers with error identification and recovery. Being able to provide such assistance is particularly important in time-critical processes, where harm could be done if process performers do not recover from an error within a short period of time after the existence of an error has been recognized.

4.5.1 Error Localization

One kind of information that could be useful for error identification upon deviation detection is where in the sequence of performed steps an error occurred, i.e., at what index(es) in the sequence of performed steps a step(s) was performed that causes the sequence of performed steps to differ from the intended sequence. We call such indexes *potential error indexes (PEIs)*. As previously discussed, the intent of process performers is often unknown and, thus, determining PEIs usually cannot be done with certainty. In this section, we discuss a preliminary approach, which we call *error localization*, for identifying and ranking PEIs.

The error localization approach consists of three phases: *legal sequence selection*, *alignment computation*, and *PEI identification*. The *legal sequence selection* phase selects a set of legal sequences from the process model that are likely candidates for the sequence of steps the process performers had planned to carry out, i.e., the intended sequence of steps. Intuitively, the more similar a legal sequence is to the sequence of performed steps, the more likely it is that that legal sequence is the intended sequence. The notion of *similarity* between two sequences is essential to the proposed error localization approach. We use the *edit distance* [118] (described in more detail in section 4.5.1.1) between two sequences as a measure of similarity.

The differences between the sequence of performed steps and the selected legal sequences could suggest potential errors. To identify such differences, the alignment computation phase finds *alignments* (defined in section 4.5.1.2) between the sequence of performed steps and each of the selected legal sequences. These alignments minimize the edit distance between the sequence of performed steps and each legal sequence.

Finally, the PEI identification phase interprets the differences between the sequence of performed steps and each of the selected legal sequences to hypothesize PEIs. Each PEI is based on a pair (sequence of performed steps, legal sequence) and is ranked according to the edit distances between the sequence of performed steps and the legal sequence in that pair.

4.5.1.1 Legal Sequence Selection

After a deviation has been detected, i.e., the sequence of performed steps has been detected to not be a legal sequence through the process model, the first phase of the error localization, namely legal sequence selection, starts. This phase involves choosing a subset of all the legal sequences specified by the process model. The comparison of the legal sequences from this subset to the sequence of performed steps

will then be used to hypothesize PEIs. For a simple process model, like the one in Figure 4.2, it is feasible to obtain all possible legal sequences and compare them to the sequence of performed steps. A more realistic process model, however, can specify a very large, or even infinite, number of legal sequences and comparing all of them to the sequence of performed steps may be infeasible. Thus, criteria for selecting a subset of legal sequences are needed.

Based on the intuition given earlier, we expect that legal sequences that are more similar to the sequence of performed steps will be more useful for obtaining information about errors. We use the *edit distance* between two sequences [118] as the measure of similarity, where the edit distance is a function of the costs of the operations (often called *edit operations*) needed to transform one sequence into the other. Other measures of similarity could be chosen for the purposes of legal sequence selection, but how this choice is made requires further investigation.

Computing edit distance is expensive (the worst case complexity of the algorithms is usually quadratic in the length of the two sequences [118]) and computing the edit distance between the sequence of performed steps and a large number of legal sequences could certainly be infeasible. There are techniques, however, for computing the edit distance incrementally and discarding a large number of sequences before they need to be compared in full length to the sequence of performed steps. One such technique is to keep track of legal sequences similar to the sequence of performed steps during the breadth-first exploration of the process model that is performed during deviation detection (this technique is used in [42] for process model validation). In addition to legal sequences that match the sequence of performed steps so far (i.e., legal sequences that are exactly the same as the sequence of performed steps), other legal sequences that are within some edit distance of the sequence of performed steps can also be kept under consideration.

Legal sequence	Distance to sequence of performed steps
p	2
pq	2
ps	1
pqr	2
pst	1
pstu	1
pstuv	2

(a) Distances between the sequence of performed steps psu and selected process legal sequences.

p	p	p	p	p	p
s	s	s	s	s	s
-	u	t	u	t	-
				u	u

(b) Alignments between the sequence of performed steps (2nd column) and the closest legal sequences (1st column).

Figure 4.6: Example application of the error localization approach given the sequence of performed steps psu and the process model in Figure 4.2.

There are different kinds of edit distances, depending on the kinds of edit operations allowed. Edit operations could be used to encode different kinds of errors (e.g., deletion of a single step vs. deletion of multiple steps could encode omission of a single step vs. omission of an entire subprocess respectively). Different edit operation costs could be used to represent domain knowledge about errors, such as the likelihood or the severity of an error.

Figure 4.6(a) shows the edit distances between the sequence of performed steps psu and the legal sequences from the model in Figure 4.2 within edit distance of 2. In this example, for simplicity, only the edit operations of deletion, insertion, and substitution of a single step all with the same cost of 1 are used to compute the edit distance. In particular, the Levenshtein edit distance [87] is used to measure of similarity between the sequence of performed steps and the legal sequences from the process model. The Levenshtein distance is the minimum sum of the costs of the edit operations (where the edit operations are deletion, insertion, and substitution of a single step) needed to transform one sequence into another.

Deciding what set of edit operations and what associated costs to use depends on factors such as the availability of domain knowledge (e.g., common errors and their

Legal sequence	Distance to sequence of performed steps
abc	3
acd	1
abcd	2
acd g	1
abcde	1
acd gh	2

(a) Distances between the sequence of performed steps $acde$ and some legal sequences.

a	a	a	a	a	a
c	c	c	c	b	-
d	d	d	d	c	c
-	e	g	e	d	d
				e	e

(b) Alignments between the sequence of performed steps (2nd column) and the closest legal sequences (1st column).

Figure 4.7: Example application of the error localization approach given the sequence of performed steps $acde$ and the process model in Figure 4.3.

frequency) and the richness of the information in the process model. For instance, if it is known that process performers omit subprocess A as frequently as they omit the single step x and the process model contains information (such as hierarchical decomposition) to determine what steps are part of subprocess A , then deletion of the single step x and deletion of all the substeps of A could be used as edit operations. Furthermore, these edit operations could be given equal cost since the two corresponding errors are known to be equally likely. Empirical evaluation could also be used to choose the set of edit operations and associated costs, but how to make this choice is certainly an issue that requires further research.

4.5.1.2 Alignment Computation

Once a set of legal sequences is selected, each legal sequence in that set is compared to the sequence of performed steps to examine how that legal sequence differs from the sequence of performed steps. This is done by computing *alignments* between each selected legal sequence and the sequence of performed steps. An alignment of two sequences is a list of ordered pairs (a, b) such that (i) a is an element of the first sequence or is the “blank” element “-”, (ii) b is an element of the second sequence or is “-”, (iii) the pair $(-, -)$ does not appear in the list, and (iv) the order of the non-

blank elements in the first and second slots of the pairs in the list is the same as the order of elements in the first and second sequences, respectively. Figure 4.6(b) shows some alignments. An alignment indicates how one sequence could be transformed into the other, where the blank elements indicate that elements were inserted in one sequence or deleted from the other at the corresponding places.

Optimal alignment(s) (i.e., the alignment(s) that minimize the edit distance between two sequences) are computed by sequence comparison techniques for computing edit distances [118]. There could be more than one alignment between the same two sequences depending on the choice of edit operations and their associated costs. In fact, there could be more than one optimal alignment between two sequences for a fixed set of edit operations and costs. As mentioned earlier, this choice of edit operations and associated costs depends on various factors, such as domain knowledge, and approaches to make this choice require further research.

4.5.1.3 Potential Error Index Identification

Once alignments are computed, they can be used to obtain information about PEIs. The intuition behind using alignments to obtain PEIs is that non-matching alignment pairs (such as the shaded pairs in Figure 4.6(b)) may represent locations where an error has been committed.

For example, based on the three alignments in Figure 4.6(b), it could be hypothesized that the nurse adhered to the recommended ways to perform the process while performing the first two steps, p and s , but not after that. Thus, 3 could be a PEI as the nurse might have committed an error by performing the third step. This seems to be a reasonable hypothesis in this example as the nurse performed u (*check blood product expiration date*) when the nurse should have performed t (*verify patient's identity*) instead, which would have kept the sequence of performed steps a legal sequence.

An alignment can have more than one non-matching alignment pair and, thus, more than one PEI could be identified based on these pairs. A strategy that we currently use is to select a single PEI per alignment. This PEI is based on the first non-matching alignment pair. One reason for this strategy is the assumption that if the first non-matching alignment pair represents the first erroneous step, then subsequent non-matching alignment pairs might be less informative about possible errors. This is because after the error, the process performers might not have “returned to” the legal sequence in the alignment under consideration, assuming this was the legal sequence they were following before the deviation. Thus, comparing the suffixes of the sequence of performed steps and the legal sequence after the first non-matching alignment pair might not reveal useful error information. Deciding how PEIs should be identified from an alignment is subject to further research.

Given that the set of selected legal sequences could be large (especially for a realistic process model and a realistic sequence of performed steps), that there could be multiple alignments between each selected legal sequence and the sequence of performed steps, and that there could be multiple PEIs per alignment, the number of PEIs could be large. Providing all PEIs to process performers upon deviation detection, however, could be overwhelming rather than useful. Thus, a strategy may be needed to rank the possible PEIs in terms of usefulness for error localization.

The ranking strategy currently used in the error localization approach is based on the edit distance between the sequence of performed steps and each of the selected legal sequences. A PEI is ranked according to the minimum edit distance to a legal sequence with an alignment suggesting that PEI. Using this PEI ranking strategy and the strategy discussed above for identifying a PEI from an alignment, 3 would be the single top-ranked PEI in the sequence of performed steps *psu* in the example in Figure 4.6.

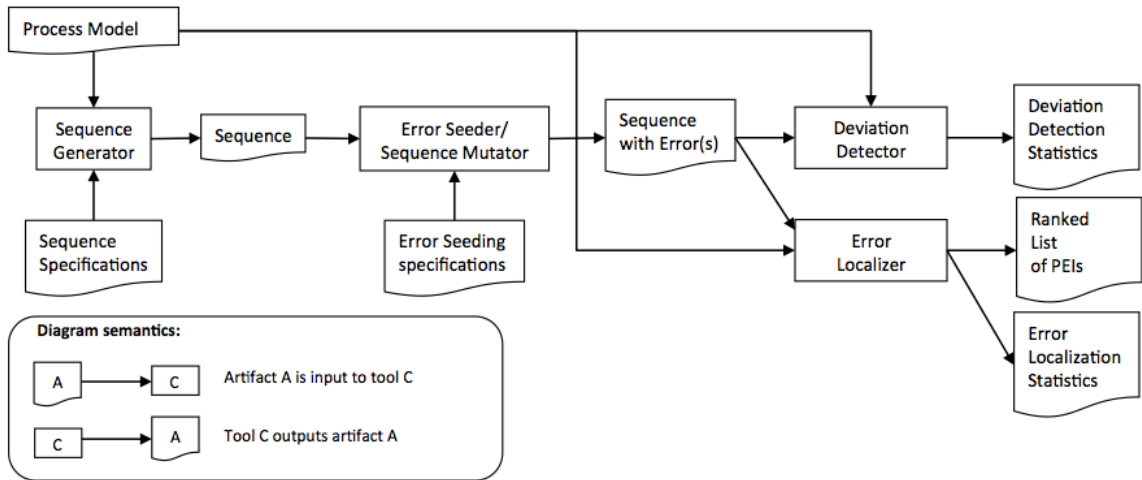


Figure 4.8: Deviation detection and error localization experimental framework.

In general, there could be multiple top-ranked PEIs, however. For example, in Figure 4.7 the alignments based on aligning the sequence of performed steps *acde* to the closest legal sequences suggest two PEIs—2 and 4. Some research questions related to PEI identification are how often there are multiple equally top-ranked PEIs and how many PEIs there are in such cases in realistic process models and sequences of performed steps. These questions are also related to the issue of delayed deviation detection since multiple highly-ranked PEIs seem to arise in situations where the deviation is detected with delay (as is the case in Figure 4.7). A preliminary investigation of these questions is discussed in the next section.

4.5.2 Evaluation of Error Localization Approach

To evaluate the error localization approach, we used the experimental framework described in section 4.2 and extended it by adding the *Error Localizer* component, which implements the three error localization phases described above. The extended framework is shown in Figure 4.8.

The Error Localizer takes as input a Process Model and a sequence of steps that the Deviation Detector has determined to not be a legal sequence through the Process

Model. The Error Localizer outputs a ranked list of potential error indices (PEIs) for the given sequence of steps and can also be instrumented to collect error localization statistics, such as percentage of sequences with multiple PEIs and average number of PEIs per sequence.

The current implementation of the Error Localizer uses a brute-force approach to accomplish the Legal Sequence Selection phase—the Error Localizer simply obtains all legal sequences up to a certain length. This brute-force approach does not scale for process models containing a large number of sequences and this limitation is discussed in section 4.5.2.4. As mentioned in section 4.5.1.1, however, there exist approaches for incrementally computing the edit distance between the growing sequence of performed steps and legal sequences from the process model, discarding legal sequences whose edit distance to the sequence of performed steps exceeds a threshold. Utilizing such an approach during the Legal Sequence Selection phase is an interesting direction for future research.

To compute alignments between the sequence of performed steps and the set of legal sequences obtained during the Legal Sequence Selection phase, the Error Localizer uses a standard dynamic programming algorithm ([118], page 23) to compute the edit distance between two sequences, which also computes the optimal alignments associated with the optimal edit distance. The current implementation of the Error Localizer uses the Levenshtein edit distance [87], where different weights for the deletion, insertion, and substitution edit operations can be set.

Once alignments between the sequence of performed steps and the set of selected legal sequences are computed, PEIs are identified and ranked as described in section 4.5.1.3.

4.5.2.1 Experimental Design

To perform an initial evaluation of the proposed error localization approach, we applied it to the model of the blood transfusion process used for the evaluation of the deviation detection approach and described in section 4.3.1.1. We replaced each of the *verify patient's identity* and *specimen labeling* subprocesses by a single step to make the model smaller. We did that because the current implementation of the Legal Sequence Selection phase of the error localization approach is brute force, i.e., it finds all sequences of steps through the model up to certain length. Removing the above two subprocesses reduced the size and the complexity of the model, allowing for longer sequences of steps to be obtained in a reasonable amount of time during the Legal Sequence Selection phase.

The resulting model was still of significant size and complexity. It contained 144 Little-JIL steps (53 leaf steps) and it specified 18 exception handling situations. All legal sequences consisting of up to 15 leaf steps (a total of 164 such sequences) were generated⁸. These generated legal sequences were then mutated to represent sequences of performed steps where the process performers have made an error. We applied the same single-step mutations—deletion, insertion, and substitution of a single step at (almost) every position of every legal sequence—as in the evaluation of the deviation detector. These mutations are described in detail in section 4.3.1.2.1. For all three mutation kinds, mutated legal sequences (mutants) that remained legal sequences through the process model were discarded. Mutating the generated legal sequences resulted in more than 1,700 mutants for each mutation kind (Figure 4.9).

⁸Only leaf steps were included in generated sequences, because in Little-JIL, the leaf steps are the ones that agents perform and are thus most likely the steps to be recorded. Non-leaf steps are primarily used to provide abstraction and specify control flow among leaf steps in a Little-JIL process model. The length of the generated sequences could increase by up to 9 steps if non-leaf Little-JIL steps are counted, depending on the depth of the subprocesses in which the leaf steps in a given sequence are located.

	Del.	Ins.	Subst.
Number of generated mutants	1762	1926	1926
Percentage of mutants with single PEI	74.3%	97.5%	98.9%
Number of mutants with single PEI that does not match the mutation index	0	0	0
Percentage of mutants with multiple PEIs	25.7%	2.5%	1.0%
Number of mutants with multiple PEIs that do not contain the mutation index	0	0	0
Average number of PEIs for mutants with multiple PEIs	2.1	2.0	2.0
Maximum number of PEIs for mutants with multiple PEIs	3	2	3

Figure 4.9: Applying the error localization approach to a blood transfusion process model.

The error localization approach was applied to each mutant and statistics related to PEI identification were collected. To compute PEIs, all legal sequences from the process model up to 17 steps long were selected to be compared against the mutants. The Levenshtein distance was used as a sequence similarity measure. The edit operations were deletion, insertion, and substitution of a single step with equal cost of 1. The PEIs for each mutant were based on the alignments between that mutant and the legal sequences that have minimum distance to it. For each such alignment, a single PEI was produced based on the first non-matching alignment pair.

4.5.2.2 Results

Figure 4.9 shows the results of applying the error localization approach to the blood transfusion process.

4.5.2.3 Discussion

For most of the insertion and deletion mutants (more than 97.5%), a single top-ranked PEI was identified. For almost 75% of the deletion mutants a single top-ranked PEI was identified. The percentage for deletion mutants is lower, again, mostly due to mutations in shuffle subsequences. For all three kinds of mutants with a single top-ranked PEI, that PEI was the same as the mutation index. These results mean that in most cases the error localization approach was able to accurately identify the location of the error in the sequence of performed steps. Furthermore, in these cases

there were no spurious top-ranked PEIs that could distract process performers in diagnosing the error. In the cases when there were more than one top-ranked PEIs, the number of PEIs was small—maximum 3 and less than 2.1 on average—and the mutation index was always among the top-ranked PEIs.

4.5.2.4 Threats to Validity

The threats to the validity of the evaluation of the deviation detection approach, discussed in section 4.3.4, also apply to the evaluation of the error localization approach, given that the experimental design was essentially the same. The fact that only a single process model was used to evaluate the error localization approach is a further threat.

As previously discussed, the Legal Sequence Selection phase was implemented by finding all legal sequences of steps up to a certain length. In particular, all legal sequences of up to 17 steps were found⁹. Given that the longest generated mutant was 16 steps long (the longest generated sequence was 15 steps long and the single-step mutations can increase the size of a sequence by at most one step), the original sequence from which a mutant was created was always among the set of legal selected sequences. This inevitably affected favorably the error localization results in Figure 4.9. In a real-world situation, however, where large and complex HIPs are involved and process executions could be much longer than 16 steps, an error localizer would not have the luxury to compare the sequence of performed steps against a set of legal sequences that is guaranteed to contain the intended sequence of steps, which would diminish the accuracy of the error localization results.

Since the mutations applied to the generated sequences of steps were known, we knew what corresponding edit operations to use for computing edit distances utilized

⁹As mentioned in section 4.5.2.1, the generated sequences contained only Little-JIL leaf steps, because in Little-JIL leaf steps correspond to the activities done by process performers.

in the error localization approach. When the error localization approach is applied to a real executing process, however, the errors that could occur might not be known in advance and, thus, selected edit operations might not be as useful as in our evaluation for accurately identifying PEIs.

4.5.3 Limitations of the Error Localization Approach

The limitations of the deviation detection approach discussed in section 4.4 are also limitations of the error localization approach—the error localization approach needs to receive from the Process Execution Monitor an accurate sequence of performed steps of interest and the success of the approach also depends on the accuracy and the completeness of the process model.

The error localization approach is highly dependent on the selection of edit operations and their costs, which are used in the computation of edit distance. If the selected edit operations do not accurately represent the kinds of errors that might occur during a process execution and/or the costs of these edit operations do not reflect characteristics of process errors, such as frequency and severity, then the top-ranked PEIs computed by the error localization approach might not represent the locations in the sequence of performed steps where errors have occurred. Thus, it is important that domain knowledge about process errors is available when the error localization approach is applied to a real executing processes. Ways to obtain information about process errors include discussions with domain experts, surveying literature on common errors in the domain, process vulnerability analysis approaches such as fault-tree analysis [128] and failure mode and effects analysis [121], and mining process execution logs. The deployment of a deviation detection and explanation system could contribute to obtaining higher quality process execution logs than currently existing ones as the Process Execution Monitor would need to collect various events of interest to support deviation detection and explanation.

CHAPTER 5

VISUALIZATION OF PROCESS EXECUTION STATE

5.1 Overview

Providing information about the state of a system is a widely used practice to assist humans who operate within complex systems. For example, consider the system consisting of vehicles operated by human drivers on a given road. A typical vehicle has means to provide the driver with various kinds of information about the state of different components of the system for the sake of the safety of everyone on the road and for efficiency. The various indicators on the dashboard (e.g., fuel level indicator, oil level indicator, tire pressure indicator, engine temperature indicator) provide the driver with information related to the state of the vehicle to help the driver make decisions whether it is safe to continue to operate the vehicle and if not, what the potential problems are, so that the driver can plan how to address these problems. The speedometer provides the driver with information to help decide whether the driver is complying with a regulation established within the system, namely the speed limit. The mirrors provide the driver with information about the state of other components of the system, namely the position of other vehicles, to assist the driver with planning maneuvers on the road. The seat-belt signs and audio warnings provide the driver with information related to the safety of the humans in the vehicle. GPS units, which are widely used nowadays, provide driving directions information that could help the driver plan road maneuvers ahead of time and could also be very helpful for navigating in areas new for the driver, including detours that the driver needs to take in case some exceptional condition arises, such as temporary road closures.

We believe that such an approach for providing information about the state of a system could be particularly useful for performers of HIPs, especially given the criticality of some HIPs (e.g., medical processes) and their increasing complexity. Such an approach could help reduce the number of errors in HIPs and improve their efficiency. For example, consider the HIP of treating a patient in a hospital. This HIP involves the patient and could involve various medical professionals, such as physicians, nurses, and medical assistants, various kinds of medical equipment, such as X-ray machine and lab equipment, and various information systems such as an electronic medical record system, a scheduling system, and a computerized order entry system. Information about the execution state of that HIP could be useful to the medical professionals carrying it out. For instance, a list of activities that medical professionals need to perform can reduce the number of omission errors committed by these medical professionals. Information about the progress of activities that others are performing and who is performing them could help each medical professional determine the overall progress of the process, improve team communication, and facilitate planning. Information about deadlines, such as that the blood product needs to be administered within certain period of time after it has been prepared by the blood bank, could reduce errors related to timing constraint violations. Information about process execution history, such as who performed which activity using what resources and at what time, could help process performers understand exceptional conditions and choices that might affect future decision. Process execution history information could also be helpful for identifying errors, identifying the root causes of errors, and planning error recovery. Information about resource availability, such as the availability of an X-ray machine, could help with scheduling, thus improving the efficiency of the HIP.

Providing information about the state of an executing HIP to process performers, however, is challenging. It is difficult to decide what information about process

execution state would be useful to process performers at different points of a process execution, to design effective process state visualizations, and to determine the state of an executing HIP.

Deciding what process execution state information to show process performers. The execution state of a HIP has multiple components. These include the state of process activities (e.g., performed, currently being performed, pending), the state of the resources utilized in the process (including who is responsible for which process activities), the state of the artifacts that have been produced in the course of the process execution, problems that have arisen and the status of the handling of these problems. The execution state of a HIP can also have domain-specific components, such as the physiological condition of a patient who is part of a medical HIP.

Deciding what information pertaining to a process execution state component and what combination of these components would be useful to show process performers is challenging. The decision might need to depend on multiple factors, such as characteristics of individual process performers (e.g., role and level of expertise), the current process execution state itself (e.g., if a problem arises, then different kinds of information might need to be shown compared to when the process execution is nominal), and the domain. Furthermore, in some circumstances it might be useful to show information about not only the current process execution state, but also information about past or even possible future process execution states.

Designing visualizations of process execution state. Besides deciding what process execution state information to show process performers, it is also important to consider how this information could be effectively presented to process performers. There are multiple ways, for example, to visualize process activities and their relationships, to make salient pending deadlines or problems that arise during a process

executions, and to display information related to process resources. The design of effective process execution state visualizations needs to involve the use of human factors approaches, user studies to empirically evaluate alternative visualizations, and consideration of visual metaphors with which process performers in a given domain are already comfortable.

Determining process execution state. To show information about the state of an executing HIP, that state needs to be first determined. To keep track of the changing state of a HIP as that HIP is being executed, a process execution monitor, like the one used in the Deviation Detection and Explanation approach (Chapter 4), is needed. This process execution monitor needs to be able to capture process execution events. The human element of HIPs, however, poses difficulties for capturing such events. For example, determining when certain process activities have been performed could be very difficult, because certain process activities performed by humans are difficult to observe. Cognitive steps, discussed in section 4.4, are an example of such activities.

In addition to capturing process execution events, a representation of process execution state is needed so that this representation can serve as the basis for selectively providing process execution state information to process performers. This representation needs to be rich enough to capture the various aspects of process execution state discussed above. In particular, this representation needs to be able to capture process activities (past, current, and possibly even upcoming), information about artifacts created or resources utilized during the execution of the process, as well as information about problems that might have arisen and how they have been handled. Appropriately updating the state of such a representation based on the events received from a process execution monitor and keeping the state of that representation “in sync” with the state of the unfolding process is also an important challenge.

5.2 The Smart Checklist Metaphor

To address the problem of human errors in HIPs discussed in Chapter 1, we are investigating an approach that uses a *smart checklist* metaphor for visualizing process execution state. This approach is based on the observation that checklists have been long used to successfully support human process performers of HIPs in certain domains, such as aviation and space. Checklists have also been recently introduced in HIPs from other domains where human errors can have critical consequences, such as medical processes [22, 67], but these checklists are known to have important limitations [68, 132].

The smart checklist approach aims to address these limitations. A smart checklist, like a traditional checklist, focuses on the process activities. To guide process performers during a process execution, the smart checklist shows activities that process performers need to do, activities that have already been performed, and potentially activities that might need to be performed in the future. Unlike traditional checklists, however, which tend to be static and to specify only the major steps during normal flow, omitting important details such as exceptional scenarios and concurrent process execution [31, 70], the smart checklist is dynamic, context-sensitive, and it provides guidance on a larger set of process scenarios. This is achieved by basing the smart checklist on a detailed process model and relying on a mechanism for capturing a rich set of process execution events as a process is being executed. In addition to showing process activities, the smart checklist provides capabilities for accessing other kinds of process execution state information, such as the resources needed to perform activities, the artifacts produced by activities, problems that have arisen while performing activities and how such problems should be or have been handled.

The smart checklist approach is part of the process improvement environment discussed in Chapter 2 and shown in Figure 2.1. The Interpreter takes as input a Process Model and feeds the Process State Visualizer with the next step(s) that need to be

performed based on the current execution state of the Process Model. A step in this context encapsulates not only the name of an activity, but also other associated information, such as resources needed to perform that activity and problems (exceptions) that might arise while performing that activity. Given this information, the Process State Visualizer issues the appropriate updates to the corresponding visual elements of the Process State Visualization (represented by the visuals to the immediate left of the stick figures in Figure 2.1).

As process performers are executing the process, they interact with the Process State Visualization. There could be different instances of that visualization. Some instances could be tailored to individual process performers and each such instance would be used only by the corresponding individual. Other instances could be used by multiple process performers and could show information related to the activities done by all these performers. Process performers could use the Process State Visualization to see what steps need to be done and access other process execution state information. Process performers could also indicate that a step has been successfully completed or that problems have arisen while performing the step and what these problems are. The Event Interaction Manager captures the input of process performers and then sends the appropriate events to the Interpreter to inform the interpreter how the process execution has advanced (e.g., done steps, or problems that have arisen). The Interpreter then consults the Process Model and, given the current execution state of the Process Model and the newly received information, the Interpreter determines what steps need to be done next or whether the process has been completed. The Interpreter then feeds the Process State Visualizer with the next step(s) that need to be performed and starts the cycle described above again.

The Process State Visualizer could query the Retrospector to obtain and then visualize information about the process execution history, such as performed activities, time when they were performed, problems that have arisen and how they have

been handled, and resources that have been used while performing process activities. Similarly, the Process State Visualizer could query the Prospector to obtain information about the potential future of the current process execution, such as possible upcoming activities and resources that might be needed.

The next section presents an initial prototype of the smart checklist approach and section 5.4 discusses a preliminary evaluation of that approach with respect to the above issues.

5.3 The Smart Checklist Prototype

5.3.1 Back-end Implementation

The smart checklist prototype is implemented within the Little-JIL process execution environment [27]. In this environment, the Little-JIL interpreter takes as input a Little-JIL process model and based on that model assigns work to process performers (called *agents* in Little-JIL terminology). A work item is encapsulated in a Little-JIL step, which contains the name of the activity to be performed, but also additional information such as any resources needed to perform that activity, artifacts that the activity needs to produce, and problems (called *exceptions* in Little-JIL terminology) that might arise while the agent is performing the activity and that the agent can report to the Little-JIL interpreter. A step is assigned to an agent by placing that step on the agent's *agenda*. The *Agenda Management System (AMS)* is the interface between agents and the Little-JIL interpreter, defining the communication protocol between them. The AMS is the instantiation of the Event Interaction Manager in the prototype implementation of the smart checklist approach.

5.3.2 Visualization

As a first step towards designing, implementing, and evaluating the smart checklist approach, we concentrated on a visualization for a single process performer. This

visualization focuses on the activities that are assigned to that process performer. As previously discussed, however, the smart checklist is intended to support other kinds of visualizations, such as ones that could be used by multiple process performers at the same time and that contain process execution state information relevant to multiple process performers. Such visualizations are left for future work.

Figure 5.1 shows the smart checklist GUI automatically generated from the Little-JIL process model of the simplified blood transfusion process shown in Figure 2.3 and discussed in Chapter 1. This GUI is from the perspective of the nurse who is about to start performing the blood transfusion process and has not performed any steps yet. The visualization in Figure 5.1 consists of three main parts—the domain-specific top panel, the process header panel, and the bottom checklist panel. For this particular example from the medical domain, the domain-specific top panel contains the personal information of the patient who is receiving the transfusion as well as patient physiological data. The process header panel contains the name of the process (“blood transfusion process”), the status of the process (in this example, “in progress”), and a notes button that the process performer can use to enter notes about the current process execution.

Below the process header panel is the checklist panel. The checklist panel shows process steps that need to be (or have been) performed. To provide context to process performers, the checklist panel also shows the hierarchical decomposition of process steps. A concrete step (corresponding to a leaf Little-JIL step¹) is shown as a solid rectangle. Subprocesses (corresponding to non-leaf Little-JIL steps), which contain concrete steps and can also contain other subprocesses, are shown as transparent rounded rectangles enclosing the contained concrete steps and subprocesses. The

¹As discussed in Chapter 4, in Little-JIL, the leaf steps are the ones that agents perform. Non-leaf steps are used primarily to provide abstraction and specify control flow among leaf steps in a Little-JIL process model.

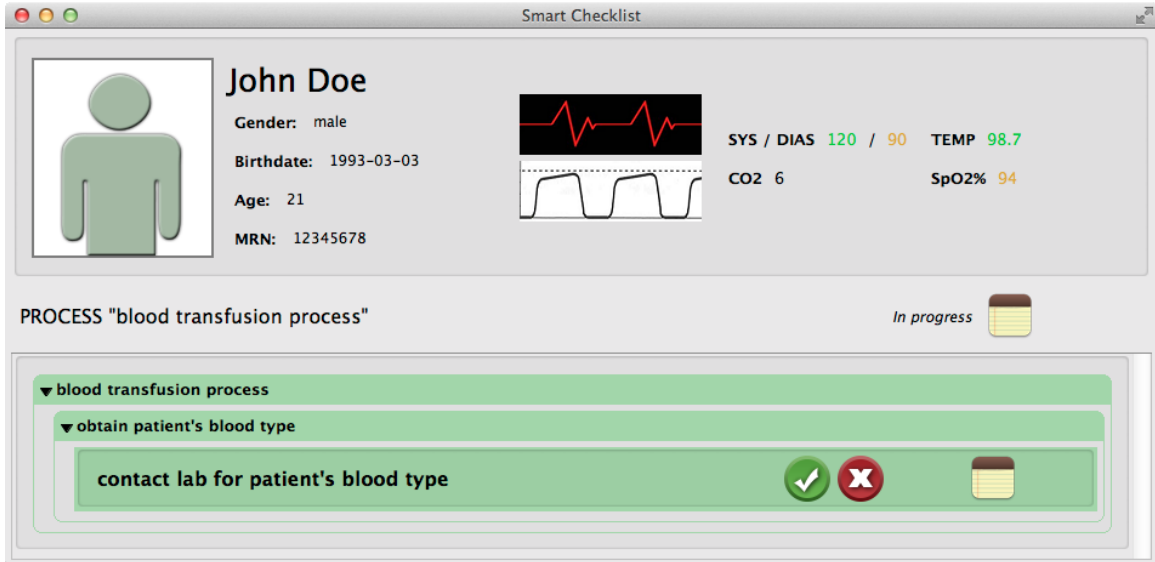


Figure 5.1: The smart checklist at the beginning of executing the simplified blood transfusion process.

first concrete step that the nurse needs to perform according to the smart checklist in Figure 5.1 is *contact lab for patient's blood type*. This step is part of the subprocess *obtain patient's blood type*, which in turn is part of the overall *blood transfusion process*.

Each concrete step has several buttons that the process performer can use to interact with the smart checklist. The rounded button with the checkmark can be clicked to indicate that a step has been successfully completed. Steps during whose performance a problem can arise have a rounded button with an “X” on it (this button is explained later in this discussion). The rounded-rectangular note button (the right-most button on the rectangle corresponding to the step *contact lab for patient's blood type* in Figure 5.1) can be clicked to bring up a text editor that allows process performers to write notes related to the performance of the current step.

Figure 5.2 shows the smart checklist for the simplified blood transfusion process after the nurse has already performed several steps. Once a step has been performed, the buttons on the right are replaced with a status indicator symbol and a time stamp. The status indicator symbol could be a checkmark, indicating that a step has been

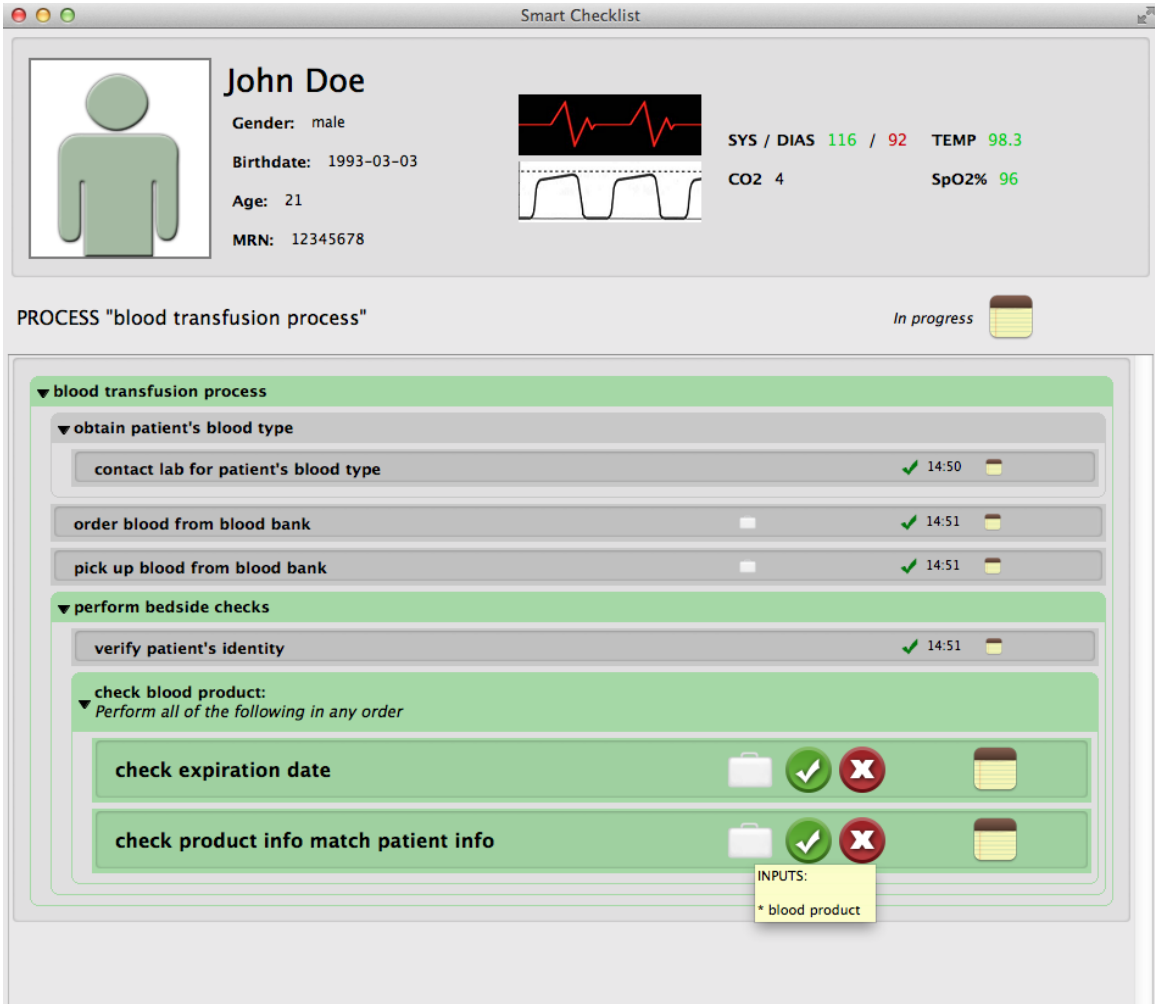

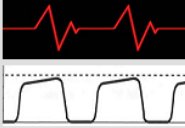


Figure 5.2: The smart checklist after the nurse has performed several steps from the simplified blood transfusion process.


Smart Checklist



John Doe
 Gender: male
 Birthdate: 1993-03-03
 Age: 21
 MRN: 12345678






SYS / DIAS 117 / 91 TEMP 98.4
 CO2 6 SpO2% 94

PROCESS "blood transfusion process" Successfully completed ✓ 14:52 

▼ blood transfusion process

▼ obtain patient's blood type

contact lab for patient's blood type	<input checked="" type="checkbox"/>	14:50	
order blood from blood bank	<input type="checkbox"/>	14:51	
pick up blood from blood bank	<input type="checkbox"/>	14:51	

▼ perform bedside checks





verify patient's identity	<input checked="" type="checkbox"/>	14:51	
▼ check blood product: <i>Perform all of the following in any order</i>			
check expiration date	<input type="checkbox"/>	14:52	
check product info match patient info	<input type="checkbox"/>	14:52	
infuse blood	<input type="checkbox"/>	14:52	

Figure 5.3: The smart checklist after the nurse has successfully completed the simplified blood transfusion process.

successfully completed, or a red “X”, indicating that problems arose while performing the step. The time stamp indicates the time when the step was successfully completed or the time when the problems arose. For example, in Figure 5.2, the steps *contact lab for patient’s blood type*, *order blood from blood bank*, *pick up blood from blood bank*, and *verify patient’s identity* have all been successfully completed and, thus, their buttons on the right have been replaced by the status indicator for successful completion, the checkmark, and a time stamp next to that checkmark. The height of the rectangle corresponding to steps that have already been completed is also decreased to save vertical space. Once a step has been performed, its corresponding solid rectangle changes background color to distinguish performed from currently active steps. Currently active steps are shown in green, whereas grey background indicates that a step has already been performed².

Figure 5.2 illustrates two more visualizations part of the smart checklist—support for shuffled steps and artifacts. The steps *check expiration date* and *check product info match patient info* are shuffled steps, meaning that the nurse needs to perform both of them, one after the other, but they can be performed in any order. The smart checklist visualizes this shuffle concept by showing all shuffled steps as currently available to perform (indicated by their corresponding green rectangles with rounded buttons on the right) and by showing the text “Perform all of the following in any order” on the subprocess that encloses the shuffled steps.

The briefcase icon on some of the steps in Figure 5.2 represents resources that are needed to perform the step. For example, clicking on the briefcase icon on the step *check product info match patient info*, brings up a list of resources needed for that step—in this case there is only one resource on the list, namely the *blood product*.

²The colors may not be easily distinguishable if the figures are printed in black and white, but the substitution of the buttons on the right of the step rectangle with a status indicator and a time stamp are visual cues that unambiguously distinguish currently active from already performed steps.

Figure 5.3 shows the smart checklist after the nurse has successfully completed the blood transfusion process. All steps in the checklist panel that had buttons on the right now have a status indicator and a time stamp instead. Also, the background color of all steps is now grey. Furthermore, the process status in the process header also changes—it now says “Successfully completed” and it shows a status indicator icon (a checkmark) and a time stamp for the overall process completion time.

Figures 5.4, 5.5, and 5.6 show a scenario where problems arise while the nurse is performing the blood transfusion process. Figure 5.4 shows the smart checklist at the same stage of the process as Figure 5.1, but this time when the nurse contacts the lab for the patient’s blood type, the patient’s blood type is unavailable. The nurse can bring up a list of potential problems that could arise during the performance of a step by clicking the rounded button with the “X”. In the example in Figure 5.4, this list contains only a single problem—“patient blood type unavailable”—but, in general, there could be more problems, depending on how many exceptions are specified in the corresponding step in the process model.

Once the nurse indicates that the “patient blood type unavailable” problem has arisen, the smart checklist dynamically updates the next steps that need to be performed. Figure 5.5 shows the smart checklist for the scenario when this problem has arisen and the nurse has subsequently performed several other steps. Note that unlike the scenario in Figure 5.2 where the next step to perform after *contact lab for patient’s blood type* is *order blood from blood bank*, in the scenario in Figure 5.5 the nurse needs to perform the *test patient’s blood type* subprocess. This subprocess consists of two substeps—*obtain blood specimen from patient* and *send blood specimen to lab for testing*—and needs to be performed to handle the problem that the patient’s blood type is unavailable. After the nurse tests the patient’s blood type, the process execution goes back to normal process flow with the step *order blood from blood bank* to be performed next.

In the scenario shown in Figure 5.5, another problem arises, this time while the nurse checks that the information on the blood product matches the patient information—this information does not match and the nurse indicates that product check has failed. The smart checklist then suggests that the blood transfusion process should be terminated as there is the risk of giving wrong blood to the patient. In the snapshot of the smart checklist in Figure 5.6, there are no more active steps, the process status message in the process header panel has updated to “Failed to complete because of problems”, and the process status indicator icon next to that message is now an “X”. In this simplified example, the process model we wrote assumes that the process terminates after a problem arises during the subprocess of checking the blood product. A more realistic process model, however, would specify behavior for handling problems that arise during the blood product check. If this is the case, the smart checklist would provide guidance to the nurse based on this specified behavior in the process model.

5.4 Preliminary Evaluation

We have performed a preliminary evaluation of the smart checklist approach and, in particular, of the single-agent smart checklist visualization. In the process of building the smart checklist prototype, several informal presentations of the smart checklist were given to a medical doctor, a nursing professor, an industrial engineering professor, and several computer science professors and students (both graduate and undergraduate), to seek their comments and update the prototype accordingly. The initial prototype was formally demonstrated to a panel of medical experts (four medical doctors and a registered nurse (RN)), who would be potential users of such a smart checklist, and they were asked for feedback.

The example process used during the panel meeting was the subprocess of setting up an infusion pump for patient-controlled analgesia (PCA), usually performed

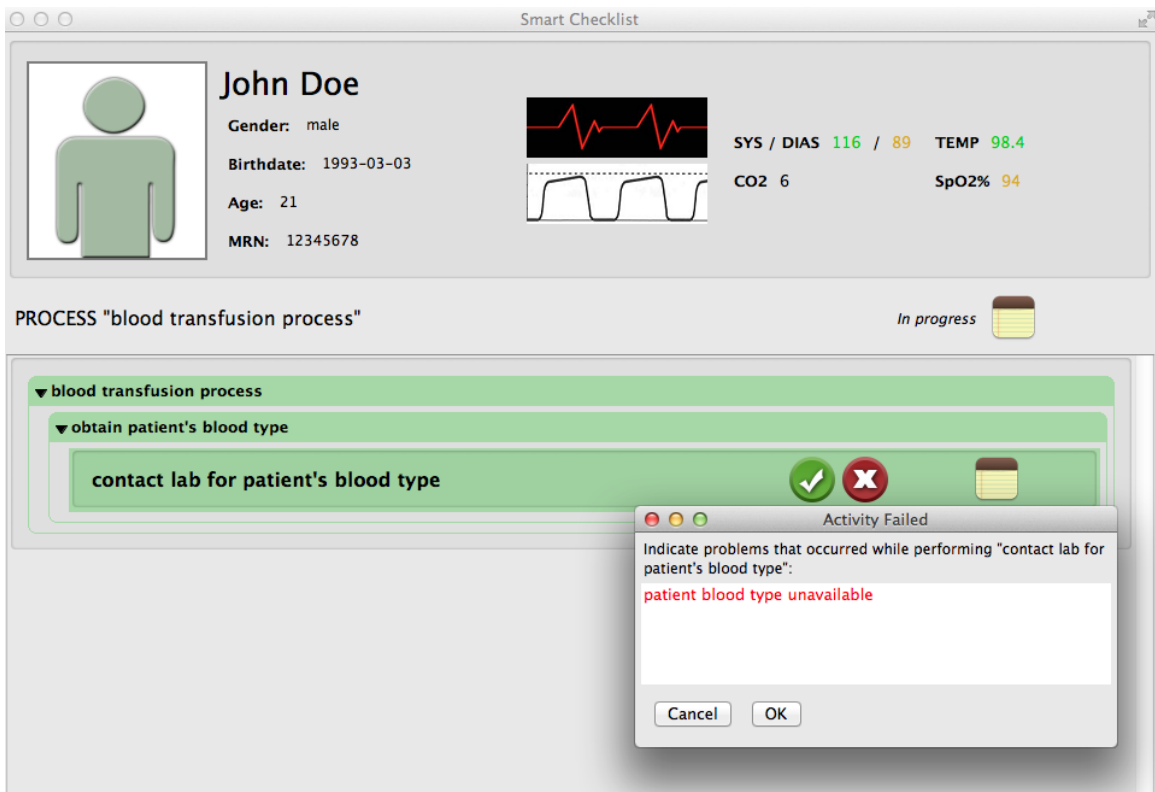


Figure 5.4: The smart checklist at a point of the executions of the simplified blood transfusion process where the nurse is about to indicate that problem has arisen.

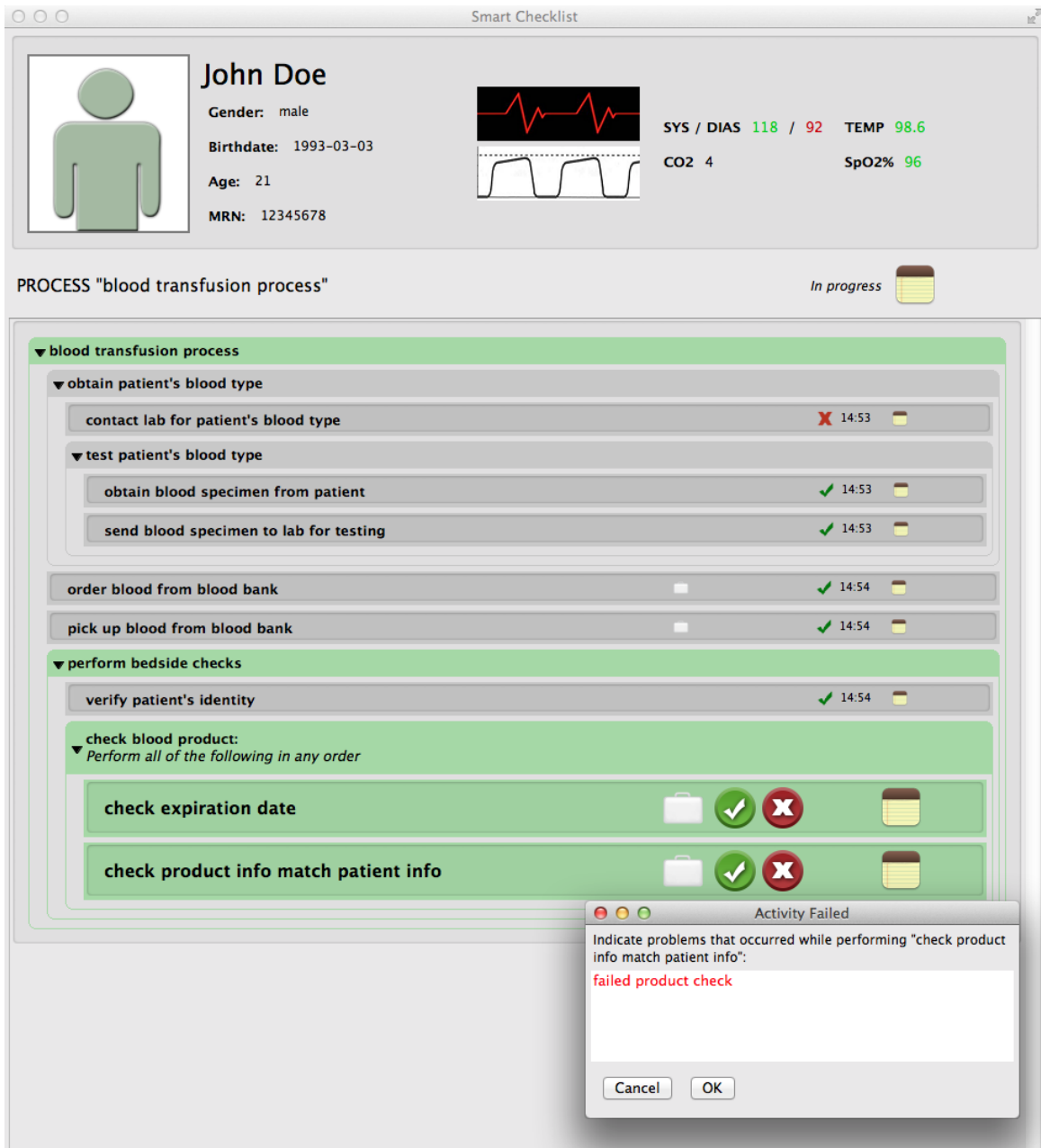

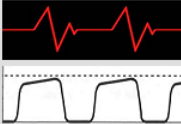


Figure 5.5: The smart checklist after the nurse has performed several steps from the simplified blood transfusion process and is about to indicate that another problem has arisen.


Smart Checklist



John Doe
 Gender: male
 Birthdate: 1993-03-03
 Age: 21
 MRN: 12345678




SYS / DIAS 117 / 90 TEMP 98.4
 CO2 5 SpO2% 95


PROCESS "blood transfusion process" Failed to complete because of problem(s) **X** 14:54 


▼ blood transfusion process


▼ obtain patient's blood type


contact lab for patient's blood type **X** 14:53 

▼ test patient's blood type


obtain blood specimen from patient ✓ 14:53 

send blood specimen to lab for testing ✓ 14:53 


order blood from blood bank ✓ 14:54 

pick up blood from blood bank ✓ 14:54 

▼ perform bedside checks

verify patient's identity ✓ 14:54 

▼ check blood product:
Perform all of the following in any order

check expiration date 


check product info match patient info **X** 14:54 

Figure 5.6: The smart checklist after the simplified blood transfusion process was not successfully completed due to problems during the execution of the process.

by a nurse as part of an overall infusion therapy process. The subprocess of setting up the infusion pump includes activities such as obtaining the pump, assessing patient infusion-related hazards, physically preparing the pump, and performing various safety checks before the infusion can be started. This process was captured in the Little-JIL process modeling language [27] (Little-JIL is described in detail in section 2.3) and the resulting process consisted of 37 Little-JIL steps, 24 of which were leaf steps.

A smart checklist was automatically generated from the Little-JIL model of the subprocess of setting up the infusion pump. Several process scenarios, including scenarios where problems arose (such as there are patient infusion-related hazards and extra work needs to be done to assess whether it is safe to perform the infusion) were shown to the panel of domain experts. The RN in the panel had extensive experience with performing the process of setting up the infusion pump and the medical doctors had experience with ordering such infusions and overseeing their execution. The domain experts in the panel were encouraged to provide open feedback on the smart checklist and our research group had also prepared a detailed list of questions related to the issues discussed in section 5.1.

The smart checklist was well received at the panel meeting as well as at the informal demonstrations. The domain experts recognized the potential of the approach to reduce errors, improve process documentation, and serve as a training aid. In the rest of this section, we discuss the smart checklist approach with respect to the issues presented in section 5.1 and based on the feedback obtained during the preliminary evaluation.

Deciding what process execution state information to show process performers. The domain experts confirmed that it would be useful to show past process activities in addition to the activities that need to be currently performed. They

thought that being able to access the process execution history would be particularly helpful during hand-offs.

Some of the domain experts suggested that the smart checklist should support different modes, depending on the level of expertise of the process performers who use it. For example, a smart checklist for novice process performers can provide more detail than a smart checklist for expert process performers. The domain experts pointed out, however, that a detailed list of possible problems that might arise during the performance of a step would be useful even for expert process performers.

As previously discussed, the smart checklist GUI in the current implementation of the prototype is from the perspective of a single process performer. The domain experts confirmed our expectations that in some situations it would be useful to access the tasks of other performers as well to get a better sense of the overall execution state of the process. Furthermore, it might be useful if the smart checklist supports ways for communication between different process performers that are working together as a team, even suggesting integration of the smart checklist with existing hospital communication systems.

Designing visualizations of process execution state. Domain experts found the step hierarchy useful as a way to visualize the structure of the process. Some of them mentioned that the hierarchy would be particularly useful when browsing through the process execution history as this history could be potentially long, but the hierarchy may not be necessary for the currently active steps, as usually there will be a small number of such steps. In fact, some of the domain experts proposed providing an even stronger distinction between current and past process steps—for instance, by placing them on different tabs as opposed to having them in the same scrollable hierarchical list.

The icons used in the smart checklist prototype were controversial. Some domain experts found them intuitive, whereas others did not. For instance, the RN who

participated in the panel meeting mentioned that the green checkmarks and red “X”s were similar to the ones that appear on the interface of a pump that this RN had used. Some of the physicians, however, found the checkmark and the “X” icons confusing—one physician, for example, pointed out that it is not clear whether a red “X” means that a step was not performed or that the step was performed, but unsuccessfully. There will not be a single set of icons that will be intuitive for all users and, thus, training will be required regardless of which set of icons is selected. The iconography used on the smart checklist GUI, however, requires further research.

The current smart checklist prototype supports several customizations, including user-selected colors for representing the different states of process steps and deciding which intermediate Little-JIL steps to hide from the step hierarchy shown in the smart checklist GUI. The domain experts found the support for customization useful, but pointed out that decisions about what should be customizable and by whom should be made very carefully. For example, not every process performer should be allowed to change the colors of process step states, because this might be confusing to another process performer taking over after a handoff. Different hospitals, however, might want to use different colors, perhaps based on the user interfaces of other systems deployed in a particular hospital. Thus, customizing the colors of steps on the smart checklist should be allowed to an administrator responsible for deploying the system in a given hospital.

Determining process execution state. Our focus for the initial smart checklist prototype and its evaluation was on the first two issues described above, namely deciding what information related to process execution state to show and how to visualize this information. To determine the state of an executing process, the current prototype of the smart checklist makes the simplifying assumption that process performers would provide *all* process execution events of interest by interacting with the smart checklist GUI. For example, for every successfully completed step, process

performers need to indicate that the step has been successfully completed by clicking the green button with checkmark. Reporting all process execution events, however, would add to the workload of already busy process performers. Thus, in future research we plan to investigate ways to automatically recognize some process execution events. For example, if a nurse starts an infusion pump, that pump could directly send an event to the smart checklist that the step *perform infusion* has been started. To automate the capturing of some process execution events, the smart checklist approach would benefit from a mechanism for monitoring a process execution like the one required for the deviation detection and explanation approach and discussed in section 4.4.

An interesting issue related to keeping track of process execution state is support for *undo*. Both at the demonstration to the panel of domain experts and at other informal demonstrations of the smart checklist prototype, it has been requested that the smart checklist allows process performers to roll back the smart checklist by indicating that an event that has been reported to have happened has not actually happened. For example, if process performers have indicated via the smart checklist GUI by mistake that they have performed a step, they should be able to indicate that, in fact, this step has not been performed. There are important concerns that need to be considered, however, when providing support for undo. Unlike a word processor, where any editing action, such as typing or deleting a character, can be easily undone and redone, in the domain of HIPs it is not clear what it means for a step to be undone. For example, if a medication has been administered to a patient, that medication cannot easily be “unadministered” to the patient. Even if a step is not performed in reality, supporting undo might still need to be done with extra care, because the sheer indication that a step has been performed might have side effects and influence the process execution. For instance, suppose a nurse indicates by mistake that the step “administer medication” has been performed, but the medication was never

given. The nurse would then want to undo the step “administer medication”, but if indicating that this step has been done triggered the smart checklist system to show other process performers that they can start new activities, and some of these activities are actually started, undo becomes problematic. Thus, the support for undo in the context of HIPs is a challenging issue that requires further investigation.

Another interesting issue related to determining the state of an executing process is determining possible future executions. Knowing what steps are coming up ahead of time can help process performers better plan their work. It is difficult, however, to predict how a process execution will unfold and, thus, some strategies are needed to determine which of the possible future executions should be shown to process performers. Furthermore, there seems to be a controversy between domain experts of whether showing information about potential future process executions would be helpful. Some of the domain experts with whom we interacted, for example, suggested that it might be useful to be able to see on the smart checklist GUI what steps in the process are left to be done, whereas others were skeptical of how accurately future executions can be predicted. This issue about determining and showing possible future executions also requires further research.

CHAPTER 6

RELATED WORK

This chapter starts with an overview of related work on process elicitation and modeling. It then continues with a discussion of existing approaches for online process guidance, including recent work on visualization of process execution state and work on traditional checklists and process guides. Existing approaches for dealing with process deviation are presented next, followed by a discussion of approaches related to error localization. The chapter concludes with an overview of work on characterizing and classifying human errors.

6.1 Process Elicitation and Modeling

6.1.1 Process Elicitation

Eliciting information about complex HIPs is important for reasoning about them and for potentially improving them. Methods to facilitate such elicitation have been proposed. For example, Woods and Hollnagel discuss detailed observations of activities of process performers in context as a key method of discovering patterns of work in what they call “joint cognitive systems” [133]. In an investigation of approaches towards measuring macrocognition (where macrocognition is a set of cognitive processes that occur across multiple individuals involved in a complex task), [103] outlines several techniques for observing macrocognition processes. These techniques include presenting humans with scripted scenarios and asking them to solve a problem or make a decision within the presented scenarios; gradually presenting new data and asking humans to look for connections and patterns; or asking humans to prepare a

plan for accomplishing a task. Graesser and Murray report that a question-answering methodology was able to provide rich information on human-computer interaction tasks, such as logging into a computer and creating and modifying a text file [63]. This methodology was a form of semi-structured interviews as several rules were used to determine what questions to ask in what situations.

The process elicitation techniques that we studied in this work (section 3.1) build on the techniques described above for eliciting information about complex socio-technical systems. We focused, however, on three particular kinds of information—nominal steps, exceptional situations, and responses to exceptional situations—that need to be included in process models that can drive the proposed online process guidance approaches. We then compared qualitatively and quantitatively the process elicitation techniques we studied in terms of their ability to elicit these particular kinds of information.

6.1.2 Process Modeling

In this section, we discuss other work related to the process modeling aspects we investigated in this thesis, namely the modeling of exception handling in HIPs and the use of different process model representations to facilitate process model understanding.

6.1.2.1 Exception Handling in HIPs

Exceptional situations arise frequently during the execution of HIPs. Thus, many process and workflow languages include constructs to allow for the definition of exception handlers (for example, Little-JIL [27], WIDE [26], OPERA [66], GAT [95]). While researchers continue to study how best to provide exception handling mechanisms within process languages, exception handling has become more mainstream with its inclusion in languages like WS-BPEL [10, 46] and products like IBM's WebSphere [58].

The need to identify exception handling patterns in processes has also been recognized. Russell, van der Aalst, and ter Hofstede [116] discuss exception handling patterns in workflows. They approach exception handling patterns by identifying four dimensions associated with exception handling mechanisms: the nature of the exception, if and how the work item that encounters the exception continues, whether other work items are cancelled as a result of the exception, and whether there is any roll-back or compensation performed. Based on this analysis, they suggest that these four dimensions be used to derive a universe of exception handling patterns. They suggest essentially that each of the different combinations of the possible choices from each of these four dimensions be considered to be an exception handling pattern, without regard to whether these combinations are commonly used in practice and without providing a description of the workflow problems that the pattern might be suitable for addressing. Thus, it is still left to the workflow designer to understand the mechanisms at their most basic level. Identifying those combinations may be useful as a benchmark to determine the exception handling capabilities of a process language. On the other hand, these combinations do little to aid process designers in identifying and reusing existing high-level solutions since no general higher-level purpose for a particular combination is provided to guide the designer in choosing a pattern to use. The combinations that Russell et al. identify also lack names that might suggest their usefulness. Instead, they name the patterns with acronyms derived from the choices made out of each of the four dimensions. For example, they identify 30 patterns associated with expired deadlines alone, two of which are called OCO-CWC-NIL and ORO-CWC-NIL. Our approach differs from the approach of Russell et al. in that it is driven by recognition of patterns that we have identified through our work in defining processes in multiple domains. We thus approach the identification of patterns in a top-down manner, analyzing uses of exception handling to generalize and extract

patterns that we believe to be useful beyond the specific processes in which we have found them.

6.1.2.2 Process Model Representations

The importance of describing processes to various stakeholders, perhaps from different backgrounds, is well-established. In some early efforts to accomplish this, organizations created paper-based process guides or manuals that describe, largely in natural language, the process of interest. It has been observed, however, that such paper-based descriptions are difficult to navigate (due to their inherently linear structure and significant size), are time-consuming to develop, are hard to keep in sync with the evolving process (or its formal description), and are nearly impossible to customize. Thus, such paper-based descriptions are not very effective and rarely created or used [80].

To alleviate some of the disadvantages of paper-based process descriptions, organizations have resorted to electronic process guides (EPGs), which usually contain hyperlinks that facilitate navigation. Using hyperlinks addresses to some extent the navigation problems with paper-based process descriptions. Manually creating and maintaining EPGs remains a large problem. For example, [20] reports that during the maintenance of a V-Modell guide, “when changing the glossary structure to tooltip style, some 2000 links had to be updated.” As a result, some tools have been proposed to automatically generate EPGs, such as Adonis [1], ARIS [2], Eclipse Process Framework (EPF) Composer [69], and Spearmint [19]. Such tools use a process model as input to a generator that automatically creates a hyperlinked EPG based on that process model. The process model is usually captured in some notation that supports constructs such as activities and their hierarchical decomposition, artifacts and resources, roles, and the relationships between these constructs (e.g., which role is responsible for which activity, what resources are needed to perform an activity,

and what artifacts are produced by an activity). Process engineers can then associate detailed natural-language descriptions with each such component of a process model.

Once a process model is in place, an EPG generator can use that model to create a hyperlinked document describing the process. EPF Composer and Spearmint, for instance, create a document that has a section showing the decomposition of the process activities (as a summary view of the process) and another section with a detailed natural-language description of the selected activity/artifact/role. These sections, however, contain little indication of the order, or the flow of control among the process activities. The section that provides a summary view of the process, for example, shows the hierarchical decomposition of the process activities, but not the order in which they need to be executed. The detailed, natural-language description of the selected activity may contain some information about control flow, but this information is included in a non-systematic way as it is up to the person who writes the description to decide how much control flow information to include.

Another concern with the process descriptions generated by the tools mentioned above is related to the lack of semantic richness of the process notations used to create the process models that are in turn the basis for the generation of EPGs. Such process notations often lack support for complex exception handling behavior, concurrency and synchronization mechanisms, or even mechanisms for simpler constructs such as looping. For example, [106] reports that the EPF Composer was not able to represent parts of a software development process because EPF Composer could not adequately model the looping at the activity level.

The Little-JIL narrative and visual representations described in section 3.2 contain information about resources, artifacts and agents, similar to the EPGs generated by the tools discussed above, but also include a detailed description of the control flow. This is due to the fact that the Little-JIL language provides complex control flow constructs, such as support for exceptional behavior, concurrency, synchronization,

and recursion. The Little-JIL narrative and visual representations are automatically generated from the process model, eliminating the issues related to manual creation and maintenance mentioned above.

6.2 Online Process Guidance for HIPs

In this section, we discuss existing approaches for providing online process guidance. We first discuss several approaches for providing guidance by visualizing the state of executing HIPs. Then, we discuss several approaches for what we call *passive guidance*, which are approaches where there is limited or no automated support and process performers are responsible for making sure they adhere to a process specification.

6.2.1 Visualization of the Execution State of HIPs

6.2.1.1 Trauma Center Process Guidance System

Fitzgerald et al. have designed and deployed a process guidance system in a trauma center to guide medical professionals during the first 30 minutes of trauma resuscitation [57]. Trauma resuscitation medical algorithms have been modeled what Fitzgerald et al. call “branch tree logic” (no reference to that logic is provided in [57], but it seems that this logic is similar to a decision tree representation). The process guidance system has been used by a team of 6-7 medical professionals with an emergency physician as a team leader and a senior trauma nurse acting as a scribe to document the resuscitation. The visualization of the process guidance system (Figure 6.1) has been displayed on a 40-inch LCD monitor. The process guidance elements were a set of action prompts (rectangular buttons, around 3-4 on the screen at a time). The screen also contained patient data such as name and DOB, readings of patient physiological conditions (heart rate, blood pressure, etc.), the patient diagnosis(es), and the drugs and treatments given to the patient.

tified or when the required type of medical professional is unavailable (e.g., a doctor) and he/she needs to be substituted by a different type of medical professional (e.g., a nurse).

6.2.1.2 Visualization of Patient Flow

Baffoe et al. [17] describe an approach for inferring process execution state based on a process model and events generated by what they call “business process management (BPM) technology” (e.g., patient registration systems, computerized physician order entry systems, scheduling systems) and real time location systems (RTLSSs). The process model is essentially a finite-state machine, where transitions are triggered when a set of events are reported by the BPM technology and/or RTLSSs. By correlating such events with the transitions in the process model, the state of an executing process can be tracked as events are recorded.

A prototype of the approach was implemented for a medical process. In particular, the flow of cardiac patients through the Emergency Department and the Cardiac Care Unit was modeled and a case study was conducted with a community hospital in Ontario, Canada. The prototype measured service and wait times and compared them against target times for the hospital. A visualization was developed for the duration of each stage of the care process for a given patient. Color coding was used to distinguish between stages whose measured duration is within an acceptable range (green), stages whose measured duration is approaching the target limit (yellow), and stages whose measured duration has exceeded the target limit (red). Figure 6.2 shows an example of this visualization. A similar visualization is also generated for aggregate statistics, such as average time for a given process stage for all patients.

The prototype of the patient flow visualization approach focused on the nominal process flow and it appears that the underlying process model captured a single process execution. It will be interesting to evaluate the ability of this approach

Patient States

Search:

State	Start Time	End Time	Duration (mins)	Target (mins)
IN_BED_CW	2013-03-02 11:13:00.0	N/A	50	N/A
IN_TRANSPORT_CW	2013-03-02 11:02:00.0	2013-03-02 11:13:00.0	11	15
WAIT_FOR_TRANSPORT_CW	2013-03-02 10:39:00.0	2013-03-02 11:02:00.0	23	15
WAIT_FOR_BED_CW	2013-03-02 10:13:10.0	2013-03-02 10:39:00.0	26	480
IN_BED_ED	2013-03-02 10:12:00.0	2013-03-02 10:13:10.0	2	N/A
IN_PHYS_RE_ASSESS	2013-03-02 10:00:00.0	2013-03-02 10:12:00.0	12	N/A
WAIT_FOR_PHYS_RE_ASSESS	2013-03-02 09:40:00.0	2013-03-02 10:00:00.0	20	30
WAIT_FOR_ORDERS_EXECUTION	2013-03-02 08:19:00.0	2013-03-02 09:40:00.0	81	30
IN_BED_ED	2013-03-02 08:15:00.0	2013-03-02 08:19:00.0	4	N/A
IN_PHYS_INIT_ASSESS	2013-03-02 08:12:00.0	2013-03-02 08:15:00.0	3	N/A
WAIT_FOR_PHYS_INIT_ASSESS	2013-03-02 08:10:00.0	2013-03-02 08:12:00.0	2	30
TRIAGED	2013-03-02 08:08:00.0	2013-03-02 08:10:00.0	2	30

Showing 1 to 12 of 12 entries

Overall Duration: 3 hours 1 mins

Figure 6.2: Visualization of cardiac patient flow.

to support process performers on more complex flow, such as when non-nominal situations arise or when concurrent execution is involved. The process model used in the evaluation case study is also very high-level. It will be interesting to investigate how the approach could support process performers when tasks at lower level of granularity are taken into account.

We believe the patient flow visualization approach nicely complements the smart checklist approach to process execution state visualization presented in section 5.2. The two approaches focus on different kinds of process information—the former on the patient flow and the latter on the activities of various process performers—and if used together, they could improve the overall situational awareness of process performers.

6.2.1.3 Visualization of Patient Location During Perioperative Clinical Processes

PERICLES [6] was a research initiative between 2009 and 2011 whose main objective was to facilitate clinicians' process awareness. A surgical procedure is modeled

using the YAWL process modeling language [8] and a tag is attached to a patient, so that an ultrasound-based realtime locating system can determine in which room a patient is. The patient location information or information manually entered by process performers is used to determine which steps from the process model are being performed. As a result, a visualization is created that shows the location of the patient in the hospital. Resource information captured in the process model is used to assist operating room coordinators with resource assignments [100].

According to [5], PERICLES has been implemented in the central operating facility of a German partner hospital. We were not able to find published studies that report the results of this implementation. Based on the project’s description ([5,6]) and the publications resulting from this project ([84,100,101]) it appears that the process model that was used in PERICLES was relatively small and simple—it contained about a dozen steps and did not capture exceptional situations.

6.2.2 Traditional Checklists and Process Guides

In the above approaches for visualization of process execution state, the information provided to process performers is to some extent *dynamically* updated depending on how the current process execution unfolds. Traditional checklists and process guides, on the other hand, are *static* in the sense that information about potentially several complete process executions is provided and it is up to process performers to decide what information applies to the current process execution and to keep track of the progress of that execution.

6.2.2.1 Checklists

Checklists have been widely used for real time-process guidance. Examples of such checklists include the Procedure Checklist for Administering Blood Transfusion [130] included in a standard nursing reference [131], the Surgical Safety Checklist advocated by the World Health Organization (WHO) and used during surgery worldwide [136],

and the Pilot's Abbreviated Flight Crew Checklist for OV-10A aircrafts used in the Vietnam war [124]. Checklists have been effective in reducing the number of errors in aviation and are now "universally employed in commercial and military aviation and space flight" [23]. Checklists have also proven beneficial in improving safety and efficiency in other industries, such as product manufacturing [67].

Recently, healthcare has started the adoption of checklists as well. Various studies have reported the success of checklists in different domains of medicine. The use of a checklist by a nurse for obtaining medication history during the initial patient assessment has been linked to the reduction of patient medication histories that contain medication discrepancies [123]. The implementation of a Quality Rounds Checklist in an ICU has been linked to the decrease in the number of pneumonias [54]. In another ICU study, a checklist was used to ensure adherence to standard procedures (such as hand-washing and removing unnecessary catheters) and the rate of catheter-related infections decreased after the medical professionals started to use the checklist [108]. [51] reports improvement in patient outcomes after a comprehensive surgical checklist was introduced in six hospitals. This checklist included not only information about what needs to be done in the operating room, but also information related to pre- and post-surgery tasks.

The role of checklist in healthcare and their direct impact on reduction of errors, however, have been more controversial than in other industries. Birkmeyer [22] acknowledges that checklists have been linked to reduction in errors in healthcare but also points out limitations of some of the studies in terms of controlling for confounding factors (such as concurrent implementation of outcomes measurement and feedback and Hawthorne effect [85]), and lack of correlation between level of compliance with checklists and extent of improvement in outcomes. In addition, Birkmeyer raises a concern about the lack of studies supporting the durability of improvements linked to checklists. He argues that "It is . . . conceivable that the benefits of surgical

checklists could wane over time as they lose their novelty and become a perfunctory component of care.”

Hales et al. [67] point out that despite reported positive outcomes, checklists are not routinely used in the clinical setting. [67] suggests that some of the factors for checklist adoption resistance in healthcare include operational barriers, such as the difficulty to standardize certain medical processes, as well as cultural barriers. Hales et al. write that “. . . there is often an assumption that the use of memory aids is an admission of weakness or lack of medical skill or knowledge, which can contribute to negative attitudes toward the implementation of these types of resources. Furthermore, clinicians often view standardization, or the use of standardized tools such as checklists, as a limitation to their clinical judgment and autonomous decision making.” We believe the detailed and semantically rich process model used in our proposed approach for process state visualization could capture the different acceptable variations of performing a medical process, and thus the resulting smart checklist would be more flexible in terms of supporting clinical judgment and autonomous decision making than traditional checklists. Furthermore, clinicians may opt to not use a smart checklist and rely only on guidance provided by the deviation detection and explanation approach, which does not require medical professionals to explicitly use memory aids and interferes only when a deviation from the acceptable ways to perform the process is detected.

Traditional checklists are also often incomplete and tend to focus on the normative workflow, omitting important details about exceptional scenarios [31]. Checklists could also contain ambiguities. For example, the tasks specified in the Surgical Safety Checklist by the WHO [136] are often high-level and ambiguous. The accompanying implementation manual [135] provides a more detailed natural language description of each task on the checklist, but there are still some ambiguities. For example, at one point the manual says that “the coordinator verbally confirms patient identity”

but it does not specify how exactly that should be done—e.g., how many patient identifiers (and which) should be used. There has been evidence that even simple processes such as verifying patient identity are error-prone [72, 74] and leaving their specification ambiguous in process guides could lead to undesired process outcomes.

Another deficiency of checklists is that they primarily help with avoiding errors of omission [23]. However, other kinds of errors, such as commission errors, occur in HIPs [59, 71, 73, 111].

Our approaches for online process guidance can help address some of the limitations of checklist discussed above. The process models we use are written in a rich language, which could enable the inclusion of complex execution behaviors, including normative flow, exceptional flow and concurrent execution. Thus, a smart checklist generated from such a process model can support process performers on a larger variety of scenarios that traditional checklists do. A smart checklist is also context-sensitive and dynamically updated, thus, making relevant information salient and removing information that is not relevant to the current process execution. The use of a formal process model can also address the ambiguity problem from which some checklists suffer.

6.2.2.2 Process Guides

The process guides and manuals described in section 6.1.2.2 can also be used to assist process performers while a process is being executed, but they suffer from the limitations also described in section 6.1.2.2—the paper-based process guides are difficult to navigate, whereas the electronic ones are often incomplete as they tend to not represent complex control flow information well, or not at all. For complex HIPs, such as medical processes, precise activity ordering or control flow information can be very important. Medical guidelines, for instance, can define strict constraints on the order of performing certain activities. Also, medical processes exhibit a high degree

of concurrency (as different medical professionals can work on the same process in parallel) and non-deterministic choices made by agents at runtime. Thus, a process guide that does not provide in a systematic way sufficient amount of control flow information might not be able to provide the level of guidance to process performers necessary to reduce control-flow related errors (e.g., the performance of activities out of order and simultaneous performance of activities that should not be performed at the same time).

6.2.3 Approaches for Dealing with Process Deviation

This section discusses formal approaches that have been used to deal with process deviation. Most of these approaches have not been directly used for process guidance (except for the approach to deal with deviation in software design processes), but the techniques they utilize could potentially be useful. In fact, our approach for error localization shares commonalities with Cook and Wolf’s approach for validation of software process models [42].

6.2.3.1 Software Process Validation

Cook and Wolf studied techniques for measuring the discrepancies between process models and process executions [42]. In particular, they investigated how to measure the difference between an a sequence of performed steps and a sequence of steps from a process model, and given a sequence of performed steps, how to find the “closest” sequence of steps from the process model.

The problem of comparing two sequences of steps is cast as a string comparison problem and a standard dynamic programming algorithm [83] is used to compute the difference between two strings in terms of insertion and deletion edit operations. Two *validation* metrics for sequence difference are defined based on the number of insertion and deletions needed to transform one sequence into another. These metrics contain tuning parameters that can be adjusted based on properties of the process

and of the domain. In Cook and Wolf’s study, the values of the tuning parameters are set based on what has been observed to work in their case studies, but a rigorous and systematic approach to determine these values is not discussed.

The problem of finding the process model sequence that is “closest” to a sequence of performed steps is cast as a search problem. The model is assumed to be a finite-state machine. A state in the search space is then determined by a triple—a model state, an index in the sequence of performed steps and an operation (deletion/insertion/match) that led to the current state in the search space. A variation of best-first search [117] is used to estimate the closest process model sequence. Heuristics are used to estimate the solution cost and to prune the search space.

6.2.3.2 Dealing with Deviations in Software Design Processes

Da Silva et al. have been studying the problem of dealing with deviations in software design processes [47–49]. Their general approach is based on defining deviation rules in terms of logical formulas and checking in real time whether a process execution sequence satisfies these logical formulas. The deviation rules are expressed in LTL in [49], in Praxis (a language defined by the authors) in [48] and as Prolog rules in [47].

In [47], a risk level and a cause for a deviation can be specified in the deviation rules. The risk level is an estimate of the risk of not achieving the process outcomes given that the deviation occurs. The cause for a deviation is an activity that led to the deviation. For example, suppose that in a given process, activity *a* should precede activity *b*. A deviation rule (which is a logical formula) can be written saying that activity *b* occurs before activity *a* has occurred and that the cause for this deviation is activity *b*. Such a deviation rule will be checked at runtime as a sequence of activities is performed and if at one point this rule is satisfied, a deviation will be reported and its cause will be determined as activity *b*.

Once a deviation has been detected, [47] proposes an approach for identifying sequences of corrective actions to address the deviation. This is achieved by specifying another set of rules (called *generator functions*), where each rule specifies a sequence of actions that need to be taken once a specific deviation caused by a specific activity occurs. There could be multiple sequences of actions (called *correction plans*) for a given (set of) deviation(s) and these correction plans are ranked by risk level of the deviations they fix—a correction plan that leaves the least number of high-risk deviations unfixed is considered the best one.

Instead of proposing correction plans once a deviation has been detected, in [49] the idea of *tolerance* is used. There are two kinds of logical rules—deviation rules, based on the constraints coming from the process model, and *tolerance rules*, which are relaxations of the deviation rules based on some predefined tolerance levels for the violation of a given deviation rule. Both the deviation and tolerance rules are checked at runtime. Two separate reports are produced at each step — a deviation report shows the deviation rules (i.e. original process constraints) that are violated; the tolerance report shows the tolerance rules that are violated. These reports are shown to the process performer who can decide to address the deviations listed in the reports or to ignore them and continue with the process execution.

In the general approach presented by Da Silva et al., only deviations for which corresponding deviation rules have been created statically (before executing the process) can be detected at runtime. In [47] and [48] creating these rules could be error-prone and time-consuming, especially for larger and more realistic process models, since most of the work needs to be done manually. Also, only “expected” deviations could be detected since deviations for which there are no deviation rules defined cannot be detected. These problems are addressed in [49], where the deviation rules are automatically created from a process model.

The strategy for identifying the cause for a deviation presented in [47] seems limited since the cause for a given deviation is hardcoded into the corresponding deviation rule. This approach will not be able to handle the issue of delayed deviation detection (described in section 4.1.2.1), where there could be several plausible, and perhaps equally likely, causes for the deviation. In general, da Silva et al. assume that the intent of process performers is always known—if a process model has multiple branches, at runtime it is always known which branch the process performers intend to execute. This way, it can be unambiguously determined what deviation rule is violated and consequently what the deviation and its causes are. In the approach presented in this thesis, we assume that the intent of process performers is not known, thus, there could be multiple plausible explanations for a detected deviation.

The correction plans that are generated upon detection of a deviation in [47] are based on rules defined prior to process execution. This is similar to specifying exception handling in process languages with rich semantics. The process guidance approach presented in this thesis uses process models specified in a language with rich semantics and it is assumed that any exception handling information is already part of the process model. A deviating sequence represents a behavior not described by the process model and therefore there is no static hardcoded knowledge about how this behavior should be addressed. Upon deviation detection, we aim to provide process performers with information to help them decide how to recover from the potential errors based domain knowledge and expertise.

6.2.3.3 A Framework for Formalizing Inconsistencies and Deviations in HIPs

Cugola et al. [45] present a framework for formalizing the notions of deviations and inconsistencies between a HIP (e.g., a software development process) and a process model-based automated system that supports the execution of the corresponding

HIP (e.g., a process-based integrated development environment). To accomplish this formalization, the HIP and its process-based support system (PBSS¹) are represented as state machines (possibly with infinite number of states). In addition there are two relations, R_s and R_t , that relate states and transitions respectively from the HIP state machine and the PBSS state machine. These state machines and relations comprise an environment.

A deviation is a concept related to transitions and an inconsistency is a concept related to states. An *environment deviation* occurs when a transition in the PBSS state machine does not correctly reflect a transition in the HIP state machine. For instance, a design document is created, but the PBSS is not updated with that information. Environment deviations can lead to environment inconsistencies. An *environment inconsistency* occurs when the HIP and its PBSS become inconsistent, i.e., the state of the PSS does not correspond to the state of the HCS as described by R_s . For example, the HIP is in a state where a design document exists, but the PBSS is in a state where a design document does not exist.

Cugola et al. also define domain deviations and inconsistencies. A *domain deviation* is an event that deviates from the expected behavior of the HIP. An example of a domain deviation is a developer writing the source code before writing its design, assuming that there is a process rule saying that design should be written before code. Unlike environment deviations, domain deviations can occur even when when no PBSS is used, but some domain rule or policy is violated. Domain deviations can lead to domain inconsistencies. An example of a *domain inconsistency* is a situation where the source code does not have a corresponding design file.

While the presented framework formalizes the notions of deviation and inconsistency, there are difficulties in utilizing it to provide online process guidance. To detect

¹The names and the abbreviations are ours. Cugola et al. call the HIP a *Human Centered System* and the PBSS a *Process Support System*.

deviations, the state machines that model the HIP and the PBSS must exist. The model of the HIP must capture all possible executions, including deviating/erroneous executions, so that these deviations can be detected when comparing the transitions of the HIP state machine to the transitions of the PBSS state machine. Having such a HIP state machine that captures all possible executions is impractical and we believe that the approach of using just one execution event sequence to represent what is happening in the HIP is more feasible.

6.2.3.4 Conformance between Process Executions and Process Models

Similar to Cook and Wolf [42], Rozinat and van der Aalst [113–115] have studied methods for quantifying the conformance between a process model and executions of the modeled process. In particular, they focus on the problem of comparing a process model to an execution log (a collection of observed process executions). Like Cook and Wolf, they are interested in measuring *fitness* (“Does the observed process comply with the control flow specified by the process model?”) but also *appropriateness* (“Does the model describe the observed process in a suitable way?”).

Rozinat and van der Aalst use a Petri net [104, 105] process model and compare it to an execution log. Each execution sequence in the log is “replayed” in the Petri net and things like number of consumed tokens, number of produced tokens, and average number of enabled transitions during the replay of the event log are counted. In addition, the authors define *precedes* and *follows* between two activities in the Petri net and in the event log. These relations and the counts mentioned above as well as the total number of places and transition in the Petri net and the part of them that correspond to events from the log are then used to compute fitness and appropriateness.

In [115], Rozinat and van der Aalst are interested in comparing a process model to *multiple* and *complete* process executions. They use aggregate information, such

as average number of enabled transitions, to compute conformance metrics. On the contrary, our deviation detection approach operates on a *single* and *partial* process execution, which gets compared to a process model as it grows.

6.2.3.5 Adherence to Medical Guidelines

In the medical domain, Advani et al. [12] have investigated methods for determining adherence of clinicians to medical guidelines by examining information in medical charts. Guidelines are represented as a hierarchical decomposition of intentions (e.g., a high-level intention of a hypertension guideline could be “Manage Hypertension” and a leaf-level intention could be “prescribing a particular drug”). The patient record is examined to determine whether the leaf-node intentions have been satisfied. The information from the leaf nodes is then propagated up the hierarchy to determine to what extent the high-level intentions have been satisfied.

Advani et al.’s approach is based on examining the patient chart “after the fact”, i.e., after process execution to determine conformance with medical guidelines. Also the representation of guidelines they use doesn’t seem to support ordering and more complex control flow. Thus, their approach does not seem applicable to checking in real time, as a process is being executed, the kind of adherence to process models we are interested in—detecting deviations and providing information to help explain these deviations.

6.2.4 Error Localization

This section describes approaches that are related to the the error localization approach described in section 4.5.1.

6.2.4.1 Fault Localization

The task of locating faults in programs (commonly referred to as *fault localization*) shares commonalities with the task of identifying possible indexes in a sequence of

performed steps where an error might have occurred. In the usual fault localization setting, a program is being tested prior to deployment or executed after being deployed. Some of the program executions fail, which indicates that there is a fault in the program. The exact location of the fault, however, is not known and locating this fault could be difficult and time-consuming. Thus, recent research has investigated techniques for automatically identifying program statements that are most likely to be faulty.

There have been various approaches to automated fault localization (e.g., [13, 39, 79, 102, 112]). The *Tarantula* approach [79] uses information about the number of times a given program statement is executed in passing and failing test cases. A simple formula is then used to compute the “suspiciousness” of a given statement to be faulty. *Set union*, described in [112], computes a “suspicious” initial set of program statements by removing the union of all statements executed by all passed test cases from the set of statements executed by a single failed test case. The nearest neighbor approach [112] arbitrarily chooses any single failed test case and then finds the passed test case that has most similar coverage to the coverage of the failed test case. Then, the set of statements executed by the passed test case is removed from the set of statements executed by the failed test case and the remainder becomes set of most “suspicious” statements. In many fault localization approaches, the “suspiciousness” score is used to rank program statements hoping to save the programmer time and effort in locating the fault.

There are two main differences between the work on fault localization and the proposed approach for error localization. Fault localization aims to identify possible faulty locations (statements) *in a program* given a set of “good” (passing) and a set of “bad” (failing) program executions (test runs). In the proposed error localization approach, we aim to identify possible indexes in a *process execution* where an error might have occurred, assuming that the program (i.e., the process model) is correct.

Fault localization also assumes that a computer program is always executed correctly by the hardware, and thus, a failing program execution must be due to a fault in the software. The second difference is that in the fault localization case, we are given *multiple* “good” and “bad” program executions to identify the faults, whereas in the online process guidance case, we are given a *prefix* of a *single* execution. Thus, fault localization techniques do not seem to be directly applicable to the problem of identifying possible indexes in a sequence of performed steps where an error might have occurred. Some fault localization work could be useful, however, in incorporating historical information about deviating executions into a strategy for identifying in real time possible indexes in a sequence of performed steps where errors might have occurred.

6.2.4.2 Anomaly Detection

Some work from the area of anomaly detection is relevant to the proposed error localization approach. Given a failing program execution (e.g., an execution that has not passed a test case), the goal of anomaly detection is to identify likely anomalous events (in the failing program executions) that caused the observed failure. Unlike fault localization, which aims to directly locate a fault in a program, anomaly detection aims to identify events in a program execution that led to the failure of that execution. Knowing such events can then help locate the fault in a program.

Some anomaly detection techniques synthesize a behavioral model of the program from correct program executions and then compare a failing program execution with that model to identify likely anomalies that caused the failure. One such technique [90] takes as input a failing program execution and a finite-state automaton (FSA) synthesized from correct executions. It then uses the kBehavior algorithm [91] to augment the FSA model so that it accepts the failing execution. The subsequences of

the program execution that correspond to the FSA augmentations are then considered to be the anomalies and are presented to the developers for inspection.

Just the anomalous events by themselves, however, are often not very useful as they “do not capture the structure and the rationale of the differences between the correct and the failing executions” [16]. The AVA technique [16] performs an automated interpretation of anomalies to provide developers with a higher-level anomaly information. For instance, if an expected event is missing from an earlier part of an execution but appears later on, then AVA could automatically interpret these two anomalies as one higher-level anomaly, namely that the event was postponed.

One difference between the approach for anomaly detection discussed above and the error localization approach we propose is that the anomaly detection approach identifies *one* set of possible anomalies per program execution. The kBehavior algorithm greedily augments the FSA to make it accept the failing program execution and, thus, produces one set of anomalies. The error localization approach we propose can identify *several* likely error indexes in a sequence of performed steps that could correspond to different, mutually exclusive errors. It might be interesting to explore the question whether the kBehavior algorithm can be extended so that it produces several likely sets of anomalies and apply it to the problem of identifying likely explanations for a deviation. Also, some of the techniques that AVA uses to infer high-level anomaly patterns might be useful in identifying possible human errors and not just error indexes based on a sequence of edit operations that transform a process model sequence into a deviating sequence.

6.2.4.3 Plan and Policy Recognition Approaches

The artificial intelligence community has studied approaches for the tasks of plan and policy recognition. These tasks are related to the task of identifying the most likely intended sequences of steps from a process model, given a sequence of steps that

process performers have executed. Identifying the most likely intended sequence of steps is a component of the approach for error localization discussed in section 4.5.1.

6.2.4.3.1 Policy Recognition Using Markov Models. Bui et al. [25] explore an approach for policy recognition based on an Abstract Hidden Markov Model, a formalism that they define. The goal is to infer agent’s policies at different levels of abstraction based on a sequence of states that an agent visits. The approach has been applied to recognizing human behavior (e.g., the person is using a computer vs. the person is passing by the computer (low-level policy), person intends to exit the building through the north exit (high-level policy)) based on a sequence of locations of the person, as recorded by video cameras.

Luhr et al. [89] use Hierarchical Hidden Markov Models (HHMMs) [56] to learn and recognize human activity. They apply their approach in the context of eldercare aiming to identify behaviors such as eating dinner, watching television, and cooking. An HHMM is learned for each high-level activity (e.g., cooking) based on sequences of observations, where an observation is a person’s location in the room at a given time. The HHMMs are then used to classify new sequences.

While the approaches proposed by Bui et al. and Luhr et al. seem promising for recognizing the intended policy/plan of process performers, they do not support the notion of error in the sense that they consider all recorded sequences of events (or states of process performers) to be error free. A sequence of events/states is considered to be more likely to follow one policy/plan than another, but no sequence can be a deviating sequence where process performers have deviated from the recommended ways to perform the process due to errors. Thus, it would be difficult to reason about human errors using the above approaches.

6.2.4.3.2 Plan Recognition Using Bayes Nets. Work in the domain of intelligent tutoring systems has also addressed similar problems. Gertner et al. [62] tackle

the problem of deciding what the tutoring system should say when a student needs help solving a problem. To achieve that, an attempt is made to estimate the student's knowledge and the solution path the student is pursuing. The model used to do the estimation is a Bayes net that consists of different kinds of nodes to represent information such as the facts that the student knows, the student's goals, rules that the student can apply to solve a problem, and so on. When a student's action is observed (e.g., the student computes some quantity needed for the problem solution), the value of the corresponding nodes in the Bayes net (e.g., the student knows fact A) is set. Nodes, whose values are not known are assigned a probability that represents the confidence that the student has performed the task associated with the node. When a student asks for help, the Bayes net is used to infer what part of the problem the student is currently working on and what the student should do next.

Similar to the approaches discussed above, Gertner et al.'s approach focuses on recognizing the intended execution sequence. Gertner et al. go one step further by proposing a hint about how to perform the supposed next activity when a student gets stuck and asks for help. No sequence of recorded actions of the process performer, however, is considered a deviation from the recommended ways to perform the process and no error identification is supported.

Gertner et al. also seem to assume that given an observation, the corresponding node in the Bayes net model can be uniquely identified. In the situations we propose to address with our deviation detection and explanation approach, an observed action can correspond to different nodes in a flow graph, as the example described in section 4.1.2.1 illustrates.

6.2.4.3.3 Plan Recognition Using DFAs and a Bayesian Classifier. Phua et al. [107] present a process guidance approach to assist dementia patients in smart homes. Information about the patient's activities is collected by a variety of sensors, such as video cameras, RFID sensors, and pressure sensors. Activity recognition

techniques are applied on the sensor data to deliver in real time an activity sequence to a plan recognition system.

A plan is defined as a sequence of activities to achieve a goal. Example activities are *moving chair*, *sitting*, *eating food*, *standing*, *walking*. Example plans are *prepare food*, *consume food*, and *prepare utensils*. The authors point out that plan definitions are subjective in terms of aspects such as breadth (type of constituent activities) and depth (granularity of constituent activities).

The main goal of Phua et al.’s approach is Erroneous Plan Recognition (EPR)—recognizing erroneous plans (subsequences of the sequence of recorded activities) in real time (as a dementia patient is performing activities of daily living) and sending timely audio and visual prompts to the patient (and potentially the caregiver) to correct the erroneous execution of a plan. To achieve this goal, they propose a 2-stage EPR system. The first stage is matching subsequences of the full activity sequence against a set of *whitelist* and a set of *blacklist* deterministic finite-state automata (DFAs). The whitelist and blacklist DFAs represent behaviors that correspond to correct and incorrect plan executions respectively. If the subsequence under consideration matches² a blacklist DFA, then the EPR system will issue an error warning³. If the subsequence under consideration does not match neither a whitelist nor a blacklist DFA, then the sequence is passed to a trained machine learning classifier which gives a probability of the subsequence being an erroneous plan execution. If this probability is higher than a pre-specified threshold, a general error warning is issued⁴.

The EPR system is constructed via a training process. A plan library is populated with a set of plans which are manually labeled as correct or erroneous. The plans can be created by domain experts or machine learned from recorded patient behavior.

²The authors do not provide details what exactly *matches* means in this context.

³No details are provided about the kind of warning that is issued.

⁴Again, no details are provided about the kind of warning that is issued.

The whitelist and blacklist DFAs discussed above are constructed from these plans. The plans are also used to train the classifier.

6.3 Human Errors

This section presents an overview of work on human errors. Taking into account the theoretical underpinnings of human errors is important for the proposed deviation detection and error localization approaches as their main goals are to catch errors before harm is done and provide information to help identify potential errors. The potential error indexes identified by the error localization approach are based on the edit operations that transform sequences from the process model into the sequence of performed steps. Taking into account the kinds of human errors identified by human error researchers could be useful with the selection of edit operations and cost function, so that the selected edit operations provide more realistic representation of the real-world errors.

For instance, knowing that omitting an activity or doing an extra activity are commonly made errors can justify the selection of deletion and insertion edit operations. In addition, recognizing that sometimes due to inattention people do a sequence of routine activities on autopilot in situations when they are actually not supposed to do this sequence of activities could justify using an edit operation that inserts an entire sequence of steps into a sequence from the process model. Also, the cost of such an edit operation could be set to be lower than the sum of the costs of individual insert operations that accomplish the same transformation. This way, such an edit operation will be favored by a sequence comparison algorithm, which would increase the probability that potential error indexes corresponding to the corresponding error interpretation will be ranked higher.

6.3.1 “Human Error” by James Reason

The work of James Reason [111] has been one of the most influential treatments of human error and is heavily used in the IOM report “To Err Is Human” [82] mentioned earlier to formalize the discussion of errors in healthcare. Reason classifies human errors based on a model of human cognition. According to that model, humans operate cognitively at three different levels—skill-based, rule-based and knowledge-based level—and each level has a corresponding category of errors. Humans work at the *skill-based* level when they perform routine actions in a familiar environment. A lot of the actions at this level are done subconsciously (since they’ve been done many times before) and only from time to time attention is needed to perform a check to decide which subconscious path of actions should be taken. Errors that occur at this level are called SLIPS and LAPSES and they are usually due to either inattention (a check is not done properly) or “overattention” (“when the human consciously interrogates the progress of an action sequence when control is best left to automatic pilot”).

Humans enter the *rule-based* level of cognition when a problem is detected at the skill-based level. At that level, the human process performer considers the current state of the process execution (situation) and tries to find a memorized rule that says “IF (situation) THEN (actions)”. Errors that occur at this level are called RB MISTAKES. RB mistakes are usually due to either a “misapplication of good rules or application of bad rules”.

Humans enter the knowledge-based level of cognition when NO appropriate rule for the current situation is found at the rule-based level. Working at this level requires most attention (not auto pilot) since no prior knowledge (rule) about how the address the problem is available and the human needs to perform more complex analysis of the current situation and decide on future actions. Errors at this level are called KB MISTAKES. KB mistakes occur for various reasons—selectivity (giving attention

to the wrong features of a problem), mental workspace limitations (finite resources of the conscious workspace), availability heuristic, confirmation bias, overconfidence, etc.

6.3.2 The Eindhoven Classification Model

Another influential work on classifying errors has been the Eindhoven classification model [126]. It was originally designed to categorize errors in the chemical industry, but has been subsequently used in other industries. For example, it has been customized for use in healthcare [127] and used in the context of event reporting systems [18, 21].

The Eindhoven model builds on Reason’s error classification by incorporating organizational and technical factors in addition to individual human errors made during process execution. Errors are divided into two main categories—latent errors and active (human) errors. *Latent errors* are technical and organizational errors and result from administrative decisions that affect technical issues, organizational policy, or allocation of resources [71]. *Active errors* are human errors made by individuals during process execution. The sub-classification of active errors follows closely Reason’s categories—skill-based, rule-based and knowledge-based errors.

Even though the Eindhoven error classification model extends the model presented by Reason with additional error categories, there has been evidence that it is not effective in categorizing errors in complex HIPs such as healthcare processes [71]. In addition, both Reason’s error classification and the Eindhoven model focus on the cognitive aspect of errors, on what is happening in the human’s mind that led to the error. For instance, an example of a skilled-based error in Reason’s error classification scheme is failing to perform a mental check that the right course of action is taken for the given circumstances; the Eindhoven model has an error category described as “the inability of an individual to apply existing knowledge to a novel situation.”

This cognitive specification of human errors makes it different to operationalize the error categorization for the purposes of deviation detection and error localization because such cognitive errors cannot be observed or detected by an automated system. What seems to be needed is a categorization based on the manifestation of errors that can be observed during process execution, rather than based on the cognitive causes of errors. In Reason's terms, an error classification at the *behavioral* level rather than the *conceptual* level is what seems to be needed.

6.3.3 Error Causes vs. Error Manifestations

Hollnagel recognizes this problem [77] and points out the lack of an operational taxonomy of human errors. He distinguishes between error cause and error manifestation and introduces the terms *error genotype* and *error phenotype* to refer to these two aspects of human errors respectively. He then proposes a set of error phenotypes and discusses their use in a system for on-line detection of errors in a process domain. This system, called RESQ [75, 76], is based on plan recognition. As actions of process performers are recorded in real time, they are matched against a *plan library* and the plan that is most likely to be currently under execution is identified. Actions that do not conform to this most likely plan are considered erroneous and an attempt to determine the kind of error that occurred is made by using a knowledge base of phenotypes. The error kinds that the RESQ system could potentially diagnose are intrusion (when the action does not belong to any plan in the plan library or is not expected in the current plan), repeated action, omitted step, a sequence of omitted steps, goal-not-as-specified (the goal of the erroneous actions does not match the goal specified by the process performer), goal-not-as-preferred (the erroneous action does not contribute to the goal recommended by the system).

CHAPTER 7

CONCLUSION AND FUTURE WORK

Human-intensive processes (HIPs) are critical for our societal infrastructure. Unfortunately, most, if not all, important HIPs, such as medical processes, suffer from human errors, leading to compromised quality of the services or of the products delivered by these processes. In medical processes, for example, human errors result in serious harm, including disabling injuries and even death.

Considerable efforts have been made to reduce the number and impact of human errors in HIPs. One approach has been to improve the design of HIPs and to train process performers to follow the resulting improved processes. Despite such efforts, however, there is evidence that human process performers still make errors while performing some important HIPs, due to various reasons such as cognitive overload, distraction, and fatigue.

To address this problem, this thesis investigates two approaches for providing guidance to human process performers *while* a process is being performed. The first approach, *deviation detection and explanation*, detects when process performers deviate from the recommended ways to perform a process. Such deviations could reflect errors and, thus, notifying process performers about detected deviations could make them aware of potential errors before harm is done. The approach also provides process performers with information to help them identify potential errors, which in turn could facilitate planning recovery from such errors. Such help could be particularly useful in complex and time-critical HIPs. The second approach, *process state visualization*, proactively guides process performers by showing them information relevant

to the current process execution. The goal of this approach is to reduce the number of errors.

Both of these approaches base the online guidance they provide on a detailed formal model of the recommended ways to perform the process. Creating such models is challenging and this thesis also investigates process elicitation techniques to help create such models. We studied several techniques for eliciting a process from domain experts—unstructured interviews, three kinds of structured interviews, and observations—in the context of a process for treatment plan review, part of an overall chemotherapy preparation and administration process. We found that using combination of elicitation techniques leads to obtaining richer process information than using any technique alone and we also compared the relative strengths and weaknesses of these techniques in terms of eliciting different kinds of process information, such as activities on the nominal process flow, exceptional situations, and responses to exceptional situations. We also conducted a small user study to compare two process modeling notations, a diagrammatic notation and a natural language notation, in terms of their ability to facilitate process understanding. For study subjects with technical background, using the diagrammatic notation was associated with better process understanding than using the narrative notation.

We identified important issues related to the deviation detection and explanation approach and developed an experimental framework to evaluate the approach with respect to these issues. Models of two medical processes—a blood transfusion process and a chemotherapy preparation and administration process—and synthetic process executions with seeded errors were used for the evaluation. We found that in the performed experiments deviations were detected with delay infrequently, the delays were short and not harmful. We also found some cases where harm could occur even if a deviation is immediately detected, recognized these cases as process vulnerabilities, and proposed ways to deal with some of these vulnerabilities. We focused on one kind

of deviation explanation, namely error localization. The preliminary experimental evaluation of the proposed error localization approach indicates that this approach can successfully identify error locations under some assumptions about potential errors that could occur in a process.

We implemented an initial prototype of the approach for visualization of process execution state and demonstrated the approach on several case studies from the medical domain. In one of these case studies—patient controlled analgesia—the approach was evaluated by a panel of medical experts. The feedback was positive and several interesting research directions were identified (discussed below).

In the rest of this chapter, we describe some future research directions.

Process elicitation and modeling. We compared several process elicitation techniques—unstructured interviews, structured interviews, and observations—in terms of their ability to elicit certain kinds of process information. These kinds of information were steps on the nominal process flow, exceptional situations, and responses to exceptional situations. It would be interesting to compare the above elicitation techniques in terms of eliciting other kinds of process information, such as the order between steps and how artifacts are used in a process. A particularly interesting future research direction would be to investigate techniques for eliciting information about process errors, such as the kinds of errors that are of concern in a particular domain, in what part(s) of the process they typically occur, and what the likelihood of such errors is. Such information could be utilized to determine potential deviation detection delays, to fine tune the error localization approach, and to inform the design of process execution state visualizations.

In our evaluation of the process elicitation techniques, we applied them in a specific order. The results might change if these techniques are applied in a different order and subsequent studies are needed to explore the effect of the order.

Deviation detection. In this work, we studied the issues related to deviation detection and explanation empirically. It would be interesting, however, to explore some of these issues analytically. For instance, it will be useful to develop analytical approaches for determining an upper bound on possible deviation detection delays given a process model and assumptions about the errors that might occur during a process execution. Such approaches could reveal how vulnerable a process is to delayed deviation detection, how harmful these vulnerabilities might be, and how a process can be modified to reduce such vulnerabilities. We have begun investigating such approaches, but their feasibility seems to be a major challenge due to the very large number of situations that need to be explored to determine what the deviation detection delay would be when different kinds of errors occur that could involve potentially any step in a process.

Deviation explanation. There are several aspects of the error localization approach described in this work that need further research. For large and complex process models, criteria for selecting legal sequences to compare to the sequence of performed steps are needed, because such models can contain a large, or even potentially infinite, number of legal sequences and, thus, choosing a subset of all legal sequences would be required to keep the approach feasible. As previously discussed, one possible technique to deal with this issue is to incrementally compare prefixes of legal sequences to a prefix of the sequence of performed steps and discard legal sequences whose edit distance to a prefix of the sequence of performed steps is above a certain threshold. Such technique is based on the assumption that there will be a small number of distinct legal sequence prefixes up to a certain length, which is usually the case as many processes have a relatively small number of initial steps.

The proposed error localization approach is highly sensitive to the measure of similarity between two sequences and deciding what measure of similarity to use under what circumstances requires further research. We used the edit distance between two

sequences as the measure of similarity, where the edit distance is a function of the costs of the edit operations needed to transform one sequence into the other. The choice of edit operations and their costs also significantly affects the accuracy of the error localization approach and, thus, further research into how to make this choice is needed. One possible strategy is to base the edit operations on typical errors that might occur during the performance of a process and to base the edit operation costs on characteristics of these errors, such as error frequency and severity.

While the information that the error localization approach provides could help process performers identify potential errors and plan recovery actions, richer deviation explanations could be even more useful. For example, a ranked list of hypotheses about actual potential errors, such as step X was omitted or subprocess Y was omitted, in addition to a ranked list of possible indexes in the sequence of performed steps where an error might have occurred, could further facilitate recovery from potential errors.

Classification of human errors. Most of the existing classification schemes for human errors focus on the cognitive aspect of errors, but such aspects cannot be observed while a process is being performed. There has been some work on categorizing errors based on their manifestation during a process execution, but further research is needed to identify error classification schemes, such that the different error kinds captured by such schemes are observable during process execution, cover important errors that could occur in a given process, and, if detected by an online guidance system, informing process performers about such errors could prevent potential harm. Constructing such error classification schemes would have direct impact on the deviation detection and explanation approaches—it could facilitate the analytic determination of bounds of deviation detection delays by potentially decreasing the number of possible error situations that need to be considered, and it would direct the selection of edit operations and associated costs used for error localization.

Process model optimizations. A search through a process model is performed for the purposes of both deviation detection and deviation explanation. For large and complex HIPs, the search space could be large, negatively affecting the performance of deviation detection and deviation explanation. In this work, we performed some optimizations to reduce the size of the low-level model that is translated from a Little-JIL process model and some optimizations to specifically reduce the size of the search space determined by this low-level model. Further optimizations might be needed to apply the deviation detection and explanation approach to models of larger and more complex HIPs than the ones used in this work. Such optimizations would also be useful for the offline analysis part of the process improvement environment.

Visualization of process execution state. The current prototype for visualization of process execution state supports visualization from the perspective of a single agent, but domain experts who saw that prototype indicated that it might be useful to have a view with information about the overall process, such as different activities that have been or are currently being performed by different process performers. Such a view might improve the overall understanding of an executing process as well as facilitate team communication. Supporting different process state visualization modes, for example based on the expertise of a process performer, is also an interesting future research direction to explore.

The design of the user interface for visualization of process execution state also needs further improvement and evaluation. For instance, some domain experts expressed concerns about ambiguity of some of the iconography in the current prototype and also expressed preference for stronger distinction between the visualization of current process execution state and the visualization of process execution history. It is very unlikely that a single set of visualizations will appeal to all performers of a given process and, thus, the user interface needs to be carefully designed, by using human factors approaches and performing user studies.

Determining how a process might unfold and visualizing information related to possible future process execution states could be useful to process performers to plan their work. Supporting such visualization would require developing approaches for exploring the process model that drives the visualization and determining most likely future process executions based on the current process execution.

An interesting issue to explore is the issue of undo—sometimes process performers might need to roll back the process state visualization because an event has mistakenly been reported to have happened (e.g., a process performer marked the wrong step as completed). Supporting undo for systems that visualize the state of human-intensive processes and also guide process performers, however, is not trivial, because reporting by mistake that a process event has occurred might influence the process execution, in some cases rendering the action of mistakenly reporting that process event irreversible.

Impact of the proposed process guidance approaches. The ultimate goal of the deviation detection and explanation approach is to catch human errors before harm is done; the goal of the process state visualization approach is to prevent human errors. Thus, studies need to be performed to evaluate the effect of these approaches on human errors in HIPs. Ideally, these approaches would be deployed in real HIPs and error rates will be compared between cases when the online guidance approaches are used and cases when they are not. Before these approaches can be deployed in critical HIPs, such as a medical process, however, pilot studies in a simulated setting would need to be performed.

APPENDIX A

ARTIFACTS USED IN PROCESS ELICITATION STUDY

A.1 Open-ended Prompts

1. A triage MA leaves a treatment plan and orders for a patient in your tray. You confirm that pretesting has been done. What do you do next?
2. A triage MA leaves a treatment plan and orders for a patient in your tray. You confirm that pretesting has been done, confirm existence and not staleness of height/weight data in CIS, confirm treatment plan is created from a careset. What else, if anything, do you do before signing the treatment plan?
3. What steps do you take to verify the doses?
4. A triage MA leaves a treatment plan and orders for a patient in your tray. You notice that labs have not been done for the patient. The chemo drugs for this patient are not platinum-based. How do you proceed?
5. A triage MA leaves a treatment plan and orders for a patient in your tray. You notice that labs have not been done for the patient. The chemo drugs for this patient are platinum- based. How do you proceed?
6. A triage MA leaves a treatment plan and orders for a patient in your tray. You notice that a scan has not been done for the patient. How do you proceed?
7. When you go to check that a patients height and weight have been entered in the CIS, you notice they are missing. How do you proceed?

8. When you go to check that a patients height and weight have been entered in the CIS, you notice they were taken in another building. How do you proceed?
9. When you go to check that a patients height and weight have been entered in the CIS, you notice a patients height and weight measurements are stale. How do you proceed?
10. You receive new height and weight measurements for a patient. There is a 6% change in the dose based on these new values. How do you proceed?
11. While reviewing a patients treatment plan, you notice that the treatment plan was not created from a careset. How do you proceed?
12. While reviewing a patients treatment plan, you notice that orders are missing for the patient. How do you proceed?
13. While reviewing a patients treatment plan and orders, you notice the orders were entered by a Fellow. How do you proceed?
14. While verifying doses for a patient, you notice that the height and weight in the treatment plan doesnt match the height and weight in the CIS or in the patient chart. How do you proceed?
15. While verifying doses for a patient, you calculate the BSA for the patient and notice the calculated dose is greater than the dose in the orders. How do you proceed?

A.2 Complete Process Traces

A.2.1 Trace 1

1. You pick up treatment plan and orders that Triage MA has left.
2. You confirm that labs have been done.

3. You confirm that the scans have been done.
4. You confirm existence of patient height/weight data in CIS.
5. You confirm that height/weight are not stale (i.e more than 2 weeks old).
6. You confirm that the treatment plan is created from a careset.
7. You confirm existence of chemo orders for the patient.
8. You confirm that the orders have been entered by an Attending.
9. You verify the doses:
 - (a) You confirm that height/weight on treatment plan, in CIS, and in the patient chart all match.
 - (b) You calculate BSA using height/weight from CIS.
 - (c) You calculate doses using the BSA just calculated and the information from the treatment plan.
 - (d) Confirm calculated doses match the ones on the chemo orders.
 - (e) Confirm dose base on treatment plan is consistent with doses on orders.
10. Check sticky notes to make sure that everything is completed and it turns out that the labs have been done.
11. You sign the treatment plan.
12. You leave the treatment plan in Triage MAs tray.

A.2.2 Trace 2

1. You pick up treatment plan and orders that Triage MA has left
2. You confirm that labs have been done.

3. You confirm that the scans have been done.
4. You confirm existence of patient height/weight data in CIS.
5. You confirm that height/weight are not stale (i.e more than 2 weeks old). You find that height/weight are stale.
6. You tell Clinic MA to schedule an appointment with patient to measure height/weight. You put a sticky note on treatment plan that height/weight need to be remeasured. You stop here and wait until height/weight are remeasured.
...
7. You get up-to-date height/weight
8. You confirm that the treatment plan is created from a careset. You confirm existence of chemo orders for the patient.
9. You verify the doses:
 - (a) You confirm that height/weight on treatment plan, in CIS, and in the patient chart all match.
 - (b) You take BSA from the patient record on CIS.
 - (c) You calculate doses using the BSA and the information from the treatment plan.
 - (d) Confirm calculated doses match the ones on the chemo orders. You sign the treatment plan.
10. You leave the treatment plan in Triage MAs tray.

A.2.3 Trace 3

1. You pick up the treatment plan and the orders that the Triage MA has left.

2. You confirm that labs have been done.
3. You discover that a lab result is missing and the drugs are not platinum based.
4. You tell an MA to draw the labs next time the patient comes.
5. You find out that the patient does not have a scheduled appointment and you tell the MA to schedule one.
6. You put a sticky note on the treatment plan to check for the labs before signing the plan.
7. You confirm that the scans have been done.
8. You confirm the existence of patient height/weight data in the CIS.
9. You confirm that the patients height/weight are not stale (i.e more than 2 weeks old).
10. You confirm that the treatment plan is created from a careset.
11. You confirm the existence of chemo orders for the patient but you find out that they are missing.
12. You call the MD to enter the orders in the system.
13. You put a sticky note on the treatment plan to check for the orders.
14. You stop your work on the treatment plan for this patient and wait until the MD enters the orders.
...
15. (in 2 days) You find out that the MD has entered the orders for that patient.
16. You confirm that the orders have been entered by an Attending. 18. You verify the doses:

- (a) You confirm that the height/weight on treatment plan, in CIS, and in the patient chart all match.
 - (b) You calculate the patients BSA using height/weight from CIS.
 - (c) You calculate doses using the BSA just calculated and the information from the treatment plan.
 - (d) You confirm that the calculated doses match the ones on the chemo orders.
 - (e) You confirm that the dose base on treatment plan is consistent with the doses on orders.
17. You check all sticky notes to make sure that everything is completed and you confirm that the labs have been done.
18. You sign the treatment plan.
19. You leave the treatment plan in Triage MAs tray.

A.3 Complete Process Model (Textual Description)

1. Pick up treatment plan and the orders.
2. Confirm labs have been done
 - (a) If labs haven't been done and the chemo drugs are not platinum-based
 - i. Tell MA to draw labs next time when patient comes. (keep tr. plan)
 - ii. If the patient does not have a scheduled appointment, either the MA or the Practice RN schedules an appointment.
 - iii. Put a sticky note on the treatment plan to check for the labs before signing it.
 - iv. Continue to 3 (if not done yet)
 - (b) If labs haven't been done and some of the chemo drugs is platinum-based

- i. Tell MA to draw labs next time when patient comes.
 - ii. If the patient does not have a scheduled appointment, either the MA or the Practice RN schedules an appointment.
 - iii. Stop here and wait for the labs before continuing with the rest of the steps.
3. Confirm scans have been done
 - (a) If some of the scans haven't been done
 - i. Obtain signed scans order from MD
 - ii. Give scans order to an MA (outtake MA, downstairs)
 - iii. MA schedules a separate appointment for the scans.
 - iv. Put a sticky note on treatment plan to check for scans before signing
 - v. Continue to 4
4. Confirm existence of patient's height/weight data in CIS
 - (a) If patient height/weight are not entered in CIS but they have been measured in the building
 - i. Enter in CIS height/weight from patient chart
 - ii. Continue to 5
 - (b) If the patient's height/weight haven't been measured in the building
 - i. Option 1
 - Schedule an appointment before teaching so that the patient's height/weight will get measured (whoever can reach the patient will schedule it either Triage MA or Practice RN)
 - Continue to 6
 - ii. Option 2

- Indicate height/weight need to be measured during teaching (the patient gets scheduled for teaching (but not chemo) and his/her height/weight get measured then.)
- Put a sticky note on treatment plan to ensure height/weight re-measured before signing
- Continue to 6

5. Confirm height/weight are not stale.

(a) If height/weight are stale

- i. Tell clinic MA to schedule an appointment with patient.
- ii. Put a sticky note on the treatment plan that height/weight need to be remeasured
- iii. Wait for height/weight to be remeasured

6. Confirm that treatment plan is created from careset.

(a) If treatment plan is not from a careset

- i. Check if the doctor gave a reference to the primary literature in the treatment plan.
- ii. If there is no reference
 - (Optionally) look on Google or Pubmed for reference.
 - If there is no reference on Google or PubMed
 - (Optionally) call the Pharmacy and then MD.
 - Continue to 7.
 - Continue to 7.

7. Confirm existence of chemotherapy orders for that patient in CIS.

(a) If there are no orders

- i. Call MD to enter the orders in the system
 - ii. Put a sticky note to check for orders
 - iii. Wait until orders are entered
8. If the orders are entered by a Fellow MD, confirm existence of an MD-to-RN order in CIS saying that the Attending MD has approved the Fellow MD's orders.
 - (a) If there is no MD-to-RN order
 - i. E-mail both attending and fellow MDs.
 - ii. Put a sticky note on the treatment plan to ensure that MD-to-RN order is entered before signing.
 - iii. Continue to 9.
9. Verify doses (make sure they are correct for the patient's height/weight)
 - (a) Confirm height/weight on treatment plan, in CIS, and in the patient chart all match
 - i. If they don't match
 - Contact MD and ask how to continue from that point on.
 - Option 1
 - Physician enters an order expressing awareness of the difference in height/weight
 - Continue to 9b
 - Option 2
 - Physician enters new orders with dose change
 - Put a sticky note to check for orders
 - Wait until a new order is entered

- Continue to 9
- (b) Calculate BSA using height/weight from CIS.
- (c) Calculate doses using the BSA just calculated and the information from the treatment plan.
- (d) Confirm calculated doses match the ones on the chemo orders.
 - i. If there is more than 5% discrepancy
 - Contact MD to resolve the discrepancy.
 - Option 1
 - MD says he/she will enter an MD-to-RN order that the current dose is OK
 - Continue to 9e
 - Option 2
 - MD enters new orders with dose change
 - Put a sticky note to check for orders
 - Wait for the new orders
 - Continue to 9d
- (e) Confirm dose base on treatment plan is consistent with doses on orders.
 - i. If the dose base is not consistent
 - Contact MD to resolve the mismatch.
 - Option 1
 - MD enters new order with correct dose base.
 - Wait until new order is entered
 - Continue to 9e
 - Option 2
 - MD decides to keep the dose on the order

– MD re-enters the treatment plan and the process starts over.

10. Check sticky notes and make sure that everything is done

(a) If something is still not done

i. Continue to 10.

11. Sign treatment plan. (All the pretesting needs to be completed at this point, all issues with height and weight need to be resolved, and doses on the orders need to be verified.)

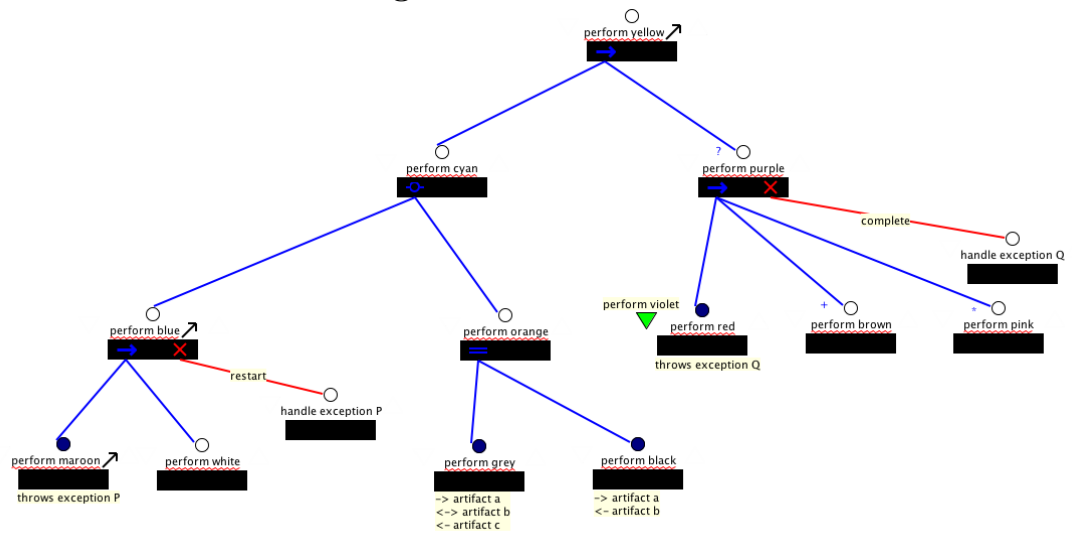
12. Leave treatment plan in Triage MA's tray.

APPENDIX B

ARTIFACTS USED IN STUDY OF PROCESS REPRESENTATIONS

B.1 Training Materials

B.1.1 Process Model—Diagram



B.1.2 Process Model—Diagram, Training script

Training script for the diagrammatic notation

In the Little-JIL diagrammatic notation, a process is described as a hierarchical decomposition of steps. A step represents an activity, or a unit of work that needs to be performed as a part of a process. Higher-level steps are decomposed into lower-level steps, which could be further decomposed to an arbitrary level of detail. A step is diagrammatically represented as a black bar (*illustrate with an example*).

A step can be started and later completed. When a higher-level step is started, it cannot be completed until all or some of its substeps (depending on the step's sequencing badge, explained next) are started and then completed themselves.

The badge on the left side of the step bar (called *sequencing badge*) specifies the order in which the substeps need to be performed. An arrow means that the substeps need to be performed in left-to-right sequential order and all substeps need to be completed before the parent step can be completed. (*Illustrate with an example: start parent, start substep 1, complete substep 1, start substep 2, complete substep 2, complete parent*).

An equal sign means that the substeps can be performed in any order, including simultaneously, and all substeps need to be completed before the parent step can be completed. (*Example: start parent, start substep 1, start substep 2, complete substep 1, start substep 3, etc.*).

A circle with a horizontal line across means that one of the substeps need to be chosen to complete the parent step. If the chosen substep fails to complete successfully, then one of the remaining substeps can be chosen and this process can repeat until one of the substeps completes successfully. (*Example: ...*)

Steps can have prerequisites. A prerequisite is usually another step that needs to be completed before the step with the prerequisite can start. A prerequisite is represented diagrammatically with a filled triangle to the left of the step name and an annotation above that triangle explains what the prerequisite is. (*Example*)

Steps can also have cardinality, where the cardinality of a step specifies how many times the step should be/can be done. The cardinality is shown as a symbol associated with the edge between the step and its parent. A "*" means that the step can be done 0 or more times; "+" means that the step can be done 1 or more times; "?" means that the step is optional, i.e. that it can be done 0 or 1 times. The default cardinality is 1, i.e., the step needs to be done exactly once. (*Example*)

Steps can use and produce artifacts (also referred to as "parameters"). Artifacts represent objects that are required to perform some real-world activity or are produced as a result of performing a real-world activity. There are several kinds of artifacts: in-artifacts (represented by an arrow pointing to the left, *example*) are input parameters to the step

passed from the parent step and are required before the step can start. Out-artifacts (represented by an arrow pointing to the right, *example*) are output parameters of a step and are passed from the step to its parent when the step completes. In-out artifacts (represented by a bi-directional arrow, *example*) are both input and output parameters of a step. In-out artifacts are required before the step can start, can be modified during step execution, and are then passed to the parent step when the step completes.

Steps can throw exceptions. Exceptions represent the occurrence of unusual or abnormal circumstances during process execution that might need to be dealt with outside of the normal flow of the process. If a step throws an exception, this is specified in the annotation associated with the step (example). Once an exception is thrown, it needs to be dealt with by an exception handler. The exception is propagated up the step tree until a matching exception handler is found (example). Exception handlers are connected to a red X on the right side of a step bar. An exception handler could be another regular step that needs to be performed in response to the exception, and once a matching exception handler is found, it is executed in response to the exception.

Once an exception handler completes, the process can resume normal execution. Where the process starts from is determined by the continuation action associated with the handler. If the continuation action is “restart”(illustrated by a “restart” annotation on the edge that connects the exception handler to a parent step), then the parent step of the exception handler is restarted once that exception handler completes. If the continuation action is “complete” (illustrated by a “complete” annotation on the edge that connects the exception handler to a parent step), then the parent step of the exception handler is completed.

B.1.3 Process Model—Narrative

TABLE OF CONTENTS FOR PERFORM YELLOW

Table Of Contents

- [perform yellow](#)
- ◊ [perform cyan](#)
 - [perform blue](#)
 - [perform maroon](#)
 - [perform white](#)
 - ⊖ [handle exception P](#)
 - ▬ [perform orange](#)
 - [perform grey](#)
 - [perform black](#)
- [perform purple](#)
 - [perform red](#)
 - [perform brown](#)
 - [perform pink](#)
 - ⊖ [handle exception Q](#)

[Legend](#) [Index of step names](#)

Perform Yellow

→ To "perform yellow", [perform cyan](#) and then [perform purple](#) (This step is optional.) .

Perform Cyan

↻ To "perform cyan", one of the following should be chosen to perform: [perform blue](#) or [perform orange](#).

Perform Blue

→ To "perform blue", [perform maroon](#) and then [perform white](#).

Perform Maroon

E If *Exception P Occurs*, then [handle exception P](#) and then restart "perform blue".

Perform White

Handle Exception P

Perform Orange

⇒ To "perform orange", the following need to be done in any order (including simultaneously), [perform grey](#) and [perform black](#).

Perform Grey

↓ The *artifact a* is required to "perform grey".

↑ Successful completion of the step "perform grey" should yield the *artifact c*.

↕ The *artifact b* is required to "perform grey" and may be modified during this step.

Perform Black

↑ Successful completion of the step "perform black" should yield the *artifact a* and *artifact b*.

Perform Purple

→ To "perform purple", [perform red](#), then [perform brown](#) (This step must be done at least once.) , and finally [perform pink](#) (This step is optional.) .

Perform Red

▼ Before beginning to "perform red", the step "perform violet" must be completed successfully.

PERFORM YELLOW

E If *Exception Q Occurs*, then [handle exception Q](#) and then complete "perform purple".

Perform Brown

Perform Pink

Handle Exception Q

2 of 2

B.1.4 Process Model—Narrative, Training script

Training script for the narrative notation

In the Little-JIL narrative notation, a process is described as a hierarchical decomposition of steps. A step represents an activity, or a unit of work that needs to be performed as a part of a process. Higher-level steps are decomposed into lower-level steps, which could be further decomposed to an arbitrary level of detail. The step decomposition is shown in the table of contents of the narrative (*illustrate with an example*). The highest-level step is listed on top. Its substeps are listed underneath, slightly indented to the right and are at the same level of indentation (*example*). The substeps of the substeps are listed in a similar way using indentation (*example*). Each step shown in the Table of Contents has a dedicated section on the right-hand side containing more detailed information about that step. (*example*)

A step can be started and later completed. When a higher-level step is started, it cannot be completed until all or some of its substeps (depending on the step's sequencing requirements, explained next) are started and then completed themselves. The order in which substeps need to be done is described on the right side of the narrative in the section describing the parent step (this section can be reached by clicking on the parent step in the Table of Contents) (*example*).

Each step section starts with the name of the step in large-size, bold font (*example*). The substeps and the order in which they need to be started and completed are described by a sentence. There are several options for the order in which substeps need to be performed before the parent step can be completed.

One option is that the substeps need to be started and completed in a sequential order and the parent step can be completed only when all of its substeps are completed. In this case, the sentence looks like this (*example*). In this case, the order of execution is (*Illustrate with an example: start parent, start substep 1, complete substep 1, start substep 2, complete substep 2, complete parent*). The arrow pointing to the right at the start of the sentence is an iconic representation that all the substeps need to be completed in the listed sequential order and this arrow is also shown in the table of contents.

Another option for the order of substep execution is that the substeps can be performed in any order, including simultaneously, and all substeps need to be completed before the parent step can be completed. (*Example: start parent, start substep 1, start substep 2, complete substep 1, start substep 3, etc*). A sentence describing this situation looks like this (*example*). The equal sign represents this situation.

Another option for the way substeps need to be executed is that one of the substeps needs to be chosen to complete the parent step. If the chosen substep fails to complete successfully, then one of the remaining substeps can be chosen and this process can repeat until one of the substeps completes successfully. (*example*). A sentence describing this situation looks like this (*example*). The circle with a horizontal line across represents this situation.

Clicking on the step names in the substep section would take you to the detailed description of the corresponding step.

In addition to having a sentence describing the order in which substeps need to be performed, each step section has a sentence (at the end) describing what needs to be done next after the step itself is completed. For example, the sentence at the end of the section corresponding to the step “perform yellow” indicates that after successful completion of that step, the entire process is considered completed. This is consistent with the Table of Contents, where we can see that “perform yellow” is the top-level step.

Similarly, looking at the end of the section for the step “perform white”, a sentence indicates that after “perform white” is completed, its parent, “perform blue” is completed. We can then click on “perform blue” and look at the sentence at the end of its section to determine what needs to be done after “perform blue” is completed. In this case, the step “perform orange” needs to be done next.

A step section can contain other information related to a step. A step can have a prerequisite. A prerequisite is usually another step that needs to be completed before the step with the prerequisite can start. A prerequisite is represented diagrammatically with a filled triangle to the left of the step name and is expressed with a sentence like this. (*example*)

Steps can also have cardinality, where the cardinality of a step specifies how many times the step should be/can be done. The number of times a step should be done is listed in the substep section of the parent step. For instance (*example*). The default number of times a step should be done is exactly once. A step can be done 0 or more times; 1 or more times (i.e. at least once); a step can be optional, i.e. that it can be done 0 or 1 times.

Steps can use and produce artifacts (also referred to as “parameters”). Artifacts represent objects that are required to perform some real-world activity or are produced as a result of performing a real-world activity. There are several kinds of artifacts: in-artifacts are input parameters to the step passed from the parent step and are required before the step can start. In-artifacts are described with a sentence like (*example*) and are represented by an arrow pointing down. Out-artifacts are output parameters of a step and are passed from the step to its parent when the step completes. Out-artifacts are represented by a sentence like (*example*) and an arrow pointing up. In-out artifacts are both input and output parameters of a step. In-out artifacts are required before the step can start, can be modified during step execution, and are then passed to the parent step when the step completes. In-out artifacts are described by a sentence like (*example*) and are represented by a bi-directional arrow.

Steps can throw exceptions. Exceptions represent the occurrence of unusual or abnormal circumstances during process execution that might need to be dealt with outside of the normal flow of the process. If a step throws an exception, this is specified by a sentence like (*example*) and the big red “E” denotes the section dedicated to exceptions. Once an exception is thrown, it needs to be dealt with by an exception handler. An exception

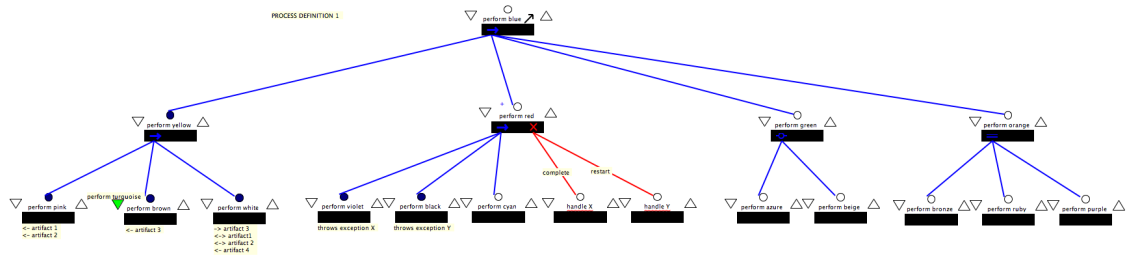
handler could be another regular step that needs to be performed in response to the exception (*example*).

Once an exception handler completes, the process can resume normal execution. The sentence associated with the exception that a step throws specifies how the process should continue after the exception has been handled. For example, this sentence may specify that certain step is completed (*example*). In that case, we can click on the step that is completed, and then look towards the end of its section to see what needs to be done next. Alternatively, the table of contents can be examined to determine the parent of the completed step, click on that parent step and see what its description says about the step that needs to be started after the completed step. (*example*).

In another case, the sentence associated with the exception might specify that a certain step needs to be restarted after the exception has been handled. (*example*).

B.2 Materials on which the study subjects were evaluated

B.2.1 Process Model 1—Diagram



B.2.2 Process Model 1—Narrative

TABLE OF CONTENTS FOR PERFORM BLUE

Table Of Contents

- [perform blue](#)
- [perform yellow](#)
 - [perform pink](#)
 - [perform brown](#)
 - [perform white](#)
- [perform red](#)
 - [perform violet](#)
 - [perform black](#)
 - [perform cyan](#)
- ⊖ [handle X](#)
- ⊖ [handle Y](#)
- ⊕ [perform green](#)
 - [perform azure](#)
 - [perform beige](#)
- ▬ [perform orange](#)
 - [perform bronze](#)
 - [perform ruby](#)
 - [perform purple](#)

PERFORM BLUE

[Legend](#) [Index of step names](#)

Perform Blue

→ To "perform blue", the following need to be done in the listed order

- [perform yellow](#)
- [perform red](#)
 - This step must be done at least once.
- [perform green](#)
- [perform orange](#)

Perform Yellow

→ To "perform yellow", [perform pink](#), then [perform brown](#), and finally [perform white](#).

Perform Pink

↑ Successful completion of the step "perform pink" should yield the *artifact 2* and *artifact 1*.

Perform Brown

▼ Before beginning to "perform brown", the step "perform turquoise" must be completed successfully.

↑ Successful completion of the step "perform brown" should yield the *artifact 3*.

Perform White

↓ The *artifact 3* is required to "perform white".

↑ Successful completion of the step "perform white" should yield the *artifact 4*.

↕ The *artifact 2* and *artifact 1* are required to "perform white" and may be modified during this step.

Perform Red

→ To "perform red", [perform violet](#), then [perform black](#), and finally [perform cyan](#).

Perform Violet

E If *Exception X Occurs*, then [handle X](#) and then continue with the next step.

Perform Black

E If *Exception Y Occurs*, then [handle Y](#) and then restart "perform red".


Perform Cyan

PERFORM BLUE

Handle X

Handle Y


Perform Green

 To "perform green", one of the following should be chosen to perform: [perform azure](#) or [perform beige](#).

Perform Azure

Perform Beige

Perform Orange

 To "perform orange", the following need to be done in any order (including simultaneously), [perform bronze](#), [perform ruby](#) and [perform purple](#).

Perform Bronze

Perform Ruby

Perform Purple

B.2.3 Process Model 1—Questions

Participant ID: _____

Questions for process definition 1

Answer the first six questions using the following steps (these are the lowest-level steps in the process description and are listed in alphabetical order here):

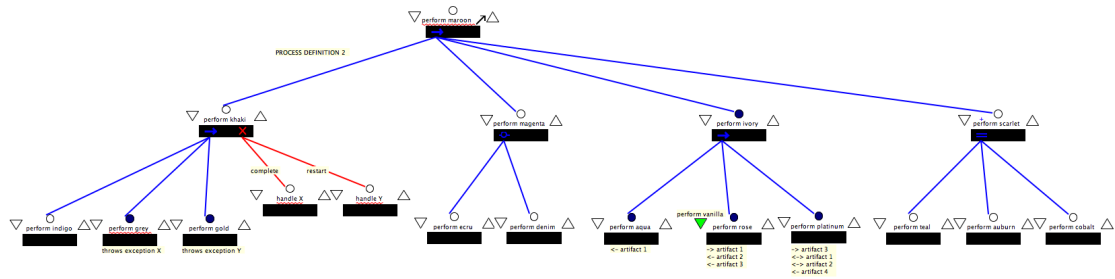
- *perform azure*
- *perform beige*
- *perform black*
- *perform bronze*
- *perform brown*
- *perform cyan*
- *perform pink*
- *perform purple*
- *perform ruby*
- *perform turquoise*
- *perform violet*
- *perform white*

For some of the answers, you might need to provide a sequence, or several sequences, of steps. List the steps in a sequence on the same line, separating consecutive steps by commas. If an answer consists of more than one sequence, write “#” at the end of every sequence and list different sequences on separate lines. For instance, if there are two possible sequences, list them in the following way:

perform violet, perform cyan, perform beige #
perform violet, perform beige, perform violet, perform bronze #

1. What step will be completed first? *(Remember to answer questions 1-6 using only steps from the bulleted list above).*
2. What are the possible sequences of the next 3 completed steps (i.e., sequences of the second, third and fourth completed steps in the process)?
3. Suppose the step “perform orange” has been started. What are the possible sequences of the next 3 completed steps?
4. Suppose the step “perform green” has been started. What are the possible sequences of the next 2 started steps?
5. Suppose that after the step “perform violet” has been started, the exception X occurs and that this exception is handled by completing step “handle X”. What step should be started next?
6. Suppose after the step “perform black” has been started, the exception Y occurs and that this exception is handled by completing step “handle Y”. What step should be started next?
7. How many times should the step “perform yellow” be completed? How many times should the step “perform red” be completed?
8. What artifacts are required to start step “perform white”?
9. What artifacts are required to start step “perform brown”?

B.2.4 Process Model 2—Diagram



B.2.5 Process Model 2—Narrative

TABLE OF CONTENTS FOR PERFORM MAROON

Table Of Contents

- [perform maroon](#)
- [perform khaki](#)
 - [perform indigo](#)
 - [perform grey](#)
 - [perform gold](#)
- ⊘ [handle X](#)
- ⊘ [handle Y](#)
- ⊕ [perform magenta](#)
 - [perform ecru](#)
 - [perform denim](#)
- [perform ivory](#)
 - [perform aqua](#)
 - [perform rose](#)
 - [perform platinum](#)
- ▬ [perform scarlet](#)
 - [perform teal](#)
 - [perform auburn](#)
 - [perform cobalt](#)

[Legend](#) [Index of step names](#)

Perform Maroon

→ To "perform maroon", the following need to be done in the listed order

- [perform khaki](#)
 - This step must be done at least once.
- [perform magenta](#)
- [perform ivory](#)
- [perform scarlet](#)

Perform Khaki

→ To "perform khaki", [perform indigo](#), then [perform grey](#), and finally [perform gold](#).

Perform Indigo

Perform Grey

E If *Exception X Occurs*, then [handle X](#) and then continue with the next step.

Perform Gold

E If *Exception Y Occurs*, then [handle Y](#) and then restart "perform khaki".

Handle X

Handle Y

Perform Magenta

⊖ To "perform magenta", one of the following should be chosen to perform: [perform ecru](#) or [perform denim](#).

Perform Ecu

Perform Denim

Perform Ivory

→ To "perform ivory", [perform aqua](#), then [perform rose](#), and finally [perform platinum](#).

Perform Aqua

↑ Successful completion of the step "perform aqua" should yield the *artifact 1*.

Perform Rose

PERFORM MAROON

▼ Before beginning to "perform rose", the step "perform vanilla" must be completed successfully.

↓ The *artifact 1* is required to "perform rose".

↑ Successful completion of the step "perform rose" should yield the *artifact 2* and *artifact 3*.

Perform Platinum

↓ The *artifact 3* is required to "perform platinum".

↑ Successful completion of the step "perform platinum" should yield the *artifact 4*.

↕ The *artifact 2* and *artifact 1* are required to "perform platinum" and may be modified during this step.

Perform Scarlet

▬ To "perform scarlet", the following need to be done in any order (including simultaneously), [perform teal](#), [perform auburn](#) and [perform cobalt](#).

Perform Teal

Perform Auburn

Perform Cobalt

B.2.6 Process Model 2—Questions

Participant ID: _____

Questions for process definition 2

Answer the first six questions using the following steps (these are the lowest-level steps in the process description and are listed in alphabetical order here):

- *perform aqua*
- *perform auburn*
- *perform cobalt*
- *perform denim*
- *perform ecru*
- *perform gold*
- *perform grey*
- *perform indigo*
- *perform platinum*
- *perform rose*
- *perform teal*
- *perform vanilla*

For some of the answers, you might need to provide a sequence, or several sequences, of steps. List the steps in a sequence on the same line, separating consecutive steps by commas. If an answer consists of more than one sequence, write “#” at the end of every sequence and list different sequences on separate lines. For instance, if there are two possible sequences, list them in the following way:

perform teal, perform gold, perform auburn #
perform teal, perform auburn, perform teal, perform grey #

1. What step will be completed first? *(Remember to answer questions 1-6 using only steps from the bulleted list above).*
2. Suppose that step “perform ivory” has been started. What are the possible sequences of the next 3 completed steps?
3. Suppose the step “perform scarlet” has been started. What are the possible sequences of the next 3 completed steps?
4. Suppose the step “perform magenta” has been started. What are the possible sequences of the next 2 started steps?
5. Suppose that after the step “perform grey” has been started, the exception X occurs and that this exception is handled by completing step “handle X”. What step should be started next?
6. Suppose after the step “perform gold” has been started, the exception Y occurs and that this exception is handled by completing step “handle Y”. What step should be started next?
7. How many times should the step “perform scarlet” be completed? How many times should the step “perform ivory” be completed?
8. What artifacts are required to start step “perform platinum”?
9. What artifacts are required to start step “perform aqua”?

APPENDIX C

LOW-LEVEL PROCESS MODEL REPRESENTATION

To make the implementation of the deviation detection experimental framework independent of a particular process modeling language, we used a low-level process model representation into which various high-level process modeling languages can be translated. For our experimental evaluation, we used the Little-JIL translation toolset [29] to translate Little-JIL models into this low-level representation. The process model translation is performed in two stages (Figure C.1). In the first stage, the Little-JIL Translator translates a Little-JIL process model into an intermediate process model written in the Bandera Intermediate Representation (BIR) [43]. In the second stage, the BIR Translator translates the intermediate process model into a low-level process model written into a low-level representation (trace flow graph and constraints [55]). The details of the process model translation are outside the scope of this work and are described in [29]. In this chapter, we provide an overview and selected details of the low-level representation to give a sense of the models that the deviation detection experimental framework is currently utilizing, to illustrate some of the issues we encountered, and explain some of the optimizations we implemented to address these issues. We also discuss the correspondence between this representation and the abstract ECFG notation used for discussion purposes in Chapter 4.

C.1 Trace flow graph and constraints

The low-level process model consists of a trace flow graph (TFG) and a set of constraints [55]. A *trace flow graph* is a collection of control flow graphs (CFGs),

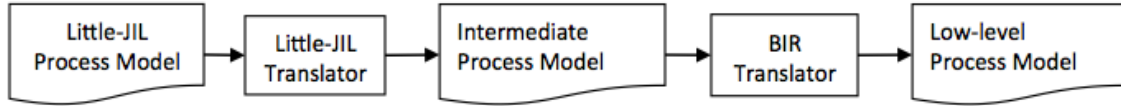


Figure C.1: The stages of translating a Little-JIL process model into a low-level process model.

where each CFG is called a *task* and it represents a subprocess or an activity that could potentially be concurrently executed with other subprocesses and/or activities. Figure C.2 shows a TFG derived from the Little-JIL model of a simplified final stage of a chemotherapy process shown in Figure C.3. Each TFG has a *main task*, which is the CFG where control starts. The main task can fork additional tasks to represent concurrent execution. In the example TFG in Figure C.2, there are two additional tasks, TASK 1 and TASK 2, which represent the Little-JIL substeps from Figure C.3 *prepare chemo drugs* and *administer pre-meds*. These steps can be executed in any order with each other—after the clinic nurse orders the chemo drugs from pharmacy, the clinic nurse can administer the pre-medications to the patient while the pharmacy is preparing the chemo drugs.

A TFG is a single-entry single-exit graph, meaning that there is a single *start node*, at which control starts, and a single *end node*, at which control ends. Each TFG node has a unique identifier, shown in Figure C.3 as an integer next to the corresponding node. For TFGs derived from Little-JIL process models, nodes correspond to different aspects of the execution of the Little-JIL process model. For example, some nodes correspond to starting of a Little-JIL step, some correspond to completing of a Little-JIL step, some correspond to forking a task, etc. Nodes are labeled with events based on this correspondence. For instance, node 2 in Figure C.2 corresponds to starting the root step *perform final stages of chemo process* in Figure C.3 and it is labelled accordingly; node 7 represents the forking of the task 2, which is the task that

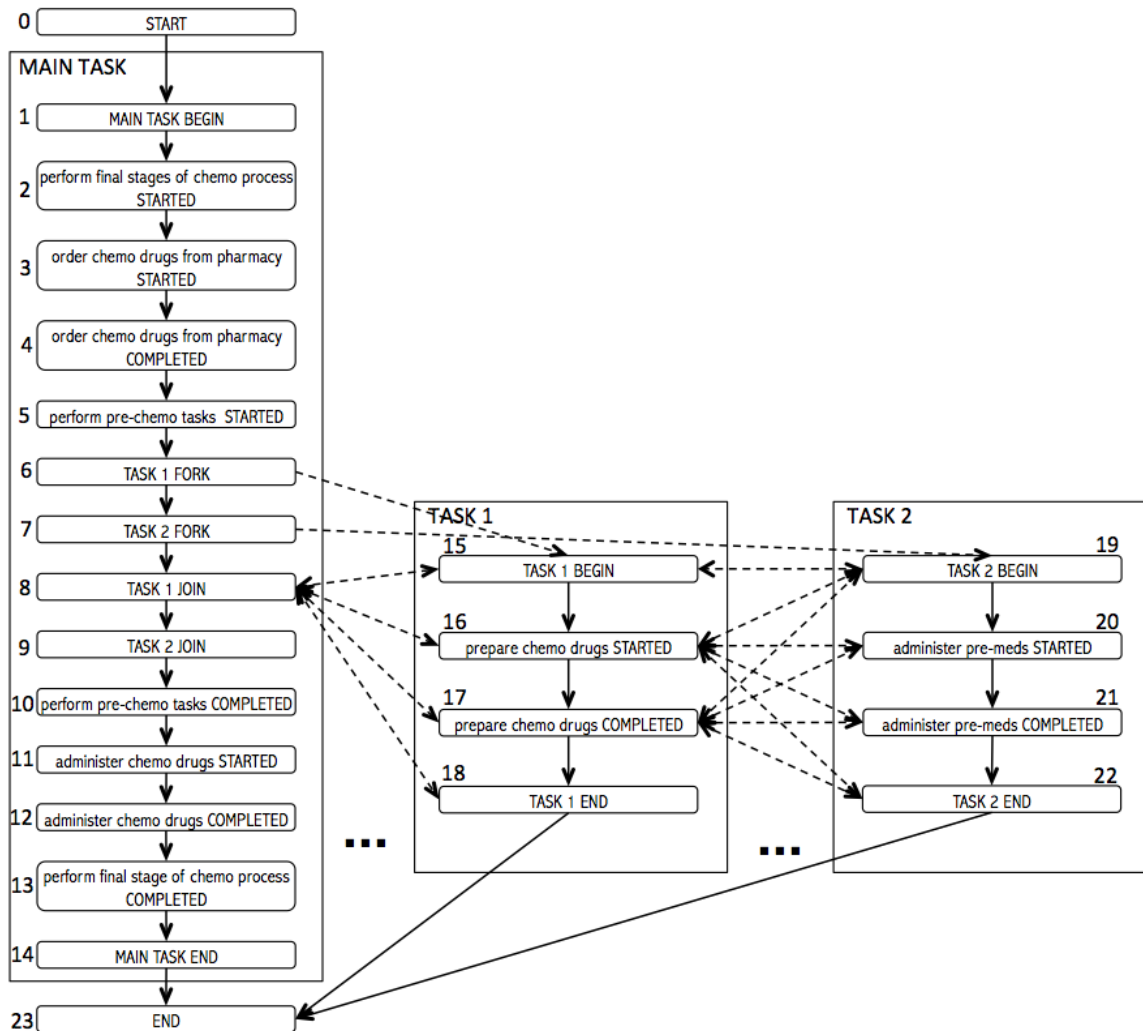


Figure C.2: Trace flow graph derived from the Little-JIL model in Figure C.3.

represents the step *administer pre-meds*, which can be done in parallel with the step *prepare chemo drugs*¹. The set of node labels in a TFG forms the TFG's *alphabet*.

There is an edge from node A to node B if the process event (such as starting of a step) that node A represents can immediately precede the process event node B represents. There are two kinds of edges in a TFG: an *intra-task edge* (shown as solid arrows) connects nodes within a single task; *may immediately precede edge (MIP)*

¹To keep the TFG in Figure C.2 relatively small and simple, we elided TFG nodes that correspond to the *posting* of a Little-JIL step on the agenda of the agent responsible for that step (posting a step represents assigning that step to an agent).

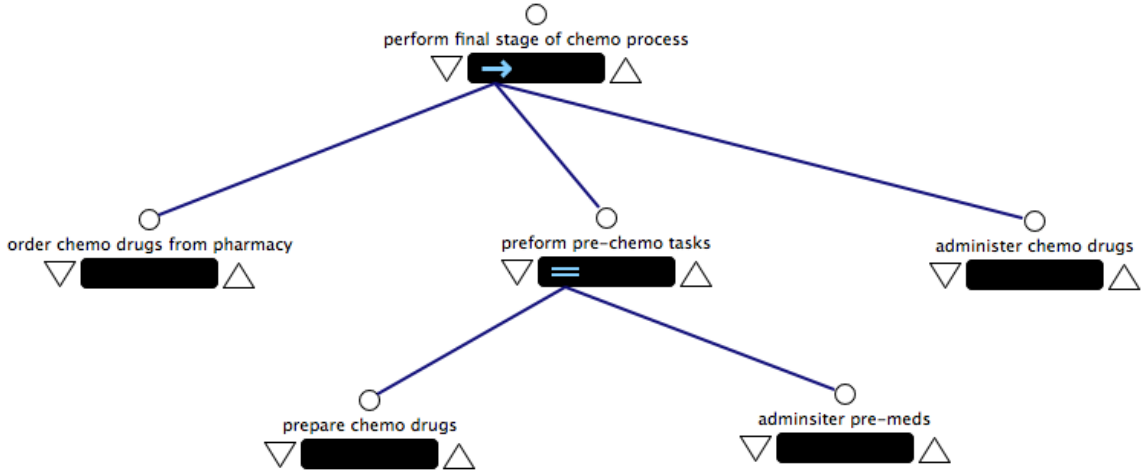


Figure C.3: Little-JIL model of a simplified final stage of a chemotherapy process.

(shown as a dashed line) connects nodes in different tasks. MIP edges represent possible interleavings between nodes in different tasks. A sequence of nodes (a path) from the start node represents a process execution.

A TFG, like most CFG-based models that preserve all possible process executions but abstract away some information (e.g., variable values) for the sake of compactness of the model, is an over-approximation of the executions specified in the high-level model it is translated from (in our case, the Little-JIL process model). This means that a TFG might allow paths from the start node that correspond to sequences of steps not allowed by the original high-level model.

To improve the precision of a TFG and eliminate such infeasible paths, the TFG is augmented with a *set of constraints*. A constraint is used to selectively add back information that was abstracted away from the TFG during the translation of the original high-level model, without making the size of the low-level model (i.e., the TFG and the constraints) prohibitively large. For example, constraints are used to ensure that the control flow within a single task is properly followed as the introduction of MIP edges could allow certain paths to skip a node that should not be skipped or allow a node to be traversed multiple times, when it should not be. Such constraints

are called *task automata*. Constraints are also used to represent other concurrency semantics, such as a task in the TFG cannot start before it has been forked from its parent task, and to represent boolean variables that capture the information whether an exception has been thrown during a process execution.

Constraints are expressed as finite state machines (FSMs) whose alphabets consist of TFG node identifiers. Figure C.4 shows the task automaton corresponding to Task 1 from the TFG in Figure C.2. The alphabet of that task automaton are the identifiers of all nodes in Task 1 in the TFG, and the identifiers of the start and the end nodes. The initial state of that automaton (state 0) is marked with an arrow without a source node. Transitions on labels from the alphabet of the task automaton that are not shown go to the violation state by default. As a path from the start node of the TFG is being followed (representing a process execution in progress), the task automaton in Figure C.2 is updated based on the visited nodes. Suppose that during a given TFG traversal, the following path of TFG nodes is taken: 0, 1, 2, 3, 4, 5, 6, 15, 19, 17. Once node 1 in the TFG has been visited, the task automaton transitions from state 0 to state 1. When nodes 2, 3, 4, 5, and 6 are visited, the task automaton stays in state 1 as these node identifiers are not in the alphabet of the task automaton and do not affect it. After node 15 of the TFG is visited, the constraint transitions to state 2 and stays in that state after node 19 is visited. When node 17 is visited, the constraint enters the violation state. This means that the path through the TFG is not feasible, which is indeed the case, as node 17 from Task 1 was visited before its predecessor, node 16, was visited—this violates the control flow of Task 1.

If any of the constraint FSMs associated with a TFG enters its violation state (i.e., a non-accepting state all of whose outgoing transitions are self-loops) during a traversal of the TFG, the path is considered infeasible. Only paths from the start node of the TFG that do not violate any constraints are considered feasible.

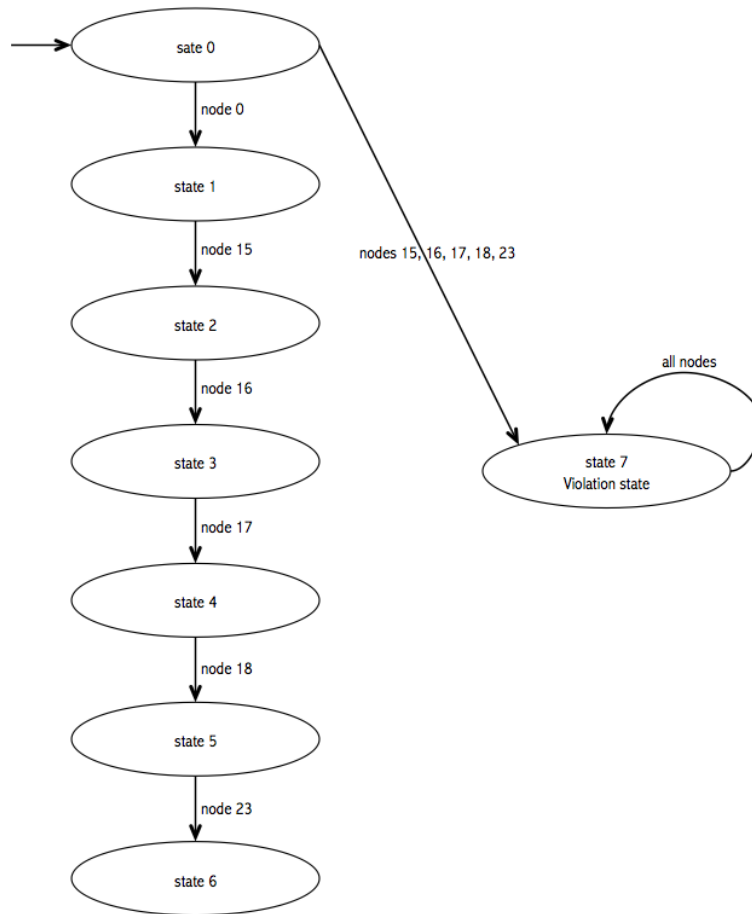


Figure C.4: The task automaton constraint for Task 1 of the TFG in Figure C.2.

C.1.1 Trace Flow Graph Traversal

Several of the components of the deviation detection experimental framework (Figure 4.4) presented in section 4.2 rely on a traversal of a trace flow graph derived from a high-level process model. To generate a sequence of steps, the Sequence Generator performs a walk from the start node of the TFG, using the Sequence Specifications to decide what node to visit at each step of the walk and when to stop the walk. During that walk, the Sequence Generator updates the constraint FSMs accompanying the TFG and ensures that none of the constraints is violated as the walk advances. Thus, the Sequence Generator performs a walk through the space determined by the cross product of the set of TFG nodes and the set of states of each constraint. We refer to this space as *node-tuple space*, where each position of a tuple is occupied by a state of a constraint FSM.

During the TFG traversal, the Sequence Generator keeps track of the labels of the visited nodes. The generated sequence is the sequence of labels of the visited nodes where only *relevant labels* are kept. Relevant labels are provided as part of the Sequence Specifications. Any subset of the TFG node labels can be chosen to be the relevant labels; relevant labels are usually determined by what process execution events are of interest and/or are captured by the Process Execution Monitor. For example, suppose that the process execution events of interest from the process modeled by the Little-JIL diagram in Figure C.3 are the starting of the leaf steps². This would result in the following set of relevant labels of nodes from the TFG in Figure C.2: $\{order\ chemo\ drugs\ from\ pharmacy\ STARTED, prepare\ chemo\ drugs\ STARTED, administer\ pre-meds\ STARTED, administer\ chemo\ drugs\ STARTED\}$. Suppose the Sequence Generator visits the following nodes on a TFG traversal: 0, 1,

²In Little-JIL, the leaf steps are the ones that agents perform and are thus most likely the steps to be captured by the Process Execution Monitor. Non-leaf steps are used primarily to provide abstraction and to specify control flow among leaf steps in a Little-JIL process model.

2, 3, 4, 5, 6, 7, 15, 16, 17, 18, 19, 20, 21, 22, 8, 9, 10, 11, 12, 13, 14, 23. This would result in a sequence of 24 labels (corresponding to the above sequence of nodes), but since only the labels of nodes 3, 16, 20, and 11 are relevant, the final generated sequence will be: *order chemo drugs from pharmacy STARTED*, *prepare chemo drugs STARTED*, *administer pre-meds STARTED*, *administer chemo drugs STARTED*. To keep the discussion at a higher level, in Chapter 4 the low-level process representation (i.e., the TFG and constraints) is not introduced and we refer to such a sequence of labels as a sequence of steps.

The Deviation Detector also relies on a traversal of the TFG (more precisely, a traversal of node-tuple space). The Deviation Detector takes as input a sequence of labels, the set of relevant labels, and the low-level process model (TFG and constraints), and then determines whether the given sequence is a legal sequence through the given model. To accomplish that, the Deviation Detector performs a guided breadth-first exploration of the node-tuple space determined by the TFG and the constraints. The deviation detector begins with the node-tuple consisting of the TFG's start node and the tuple containing the initial states of all constraint FSMs. The deviation detector then follows all paths from the start node-tuple, discarding paths that violate some of the constraints or paths whose sequence of labels (i.e., the sequence of labels of the nodes on the path) is not the same as the given sequence of labels³. If all paths are discarded before the Deviation Detector finds a path whose sequence of labels is the same as the given sequence of labels, a deviation is detected.

For efficiency, the Deviation Detector explores node-tuple space incrementally—beginning from the TFG start node, the Deviation Detector first tries to find paths to nodes whose label is the same as the first label in the given sequence of labels. If it finds paths to such nodes, the Deviation Detector places these nodes (and their

³Similar to the Sequence Generator, the deviation detector ignores node labels that are not relevant during the breadth-first traversal.

corresponding tuples) on a frontier⁴ (we refer to such node-tuples, i.e., node-tuples where the node’s label is relevant, as *relevant node tuples* and to the node itself as a *relevant node*). Then, starting from the nodes on the frontier, the Deviation Detector looks for paths to other relevant nodes with a label that is the same as the second label in the given sequence of labels, and places such nodes on the frontier. The Deviation Detection continues this exploration until it either reaches the end of the given sequence of labels (in this case, the sequence of labels is a legal sequence through the TFG) or until the frontier is empty before the end of the given sequence of labels is reached, meaning that the given sequence of labels is not a legal sequence through the TFG, and therefore a deviation is detected.

C.2 Issues and Optimizations

This section presents some of the issues associated with using the low-level process model representation and some optimization we performed to tackle these issues.

C.2.1 Construction of Low-level Process Model

We ran into several issues related to the construction of the low-level process model. As previously discussed, important real-world HIPs, such as medical processes, often encompass complex behaviors, such as concurrent execution and the handling of various exceptional situations. To adequately represent real-world HIPs and thus be useful for process guidance, models of HIPs need to capture such complex behaviors. This results in large and intricate process models. For example, the Little-JIL model of a blood transfusion process we used in our studies contained 220 Little-JIL steps (including 102 leaf steps) and captured 63 exception handling situations.

⁴A TFG node can have multiple corresponding tuples, because there could be different paths through the TFG leading to the same node, where each path changes the states of the constraints differently.

The corresponding low-level process models are even larger as a high-level construct, such as a Little-JIL step, is translated into multiple low-level constructs, such as TFG nodes and edges and states and transitions of FSM constraints associated with the TFG. For instance, the very simple Little-JIL process model in Figure C.3, which has 6 steps, is translated into a TFG with more than 24 nodes and close to 100 edges⁵. When a high-level process model is larger and more complex (e.g, it contains exception handling, other more complex control flow (such as Little-JIL try and choice steps), and artifacts), than the one in Figure C.3, then the explosion of the size of the low-level model with respect to the size of the high-level model from which it was derived is more severe. This could result in low-level process models that are too large for their construction to be practical. For example, when we tried to construct the low-level process models corresponding to the Little-JIL models of the blood transfusion and chemotherapy processes we studied, the translation did not finish in more than several days and eventually the Java Virtual Machine (the translation is implemented in Java) ran out of heap memory.

In previous work on applying model checking to low-level process models derived from high-level Little-JIL models [29], the explosion of the size of the low-level process model was tackled by not constructing parts of the model that are not relevant to the property of interest. Since in this work the properties of interest contained a small number of events (usually fewer than 5) that correspond to a small number of steps in the Little-JIL process model, most of the steps in the process model were not relevant to the property of interest resulting in a significant reduction of the size of the low-level process model.

For the purposes for deviation detection, however, most of the steps in the process model are relevant, because process performers could potentially perform any of the

⁵As previously discussed, not all nodes and edges of the TFG in Figure C.2 are shown to reduce visual clutter.

executions allowed by the process model. Thus, applying the above optimization did not result in a significant reduction of the size of the low-level process models needed for deviation detection. We applied a similar optimization, called *alphabet refinement*, where we specified certain TFG node labels from the TFG alphabet as irrelevant (e.g., labels that represent the posting of a Little-JIL step or labels that represent some action on an artifact) and removed nodes with such labels from the TFG. This did not result in a significant reduction of the size of the low-level process model, however, as many nodes with irrelevant labels need to be kept in the TFG to conserve paths that represent feasible process executions.

Given that to support deviation detection it is necessary to keep most of the low-level process model, we tackle the problem of feasibly constructing the low-level model by not explicitly constructing parts of that model and storing them only implicitly. In particular, we identified two parts of the model that are expensive to compute and too large to store—MIP edges and certain task automata constraints.

MIP edges represent possible interleavings between nodes in tasks that can happen in parallel. Thus, there could be a large number of MIP edges between parallel tasks and the number of MIP edges could potentially be quadratic in the number of nodes in these tasks. We modified the translation from Little-JIL to TFG to not explicitly compute and store MIP edges. Instead, the deviation detection experimental framework provides a routine to compute MIP edges at run-time and on-demand. The Sequence Generator and the Deviation Detector use that routine when they traverse the TFG.

Task automata constraints, as previously described, accompany a TFG to ensure that the control flow within a single task is not violated during a traversal of the TFG. TFGs derived from large and realistic process models can have large tasks, which in turn can have very large corresponding task automata. In fact, when creating the low-level models for the blood transfusion and chemotherapy processes, we ran into

a similar problem as with the MIP edges—some task automata were too expensive to compute and also eventually caused the translator to run out of heap memory. To tackle this problem, we modified the translation from Little-JIL to the low-level process model to not explicitly compute and store all transitions in a task automaton. Instead, we associated rules with each task automaton that the deviation detection framework uses at run-time to compute task automata transitions on demand.

Before we applied the optimizations for implicit MIP edges and task automata transitions, we were not able to construct the low-level models for the Little-JIL models of the blood transfusion and chemotherapy processes. After these optimizations, we could construct the low-level models in less than 20 minutes each (for the construction of the low-level model we used a MacBook laptop with 2.4 GHz Intel Core 2 Duo processor, running Java with 2.5 GB maximum heap size). The low-level models for the blood transfusion and chemotherapy processes were 328 MB and 19 MB respectively.

C.2.2 Traversal of Low-level Process Model

The optimizations discussed above help with the construction of the low-level process model by making parts of it implicit and not constructing these parts. These optimizations, however, do not reduce the size of the model at runtime (i.e., while the model is being traversed), resulting in a large search space that needs to be explored. The size of that search space becomes a serious problem for the Deviation Detector, which, beginning from the TFG’s start node, needs to keep track of all paths whose sequence of relevant labels⁶ is the same as the given sequence of labels. As discussed in section C.1.1, the Deviation Detector traverses the node-tuple space incrementally,

⁶A path (i.e., a sequence of nodes) through the TFG has a corresponding sequence of labels, which are the labels of the nodes on the path. If we delete from that sequence all the labels that are not relevant (as previously discussed, the relevant labels are given as input to the Deviation Detector), we are left with what we call the *relevant sequence of labels of the path*.

beginning from the TFG start node-tuple, first finding all relevant node-tuples whose node label is the same as the first label in the given sequence of labels, then from these node-tuples finding all node-tuples whose node label is the same as the second label in the given sequence, and so on. This exploration of node-tuple space could be quite expensive, however, as there could be multiple paths between two relevant nodes in the TFG and there could be multiple nodes with the same label, resulting in a large number of paths.

There could be multiple TFG nodes with the same label, because the same step could occur in different parts of a process and it will have different corresponding TFG nodes to represent it. There could be multiple paths between two relevant nodes, because one could potentially be reachable from the other on a different sequence of irrelevant nodes in between. For example, consider the TFG in Figure C.2 and the set of relevant labels we used in the discussion before: $\{order\ chemo\ drugs\ from\ pharmacy\ STARTED, prepare\ chemo\ drugs\ STARTED, administer\ pre-meds\ STARTED, administer\ chemo\ drugs\ STARTED\}$. To reach relevant node 16 from relevant node 3, for example, at least three different paths can be taken: 2, 3, 4, 5, 6, 15, 16; 2, 3, 4, 5, 6, 7, 19, 15, 16; and 2, 3, 4, 5, 6, 7, 19, 20, 15, 16. Each of these paths would also result in a different node-tuple that would need to be placed on the frontier that the Deviation Detector is maintaining and each such node-tuple would need to be explored.

To reduce the number of paths between two relevant nodes that the Deviation Detector needs to explore, we take advantage of domain knowledge. In particular, we identify situations in processes (if such situations exist) where two or more subprocesses can happen in any order with each other, but not in parallel, meaning that steps in different subprocesses cannot be interleaved (we call such subprocesses *shuffled* subprocesses). In such situations, we add additional constraints to the low-level process model to eliminate paths that interleave steps of different shuffled subpro-

cesses. For example, such constraints can be added for the low-level model of the simplified blood transfusion process in Figure 4.2 where *check blood product expiration date* and *check blood product info matches patient info* can happen in any order with each other, but not in parallel. Such constraints cannot be added to the low-level model of the simplified chemotherapy process in Figure C.3, however, because *prepare chemo drugs* and *administer pre-meds* can happen in any order, including in parallel.

Another optimization we performed is related to speeding up the finding of relevant nodes from a given node in the TFG. As the TFG is traversed, every time a successor node is obtained, it needs to be checked that all TFG constraints are not violated. Given that the number of constraints for the low-level model of realistic high-level process models could be large, this check could be expensive. Furthermore, this check is performed many times during the traversal of a TFG. To speed this check up, we precomputed which constraints are affected when visiting a TFG node. Given that most TFG nodes affect only a small number of constraints, this optimization resulted in a significant speed-up of the check of whether any TFG constraint is violated upon visiting a node.

C.3 Correspondence between TFG with constraints and ECFG

The ECFG process model representation used in Chapter 4 is a simplified representation that is introduced to keep the discussion in that chapter at a high level. The ECFG representation is less expressive than the Little-JIL representation and the Little-JIL's corresponding low-level representation—the TFG with constraints. In this section, we discuss the correspondence of the ECFG representation to the TFG with constraints representation.

The ECFG nodes are a subset of the TFG nodes. In particular, the ECFG nodes are the nodes of interest, the relevant nodes, from the TFG. These nodes of interest

consist of nodes that correspond to events captured by the Process Execution Monitor (e.g., starting of a process step) and nodes that represent the forking and joining of concurrent tasks. TFG nodes that represent forking and joining of tasks correspond to fork and join nodes in the ECFG, respectively.

There is an edge from node A to node B in the ECFG if A and B are in the same CFG (task) in the TFG and there is a path from node A to node B that does not violate any of the TFG constraints (i.e., there is a feasible path from A to B). MIP edges from the TFG are not explicitly represented in the ECFG; the possible interleavings of nodes from different CFGs in the TFG is captured by the fork/join semantics of the ECFG.

Unlike the TFG, the ECFG does not have any accompanying constraints. Thus, any path from a start node of the ECFG is considered feasible.

BIBLIOGRAPHY

- [1] “Adonis,” www.boc-group.com/at.
- [2] “ARIS,” www.ids-scheer.de.
- [3] “Eclipse Process Framework (EPF),” www.eclipse.org/epf.
- [4] “Modelplex, IST European Project contract IST-3408,” <http://www.modelplex-ist.org>.
- [5] “PERICLES-GECKO,” <http://www.gecko.de/referenzen/forschung.html>.
- [6] “Project PERICLES,” <http://www.perikles.org/index.html>.
- [7] “XSL transformations (XSLT) version 2.0,” www.w3.org/TR/xslt20.
- [8] “YAWL: Yet another workflow language,” <http://www.yawlfoundation.org/>.
- [9] “DoD modeling and simulation management,” *Department of Defense Directive 5000.59*, p. 7, 2007.
- [10] “Web services business process execution language version 2.0,” <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf>, April 2007.
- [11] “Chemotherapy administration safety standards,” *American Society of Clinical Oncology*, 2009.
- [12] A. Advani and Y. Musen, “Medical quality assessment by scoring adherence to guideline intentions,” in *Proceedings of the AMIA Annual Symposium*, 2001, pp. 2–6.
- [13] H. Agrawal, J. Horgan, S. London, and W. Wong, “Fault localization using execution slices and dataflow tests,” in *Proceedings of the Sixth International Symposium on Software Reliability Engineering.*, 1995, pp. 143–151.
- [14] G. S. Avrunin, L. A. Clarke, L. J. Osterweil, S. C. Christov, B. Chen, E. A. Henneman, P. L. Henneman, L. Cassells, and W. Mertens, “Experience modeling and analyzing medical processes: UMass/Baystate medical safety project overview,” in *Proceedings of the 1st ACM International Health Informatics Symposium*, 2010, pp. 316–325.

- [15] G. S. Avrunin, L. A. Clarke, L. J. Osterweil, J. M. Goldman, and T. Rausch, "Smart checklists for human-intensive medical systems," in *42nd International Conference on Dependable Systems and Networks, Workshop on Open, Resilient, Human-aware, Cyber-physical Systems, IEEE/IFIP*, 2012, pp. 1–6.
- [16] A. Babenko, L. Mariani, and F. Pastore, "AVA: automated interpretation of dynamically detected anomalies," in *ISSTA '09: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, 2009, pp. 237–248.
- [17] S. Baffoe, A. Baarah, and L. Peyton, "Inferring state for real-time monitoring of care processes," in *Proceedings of the 5th International Workshop on Software Engineering in Health Care (SEHC)*, May 2013, pp. 57–63.
- [18] J. B. Battles, H. S. Kaplan, T. W. Van der Schaaf, and C. E. Shea, "The attributes of medical event-reporting systems: experience with a prototype medical event-reporting system for transfusion medicine," *Archives of Pathology and Laboratory Medicine*, vol. 122, no. 3, pp. 231–238, 1998.
- [19] U. Becker-Kornstaedt, D. Hamann, R. Kempkens, P. Rösch, M. Verlage, R. Webby, and J. Zettel, "Support for the process engineer: the Spearmint approach to software process definition and process guidance," in *Proceedings of the 11th International Conference on Advanced Information Systems Engineering*, 1999, pp. 119–133.
- [20] U. Becker-Kornstaedt and M. Verlage, "The V-modell guide: experience with a web-based approach for process support," in *Proceedings of Software Technology and Engineering Practice (STEP)*, 1999, pp. 161–168.
- [21] C. Billings, "Some hopes and concerns regarding medical event-reporting systems: lessons from the NASA aviation safety reporting system," *Archives of Pathology and Laboratory Medicine*, vol. 122, no. 3, pp. 214–215, 1998.
- [22] J. D. Birkmeyer, "Strategies for improving surgical quality—checklists and beyond," *New England Journal of Medicine*, vol. 363, pp. 1963–11 965, 2010.
- [23] D. Boorman, "Today's electronic checklists reduce likelihood of crew errors and help prevent mishaps," *International Civil Aviation Organization Journal*, vol. 56, no. 1, pp. 17–36, 2001.
- [24] A. Bryman, "Integrating quantitative and qualitative research: how is it done?" *Qualitative Research*, vol. 6, no. 1, pp. 97–113, 2006.
- [25] H. H. Bui, S. Venkatesh, and G. West, "Policy recognition in the abstract hidden Markov model," *Journal of Artificial Intelligence Research*, vol. 17, pp. 451–499, 2002.
- [26] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi, "Specification and implementation of exceptions in workflow management systems," *ACM Transactions on Database Systems*, 1999.

- [27] A. G. Cass, B. S. Lerner, J. Stanley M. Sutton, E. K. McCall, A. Wise, and L. J. Osterweil, “Little-JIL/Juliette: a process definition language and interpreter,” in *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, 2000, pp. 754–757.
- [28] A. G. Cass and L. J. Osterweil, “Process support to help novices design software faster and better,” in *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005, pp. 295–299.
- [29] B. Chen, “Improving processes using static analysis techniques,” Ph.D. dissertation, University of Massachusetts Amherst, 2011.
- [30] B. Chen, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil, “Automatic fault tree derivation from Little-JIL process definitions.” in *Software Process Workshop (SPW 2006) and Process Simulation Workshop (PROSIM 2006)*, ser. LNCS, vol. 3966, 2006, pp. 150–158.
- [31] B. Chen, G. S. Avrunin, E. A. Henneman, L. A. Clarke, L. J. Osterweil, and P. L. Henneman, “Analyzing medical processes,” in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 623–632.
- [32] S. C. Christov, B. Chen, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, D. Brown, L. Cassells, and W. C. Mertens, “Rigorously defining and analyzing medical processes: an experience report,” *Models in Software Engineering: Workshops and Symposia at MoDELS 2007, Reports and Revised Selected Papers*, pp. 118–131, 2008.
- [33] S. C. Christov, G. S. Avrunin, and L. A. Clarke, “Considerations for online deviation detection in medical processes,” in *Proceedings of the 5th International Workshop on Software Engineering in Health Care (SEHC)*, 2013, pp. 50–56.
- [34] S. C. Christov, G. S. Avrunin, and L. A. Clarke, “Online deviation detection for medical processes,” in *American Medical Informatics Association Annual Symposium (AMIA)*, 2014.
- [35] S. C. Christov, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, and E. A. Henneman, “A benchmark for evaluating software engineering techniques for improving medical processes,” in *Proceedings of the 2010 ICSE Workshop on Software Engineering in Health Care*, 2010, pp. 50–56.
- [36] S. C. Christov, B. Chen, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, D. Brown, L. Cassells, and W. C. Mertens, “Formally defining medical processes,” *Methods of Information in Medicine. Special Topic on Model-Based Design of Trustworthy Health Information Systems*, vol. 47, no. 5, pp. 392–398, 2008.
- [37] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2000.

- [38] L. A. Clarke, A. Gaitenby, D. Gyllstrom, E. Katsh, M. Marzilli, L. J. Osterweil, N. K. Sondheimer, L. Wing, A. Wise, and D. Rainey, “A process-driven tool to support online dispute resolution,” in *Proceedings of the 2006 International Conference on Digital Government Research*, 2006, pp. 356–357.
- [39] H. Cleve and A. Zeller, “Locating causes of program failures,” in *Proceedings of the 27th international conference on software engineering*, 2005, pp. 342–351.
- [40] H. M. Conboy, J. K. Maron, S. C. Christov, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, and M. A. Zenati, “Process modelling of aortic cannulation in cardiac surgery: toward a smart checklist to mitigate the risk of stroke,” in *5th Workshop on Modeling and Monitoring of Computer Assisted Interventions (M2CAI)*, 2014.
- [41] L. G. Connelly and A. Bair, “Discrete event simulation of emergency department activity: a platform for system-level operations research,” *Academic Emergency Medicine*, vol. 11, no. 11, pp. 1177–1185, 2004.
- [42] J. E. Cook and A. L. Wolf, “Software process validation: quantitatively measuring the correspondence of a process to a model,” *ACM Transactions on Software Engineering and Methodology*, vol. 8, pp. 147–176, 1999.
- [43] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng, “Bandera: extracting finite-state models from Java source code,” in *Proceedings of the 22nd International Conference on Software Engineering*, 2000, pp. 439–448.
- [44] B. Crandall, G. A. Klein, and R. R. Hoffman, *Working minds: a practitioner’s guide to cognitive task analysis*. MIT Press, 2006.
- [45] G. Cugola, E. D. Nitto, A. Fuggetta, and C. Ghezzi, “A framework for formalizing inconsistencies and deviations in human-centered systems,” *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, pp. 191–230, 1996.
- [46] F. Curbera, R. Khalaf, F. Leymann, and S. Weerawarana, “Exception handling in the BPEL4WS language,” in *Conference on Business Process Management*, 2003.
- [47] M. A. A. da Silva, R. Bendraou, J. Robin, and X. Blanc, “Flexible deviation handling during software process enactment,” in *Enterprise Distributed Object Computing Conference Workshops (EDOCW)*, 2011, pp. 34–41.
- [48] M. A. A. da Silva, R. Bendraou, X. Blanc, and M.-P. Gervais, “Early deviation detection in modeling activities of MDE processes,” in *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part II*, 2010, pp. 303–317.

- [49] M. A. A. da Silva, X. Blanc, and R. Bendraou, "Deviation management during process execution." in *ASE 2011: 26th IEEE/ACM International Conference On Automated Software Engineering*, 2011, pp. 528–531.
- [50] C. Damas, B. Lambeau, F. Roucoux, and A. van Lamsweerde, "Analyzing critical process models through behavior model synthesis," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 441–451.
- [51] E. N. de Vries, H. A. Prins, R. Crolla, A. J. den Outer, G. van Anandel, S. van Helden, W. S. Schlack, M. A. van Putten, D. J. Gouma, M. G. Dijkgraaf, S. M. Smorenburg, and M. A. Boormeester, "Effect of a comprehensive surgical safety system on patient outcomes," *New England Journal of Medicine*, vol. 363, pp. 1928–1937, 2010.
- [52] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.
- [53] W. E. Deming, *Out of the Crisis*. Cambridge: MIT Press, 1982.
- [54] J. Dubose, P. G. Teixeira, K. Inaba, L. Lam, P. Talving, B. Putty, D. Plurad, D. J. Green, D. Demetriades, and H. Gelzberg, "Measurable outcomes of quality improvement using a daily quality rounds checklist: one-year analysis in a trauma intensive care unit with sustained ventilator-associated pneumonia reduction," *Journal of Trauma*, vol. 69, no. 4, pp. 855–60, 2010.
- [55] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich, "Flow analysis for verifying properties of concurrent software systems," *ACM Transactions on Software Engineering and Methodology*, vol. 13, no. 4, pp. 359–430, 2004.
- [56] S. Fine, Y. Singer, and N. Tishby, "The hierarchical hidden Markov model: analysis and applications," *Machine Learning*, vol. 32, no. 1, pp. 41–62, 1998.
- [57] M. Fitzgerald, P. Cameron, C. Mackenzie, N. Farrow, P. Scicluna, R. Gocentas, A. Bystrzycki, G. Lee, G. O'Reilly, N. Andrianopoulos, L. Dziukas, D. J. Cooper, A. Silvers, A. Mori, A. Murray, S. Smith, Y. Xiao, D. Stub, F. T. McDermott, and J. V. Rosenfeld, "Trauma resuscitation errors and computer-assisted decision support," *Archives of Surgery*, vol. 146, no. 2, pp. 218–225, 2011.
- [58] P. Fong and J. Brent, "Exception handling in WebSphere Process Server and WebSphere Enterprise Service Bus," http://www.ibm.com/developerworks/websphere/library/techar-ticles/0705_fong/0705_fong.htm.
- [59] J. Fordyce, F. S. J. Blank, P. Pekow, H. A. Smithline, G. Ritter, S. Gehlbach, E. Benjamin, and P. L. Henneman, "Errors in a busy emergency department," *Annals of Emergency Medicine*, vol. 42, no. 3, pp. 324–333, 09 2003.

- [60] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [61] T. K. Gandhi, S. B. Bartel, L. N. Shulman, D. Verrier, E. Burdick, A. Cleary, J. M. Rothschild, L. L. Leape, and D. W. Bates, “Medication safety in the ambulatory chemotherapy setting,” *Cancer*, vol. 104, no. 11, pp. 2477–2483, 2005.
- [62] A. S. Gertner, C. Conati, and K. VanLehn, “Procedural help in Andes: generating hints using a Bayesian network student model,” in *Proceedings of the Fifteenth National Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*. American Association for Artificial Intelligence, 1998, pp. 106–111.
- [63] A. C. Graesser and K. Murray, “A question-answering methodology for exploring user’s acquisition and knowledge of a computer environment,” *Cognition, Computing, and Cooperation*, pp. 237–267, 1990.
- [64] G. Guest, A. Bunce, and L. Johnson, “How many interviews are enough?: an experiment with data saturation and variability,” *Field Methods*, vol. 18, no. 1, pp. 59–82, 2006.
- [65] K. Hafner, “A busy doctor’s right hand, ever ready to type,” *The New York Times*, January 12 2014.
- [66] C. Hagen and G. Alonso, “Exception handling in workflow management systems,” *IEEE Transactions on Software Engineering*, vol. 26, no. 10, pp. 943–958, 2000.
- [67] B. M. Hales and P. J. Pronovost, “The checklist: a tool for error management and performance improvement,” *Journal of Critical Care*, vol. 21, pp. 231–235, 2006.
- [68] L. A. Hassell, A. V. Parwani, L. Weiss, M. A. Jones, and J. Y, “Challenges and opportunities in the adoption of College of American Pathologists checklists in electronic format: perspectives and experience of reporting pathology protocols project (RPP2) participant laboratories.” *Archives of Pathology and Laboratory Medicine*, vol. 134, no. 8, pp. 1152–1159, 2010.
- [69] P. Haumer, “Increasing development knowledge with EPFC,” *Eclipse Review*, pp. 26–33, 2006.
- [70] E. A. Henneman, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, C. Andrzejewski Jr., K. Merrigan, R. Cobleigh, K. Frederick, E. Katz-Bassett, and P. L. Henneman, “Increasing patient safety and efficiency in transfusion therapy using formal process definitions,” *Transfusion Medicine Review*, vol. 21, no. 1, pp. 49–57, 2007.

- [71] E. A. Henneman, F. S. Blank, S. Gattasso, K. Williamson, and P. L. Henneman, "Testing a classification model for emergency department errors," *Journal of Advanced Nursing*, vol. 55, no. 1, pp. 90–99, 2006.
- [72] P. L. Henneman, D. L. Fisher, E. A. Henneman, T. A. Pham, Y. Y. Mei, R. Talati, B. H. Nathanson, and J. Roche, "Providers do not verify patient identity during computer order entry," *Academic Emergency Medicine*, vol. 15, no. 7, pp. 641–648, 2008.
- [73] P. L. Henneman, F. S. J. Blank, H. A. Smithline, H. Li, J. S. Santoro, J. Schmidt, E. Benjamin, and E. A. Henneman, "Voluntarily reported emergency department errors," *Journal of Patient Safety*, vol. 1, no. 3, 2005.
- [74] P. L. Henneman, D. L. Fisher, E. A. Henneman, T. A. Pham, M. M. Campbell, and B. H. Nathanson, "Patient identification errors are common in clinical simulation," *Annals of Emergency Medicine*, vol. 55, no. 6, pp. 503–509, 2009.
- [75] E. Hollnagel, "Plan recognition in modeling of users," in *Accident Sequence Modeling: Human Actions, System Response, Intelligent Decision Support*, G. Apostolakis, P. Kafka, and G. Mancini, Eds. Elsevier Science Publishers Ltd., 1988.
- [76] E. Hollnagel, "The design of fault tolerant systems: prevention is better than cure," *Reliability Engineering and System Safety*, vol. 36, no. 3, pp. 231–237, 1992.
- [77] E. Hollnagel, "The phenotype of erroneous actions," *International Journal of Man-Machine Studies*, vol. 39, pp. 1–32, July 1993.
- [78] J. T. James, "A new, evidence-based estimate of patient harms associated with hospital care," *Journal of Patient Safety*, vol. 9, no. 3, pp. 122–128, 2013.
- [79] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. ACM, 2002, pp. 467–477.
- [80] M. Kellner, U. Becker-Kornstaedt, W. Riddle, J. Tomal, and M. Verlage, "Process guides: effective guidance for process participants," in *Proceedings of the International Conference on the Software Process*, 1998, pp. 11–25.
- [81] B. Kirwan and L. K. Ainsworth, *A Guide To Task Analysis: The Task Analysis Working Group*. Taylor & Francis, 1992.
- [82] L. T. Kohn, J. M. Corrigan, and M. S. Donaldson, Eds., *To Err is Human: Building a Safer Health System*. National Academies Press, 1999.
- [83] J. B. Kruskal, "An overview of sequence comparison: time warps, string edits, and macromolecules," *SIAM Review*, vol. 25, no. 2, pp. 201–237, 1983.

- [84] R. Kühn, A. Dittmar, and P. Forbrig, “Alternative representations of workflow control-flow patterns using HOPS,” in *Perspectives in Business Informatics Research*. Springer Berlin Heidelberg, 2010, vol. 64, pp. 115–129.
- [85] H. A. Landsberger, *Hawthorne revisited*. Cornell University, 1958.
- [86] B. S. Lerner, S. C. Christov, L. J. Osterweil, R. Bendraou, U. Kannengiesser, and A. Wise, “Exception handling patterns for process modeling,” *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 162–183, 2010.
- [87] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” *Doklady Akademii Nauk SSSR*, vol. 163, no. 4, pp. 845–848, 1965.
- [88] N. Leveson, *Safeware: System Safety and Computers*. Addison-Wesley, 1995.
- [89] S. Lühr, H. H. Bui, S. Venkatesh, and G. A. W. West, “Recognition of human activity through hierarchical stochastic learning,” in *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*. IEEE Computer Society, 2003, pp. 416–422.
- [90] L. Mariani and F. Pastore, “Automated identification of failure causes in system logs,” in *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2008, pp. 117–126.
- [91] L. Mariani and M. Pezzè, “Dynamic detection of COTS component incompatibility,” *IEEE Software*, vol. 24, no. 5, pp. 76–85, 2007.
- [92] W. C. Mertens, D. E. Brown, R. Parisi, L. J. Cassells, D. Naglieri-Prescod, and D. J. Higby, “Detection, classification, and correction of defective chemotherapy orders through nursing and pharmacy oversight,” *Journal of Patient Safety*, vol. 4, no. 3, pp. 195–200, 2008.
- [93] W. C. Mertens, S. C. Christov, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, L. J. Cassells, and J. Marquard, “Using process elicitation and validation to understand and improve chemotherapy ordering and delivery,” *The Joint Commission Journal on Quality and Patient Safety*, vol. 38, no. 11, pp. 497–505, 2012.
- [94] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of unix utilities,” *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.
- [95] S. Nepal, A. Fekete, P. Greenfield, J. Jang, D. Kuo, and T. Shi, “A service-oriented workflow language for robust interacting applications,” in *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, ser. Lecture Notes in Computer Science, no. 3760. Springer, 2005, pp. 40–58.
- [96] OMG, “Business process modeling notation (BPMN), version 2.0.1,” <http://www.omg.org/spec/BPMN/2.0.1/>.

- [97] OMG, “Unified modeling language (UML), version 2.4.1,” <http://www.omg.org/spec/UML/2.4.1/>.
- [98] OMG, “Software process engineering meta-model, version 2.0,” <http://www.omg.org/spec/SPEM/2.0/>, 2008.
- [99] L. J. Osterweil, C. M. Schweik, N. K. Sondheimer, and C. W. Thomas, “Analyzing processes for e-government development: the emergence of process modeling languages,” *Journal of E-Government*, vol. 1, no. 4, pp. 63–89, 2004.
- [100] C. Ouyang, M. T. Wynn, J.-C. Kuhr, M. J. Adams, T. Becker, A. H. ter Hofstede, and C. J. Fidge, “Workflow support for scheduling in surgical care processes,” in *The 19th European Conference on Information Systems : ICT and Sustainable Service Development (ECIS 2011)*, 2011.
- [101] C. Ouyang, M. T. Wynn, C. Fidge, A. H. ter Hofstede, and J.-C. Kuhr, “Modelling complex resource requirements in business process management systems,” in *21st Australasian Conference on Information Systems : Defining and Establishing a High Impact Discipline (ACIS 2010)*, M. Rosemann, P. Green, and F. Rohde, Eds. ACIS, 2010.
- [102] H. Pan and E. H. Spafford, “Heuristics for automatic localization of software faults,” Purdue University, Tech. Rep. SERC-TR-116-P, 1992.
- [103] E. S. Patterson and J. E. Miller, *Macrocognition Metrics and Scenarios: Design and Evaluation for Real-World Teams*. Ashgate Publishing, 2010.
- [104] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.
- [105] C. A. Petri, “Communication with automata,” Ph.D. dissertation, University of Darmstadt, 1962.
- [106] M. Phongpaibul, S. Koolmanojwong, A. Lam, and B. Boehm, “Comparative experiences with electronic process guide generator tools,” in *Proceedings of the International Conference on Software Process*, 2007, pp. 61–72.
- [107] C. Phua, V.-F. Foo, J. Biswas, A. Tolstikov, A.-P.-W. Aung, J. Maniyeri, W. Huang, M.-H. That, D. Xu, and A.-W. Chu, “2-layer erroneous-plan recognition for dementia patients in smart homes,” in *11th International Conference on e-Health Networking, Applications and Services, 2009. Healthcom 2009*, 2009, pp. 21–28.
- [108] P. Pronovost, D. Needham, S. Berenholtz, D. Sinopoli, H. Chu, S. Cosgrove, B. Sexton, R. Hyzy, R. Welsh, G. Roth, J. Bander, J. Kepros, and C. Goeschel, “An intervention to decrease catheter-related bloodstream infections in the ICU,” *New England Journal of Medicine*, vol. 355, no. 26, pp. 2725–2732, 2006.

- [109] M. S. Raunak, L. J. Osterweil, A. Wise, L. A. Clarke, and P. L. Henne-
man, “Simulating patient flow through an emergency department using process-
driven discrete event simulation,” in *SEHC '09: Proceedings of the 2009 ICSE
Workshop on Software Engineering in Health Care*. IEEE Computer Society,
2009, pp. 73–83.
- [110] M. S. Raunak, B. Chen, A. Elssamadisy, L. A. Clarke, and L. J. Osterweil, “Def-
inition and analysis of election processes,” *Software Process Workshop (SPW
2006) and 2006 Process Simulation Workshop (PROSIM 2006)*, vol. 3966, pp.
178–185, 2006.
- [111] J. Reason, *Human Error*. Cambridge University Press, 1990.
- [112] M. Renieris and S. P. Reiss, “Fault localization with nearest neighbor queries,”
in *18th International Conference on Automated Software Engineering*, 2003, pp.
30–39.
- [113] A. Rozinat, “Conformance testing: measuring the alignment between event logs
and process models,” Eindhoven University of Technology, Tech. Rep., 2005.
- [114] A. Rozinat, “Conformance testing: measuring the fit and appropriateness of
event logs and process models,” in *BPM 2005 Workshops (Workshop on Busi-
ness Process Intelligence)*. Springer-Verlag, 2006, pp. 163–176.
- [115] A. Rozinat and W. M. P. van der Aalst, “Conformance checking of processes
based on monitoring real behavior,” *Information Systems*, vol. 33, no. 1, pp.
64–95, 2008.
- [116] N. Russell, W. van der Aalst, and A. ter Hofstede, “Exception handling patterns
in process-aware information systems.” BPMCenter.org, BPM Center Report
BPM-06-04, 2006.
- [117] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed.
Prentice Hall, 2003.
- [118] D. Sankoff and J. Kruskal, *Time warps, string edits, and macromolecules: the
theory and practice of sequence comparison*. Addison-Wesley, 1983.
- [119] W. A. Shewhart, *Economic control of quality of manufactured product*. D. Van
Nostrand Company, Inc, 1931.
- [120] B. I. Simidchieva, M. S. Marzilli, L. A. Clarke, and L. J. Osterweil, “Specifying
and verifying requirements for election processes,” in *Proceedings of the 2008 In-
ternational Conference on Digital Government Research*. Digital Government
Society of North America, 2008, pp. 63–72.
- [121] D. H. Stamatis, *Failure Mode and Effect Analysis: FMEA from Theory to Ex-
ecution*. American Society for Quality, 1995.

- [122] W. W. Stead and H. S. Lin, Eds., *Computational Technology for Effective Health Care: Immediate Steps and Strategic Directions*. National Academies Press, 2009.
- [123] E. G. Tessier, E. A. Henneman, B. Nathanson, K. Plotkin, and M. Heelon, “Pharmacy-nursing intervention to improve accuracy and completeness of medication histories,” *American Journal of Health-System Pharmacy*, vol. 15, pp. 607–611, 2010.
- [124] US Air Force, “Pilots’ abbreviated flight crew checklist, OV-10A aircraft,” *United States Air Force Series*, 1969.
- [125] W. van der Aalst, A. ter Hofstede, B. Keipuszewski, and A. P. Barros, “Workflow patterns,” *Distributed and Parallel Databases*, vol. 14, no. 3, pp. 5–51, July 2003.
- [126] T. W. van der Schaaf, “Near miss reporting in the chemical process industry: an overview,” *Microelectronics and Reliability*, vol. 35, no. 9-10, pp. 1233 – 1243, 1995.
- [127] W. van Vuuren, C. Shea, and T. van der Schaaf, “The development of an incident analysis tool for the medical field,” Eindhoven University of Technology, Tech. Rep., 1997.
- [128] W. Vesely, F. Goldberg, N. Roberts, and D. Haasl, *Fault Tree Handbook (NUREG-0492)*. U.S. Nuclear Regulatory Commission, Washington, D.C., January 1981.
- [129] D. Wang, J. Pan, G. S. Avrunin, L. A. Clarke, and B. Chen, “An automatic failure mode and effect analysis technique for processes defined in the Little-JIL process definition language,” in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE 2010)*, 2010, pp. 765–770.
- [130] J. M. Wilkinson and K. V. Leuven, “Procedure checklist for administering a blood transfusion,” http://davisplus.fadavis.com/wilkinson/Procedure_Checklists/PC_Ch36-01.doc.
- [131] J. M. Wilkinson and K. Van Leuven, *Fundamentals of Nursing*. F. A. Davis Company, 2007.
- [132] B. D. Winters, A. P. Gurses, H. Lehmann, J. B. S. ton, C. Rampersad, and P. J. Pronovos, “Clinical review: checklists—translating evidence into practice.” *Critical Care*, vol. 13, no. 6, p. 210, 2009.
- [133] D. D. Woods and E. Hollnagel, *Joint Cognitive Systems: Patterns in Cognitive Systems Engineering*. Taylor & Francis, 2006.

- [134] D. D. Woods, E. M. Roth, and K. B. Bennett, “Explorations in joint human-machine cognitive systems,” in *Cognition, Computing, and Cooperation*, S. P. Robertson, W. W. Zachary, and J. B. Black, Eds. Ablex Publishing Corp., 1990, pp. 123–158.
- [135] World Health Organization, “Implementation manual surgical safety checklist,” http://www.who.int/patientsafety/safesurgery/tools_resources/SSSL_Manual_finalJun08.pdf.
- [136] World Health Organization, “Surgical safety checklist,” http://www.who.int/patientsafety/safesurgery/tools_resources/SSSL_Checklist_finalJun08.pdf, 2008.
- [137] W. W. Zachary and S. P. Robertson, “Introduction to cognition, computation, and cooperation,” in *Cognition, Computing, and Cooperation*, S. P. Robertson, W. W. Zachary, and J. B. Black, Eds. Ablex Publishing Corp., 1990, pp. 1–19.