2017

Automatic Derivation of Requirements for Components Used in Human-Intensive Systems

Heather Conboy

AUTOMATIC DERIVATION OF REQUIREMENTS FOR COMPONENTS USED IN HUMAN-INTENSIVE SYSTEMS

A Dissertation Presented

by

HEATHER M. CONBOY

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

DOCTOR OF PHILOSOPHY

May 2017

Computer Science

AUTOMATIC DERIVATION OF REQUIREMENTS FOR COMPONENTS USED IN HUMAN-INTENSIVE SYSTEMS

A Dissertation Presented

by

HEATHER M. CONBOY

DEDICATION

To my parents

ACKNOWLEDGMENTS

First and foremost, I want to thank my dissertation advisors, Lori Clarke and George Avrunin, for their support and guidance throughout my software engineering career and graduate studies. They improved by ability to research, develop, and evaluate new software engineering techniques. They also made sure that I could give a high-level overview of such techniques as well as provide the precise lower-level details about them.

I also want to thank the other members of my thesis committee, Lee Osterweil and Maciej Ciesielski. I had many fruitful discussions with Lee Osterweil about various research projects. I benefitted from Maciej Ciesielski asking me thoughtful questions and giving me constructive feedback on the dissertation. I greatly appreciate Dimitra Giannakopoulou and Jamie Cobleigh for helping me understand how to employ the L^* learning algorithm and model checking techniques for this work.

I collaborated with many domain experts to put together case studies in the healthcare and election administration domains. I especially want to thank Beth Henneman, Jenna Marquard, the VA Boston Healthcare System cardiac surgery team, Kevin Fu, Matt Bishop, and the Marin/Yolo County election officials for sharing their knowledge about these case studies and providing feedback on the system models, their requirements, and the software engineering tools applied to them.

I want to express my appreciation for the past and current LASER members for providing such a collaborative and innovative research environment: Yuriy Brun, Rene Just, Manish Motwani, Seung Yeob Shin, Stefan Christov, Huong Phan, Bobby Simidchieva, Bin Chen, M.S. Raunuk, and Jamie and Rachel Cobleigh. I also benefitted from mentoring the masters and undergraduate students, most recently Nancy

Famigletti, Siyu Peng, and Sam Kolovson. I want to thank Sandy Wise for building many of the software engineering tools that I built on for this work and to thank Barb Lerner for mentoring me as a CS undergraduate researcher.

I also want to express my appreciation for the CS administrative and technical support staff. Leaanne Leclerc was always the voice of reason and answered any and all questions about the CS graduate program. Deb Bergeron assisted me with graduate appointments as well as travel funding. The CSCF helped me keep my personal computers alive and well.

I am grateful to my friends for helping me maintain a healthier life/work balance, especially the CS volleyball team, the CS yoga class, and the Xenophon farm community.

I want to thank my parents for always encouraging both their daughters' interest in math and science. They have always supported us in our studies and careers in computer science and civil engineering. I am particularly grateful for my parents, sister, and brother for their love, encouragement, and patience throughout my graduate studies.

ABSTRACT

AUTOMATIC DERIVATION OF REQUIREMENTS FOR COMPONENTS USED IN HUMAN-INTENSIVE SYSTEMS

MAY 2017

HEATHER M. CONBOY

B.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

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

Human-intensive systems (HISs), where humans must coordinate with each other along with software and/or hardware components to achieve system missions, are increasingly prevalent in safety-critical domains (e.g., healthcare). Such systems are often complex, involving aspects such as concurrency and exceptional situations. For these systems, it is often difficult but important to determine requirements for the individual components that are necessary to ensure the system requirements are satisfied. In this thesis, we investigated an approach that employs interface synthesis methods developed for software systems to automatically derive such requirements for components used in HISs.

In previous work, we investigated a requirement deriver that employs a regular language learning algorithm to iteratively refine the derived requirement based on counterexamples generated by model checking techniques. Since this learning-based requirement deriver often did not scale well, we investigated several learning and model checking optimizations. These optimizations significantly improved performance but affected the counterexample generation heuristics, often widely varying the permissiveness of the derived requirements. For comparison purposes, we investigated a direct requirement deriver that was purported to have poor performance but guarantees the derived requirements are adequately permissive, conceptually meaning the requirements are permissive as possible without violating the system requirements. For our evaluation, we applied these requirement derivers to case studies in two important domains, healthcare and election administration.

Based on this evaluation, the direct requirement deriver with all optimizations applied had reasonable performance and ensures the derived requirements are adequately permissive. For the learning-based requirement deriver, many of the optimizations and heuristics have been presented previously, but we recommend how to selectively combine them to obtain reasonable performance while usually producing the adequately permissive derived requirements.

Since such derived requirements often reflect the system complexity, these requirements can be easily misunderstood. Thus, we also investigated building views of the requirements that abstract away or highlight certain aspects to try to improve their understandability. Each single view appears to improve understandability and the multiple views seem to complement each other further improving understandability. Such derived requirements and their views can be used to safely develop and deploy the components used in HISs.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS v

ABSTRACT vii

LIST OF TABLES xii

LIST OF FIGURES xiii

CHAPTER

1. INTRODUCTION 1

2. BACKGROUND 7

2.1 Basic Definitions 9
2.2 Illustrative Example 10
2.3 Little-JIL Process Models 18
2.4 FLAVERS Model Checker 24
2.5 Interface Synthesis Methods 34

3. DERIVATION APPROACH 36

3.1 Basic Definitions 38
3.2 HIS Subject Translator 41
3.3 Pre-Requisite Check 45
3.4 Direct Requirement Deriver 46

3.4.1 FSA Extraction 48
3.4.2 Guarantees 51

3.5 Learning-Based Requirement Deriver 53

3.5.1 L^* Learner 57
3.5.2 FLAVERS-Based L* Teacher 60

3.5.2.1 Membership Query 60
3.5.2.2 Equivalence Query 63

3.5.3 Guarantees 66

3.6 Derived Requirement Permissiveness Classifier 70

4. DERIVATION IMPROVEMENTS 72

4.1 Derivation Optimizations 73

4.1.1 Variable Modeling Alternatives 75
4.1.2 Reset Dead Local Variables 79
4.1.3 Channel Modeling Alternatives 81
4.1.4 Incremental Membership Query 87

4.2 Permissiveness Counterexample Generation Heuristics 89

4.2.1 Ensure the System Requirement is Satisfied 91
4.2.2 Select Terminating Node-Tuples 92
4.2.3 Order of Node-Tuples in the Worklist 93
4.2.4 Determine Previously Visited Node-Tuples 93
4.2.5 Take N Steps Further 94

5. DERIVATION EVALUATION 97

5.1 Experimental Methodology 98

5.1.1 Evaluation Metrics 102
5.1.2 Requirement Deriver Results Ranking 105

5.2 Experimental Results 105
5.3 Discussion 111
5.4 Preliminary Evaluation of Java Pathfinder Heuristics 112
5.5 Threats to Validity 118

6. DERIVED REQUIREMENT VIEWS 120

6.1 Procedure Abstraction View 121
6.2 Modal Abstraction View 123
6.3 Safe Alternatives View 127
6.4 Evaluation 130

7. RELATED WORK 140

7.1 Compositional Verification 140

7.2 Automated Synthesis Methods 142

7.2.1 Scenario-based Development Frameworks 143
7.2.2 Specification Mining Methods 145
7.2.3 Interface Synthesis Methods 146

7.3 Determining Requirements about the Human Participants Interactions with Components Used in HISs 148

8. CONCLUSIONS 153

APPENDICES

A. BUILD NODE-TUPLE GRAPH ALGORITHM DESCRIPTION 159
B. LOW-LEVEL DERIVED REQUIREMENT ALPHABET DESCRIPTION 162

BIBLIOGRAPHY 165

LIST OF TABLES

Table **Page**

3.1 Correspondence between event sequences and paths 42

5.1 Derivation Optimizations Applied 99

5.2 Derivation Heuristics Applied 100

5.3 Subjects Used in Experiments 101

5.4 Comparison of Query-Based Heuristics in Terms of Permissiveness 109

5.5 Summary of the top 5 ranked requirement derivers in terms of the best performance with regards to maximum space needed 111

6.1 Mapping for the surgery derived requirement from call-return event pairs to procedure abstraction events 122

6.2 Summary of the applicable views for the derived requirements 131

6.3 Sets of Calling Context Events 132

6.4 Mapping from Mode Change Events to Mode Name 133

6.5 Summary of Modal Abstraction View Results 134

6.6 Mapping from Phase Change Events to Phase Name 135

6.7 Summary of Phase Abstraction View Results 136

6.8 Summary of Safe Alternatives View Results 138

B.1 Alphabets for the process and the component 164

LIST OF FIGURES

Figure **Page**

2.1 System requirement property for never overdose represented as an FSA 13

2.2 Pseudo-code for the abstract pump's *setDose* procedure 14

2.3 View of the derived pump component requirement represented as an FSA 16

2.4 Pseudo-code for the concrete pump's *setDose* procedure 17

2.5 View of the derived surgery process requirement represented as an FSA 19

2.6 Overall "performPumpHIS" model written in Little-JIL 20

2.7 Elaboration of the concrete "perform in-patient surgery" step 21

2.8 Elaboration of the "perform infusion" step 22

2.9 Variable constraint for the *setDose_dose* parameter with enumerated type { LOW, HIGH } 27

2.10 Portion of a TFG for the pump's *setDose* procedure 28

2.11 System requirement property for never report a dose alert (represented as event "setDose_DoseAlert=TRUE") 29

2.12 Portion of the node-tuple graph for the *setDose* procedure where each node-tuple shows the TFG node (top) and the tuple (bottom) 30

3.1 Overview of our HIS-based requirement derivation approach 36

3.2 Non-deterministic node-tuple graph 40

3.3 Overview of the direct requirement deriver 47

3.4 Overview of the learning-based requirement deriver 55

3.5 Pseudo-code for the learner's updateT method 58

3.6 Pseudo-code for the updateT method that incorporates the prefix-closed optimization 59

3.7 Pseudo-code for the teacher's membership query method 62

3.8 Pseudo-code for the teacher's permissiveness counterexample generation method 66

4.1 Portion of the TFG for the *setDose* procedure constructed when the first alternative for modeling the *currLib_*, *currDose_*, and *setDose_DoseAlert* variables is applied and the second alternative for modeling the *setDose_dose* variable is applied 77

4.2 Portion of the TFG for the *setDose* procedure shown in Figure 2.10 where the reset dead local variables optimization was applied to the *setDose_dose* variable 80

4.3 Variable constraint for channel *setDoseRetChn* that stores a message of boolean type 82

4.5 Channel constraint for channel *setDoseRetChn* that stores a message of boolean type 85

4.7 Pseudo-code for the extended permissiveness counterexample generation method that varies along the five dimensions (shown as underlined) that parameterize some of the key helper methods 91

5.1 Performance of the top 5 ranked requirement derivers in terms of the maximum space on the 5 largest subjects 110

5.2 Performance of the top 5 ranked requirement derivers in terms of the log of the time on the 5 largest subjects 111

6.1 Implicit violation view of the surgery derived requirement 121

6.2 Procedure abstraction view of the surgery derived requirement (that was also shown in Figure 2.5) 124

6.3 Modal abstraction view of the surgery derived requirement 126

6.4 Safe alternatives view of the surgery derived requirement 129

A.1 Pseudo-code for the *build node-tuple graph* algorithm 161

CHAPTER 1
INTRODUCTION

Human-intensive systems (HISs), where *human participants* need significant domain expertise to perform activities that affect the outcome of the system mission, are becoming more prevalent. Many of these systems, such as certain medical procedures, require the human participants to coordinate their activities among themselves as well as with software applications and/or hardware devices. For such systems, it is often difficult but important to determine requirements for individual software and hardware components that are needed to ensure that the overall system requirements are satisfied. For these systems, the HIS coordination is often complex, typically involving non-deterministic choices, concurrent activities, and non-nominal (or exceptional) situations. When the HIS is taken into account when determining the component requirements, the complex coordination makes it likely that some of the various component usages may be misunderstood or even overlooked. Therefore some of the component requirements may contain subtle errors or even be missing all together, potentially leading to overall system requirement violations that have disastrous outcomes, such as patients being injured or even killed. In this thesis, we investigated an automated requirement derivation approach that employs interface synthesis methods developed for software systems (e.g., [4, 46]). The derive requirements for a particular component restricts the behaviors of that component to prevent any violations of the system requirements.

Throughout this thesis, we will illustrate this requirement derivation approach on a HIS that consists of a medical process for an in-patient surgery in which an infusion pump is used to administer intravenous fluids and medications. One important system requirement for such a system is that a patient never be administered a medication overdose[1]. Many modern infusion pumps can be programmed with drug libraries for particular areas of the hospital that for the drugs commonly used in those areas specify the dosing limits (e.g., concentrations, units, and range of doses). For instance, the drug library for an operating room typically allows a wider range of drugs and dosing limits than other areas in the hospital. For the in-patient surgery medical process, a clinician must configure a given infusion pump, shortened to pump in what follows, for a particular drug library, enter the drug, enter the dose, etc., to ensure that the pump issues the appropriate alerts. If during the surgery an exceptional situation occurs, the patient with their attached pump is sometimes moved from the operating room to another area of the hospital that uses a different drug library. If the pump developers do not take into account such exceptional situations, a requirement that states the pump must be reconfigured after being moved might be overlooked or, perhaps more likely, the requirement might contain subtle errors.

Interface synthesis methods generally take as input a system implementation along with an overall system requirement. Such methods focus on the interface between a selected component and the remaining system. These methods produce an interface automaton that restricts the sequences of the component procedure executions to prevent any system requirement violations. For our work, each HIS in the case studies is formally modeled as a composition of a selected component model and a process model that describes the recommended ways to achieve the system mission, including how that component will be used. In our setting, the synthesized interface automata will be used as derived requirements on the interface between the component and the process that, when met, ensure that the overall system requirements

[1]In a dual manner, another important system requirement is that a patient never be administered a medication underdose.

are satisfied. The interface synthesis methods define the language of a synthesized interface automaton to be *safe* when that automaton disallows all actual sequences of the component procedure executions that may violate the overall system requirement (e.g., disallows sequences that lead to overdoses). For this work, it is crucial that the derived requirements are safe to prevent failures of the system mission. Ideally, these requirements are also *adequately permissive*, meaning the automaton allows all actual sequences of the component procedure executions that always satisfy the system requirement (e.g., allows sequences that lead to necessary doses).

For a *process perspective*, the model of the HIS consists of an imprecise component model and a detailed process model in which the component will be used. The imprecise component model, in the least restrictive case, essentially allows the component to behave in an arbitrary fashion, often leading to violations of the overall system requirement. The derived requirement for the component restricts the wide range of behaviors of the component to ensure that the overall system requirement is satisfied. The derived component requirements can be used to help developers understand how the recommended process impacts the component's design and implementation, to help certify that a component will be safely used by the recommended process, or to help select a component to be safely used in the recommended process.

For a *component perspective*, the HIS model consists of a detailed component model and an imprecise process model in which that component will be used. The derived requirement for the process restricts the wide range of behaviors of the process, in particular with regards to using the component, to ensure that the overall system requirement is satisfied. The derived process requirements can be used to help determine the class of processes in which such a component can be safely used, develop training materials for human participants to learn how to safely use the component, or to help monitor the process at run-time to ensure the component is safely used. The interface synthesis methods are typically applied from the component perspective.

In previous work, we investigated such an automated requirement derivation approach [23] that built on interface synthesis methods (e.g., [4]) that employ regular language learning algorithms (e.g., [6]) and model checking techniques (e.g., [19]). Specifically, the learning algorithm iteratively refines the current derived requirement based on counterexamples generated by the model checker. The final derived requirement is represented as a minimal deterministic finite-state automaton (FSA). This requirement is guaranteed to be safe but not necessarily adequately permissive. Because this learning-based interface synthesis method often did not scale well, we investigated several learning and model checking optimizations (e.g., [15, 25]). All of these optimizations improved the performance of the requirement derivation but some of the model checking optimizations interacted with the counterexample generation, affecting the permissiveness of the derived requirements. Thus, we investigated several counterexample generation heuristics (e.g., [41]) to try to increase the permissiveness of the derived requirements. Specifically, we wanted to generate more *permissiveness counterexamples*, meaning actual system executions disallowed by the derived requirement but that always satisfy the system requirement. The learning-based requirement deriver refines the derived requirement to allow the permissiveness counterexamples. Our goal is to recommend a combination of the optimizations and heuristics that makes the best tradeoff for the learning-based requirement deriver results between performance and permissiveness.

For comparison purposes, we also investigated a direct interface synthesis method (e.g., [12,36]) that first uses a model checker to generate the full reachability graph of the system model. This method then extends regular language algorithms (e.g., [1]) to convert from the reachability graph, which is essentially represented as a non-deterministic FSA, to a minimal deterministic FSA. The method produces derived requirements that are guaranteed to be safe and adequately permissive. To improve the performance of the direct interface synthesis method, we incorporated the same model checking optimizations as we did for the learning-based interface synthesis method.

The interface synthesis methods, however, often produce derived requirements that reflect the inherent complexity of the systems and therefore are difficult to understand. Some of these methods (e.g., [74]) produce derived requirements specified in higher-level representations such as statecharts to improve their understandability. Alternatively, other of the methods, including the direct and learning-based methods that we considered, produce derived requirements specified as minimal deterministic FSAs. Thus, we create different views of the FSAs that abstract away or highlight certain aspects of the FSAs to improve their understandability. This allows multiple interface synthesis methods to share the views. This also allows the same FSA to apply multiple views to further improve understandability.

For this evaluation, we developed a toolset that supports such an automated requirement derivation approach and applied that toolset to case studies from two important domains, healthcare (e.g., [23]) and election administration (e.g., [64])). This toolset provides support for a learning-based requirement deriver that incorporates the learning and model checking optimizations along with the counterexample generation heuristics. The toolset also provides support for a direct requirement deriver that incorporates only the model checking optimizations. Additionally, the requirement derivation toolset provides support to automatically build the derived requirement views. Our approach was flexible enough to automatically derive requirements that provided insights about the component, process, or both. We found that each optimization applied improved the performance of both interface synthesis methods. The direct interface synthesis method scaled well and produced derived requirements that were safe and adequately permissive. For the learning-based interface synthesis method, however, we needed to carefully tailor the permissiveness counterexample generation heuristics for the optimizations applied in order to improve permissiveness without significantly worsening performance. Based on our experimental results, we identified a single heuristic that, when combined with all optimizations being applied, usually led to the best derivation results in terms of permissiveness and performance. We applied multiple views, whenever possible, to each derived requirement which seemed to improve the understandability of that requirement and showed that the views can complement each other.

In this thesis, Chapter 2 provides background on the HIS modeling language, the model checker, and the interface synthesis methods used. Chapter 3 more fully describes our automated requirement derivation approach, which employs either a direct or learning-based interface synthesis method. Chapter 4 describes optimizations to both interface synthesis methods to improve their performance and counterexample generation heuristics for the learning-based interface synthesis method to improve the permissiveness of the derived requirements. Chapter 5 describes our experiments to evaluate the effect of the optimizations and heuristics on the performance of the requirement derivers as well as on the permissiveness of the derived requirements. Chapter 6 describes multiple views of the derived requirements and their impact on the understandability of the derived requirements. Chapter 7 discusses related work and Chapter 8 presents our conclusions and some possible directions for future work.

CHAPTER 2
BACKGROUND

As described in the introduction, our automated requirement derivation approach takes as input a formal model of the overall HIS along with one of its system requirements. For presentation purposes, we consider each HIS model to be composed of a selected component model and a process model. In general, there may be multiple component models, multiple process models, or both. The process model specifies the recommended sequences of activities that must be performed by the human participants and components to achieve the overall system mission. For this work, each component is represented by its *interface* (represented by a set of procedure definitions) that may be used by the remaining system (usually represented by procedure executions consisting of procedure call/return pairs). This approach is built on interface synthesis methods that employ model checking techniques. The derived requirements (often represented by interface automata) restrict the sequences of the selected component's procedures executed by the process to prevent any overall system requirement violations. We next provide overviews of this requirement derivation approach, the HIS modeling language, the model checking techniques, and the interface synthesis methods.

For this work, the HIS models must reflect the real-world complexity, capturing concurrent activities, non-deterministic choices, human participants communications with each other and the components, and exceptional situations. They also must have precisely defined semantics to support formal analyses such as model checking techniques. We chose to write the HIS models in the Little-JIL process modeling language [13], which is sufficiently expressive and precise for our purposes. In Little-JIL, an overall system is hierarchically decomposed into the individual activities (or *steps*) that the human and automated components must perform to achieve the system mission. These steps can be thought of as procedures.

The model checking techniques verify that all paths through a system model satisfy a given overall system requirement. The verification algorithms often explicitly or symbolically generate a reachability graph that explores the possible reachability nodes that encode the internal state of the system model such as program counters and run-time variable values. Some of the verification algorithms can also generate *counterexample paths*, meaning paths through the reachability graph that demonstrate possible system requirement violations. For this work, a given HIS model written in Little-JIL annotated with the system events can be automatically translated to what is essentially a labeled transition system [8]. That labeled transition system can then be automatically translated to the input formalisms of two model checkers, FLAVERS and Spin. The translation includes multiple optimizations. For this work, we use FLAVERS that supports various model checking optimizations and verification algorithms to generate the full reachability graph as well as to iteratively generate counterexamples.

Interface synthesis methods employ various techniques to automate the reasoning about the system requirements and system model such as reachability analysis (e.g., [36]), learning algorithms (e.g., [4]), counterexample-guided abstraction refinement (e.g., [46]), and game theory (e.g., [77]). In the next chapter, we describe a direct interface synthesis method based on reachability analysis techniques that employs a model checker to build the full reachability graph of the system model. Additionally, we describe a learning-based interface synthesis method that employs a learning algorithm to iteratively refine a derived requirement based on counterexamples generated by a model checker. Both interface synthesis methods considered use the model checker to determine whether or not a given event sequence may violate the overall system requirement. If so, the derived requirement should disallow that event sequence. If not, the derived requirement should allow the event sequence. In the following sections, we will more fully describe the illustrative example, Little-JIL, FLAVERS, and the interface synthesis methods.

2.1 Basic Definitions

We use FSAs to specify the overall system and derived requirements because they can express a wide range of interesting requirements and are formally defined to support automated analyses such as model checking techniques. Each FSA specifies the intended or unintended sequences of the system events. For the requirement derivation, the system events usually correspond to the selected component's procedure calls and their returns along with any calling contexts.

We specify that each call to a key step (or procedure) (e.g., *setLib*) on given input parameter values (e.g., ICU) corresponds to a call event denoted as follows:

- **call**($procedureName, inputParameterValues$) (e.g., **call**($setLib, ICU$))

We also specify that each return from that procedure (e.g., *setLib*) with particular output parameter values (e.g., OK) corresponds to a return event denoted as follows:

- **return**($procedureName, outputParameterValues$) (e.g., **return**($setLib, OK$))

On the other hand, we specify that each return from the procedure with specific exceptions thrown corresponds to a return event denoted as follows:

- **return**($procedureName, exceptionsThrown$)

Additionally, we specify calling context events. For instance, we specify that the start of the step "administer ICU care" corresponds to the calling context event $enterICU$ while the completion of that step corresponds to the calling context event $leaveICU$.

Thus, one partial system execution trace for the illustrative example is as follows: $enterICU$, $\mathbf{call}(setLib, ICU)$, $\mathbf{return}(setLib, OK)$, $leaveICU$.

For this work, we use FSAs that are *deterministic*, meaning for each state s in the FSA and for each event e in the FSA's alphabet there exists at most one transition from FSA state s on event e. We also use FSAs that are *total*, meaning for each state s in the FSA and for each event e in the FSA's alphabet there exists a transition from FSA state s on event e. If an FSA is not total, we add a special violation state that is a non-accepting state and a *trap state*, meaning the violation state has a transition to itself for every event in the alphabet. Therefore any event sequence that reaches the violation state will remain in the violation state. For each FSA state s and for each event e in the FSA's alphabet, if there does not exist a transition from FSA state s on event e, then we then add a transition from FSA state s on event e to the FSA's violation state. Given an FSA A with alphabet Σ_A, we use $\mathcal{L}(A)$ to denote the language of FSA A, meaning all of the event sequences from Σ_A^* that are accepted by A. In a similar manner, we use $\neg\mathcal{L}(A)$ to denote the complement of the language of FSA A, meaning all event sequences from Σ_A^* that are rejected by A.

2.2 Illustrative Example

For each of our case studies, a HIS model is composed of a component model and a process model in which that component will be used. The component models were based on available user manuals and how the components are to be used in the processes, as well as consultation with domain experts. The process models in which the components are to be used were developed in prior work (e.g., [8, 64]) and are based on extensive elicitation and validation efforts involving both computer scientists and experts in the relevant domains. In the introduction, we mentioned one of our case studies [7] that considers a model for a HIS that consists of a component model for an infusion pump and a medical process model for an in-patient surgery in which the pump will be used. The overall system requirement was taken directly from the safety criteria discussed in [7]. For this illustrative example, we first provide more details about the HIS model and the overall system requirement. We then briefly describe how to apply our automated requirement derivation approach from both the process perspective and the component perspective.

For the illustrative example, we use a very simplified model of a real-world infusion pump, shortened to pump in what follows. There are only two drug libraries modeled, a drug library for an OR and a drug library for an ICU. Each drug library contains a single drug, and the only dosing limits modeled are a minimum and maximum. The drug doses are abstracted as either LOW or HIGH. For the pump, we consider only the following three procedures: *setLib*, *setDose*, *start*. The *setLib* procedure inputs a primary care area (either OR or ICU) and configures the pump with the drug library associated with that area. The *setDose* procedure inputs a drug dose (either LOW or HIGH) and checks whether or not that dose is within the dosing limits for the configured drug library. If so, this procedure stores the configured dose and then returns OK (i.e. does not report a dose alert). If not, the procedure throws a DoseExceedsLimits exception (i.e. does report a dose alert). In the introduction, we described how the OR drug library permits both LOW and HIGH doses but the ICU drug library permits only LOW doses. The *start* procedure checks whether or not there is a configured dose. If so, this procedure administers that dose and then returns OK. If not, the procedure throws a DoseAlert exception, specifically a MissingDose exception.

In general for the in-patient surgery process model, we elaborated those steps where the medical clinicians interacted with the infusion pump, based on a demonstration given by Professor Elizabeth Henneman from the University of Massachusetts College of Nursing. At a high-level of abstraction, an in-patient surgery process model consists of five key steps: checking the patient into the hospital, performing the operation on that patient, administering ICU care if needed, monitoring the patient during recovery, and checking that patient out of the hospital. The pump is used in the context of either the OR or the ICU.

Each high-level system requirement is represented as one or more low-level properties. Each property is specified as an FSA that captures the intended or unintended behaviors as a set of event sequences. Figure 2.1 shows the system requirement property mentioned in the introduction specified as an FSA. This property informally states that when a pump is in an ICU and is set to deliver a dose over the allowed limit for the ICU, it must report a dose alert. In the simplified pump component model, the ICU drug library specifies that a LOW dose is within the dosing limits but a HIGH dose exceeds the dosing limits. For the property, the alphabet, or set of events, is {enterICU, leaveICU, **call**$(setDose, HIGH)$, **return**$(setDose, OK)$, **return**$(setDose, DoseAlert)$}. There are 4 states and 16 transitions.

In Figure 2.1, each state is shown as a circle annotated with a unique name. Since in the medical process model the medical clinicians first use the pump outside the ICU, specifically in the OR, the start state is annotated with 1 (denoted by the right arrow). Additionally since this model allows the pump to remain in the OR, start state 1 is also an accepting state (denoted by the inner concentric circle). Each transition from a given state to a specified state on a particular event is shown as an arc between those two states annotated with that event. To illustrate, the pump may be moved from an OR to an ICU so there is a transition from the start state 1 to state 2 annotated with event $enterICU$. In this figure, the violation state and its transitions are shown in red. For simplicity, the figures in the rest of this thesis that show the FSAs usually do not show the violation states and their transitions. Thus if a state does not show a transition annotated with a particular event in the alphabet, then there is implicitly a transition on that event to the violation state. For the never overdose property represented as an FSA, one event sequence that satisfies this FSA is $enterICU$, $leaveICU$. Alternatively, an event sequence that violates the FSA is $enterICU$, $\textbf{call}(setDose, HIGH)$, $\textbf{return}(setDose, OK)$, $leaveICU$.

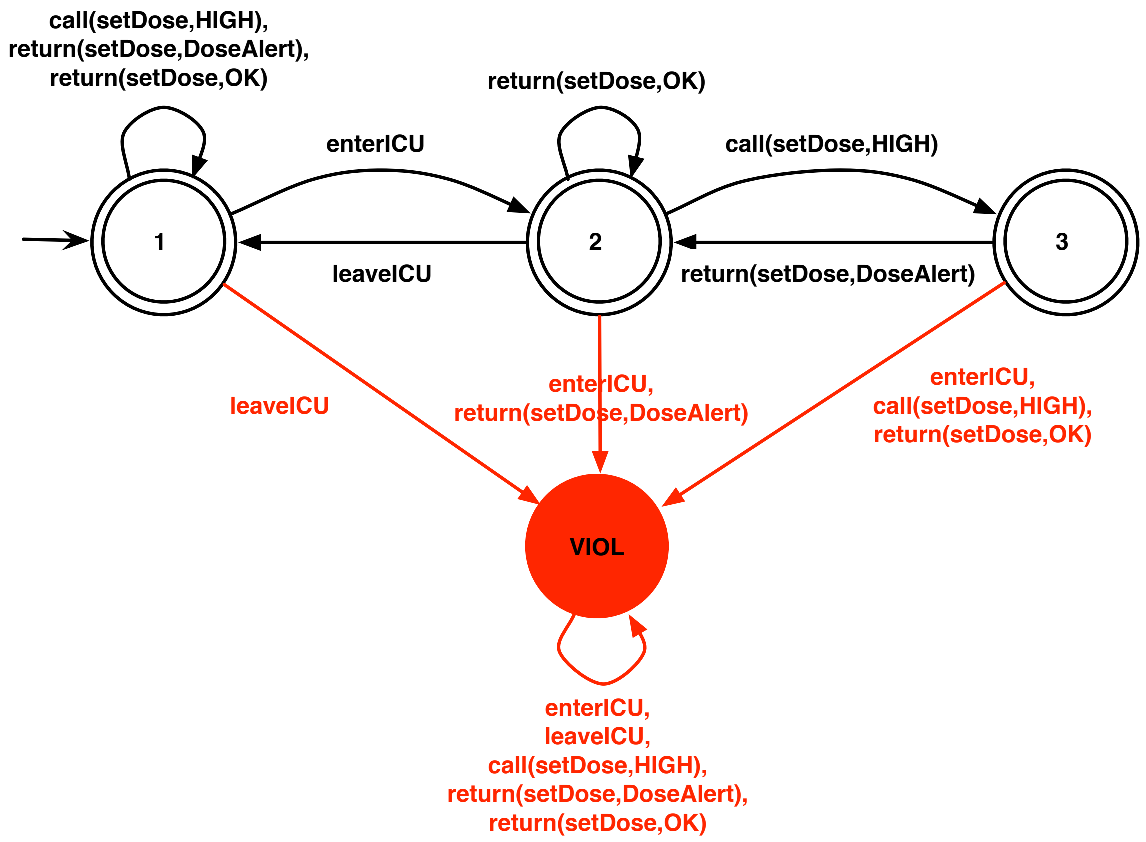


Figure 2.1: System requirement property for never overdose represented as an FSA

For the process perspective, our approach conceptually takes as input the "never overdose" system requirement along with a HIS model composed of a concrete in-patient surgery process model and a very abstract pump component model. This approach produces a derived pump component requirement. In the process model, the medical clinicians may use the pump in the context of an OR or an ICU. For the nominal scenario, the clinician must first configure the pump using the *setLib* and *setDose* procedures and then use the *start* procedure to administer the medication or fluid to the patient. Figures 2.7 and 2.8 show portions of the in-patient surgery process model written in Little-JIL. The abstract pump component model does not store the pump's internal state and does not implement the procedures' internal logic. In related work (e.g, [4, 35]), the component models are written in programming languages such as Java. For this work, the models are written in Little-JIL. For presentation purposes, the pump's procedures are written as pseudo-code that indicates how the pump's internal logic has been abstracted away or concretely implemented. To illustrate, Figure 2.2 shows pseudo-code for the abstract pump's *setDose* procedure. In this figure, this procedure non-deterministically chooses (line 2) to either not report a dose alert by returning OK (line 3) or to report a dose alert by throwing a *DoseExceedsLimits* exception (line 5). The return and throw statements are annotated with the corresponding return events. Such an abstract pump component model permits the pump to behave in an arbitrary manner, which may violate the system requirements. Given the system requirement shown in Figure 2.1 and the HIS model described in the previous sentences, our requirement derivation approach produces the derived component requirement shown in Figure 2.3. For this figure, we actually show the procedure abstraction view of the derived component requirement that will be described in Section 6.1. In this view, each procedure call event (e.g., **call**(*setDose*, *HIGH*) was paired with its corresponding return event (e.g., **return**(*setDose*, *DoseAlert*) to form a procedure execution event (e.g., setDose(HIGH)_DoseAlert)).

```
1: procedure SETDOSE(dose: in Dose) throws DoseExceedsLimits
2:     if (choose() = false) then
3:         return                          ▷ Event return(setDose, OK)
4:     else
5:         throw DoseExceedsLimits         ▷ Event return(setDose, DoseAlert)
6:     end if
7: end procedure
```

Figure 2.2: Pseudo-code for the abstract pump's *setDose* procedure

This requirement informally specifies that the nominal scenario is to set the drug library, set the dose, and then start the infusion. The requirement also specifies an exceptional scenario where after the pump is configured for the ICU that pump cannot be configured with a HIGH dose and therefore reports a dose alert. In this figure, each state was automatically annotated with a unique integer ID and manually annotated with both the configured drug library and dose. Each transition is annotated with an event that corresponds to one of the pump's procedures (e.g., the event setDose_HIGH_DoseAlert). Initially, the pump is configured with no drug library and no dose (represented by start state 1). The pump may be configured with either of the two drug libraries, in particular the ICU (represented by states 2 and 4) or the OR (represented by states 3 and 5). If the pump is configured for the ICU, then the pump may be configured with a LOW dose (represented by state 4). On the other hand, if the pump is configured for the OR, then the pump may be configured with either a LOW or HIGH dose (represented by state 5).

For the component perspective, this approach basically takes as input the "never overdose" system requirement along with a HIS model composed of a very abstract in-patient surgery process model and a concrete pump component model. The approach produces a derived surgery process requirement. In more detail, the concrete pump component model stores its internal state in global variables and for each procedure implements the internal logic to check and update that internal state, including reporting dose alerts. Specifically, the pump component model declares two global variables, one to store the configured drug library (*currLib_*) and another to store the configured dose (*currDose_*). The pump component model also declares and implements the *setLib*, *setDose*, and *start* procedures. Figure 2.4 shows pseudo-code for the concrete *setDose* procedure that indicates the pump's internal logic for that procedure. This procedure first checks the current run-time value of the *currLib_* global variable (line 2). If the *currLib_* is OR, the procedure first sets the *currDose_*

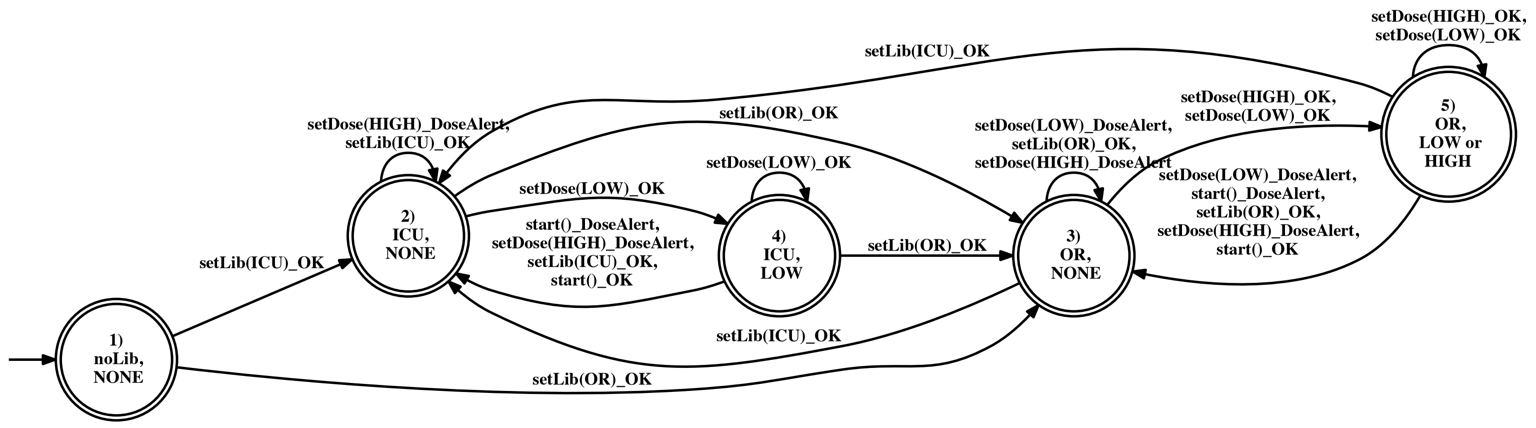

1) noLib, NONE
2) ICU, NONE
3) OR, NONE
4) ICU, LOW
5) OR, LOW or HIGH
setLib(ICU)_OK
setLib(OR)_OK
setDose(HIGH)_DoseAlert, setLib(ICU)_OK
setDose(LOW)_OK
start()_DoseAlert, setDose(HIGH)_DoseAlert, setLib(ICU)_OK, start()_OK
setLib(OR)_OK
setLib(ICU)_OK
setDose(LOW)_DoseAlert, setLib(OR)_OK, setDose(HIGH)_DoseAlert
setDose(HIGH)_OK, setDose(LOW)_OK
setDose(LOW)_DoseAlert, start()_DoseAlert, setLib(OR)_OK, setDose(HIGH)_DoseAlert, start()_OK
setDose(HIGH)_OK, setDose(LOW)_OK
setLib(ICU)_OK
setLib(OR)_OK
setLib(ICU)_OK

Figure 2.3: View of the derived pump component requirement represented as an FSA

global variable to the *dose* input parameter (line 3). The procedure does not report a dose alert and returns OK (line 4). If the *currLib_* is ICU (line 5), then the *setDose* procedure checks the current run-time value of the *dose* input parameter (line 6). If the *dose* is HIGH (line 6), then the procedure does report a dose alert by throwing a *DoseExceedsLimits* exception (line 7). If the *dose* is LOW (line 8), this procedure first sets the *currDose_* global variable to that dose (line 9) and then returns (line 10). The return and throw statements are annotated with their corresponding return events. The abstract process model for the in-patient surgery basically uses the pump's procedures in an arbitrary manner. In more detail, the surgery process model contains a main loop where on each iteration one of the pump's procedures is used. The surgery process model non-deterministically chooses when to stop looping and which of the pump's procedures is used. Given the system requirement shown in Figure 2.1 and the HIS model described in the preceding sentences, this requirement derivation approach produces the derived process requirement shown in Figure 2.5. In this figure, we actually show the procedure abstraction view of the derived process requirement.

```
 1: procedure setDose(dose: in Dose) throws DoseExceedsLimits
 2:     if (currLib_ = OR) then
 3:         $currDose\_ \leftarrow dose$
 4:         return                                   ▷ Event return($setDose, OK$)
 5:     else if (currLib_ = ICU) then
 6:         if (dose = HIGH) then
 7:             throw DoseExceedsLimits          ▷ Event return($setDose, DoseAlert$)
 8:         else if (dose = LOW) then
 9:             $currDose\_ \leftarrow dose$
10:             return                               ▷ Event return($setDose, OK$)
11:         end if
12:     end if
13: end procedure
```

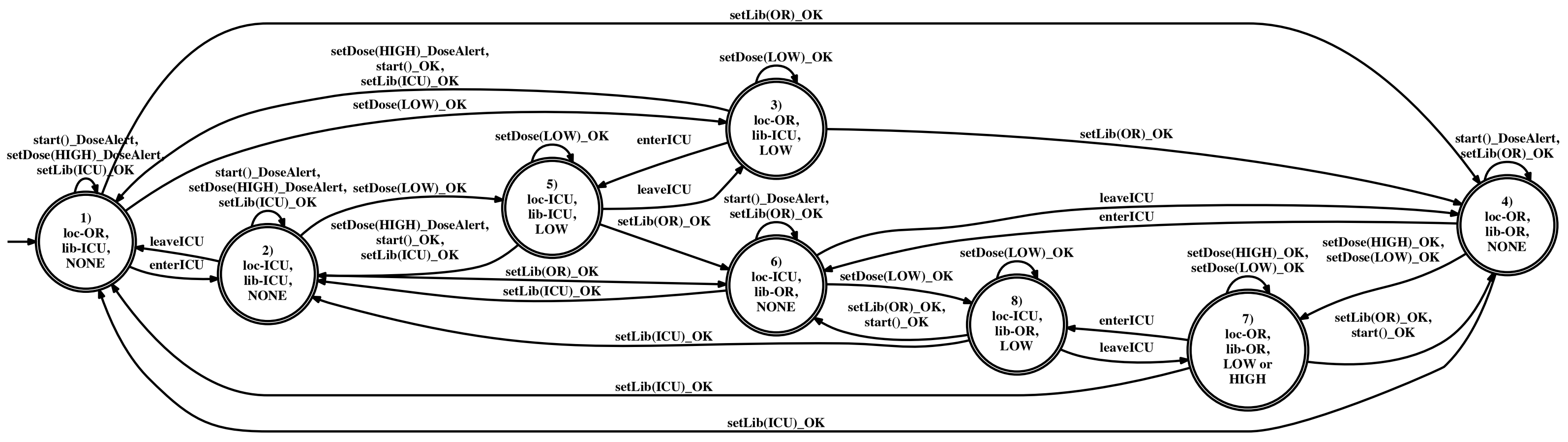
Figure 2.4: Pseudo-code for the concrete pump's *setDose* procedure

This requirement informally specifies that the nominal scenario is to move the pump to a primary care area, set the drug library for that area, set the dose, and then start the infusion. The requirement also specifies an exceptional scenario where after the pump is configured for the ICU that pump cannot be configured with a HIGH dose. In this figure, each state was automatically annotated with a unique integer ID and manually annotated with the primary care area (or location) along with both the configured drug library and dose. Each transition is annotated with an event that corresponds to one of the pump's procedures being executed (e.g., the event $setDose(HIGH, DoseAlert)$) or moving the pump to a new primary care area (e.g., the event *enterICU* corresponds to moving the pump into an ICU).[1] Within the in-patient surgery process model, the pump is initially located in the OR. Additionally, the pump is initialized with the most restrictive drug library, which in this model is the ICU's drug library, along with no dose. Thus, the start state 1 is annotated with loc-OR, lib-ICU, NONE. The pump may be configured with either of the two drug libraries, in particular the ICU (represented by states 1, 2, 3, and 5) or the OR (represented by states 4, 6, 7, and 8). If the pump is configured for the ICU, then the pump may be configured with a LOW dose (represented by state 5). On the other hand, if the pump is configured for the OR, then the pump may be configured with either a LOW or HIGH dose (represented by state 7).

2.3 Little-JIL Process Models

Each Little-JIL process model precisely captures how to perform the individual activities needed to achieve the system mission. The model contains three main parts, a resource repository, an artifact collection, and a set of coordination specification diagrams. The resource repository defines which agents, either human agents (e.g.,

[1]This illustrates a derived requirement that uses the calling context (i.e. the primary care area).



1) loc-OR, lib-ICU, NONE
2) loc-ICU, lib-ICU, NONE
3) loc-OR, lib-ICU, LOW
4) loc-OR, lib-OR, NONE
5) loc-ICU, lib-ICU, LOW
6) loc-ICU, lib-OR, NONE
7) loc-OR, lib-OR, LOW or HIGH
8) loc-ICU, lib-OR, LOW
start()_DoseAlert, setDose(HIGH)_DoseAlert, setLib(ICU)_OK
start()_DoseAlert, setDose(HIGH)_DoseAlert, setLib(ICU)_OK
start()_DoseAlert, setLib(OR)_OK
start()_DoseAlert, setLib(OR)_OK
setDose(LOW)_OK
setDose(LOW)_OK
setDose(LOW)_OK
setDose(HIGH)_OK, setDose(LOW)_OK
setDose(HIGH)_DoseAlert, start()_OK, setLib(ICU)_OK
setDose(HIGH)_DoseAlert, start()_OK, setLib(ICU)_OK
setDose(LOW)_OK
setDose(LOW)_OK
leaveICU
enterICU
enterICU
leaveICU
setLib(OR)_OK
setLib(OR)_OK
setLib(ICU)_OK
setLib(OR)_OK
leaveICU
enterICU
setDose(LOW)_OK
setLib(OR)_OK, start()_OK
enterICU
leaveICU
setDose(HIGH)_OK, setDose(LOW)_OK
setLib(OR)_OK, start()_OK
setLib(OR)_OK
setLib(ICU)_OK
setLib(ICU)_OK
setLib(ICU)_OK

Figure 2.5: View of the derived surgery process requirement represented as an FSA

anesthesiologists, nurses) or automated agents (e.g., electronic order entry system, pump), perform the activities. The artifact collection defines what artifacts are taken as input and/or produced as output by the activities. For instance, an artifact could be a (communication) channel, a parameter, or an exception. Each coordination specification diagram is a visual representation of a process or sub-process that consists of a hierarchical decomposition of steps (essentially procedures). Each step represents how an activity is performed by a particular agent using the input artifacts to produce the output artifacts.

For the motivating example, Figure 2.6 shows the "performPumpHIS" step that represents the overall HIS model written in Little-JIL. In the Little-JIL figure, the "performPumpHIS" step concurrently executes (denoted by the equal sign on the left hand side of the step bar) the "performInPatientSurgery" step that represents the process model and the "IteratePump" step that represents the component model. Both the "performInPatientSurgery" and "IteratePump" steps are elaborated in other diagrams. In the previous section, we already provided a high-level description of the "IteratePump" step elaboration. From the process perspective, we next describe the "performInPatientSurgery" step elaboration.

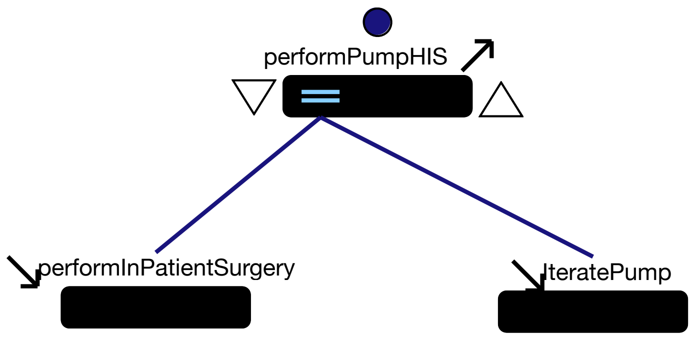


Figure 2.6: Overall "performPumpHIS" model written in Little-JIL

For the process perspective, the "performInPatientSurgery" step corresponds to a concrete in-patient surgery process model while the "IteratePump" step corresponds to a very abstract pump component model. The “performInPatientSurgery” step shown in Figure 2.7 sequentially executes (denoted by the right arrow on the left hand side of the step bar) the following substeps: “perform check-in”, “perform surgery activities”, “perform check-out.” The “perform surgery activities” step may be repeated zero or more times (denoted by the ‘*‘ on the incoming arc). This step sequentially executes the “perform operation” step, optionally followed by the “administer ICU care” step (denoted by the ’?’ on the incoming arc), and lastly the “perform post-operative care” step. Both the “perform operation” and “administer ICU care” steps use the “perform infusion” step shown in Figure 2.8 to configure the pump and then to administer the medication or fluid to the patient.

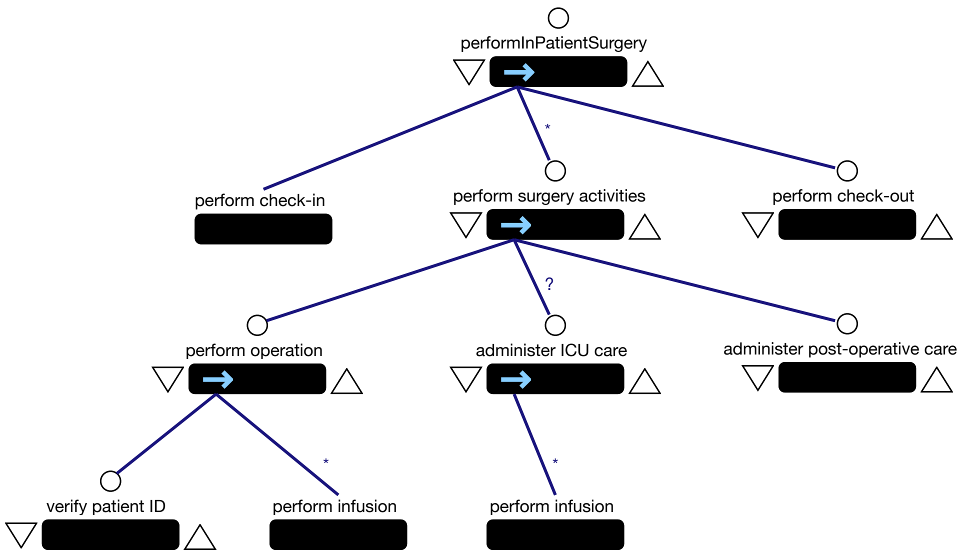


Figure 2.7: Elaboration of the concrete “perform in-patient surgery” step

In this figure, the “perform infusion” step sequentially executes the “set library for pump” substep followed by the “configure and administer dose” substep to administer a prescribed medication or fluid to the patient in a particular primary care area of the hospital. The step “set library for pump” takes as input the area’s drug library either OR or ICU and returns no outputs denoted OK (shown in the yellow note below the step bar). This step uses the pump’s *setLib* procedure. The step "configure and administer dose" may be executed zero or more times. This step first performs the step "set dose for pump" that takes as input the entered dose either LOW or HIGH and then either returns no outputs denoted OK or else throws a *DoseExceedsLimits* exception. This step uses the pump's *setDose* procedure. If a *DoseExceedsLimits* exception is thrown, then the handler "respond to dose exceeds limits alert" is performed. If not, then the step "administer infusion" may optionally be performed. This step first performs the step "start pumping" that may throw a *MissingDose* exception type. This step uses the pump's *start* procedure. If a *MissingDose* exception is thrown, then the handler "respond to missing dose alert" is performed. If not, then the step "stop pumping" is performed. The medical clinician then decides either to repeat this entire sequence, assuming that dosage is not entered correctly (i.e. mistyped a number), or to not administer the dosage at all until double checking with someone else that the dosage was the appropriate one.

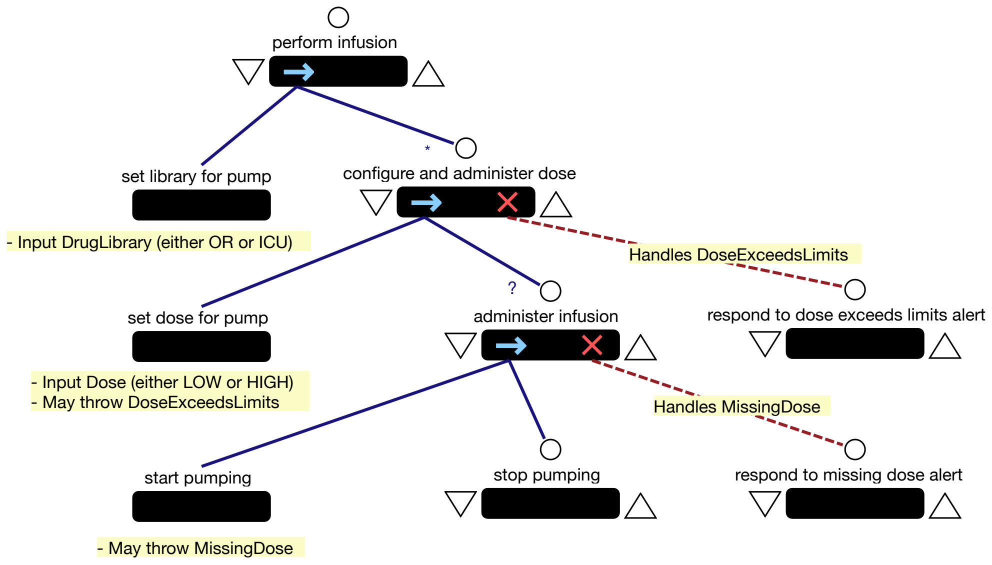

perform infusion
set library for pump
- Input DrugLibrary (either OR or ICU)
*
configure and administer dose
Handles DoseExceedsLimits
set dose for pump
- Input Dose (either LOW or HIGH)
- May throw DoseExceedsLimits
?
administer infusion
respond to dose exceeds limits alert
Handles MissingDose
start pumping
- May throw MissingDose
stop pumping
respond to missing dose alert

Figure 2.8: Elaboration of the "perform infusion" step

For both perspectives, we next provide a low-level description of how the HIS model represents the environment (e.g., in-patient surgery process) executing one of the selected component's procedures (e.g., the pump's *setDose* procedure). This model performs the following sequence of system activities:

1. The environment calls one of the selected component's procedures with any input parameter values (e.g., $\mathbf{call}(setDose, HIGH)$),

2. That component executes the named procedure on the given input parameter values (e.g., $setDose(HIGH)$)

3. The environment returns from that procedure with either any output parameter values or any exceptions thrown (e.g., $\mathbf{return}(setDose, DoseAlert)$).

For this work, the environment and component are executing in parallel and thus the procedure calls must be treated as remote procedure calls. We chose to model the remote procedure calls with (communication) channels. Each channel contains a buffer of messages. The *send* method takes as input a given message, blocks waiting until the channel's buffer is not full, and adds that message to the buffer. On the other hand, the *receive* method blocks waiting until the buffer is not empty, removes a message from the buffer, and outputs that message. For the remote procedure calls, the call system activity is broken down as follows:

1.1) The environment sends a procedure call message to a particular component's call channel (e.g., $pumpCallChn$). That procedure call message encodes the procedure name and any input parameter values (e.g., $\mathbf{call}(setDose, HIGH)$).

1.2) That component receives a procedure call message from its call channel and then decodes that message.

In a dual manner, the return system activity is decomposed as follows:

3.1) The component sends a procedure return message to that procedure's return channel (e.g., $setDoseRetChn$). That procedure return message encodes any output parameter values or exceptions thrown (e.g., $\mathbf{return}(setDose, DoseAlert)$).

3.2) The environment receives a procedure return message from the procedure's return channel and then decodes that message.

2.4 FLAVERS Model Checker

FLAVERS (FLow Analysis for the VERification of Systems) [29] checks whether or not all potential executions of a system model satisfy a set of user-defined properties. The system model and properties are event-based, where an event represents a system activity such as a communication, a procedure call[2], or a variable being defined. FLAVERS models the system as a *trace flow graph (TFG)* that is essentially a collection of control flow graphs (CFGs), one CFG for each thread of control, where extra edges are added between nodes in different CFGs to explicitly represent the possible thread inter-leavings. Each *property* is specified as an FSA. An FSA has its transitions labeled with events as opposed to the TFG that has its nodes labeled with events. FLAVERS checks whether all potential paths through the TFG satisfy the property. If not, then FLAVERS may generate a counterexample path to illustrate the potential property violation.

To make the analysis tractable, the TFG is imprecise but conservative. This means that the TFG may allow paths that do not correspond to actual executions of the system. For instance, the analyst may abstract away the run-time value of a particular variable. This could lead to taking a branch on one value of that variable and then taking a subsequent branch on a different value of the same variable. Thus, if FLAVERS determines that no path can lead to a violation of the property, we know that no actual system execution trace can violate it. But when FLAVERS finds a path that leads to a property violation, we do not know for certain that this path corresponds to an actual execution of the system. The analyst can incrementally

[2]In Little-JIL, all of the steps are essentially treated as procedures.

refine the TFG to make it more precise by specifying additional feasibility *constraint*s that restrict the system behaviors. Each feasibility constraint is also encoded as an FSA that contains a special *violation state* that is entered when infeasible paths are encountered. The violation state is a trap state that once entered cannot be exited. When the user provides constraints, FLAVERS determines whether any potential path through the TFG that is allowed by the constraints can violate the property. The worst case complexity of FLAVERS is $O(N^2 \cdot P \cdot C_1 \cdot \ldots \cdot C_m)$ where N is the number of nodes in the TFG, P is the number of states in the property, m is the number of constraints, and C_i is the number of states in constraint i. In the following, we give more details about the TFGs, the constraints, and the verification algorithms that determine whether any potential path through the TFG that is allowed by the constraints can violate the property.

The TFG supports single-entry/single-exit semantics, meaning that there is a unique initial node of the TFG and a unique final node of the TFG. Each TFG node is labeled with either a system event or τ, a special "empty" event. For instance, the initial and final nodes are labeled with τ. The local TFG edges between nodes in the same CFG capture the intra-thread control flow. Additionally, the *may happen in parallel* algorithm [57] is used to compute the *may immediately precede* (MIP) edges between nodes in different CFGs that capture the inter-thread control flow. To illustrate, the pump example contains three threads of control for the system, the in-patient surgery process, and the pump. Since the process and pump are executing concurrently, the process and pump will have MIP edges between some of the nodes of their respective CFGs. In FLAVERS, the system execution traces are represented as paths through the TFG that start at the unique initial node, follow the local and MIP edges, and end at one of the TFG nodes. The potential terminating system execution traces correspond to the paths through the TFG that end at the unique final node.

Analysts commonly use the the following three kinds of constraints in FLAVERs: *context constraints* to model aspects of the environment, *variable constraints* [29, Section 6.1] to track the run-time values of variables, and *task constraints* [29, Section 6.2] to track the run-time program counters of threads. The context constraints must be manually specified by the analyst. Both the variable and task constraints can be automatically built by FLAVERS. For instance, FLAVERS can automatically build variable constraints that track the run-time values of variables of type boolean, enumerated, or integer range. To illustrate, Figure 2.9 shows the variable constraint for variable *setDose_dose* with enumerated type {LOW, HIGH}[3]. In each variable constraint (e.g., *setDose_dose*), there is a state for every possible value of the type of the variable (e.g., LOW, HIGH) and an UNKNOWN value (shortened to UNK). The state for the UNKNOWN value is set as the start state and all states that correspond to values of the variable are set as accepting states. Additionally, there are transitions labeled with the events that set (e.g., "setDose_dose=LOW") and test (e.g., "setDose_dose==LOW") the values of the variable. Lastly, there is a special VIOLATION state (hidden to simplify the figure) that is entered when a path through the TFG is infeasible. For instance, a path through the TFG that contains a TFG node labeled with event "setDose_dose=HIGH" cannot then be immediately followed by another TFG node labeled with event "setDose_dose==LOW' because that path is infeasible. That violation state is a trap state so will never be exited.

Because the motivating example produces a TFG with around a 100 nodes, we consider only a portion of the motivating example in what follows, specifically the pump's *setDose* procedure shown in Figure 2.4. That procedure is modeled by the portion of the TFG shown in Figure 2.10. For clarity, this TFG shows the intra-thread control flow but not the inter-thread control flow, in particular the TFG does not show

[3]For presentation purposes, we used the string names of the enumerated type literals (e.g., LOW) but in practice we actually used the ordinal values (e.g., 0) of the enumerated type literals.

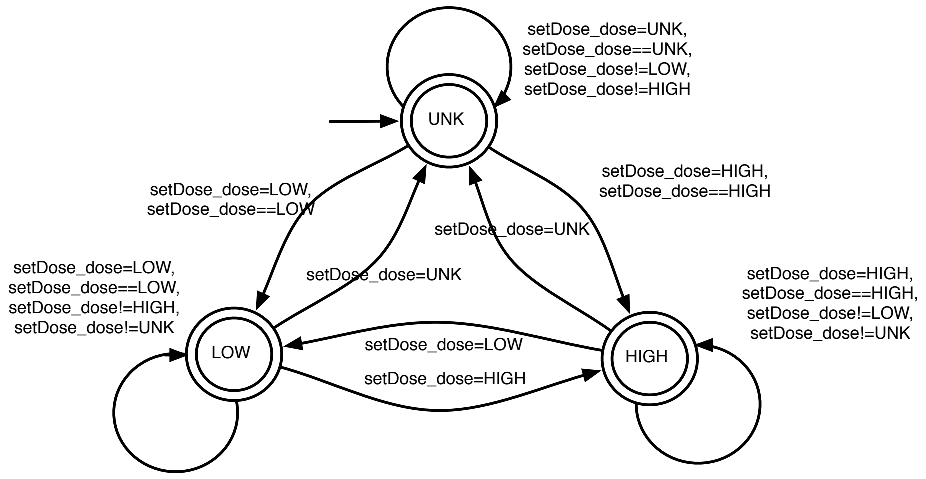


Figure 2.9: Variable constraint for the *setDose_dose* parameter with enumerated type { LOW, HIGH }

the MIP edges from/to other threads in the TFG or any necessary communication over channels, which will be described in Section 4.1.3. In the TFG figure, each TFG node is labeled with a unique ID and either a system event or τ. The TFG node with ID 0 is the unique initial node and the TFG node with ID 11 is the unique final node. The procedure is also modeled by variables constraints for the *currLib_*, *currDose_*, *setDose_dose*, and *setDose_DoseAlert* variables such as the variable constraint shown in Figure 2.9. Each Little-JIL statement (e.g., assignment statement) is translated to an atomic block. For instance, the assignment statment at line 9 is translated to an atomic block that contains TFG nodes 9a, 9b, 9c, and 9d. For the *setDose* procedure, one simple system requirement shown in Figure 2.11 states that the pump must never report a dose alert (represented as event "setDose_DoseAlert=TRUE"). One infeasible path though the TFG is 0, 1b, 5, 8 where the current value of the *dose* variable constraint is set to the value *HIGH* and then the test of the current value of that variable against the constant value *LOW* returns TRUE. One terminating path through the TFG is 0, 1b, 5, 6, 7, 11. That terminating path is allowed by the constraints but violates the simple system requirement property.

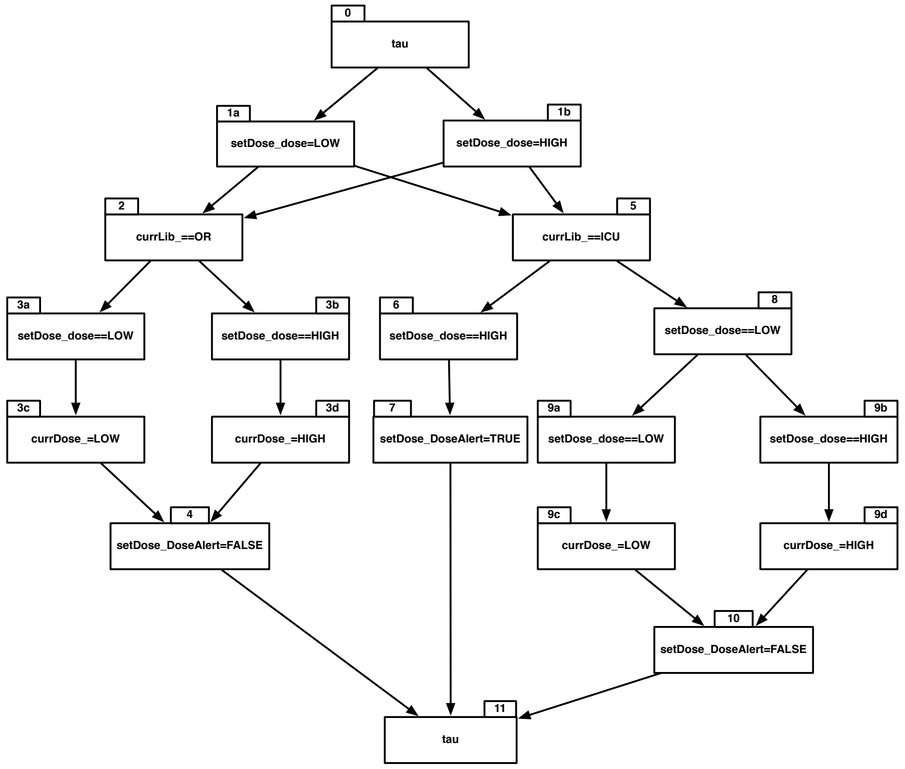


Figure 2.10: Portion of a TFG for the pump's *setDose* procedure

The verification algorithms take as input a FLAVERS *subject* that consists of a TFG, a set of constraints, and a property. These verification algorithms generate a *node-tuple graph*, whose vertices are pairs consisting of a node from the TFG and a tuple of states, one from the property and one from each constraint. (The node-tuples are essentially the nodes of a reachability graph of the system model.) In more detail, FLAVERS provides three different verification algorithms: *build node-tuple graph*, *state propagation*, *find path*. These verification algorithms make different trade-offs between the performance, especially in terms of space, and which paths through the node-tuple graph are explored. Additionally, only one of the verification algorithm can be used to generate counterexample paths. At a high-level, the verification algorithms first generate the initial node-tuple that pairs the unique initial node of the TFG with the initial tuple where each FSA is at its start state. These algorithms then iteratively generate the node-tuple graph starting from the initial node-tuple. After

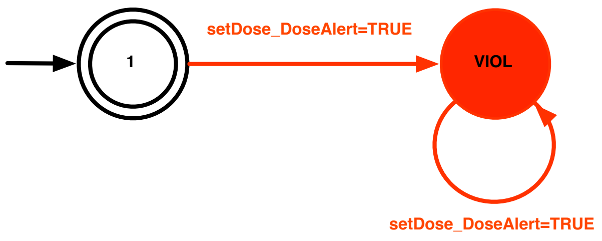


Figure 2.11: System requirement property for never report a dose alert (represented as event "setDose_DoseAlert=TRUE")

all reachable node-tuples are generated, the algorithms examine the final node-tuples, meaning the TFG node is the unique final node, that summarize the potential terminating paths to determine the verification result. For instance, Figure 2.12 shows the node-tuple graph generated from the subject that consists of the TFG for the *setDose* procedure shown in Figure 2.10, the property shown in Figure 2.11, and the variables constraints for currLib_, currDose_, setDose_dose, and setDose_DoseAlert automatically generated using the variable constraint template shown in Figure 2.9. The initial node-tuple with ID 0 pairs TFG node 0 with the initial tuple that has the property at state 0 and each variable constraint at its UNKNOWN state (denoted as UNK) represented as (1,UNK,UNK,UNK,UNK). The node-tuple 23 is infeasible because the tuple has a constraint at its violation state (shown as dotted because the infeasible paths are pruned away). The final node-tuples have IDs 6, 18, 12, and 22. The final node-tuple 22 violates the property (shown in red). In the following paragraphs, we provide a formal definition of the node-tuple graphs, give a high-level description of the verification algorithms, and then give a brief description of each verification algorithm.

For FLAVERS, we model the actual system execution traces as sequences of system events such as procedure calls, exceptions being thrown, or variables being defined or used. A node-tuple graph is an over-approximation of all possible sequence of system events that may be observed on an actual execution of the system. The

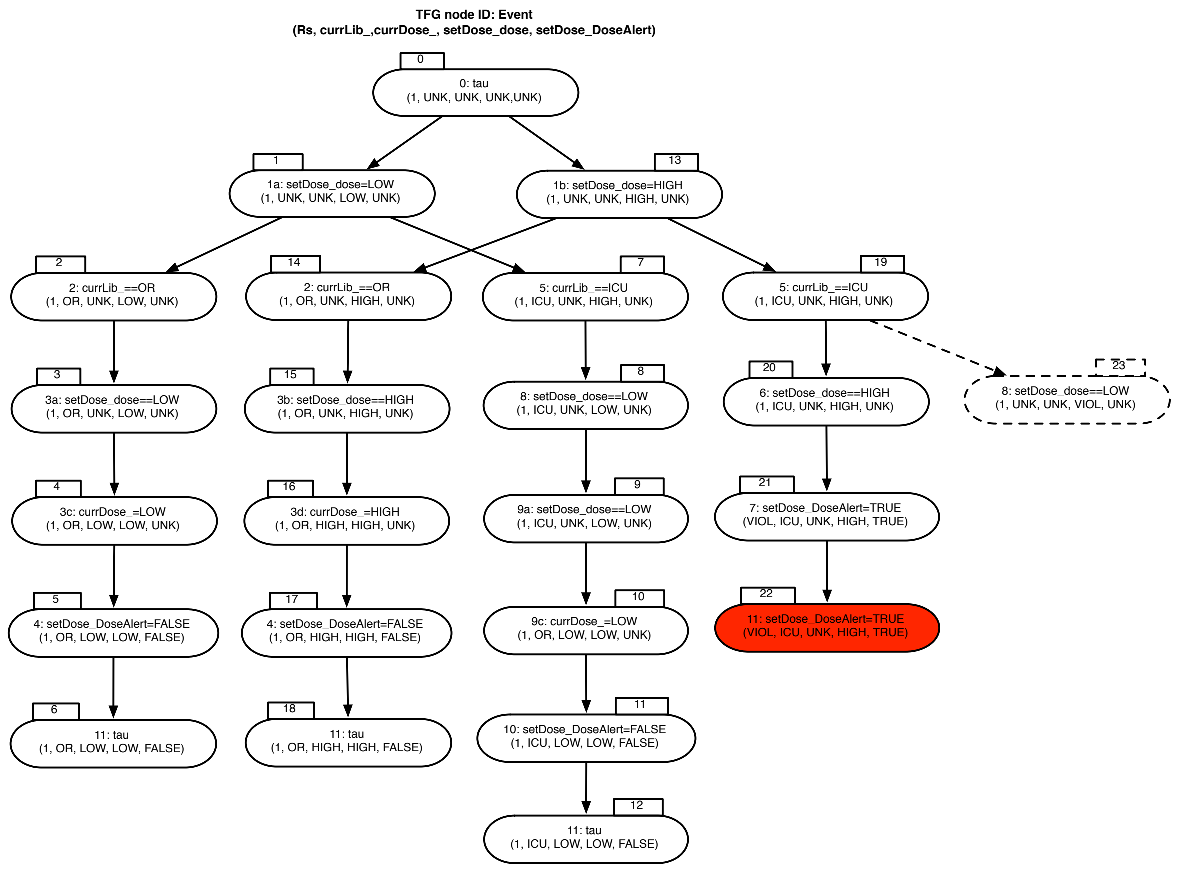


Figure 2.12: Portion of the node-tuple graph for the *setDose* procedure where each node-tuple shows the TFG node (top) and the tuple (bottom)

node-tuple graph is conservative, meaning that each actual system execution trace corresponds to a path through that node-tuple graph, but imprecise, meaning each path may correspond to zero or more actual system execution traces. Formally, a node-tuple graph is a labeled directed graph G = (N, E, n_{init}, F, Σ_G, L) where N is a set of node-tuples, $E \subseteq N \times N$ is a set of directed edges, $n_{init} \in N$ corresponds to the unique initial node-tuple, and $F \subseteq N$ is the set of final node-tuples, Σ_G is an alphabet of system events that label the node-tuples, and L : N $\rightarrow \Sigma_G \cup \{\tau\}$ is a function mapping node-tuples to their events. The *tuple transition function* takes as input a current tuple along with a next TFG node and produces the possible next tuples. This function applies the event labeling the next TFG node to each of the FSA states in the current tuple to produce the next tuples. The function can be configured to only produce the *feasible node-tuples*, meaning the tuple does not have any constraint at its violation state. As shown in Figure 2.12, the tuple transition function given the initial tuple 0 with contents (1,UNK,UNK,UNK,UNK) and next TFG node 1b labeled with event "dose=HIGH" from Figure 2.10 will produce the next tuple 13 with contents (1,UNK,UNK,HIGH,UNK). We define a path through the node-tuple graph to be a sequence of node-tuples where the first node-tuple is the initial node-tuple and each current node-tuple (at index i) can transition to its next node-tuple pair (at index i + 1). We define an *infeasible path* to be a path through the node-tuple graph that ends at an *infeasible node-tuple*, meaning a node-tuple where the tuple has one or more of the constraints at their violation states. For instance, the node-tuple graph shown in Figure 2.12 contains the path 0, 13, 19, 23 that is infeasible and the path 0, 13, 19, 20 that is feasible (i.e. not infeasible). For the feasible paths, we define a *terminating path* to be a feasible path through the node-tuple graph that ends at a *terminating node-tuple* where the TFG node is the unique final node and the tuple has each constraint at one of its accepting states. In the same figure, the right path 0, 13, 19, 20, 21, 22 is terminating but the prefix of that path 0, 13, 19, 20 is non-terminating (i.e. not terminating). For the terminating paths, we define an *unsafe path* to be a terminating path through the node-tuple graph that ends at a *violating node-tuple* where the TFG node is the final node and the tuple has each constraint at one of its accepting states but the property is at one of its non-accepting states (i.e. the property is violated). (The model checker calls this a counterexample path.) In the same figure, the right path 0, 13, 19, 20, 21, 22 is unsafe. On the other hand, we define a *safe path* to be a terminating path though the node-tuple graph that ends at a *satisfying node-tuple* where the TFG node is the final node and the tuple has each constraint at one of its accepting states and the property is at one of its accepting states (i.e. the property is satisfied). In that figure, the leftmost middle path 0, 13, 14, 15, 16, 17, 18 is safe. Additionally, we define the *non-safe paths* to be all of the paths that are not safe, meaning the unsafe paths, the non-terminating paths, and the infeasible paths.

As mentioned above, the three verification algorithms take as input a FLAVERS subject and then iteratively generate the node-tuple graph starting from the initial node-tuple. For each current node-tuple that pairs a current TFG node with a current tuple, these algorithms iterate through all of the edges of the current TFG node to obtain the possible next TFG nodes. For each next TFG node, the algorithms applies the tuple transition function to the current tuple and that next TFG node to generated the possible next node-tuples. After the verification algorithms stop iterating, these algorithms then determine the verification result by interpreting the final node-tuples that summarize the potential terminating paths. The algorithms first check whether there exist terminating paths. If not, then the verification algorithms report "INCONCLUSIVE (No potential terminating paths exist)." In this case, the verification result is INCONCLUSIVE because the system model is overconstrained and therefore the verification algorithms cannot determine whether or not all potential paths through the system model satisfy the user-defined property. If so, these algorithms next check whether there exist any unsafe paths. If so, the verification algorithms report "INCONCLUSIVE (Counterexample path exists)." In this case, these algorithms may be able to generate one or more of the unsafe paths as counterexample paths. If not, these algorithms report "CONCLUSIVE." Since the verification algorithms do not consider the infeasible paths when determining the verification result, any of the next node-tuples that are infeasible can be conservatively pruned away. This pruning does not affect the verification result but can improve the performance of the verification algorithms.

In more detail, the three verification algorithms generate different abstractions of the node-tuple graph. The *build node-tuple graph* algorithm performs reachability analysis to generate the full node-tuple graph where the vertices are represented as node-tuples and the edges are explicitly represented among the node-tuples. This algorithm explores all of the reachable node-tuples and all possible edges among them. The algorithm reports the verification result but does not generate counterexample paths. The algorithm, however, could easily be extended to generate some of the counterexample paths because both the node-tuples and the edges among them are stored. Appendix A provides more details about the *build node-tuple graph* algorithm. Alternatively, the *find path* algorithm [67] employs search techniques, such as breadth or depth first search, to iteratively generate counterexample paths. This algorithm associates each node-tuple with a parent pointer that explicitly represents the edge from the current node-tuple (at index i in the path) to the previous node-tuple (at index i - 1 in the path). The *find path* algorithm explores all of the reachable node-tuples. This algorithm, however, does not add a parent pointer to a current node-tuple that has already been reached and, thus, the algorithm often only explores some of the edges among the node-tuples. The algorithm reports the verification result based on the reachable node-tuples and generates the counterexample paths for any violating node-tuples that were found. On the other hand, the *state propagation* algorithm [29] employs qualified data flow analysis (e.g., [47]) to propagate the tuples among the nodes. This algorithm abstracts the node-tuple graph by storing a set of tuples with each TFG node to represent the reachable node-tuples and by not storing how each tuple reached that TFG node and, thus, not explicitly representing all of the possible edges among the node-tuples. The *state propagation* algorithm instead uses the TFG edges to explore all of the possible edges among the node-tuples. This algorithm does report the verification result based on the tuple set associated with the final TFG node. The algorithm, however, does not generate counterexample paths because the edges among the node-tuples were abstracted away. Both the *find path* and *state propagation* algorithms have an option to immediately terminate after the first counterexample path is found, and thus these two algorithms in practice often need to generate only a partial node-tuple graph.

2.5 Interface Synthesis Methods

The interface synthesis methods generally take as input a selected component model (or implementation) that defines a set of procedures that will be used in the remaining system. The interface synthesis methods also usually take as input an overall system requirement that employs the never E property pattern [28] that states that a given event E never occurs on any system execution trace. In particular, the event E is usually that a specific error code is not returned by or a specific exception is not thrown by any of the component's procedures. The interface synthesis methods produce a synthesized interface represented as an FSA that allows all system execution traces (represented as sequences of procedure call/return pairs) that prevent any system requirement violations (never return a specific error code or throw a specific exception). In our terminology, the interface synthesis methods are commonly applied from the component perspective. In what follows, we will refer to the synthesized interfaces as derived requirements. For instance, Figure 2.5 shows a view of the derived requirement for the process that prevents any of the pump's procedures from throwing dose alert exceptions. In what follows, we provide basic definitions first for the interface synthesis methods and then for our automated requirement derivation approach.

For the interface synthesis methods, we adopt Henzinger et. al's [46] basic definitions. In what follows, we consider a system model S with alphabet Σ_S. We also consider an overall system requirement R_S represented as an FSA with alphabet Σ_{R_S}, which is a subset of the system alphabet Σ_S, and a derived requirement D also represented as an FSA with alphabet Σ_D, which is also a subset of the system alphabet Σ_S. We use $\mathcal{L}(S)$ to denote the language of the finite-state system model S, meaning all of the event sequences from Σ_S^* that are accepted by system model S. Additionally, we use $\neg\mathcal{L}(S)$ to denote the complement of the language of the system model S, meaning all of the event sequences from Σ_S^* that are rejected by system model S. An *illegal event sequence* corresponds to at least one actual system execution trace that violates the overall system requirement, meaning there exists an event sequence from $\mathcal{L}(S) \cap \neg\mathcal{L}(R_S)$. On the other hand, a *legal event sequence* corresponds to any event sequence from Σ_S^* that is not illegal, meaning an event sequence from either $\mathcal{L}(S) \cap \mathcal{L}(R_S)$ or $\neg\mathcal{L}(S)$. In the former case, the legal event sequence corresponds to an actual system execution trace that satisfies the system requirement. In the later case, the legal event sequence, however, corresponds to no actual system execution traces at all.

A *safe* derived requirement D_{safe} is defined to disallow all illegal event sequences. In other words, each event sequence σ from $\mathcal{L}(D_{\text{safe}})$ must be legal. The *most permissive* derived requirement D_{mostPerm} is defined to allow all legal event sequences. More formally, $\mathcal{L}(D_{\text{mostPerm}}) = \{\forall\sigma \in \Sigma_D^* : \sigma \text{ is legal}\}$. Our goal is for a safe derived requirement to also be *adequately permissive*, meaning the requirement allows all event sequences from $\mathcal{L}(S) \cap \mathcal{L}(R_S)$. The language of the adequately permissive derived requirement is contained in (and might be equal to) the language of the most permissive derived requirement that allows all legal event sequences from $\Sigma_S^* \setminus (\mathcal{L}(S) \cap \neg\mathcal{L}(R_S))$, which allows the event sequences from $\mathcal{L}(S) \cap \mathcal{L}(R_S)$ as well as the ones from $\neg\mathcal{L}(S)$.

CHAPTER 3
DERIVATION APPROACH

Our HIS-based requirement derivation approach takes as input a HIS model specification, a system requirement specification, and a requirement deriver perspective. The approach, whenever possible, produces a safe derived requirement represented as a minimal deterministic FSA. Figure 3.1 shows an overview of our HIS-based requirement derivation approach.

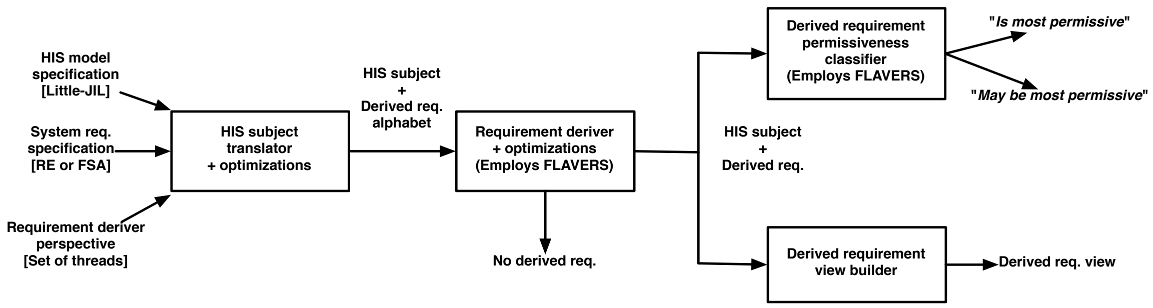

HIS model specification [Little-JIL]
System req. specification [RE or FSA]
Requirement deriver perspective [Set of threads]
HIS subject translator + optimizations
HIS subject + Derived req. alphabet
Requirement deriver + optimizations (Employs FLAVERS)
No derived req.
HIS subject + Derived req.
Derived requirement permissiveness classifier (Employs FLAVERS)
"Is most permissive"
"May be most permissive"
Derived requirement view builder
Derived req. view

Figure 3.1: Overview of our HIS-based requirement derivation approach

In more detail, the HIS subject translator takes as input a system requirement specification represented as an RE (regular expression) or FSA. Additionally, this translator takes as input a HIS model specification written in Little-JIL. In the previous chapter, we described that the overall HIS model concurrently executes the component model and the process model in which that component will be used. Thus, the translator takes as input the requirement deriver perspective specified as a subset of the threads for the overall system, the component, and the process. From the component perspective, the set of threads would contain only the thread for the selected component (e.g., IteratePump). From the process perspective, the set of threads would contain the threads for the overall system and the process (e.g., performPumpHIS, performInPatientSurgery). This translator produces a HIS subject along with the derived requirement alphabet. The translator incorporates several optimizations that will be described in the next chapter.

For this work, we investigated two *requirement derivers* that employ different interface synthesis methods that affect the requirement deriver results in terms of the performance of the requirement deriver as well as the permissiveness of the derived requirements. Each requirement deriver first performs a *pre-requisite check* to determine whether or not a derived requirement should be produced. If so, the requirement deriver then takes as input the HIS subject and the derived requirement alphabet and employs either a direct or learning-based interface synthesis method to automatically produce a derived requirement that is guaranteed to be safe. The *direct requirement deriver* basically first uses FLAVERS to build the full node-tuple graph, refines that graph, and then converts the graph to a minimal deterministic FSA. In the previous chapter, we defined a derived requirement to be safe when that requirement disallows all actual system execution traces that may violate the overall system requirement. We also defined a derived requirement to be adequately permissive when that requirement allows all actual system execution traces that always satisfy the system requirement. The direct derived requirements are guaranteed to be minimal, safe, and adequately permissive. This requirement deriver, however, may exceed all available space when explicitly converting from a non-deterministic node-tuple graph to a deterministic FSA. On the other hand, the *learning-based requirement deriver* uses a regular language learning algorithm to iteratively refine the derived requirement based on counterexamples generated by FLAVERS. This deriver uses an incremental strategy to try to not exceed the available space by not explicitly converting from a non-deterministic node-tuple graph to a deterministic FSA. The deriver, however, only guarantees that the derived requirements are safe and minimal. This deriver incorporates several learning algorithm optimizations that will be described in this chapter and the next chapter. The deriver also incorporates several counterexample generation heuristics that impact both the performance of the requirement deriver as well as the permissiveness of the derived requirements that will also be described in the next chapter.

The *derived requirement permissiveness classifier* can then be used to conservatively determine whether or not a given derived requirement that is safe is also adequately permissive. Since the derived requirements often reflect the complexity of the HIS models, we also investigated building different views of the derived requirements that use various HIS-based abstractions to try to improve the understandability of the requirements as described in Chapter 6.

Section 3.1 first gives some basic definitions for our HIS-based requirement derivation approach. Section 3.2 to Section 3.6 then provide more details about the HIS subject translator, the pre-requisite check, the direct requirement deriver, the learning-based requirement deriver, and the derived requirement permissiveness classifier respectively.

3.1 Basic Definitions

For this work, the actual systems often contain non-determinism (introduced by, e.g., the environment or human decision making) that must be reflected in the system models. Additionally, the system models often use abstractions and refinements that over-approximate the actual system execution traces. Thus, the interface synthesis methods must take into account the non-determinism and over-approximations to ensure that the derived requirements disallow all the system execution traces that may violate the system requirements and allow as many as possible of the system execution traces that always satisfy the system requirements. In more detail, each path through the node-tuple graph S with alphabet Σ_S corresponds to an event sequence from Σ_S^*

that is the concatenation (denoted by '$\cdot$') of the non-τ events labelling the node-tuples. We use $\mathcal{L}(S)$ to denote the language of the node-tuple graph S, meaning the set of all event sequences that correspond to terminating paths through node-tuple graph S. Since our systems often contain non-determinism and the system models are over-approximations, an actual system execution trace represented as an event sequence often corresponds to one or more paths through the node-tuple graph. In particular, that event sequence may correspond to both unsafe paths and safe ones. We first define the correspondence between event sequences and paths. We then define the language of the derived requirements.

For illustration purposes, Figure 3.2 shows a non-deterministic node-tuple graph with alphabet {a, b, c}. (This figure is based on an illustrative example provided by Giannakopoulou [34].) In the figure, each node-tuple is labeled with the TFG node's event (shown on the top). That node-tuple is also labeled with a simplified tuple of FSA states where the first state is from the derived requirement D and the second state is from the system requirement R_S (shown on the bottom). The FSA states have been simplified to be either OK or VIOL. In the previous chapter, we defined a safe path to satisfy both the derived requirement and the system requirement while an unsafe path satisfies the derived requirement but violates the system requirement. The node-tuple shows whether it ends a safe path (denoted by green) or ends an unsafe path (denoted by red). To illustrate, the non-deterministic node-tuple graph shown in Figure 3.2 contains path 1, 3, 4, 5 that corresponds to the event sequence a$\cdot$b. For the non-deterministic node-tuple graph S shown in Figure 3.2, $\mathcal{L}(S) = \{\lambda$, c, a$\cdot$b, a$\cdot$c$\}$.

We say that an event sequence s is *unsafe* if there exists at least one terminating path corresponding to s that is unsafe. In Figure 3.2, the unsafe event sequence a$\cdot$b corresponds to unsafe path 1, 7, 8, 9 and safe path 1, 3, 4, 5. (In the interface synthesis method terminology, the unsafe event sequences are the illegal event sequences

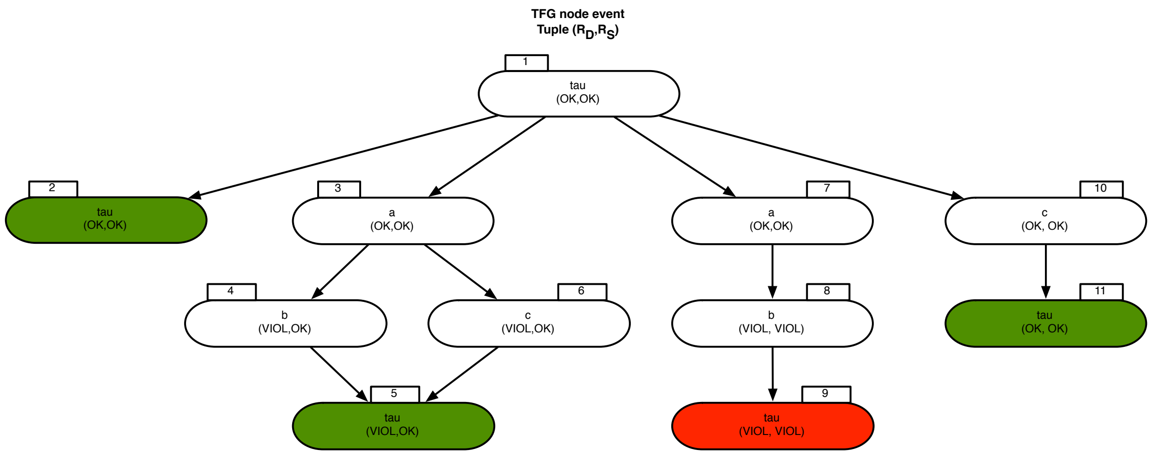


Figure 3.2: Non-deterministic node-tuple graph

from $\mathcal{L}(S) \cap \neg\mathcal{L}(R_S)$.) On the other hand, we say that an event sequence s is *safe* if all terminating paths corresponding to s are safe. In the same figure, the safe event sequence c corresponds to the safe path 1, 10, 11. (The safe event sequences correspond to all legal event sequences from $\mathcal{L}(S) \cap \mathcal{L}(R_S)$.) We say that an event sequence s is *non-terminating* if there exist no terminating paths corresponding to s that are feasible. On the other hand, we say that an event sequence s is *infeasible* if all paths corresponding to s are infeasible or else there exist no paths corresponding to s. (The non-terminating and infeasible event sequences correspond to all legal event sequences from $\neg\mathcal{L}(S)$.) Additionally, we define the *non-unsafe event sequences* to be all of the event sequences that are not unsafe, meaning the safe, non-terminating, and infeasible event sequences. (The non-unsafe event sequences correspond to the legal event sequences.) We also define the *non-safe event sequences* to be all of the event sequences that are not safe, meaning the unsafe, non-terminating, and infeasible event sequences. In more detail, Table 3.1 shows the correspondence between event sequences from Σ_D^* and paths through node-tuple graph S with alphabet Σ_S where Σ_D is a subset of or equal to Σ_S. In this table, NOT means there exist no paths of that type corresponding to the event sequence, $MUST$ means there exists one or more paths of that type corresponding to the event sequence, and MAY means there may exist paths of that type corresponding to the event sequence. If an event sequence corresponds to no paths through the node-tuple graph, then that event sequence is defined to be infeasible.

In the previous paragraph, we described how the event sequences from Σ_S^* correspond to paths. In this paragraph, we will frame our discussion in terms of event sequences from Σ_D^*, noting that Σ_D is a (usually small) subset of Σ_S. Thus, we define the projection of a given event sequence from Σ_S^* on the derived requirement alphabet Σ_D to be the concatenation of those events contained in Σ_D. We defined a safe derived requirement D_{safe} to disallow all unsafe event sequences. For the non-deterministic node-tuple graph shown in Figure 3.2, the derived requirement alphabet Σ_D contains $\{a, b, c\}$. The learning-based requirement deriver could produce a safe derived requirement D_{safe} that allows the following safe event sequences $\{\lambda, c\}$, which disallows the unsafe event sequence $a{\cdot}b$. We defined an *adequately permissive* derived requirement $D_{\text{adequatelyPerm}}$ to allow all safe event sequences. More formally, $\mathcal{L}(D_{adequatelyPerm}) = \{\sigma \in \Sigma_D^* : \sigma \text{ is safe}\}$. Because $\mathcal{L}(D_{mostPerm})$ allows all of the non-terminating and infeasible event sequences but $\mathcal{L}(D_{adequatelyPerm})$ may disallow some or all of those event sequences, $\mathcal{L}(D_{adequatelyPerm}) \subseteq \mathcal{L}(D_{mostPerm})$. For the non-deterministic node-tuple graph shown in Figure 3.2, the $\mathcal{L}(D_{adequatelyPerm})$ must allow the following event sequences $\{\lambda,\ \text{c},\ \text{a}{\cdot}\text{c}\}$. The $\mathcal{L}(D_{mostPerm})$ must allow all event sequences from $(a|b|c)^* \setminus a{\cdot}b$.

3.2 HIS Subject Translator

The HIS subject translator takes as input a system requirement specification represented as a regular expression or an FSA, a HIS model specification written in Little-JIL, and a requirement deriver perspective. This translator produces a HIS subject along with the derived requirement alphabet. In what follows, we first describe how the translator produces the HIS subject that consists of a system TFG, a

	Unsafe paths	Safe paths	Non-terminating paths	Infeasible paths
1. Unsafe event seqs.	MUST	MAY	MAY	MAY
2. Safe event seqs.	NOT	MUST	MAY	MAY
3. Non-terminating event seqs.	NOT	NOT	MUST	MAY
4. Infeasible event seqs.	NOT	NOT	NOT	MAY

Table 3.1: Correspondence between event sequences and paths

set of system constraints, and the system requirement property R_S. We then describe how the HIS subject translator produces the derived requirement alphabet Σ_D based on that HIS subject and the requirement deriver perspective.

The system requirement specification represented as a regular expression (RE) or FSA is automatically translated to the system requirement property represented as an FSA. If the system requirement specification is represented as an RE then that specification is parsed in as an RE and then a standard algorithm to convert from that RE to an FSA (e.g., [1]) is applied. If the system requirement specification is represented as an FSA, then that that specification is parsed in as an FSA. The HIS model specification written in Little-JIL is automatically translated to a HIS model represented as a TFG and a set of constraints. In the previous chapter, we explained that a HIS model specification written in Little-JIL defines the overall system as a hierarchy of steps that can be treated as procedures. Each step defines its interface with its name, input parameters, output parameters, and exceptions thrown. Additionally, that step may define local parameters. Some of the steps represent the remote procedure calls between the process and the selected component. These steps use channels to pass the input parameters from the process to that component and then to pass back either the output parameters or exceptions thrown from the component to the process. To accurately represent the actual system execution traces, the HIS models represented by TFGs with sets of constraints often must encode the channels, parameters, and exceptions relevant to the overall system requirement as follows.

In more detail, such a HIS model declares a step representing the execution of the overall system. That step concurrently executes a step that represents the process model and another step that represents the component model. Thus, the TFG will contain a thread for the overall system, the component, and the process. For each TFG thread, the TFG will contain a TFG node to fork that thread to begin executing and another TFG node to join the thread after it finishes executing. Additionally, the set of constraints will contain a task constraint for that thread to track its program counter and a thread status constraint [58] for that thread to track whether or not the thread is currently executing. For each named channel that stores messages of a given type, the system translator creates a variable constraint with that name and message type that tracks the run-time value of the message stored in the channel. Additionally, it creates TFG nodes to send messages to that channel and receive messages from the channel.The channel translation is described in more detail in Section 4.1.3. For each parameter (e.g., *currLib_*) or exception (e.g., DoseExceedsLimits), that parameter or exception is treated as a local variable in the TFG that may be tested and set. For each named parameter of a given primitive type, including boolean, enumerated, and integer range types, the system translator creates a variable constraint with that name and primitive type. Additionally, it creates TFG nodes for any tests of that parameter by the steps and any sets of the parameter because of initialization by the steps or parameter passing among the steps. For each exception type, the system translator creates a variable constraint with that exception type name and boolean type. Additionally, it creates TFG nodes that set the variable constraint to true when the exception type instance is thrown by a given step. The system translator also creates TFG nodes that set the variable constraint to false when the exception type instance is caught by a handler step. We know from previous work that alternative ways of encoding the FLAVERS variables (e.g., channels, parameters, exceptions) can have substantial impact on the cost of the verification. Thus, we explored different variable encodings to reduce the space needed to perform the requirement derivation as described in Section 4.1.

The derived requirement's alphabet conceptually contains the events that are relevant to the interface between the selected component and the process. This alphabet contains the events that correspond to all of the possible component's procedure executions (e.g., $setLib(ICU)\_OK$, $setDose(HIGH)\_DoseAlert$). The alphabet may also contain the events that correspond to the process' calling contexts (e.g., $enterICU$, $leaveICU$) for the component's procedure executions. Appendix B describes how to compute the derived requirement's alphabet where each procedure execution is represented as a remote procedure call.

From both perspectives, we consider an overall system requirement R_S with alphabet Σ_{R_S}. For the illustrative example, we consider the never overdose system requirement shown in Figure 2.1. We also consider an overall system S with alphabet Σ_S decomposed into the selected component model C with alphabet Σ_C and process model P with alphabet Σ_P in which that component will be used. From the component perspective, the derived requirement alphabet Σ_D is basically computed as $((\Sigma_C \cup \Sigma_{R_S}) \cap \Sigma_P)$. If the alphabet of the system requirement Σ_{R_S} contains calling context events, then the alphabet of the derived requirement for the process Σ_D will contain the calling context events because the alphabet of the process model Σ_P contains them. For the illustrative example, the derived requirement alphabet Σ_D is $\{setLib(ICU)\_OK$, $setLib(OR)\_OK$, $setDose(LOW)\_OK$, $setDose(LOW)\_DoseAlert$, $setDose(HIGH)$-$\_OK$, $setDose(HIGH)\_DoseAlert$, $start()\_OK$, $start()\_DoseAlert$, $enterICU$, $leaveICU\}$.

From the process perspective, the derived requirement alphabet Σ_D is essentially computed as $((\Sigma_P \cup \Sigma_{R_S}) \cap \Sigma_C)$. If the alphabet of the system requirement Σ_{R_S} contains calling context events, the alphabet of the derived requirement for the component Σ_D does not contain the calling context events because the alphabet of the component model Σ_C does not contain them. For the illustrative example, the derived requirement alphabet Σ_D is $\{setLib(ICU)\_OK$, $setLib(OR)\_OK$, $setDose(LOW)\_OK$, $setDose(LOW)\_DoseAlert$, $setDose(HIGH)\_OK$, $setDose(HIGH)\_DoseAlert$, $start()\_OK$, $start()\_DoseAlert\}$. As noted above, this alphabet is missing the calling context events $enterICU$ and $leaveICU$.

3.3 Pre-Requisite Check

The *Pre-requisite check* takes as input a HIS subject and a derived requirement alphabet and determines whether ALL, SOME, or NONE of the paths through the node-tuple graph are safe, which determines whether or not a derived requirement should be produced. This check first tests whether or not the derived requirement alphabet is empty. If so, then the check returns NONE. In this case, a derived requirement should not be produced because no derived requirement could ensure that the system requirement is satisfied. In other words, the adequately permissive derived requirement would allow no paths because they are all unsafe. If not, then the *Pre-requisite check* looks for the existence of safe paths and unsafe ones through the node-tuple graph. If there are only safe paths, then this check returns ALL. In this case, a derived requirement should not be produced because it is not needed. The adequately permissive derived requirement would allow all paths because they are all safe. If there are only unsafe paths, then the check reports NONE. If there are both safe and unsafe paths, then the pre-requisite check returns SOME. In this case, a derived requirement should be produced that is guaranteed to be safe and may or may not be adequately permissive.

Because the FLAVERS *state propagation* algorithm conservatively summarizes all potential paths through the node-tuple graph for the given HIS subject, we use this algorithm to check for the existence of safe and unsafe paths. The Pre-requisite check runs the *state propagation* algorithm on a HIS subject consisting of the system TFG, the system constraints, and the system requirement property. This check then looks for the existence of safe and unsafe paths by interpreting the final node-tuples and then returns either ALL, SOME, or NONE. If there exist only safe paths, meaning all of the final node-tuples are satisfying (and perhaps non-terminating), the Pre-requisite check returns ALL. If there exist only unsafe paths, meaning all of the final node-tuples are violating (and perhaps non-terminating), or there exist no terminating paths at all, meaning the final node-tuples are all non-terminating, then this check returns NONE. If there exist both safe and unsafe paths, meaning some of the final node-tuples are satisfying, other final node-tuples are violating, and perhaps the remainder are non-terminating, the check returns SOME.

3.4 Direct Requirement Deriver

Given a HIS subject and the derived requirement alphabet, the direct requirement deriver extends Giannakopoulou et al.'s interface synthesis method [36] that conceptually first generates the full reachability graph and then refines that graph based on the derived requirement's alphabet to produce the derived requirement represented as a minimal deterministic FSA. Figure 3.3 shows a high-level overview of the direct requirement deriver that consists of four main stages: *Pre-requisite check*, *Composition*, *Refinement*, *FSA extraction*. We describe each stage in the following paragraphs.

Section 3.3 describes the *Pre-requisite check* that uses the *state propagation* algorithm to determine whether or not a derived requirement will be produced. If not, then that determination is reported. If so, then the direct requirement deriver per-

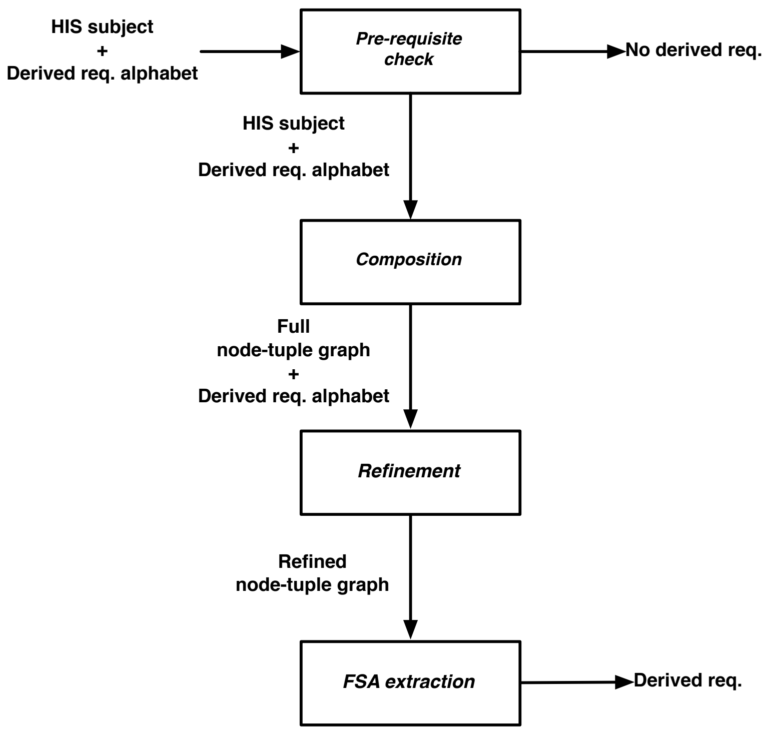


Figure 3.3: Overview of the direct requirement deriver

forms the remaining three stages to produce a derived requirement that is guaranteed to be safe and adequately permissive.

The *Composition* stage first uses the *build node-tuple graph* algorithm, which is one of the FLAVERS verification algorithms described in Chapter 2, to generate the full node-tuple graph for the given HIS subject. For the motivating example from the component perspective, the full node-tuple graph has 29 unique events, 888 node-tuples, and 3555 edges. The *Refinement* stage takes as input the full node-tuple graph and the derived requirement's alphabet and then conceptually abstracts away the nodes labeled with events not in that alphabet. At a lower-level, this refinement performs two phases: minimization and backward-error propagation. The minimization phase first relabels any nodes labeled with events not in the derived requirement's alphabet with the special τ event. The backward-error propagation phase then will remove each node labeled with τ if after that removal the node-tuple graph remains well-formed. For instance, a node that was labeled with a local variable assignment and was relabeled as τ can be removed. On the other hand, the initial node is labeled with τ but cannot be removed. The *Refinement* stage adapts the *alphabet refinement* algorithm [29] developed for FLAVERS for Ada that applies to CFGs. Given an original node-tuple graph S with alphabet Σ_S that accepts language $\mathcal{L}(S)$ along with the derived requirement alphabet Σ_D that is a subset of Σ_S, this algorithm creates the refined node-tuple graph S' with alphabet Σ_D that accepts language $\mathcal{L}(S')$. The algorithm ensures that for each event sequence σ in $\mathcal{L}(S)$ the language $\mathcal{L}(S')$ accepts the event sequence σ' that is the concatenation of the events from Σ_D. For example, the refined node-tuple graph has 13 unique events, 499 node-tuples, and 2483 edges. The *FSA extraction* stage is performed next to convert from a node-tuple graph to a minimal deterministic FSA.

The *FSA extraction* stage first converts from the refined node-tuple graph where the nodes are labeled with the events to a non-deterministic FSA (NFA) where the transitions are labeled with the events. The FSA extraction stage then uses regular language algorithms (e.g., [1]) to determinize the NFA and then minimize the resulting deterministic FSA. The next section describes the FSA extraction stage in more detail, and Section 3.4.2 discusses the guarantees provided by the direct requirement deriver.

3.4.1 FSA Extraction

At a lower level, the FSA extraction stage first converts the node-tuple graph to an NFA. This conversion will convert each node-tuple, which represents the end of a set of paths through the node-tuple graph, to a state in the NFA. To guarantee that the derived requirement is safe, the conversion will ensure that every node-tuple that represents the end of any unsafe paths will be converted to a unique error state in the NFA that is a non-accepting trap. This stage next applies the algorithm to determinize the NFA and then applies the algorithm to minimize the resulting deterministic FSA. The algorithm to determinize the NFA, however, must be specialized to direct all of the unsafe paths through the node-tuple graph to the error state in the deterministic FSA to guarantee that the derived requirement is safe. In more detail, the algorithm to determinize the NFA conceptually converts from each possible subset of the NFA states to a corresponding state in the deterministic FSA. For the requirement derivation, we need to extend this algorithm to add special handling to ensure that each subset of NFA states that contains the error state in the NFA is mapped to the error state in the deterministic FSA. The algorithm to minimize the deterministic FSA, however, does not need to have special handling for the error state. The FSA extraction stage, lastly, performs a refinement to the deterministic FSA that also redirects all infeasible paths through the node-tuple graph to the special error state in that FSA. In the following paragraphs, we describe the *node-tuple graph to NFA conversion* and the *infeasible paths refinement*.

Given the refined node-tuple graph and the derived requirement's alphabet, the node-tuple graph to NFA conversion first creates a new NFA and sets the NFA's alphabet to a copy of the derived requirement's alphabet. This conversion then creates the special *error state*. This error state is set as a non-accepting state of the NFA and made a trap state by creating a transition from the error state to itself on each event in the NFA's alphabet. Given a node-tuple, the conversion maps that node-tuple to an NFA state as follows. This conversion checks whether the current node-tuple is satisfying (i.e. corresponds to a safe path), violating (i.e. corresponds to an unsafe path), or non-terminating (i.e. corresponds to a non-terminating path). The current node-tuple will not be infeasible (i.e. corresponds to an infeasible path) because the infeasible node-tuples were pruned away. If the node-tuple is satisfying, then that node-tuple is mapped to a new NFA state that is set as an accepting state of the NFA. If the node-tuple is violating, then that node-tuple is mapped to the error state. If the current node-tuple is non-terminating, then that node-tuple is mapped to a new NFA state that is set as non-accepting. If the current node-tuple is the initial node-tuple, then the current state is set as the start state of the NFA. The node-tuple graph to NFA conversion then reiterates over the node-tuples to create the appropriate NFA transitions. Given a current node-tuple c with an edge in the node-tuple graph to a next node-tuple n, it creates a transition from the NFA state associated with c to the NFA state associated with n. This conversion then checks whether or not the current node-tuple is labeled with the τ event. If so, then that transition is labeled with λ, a special, "empty" event. If not, then the transition is labeled with the non-tau event that labels the current node-tuple. For the pump example, the NFA has 12 unique events, 499 non-error states, and 2483 non-error transitions. The node-tuple graph to NFA conversion, however, was not able to redirect the infeasible paths to the error state because the *build node-tuple graph* algorithm pruned away the infeasible paths. On the other hand, each FSA has a transition function that is *total*, meaning for every state s in that FSA, for every event e in the alphabet of the FSA, there exists a transition from state s labeled with event e. Thus, these transition functions represent both the feasible paths and the infeasible ones. We therefore perform an *infeasible paths refinement* on the resulting minimal deterministic FSA to also redirect the infeasible paths to the error state in the FSA to simplify the derived requirement.

Since the state propagation algorithm uses the feasible paths through the TFG to compute the set of reachable states in the property and constraint automata, this means that the infeasible paths will correspond to the unreachable states in those automata. We also extended the state propagation algorithm to compute the set of reachable transitions in the derived requirement FSA. The infeasible paths refinement will redirect the infeasible paths to the error state in the derived requirement FSA by removing the unreachable states and transitions. This refinement runs the extended state propagation algorithm on a HIS subject that consists of the system TFG, the system constraints along with a new constraint that is a copy of the derived requirement, and the system requirement property to compute the set of reachable states and transitions. Based on the results of the extended state propagation algorithm, the refinement first redirects every unreachable transition to the error state in the FSA and then removes the unreachable states. Since the infeasible paths refinement affects the transition function of the deterministic FSA, this refinement must reapply the algorithm to minimize that FSA. For the example, the direct requirement deriver produces a derived requirement represented as a minimal deterministic FSA that contains 12 unique events, 26 non-error states, and 64 non-error transitions. One possible view of that derived requirement is shown in Figure 2.3 represented as a minimal deterministic FSA that contains 10 unique events, 8 non-error states, and 46 non-error transitions.

3.4.2 Guarantees

For a system model that is non-deterministic, a given event sequence may correspond to both non-safe and safe paths. We defined an non-safe event sequence to be an event sequence that corresponds to at least one terminating path that is unsafe or else no terminating paths at all. On the other hand, we defined a safe event sequence to be an event sequence that corresponds to safe paths but no unsafe ones. For the direct requirement deriver, the *Composition*, *Refinement*, and *FSA extraction* stages conservatively but imprecisely model the non-safe and safe event sequences. Thus, the derived requirement is safe, meaning the FSA disallows all of the unsafe event sequences, and adequately permissive, meaning the FSA allows all of the safe event sequences. Additionally, the *FSA extraction* stage ensures that the derived requirement is a minimal deterministic FSA. We first discuss the guarantees about the derived requirements and then the worst-case complexity of the requirement deriver.

In more detail, the *Composition* stage uses the *build node-tuple graph* algorithm to generate the full node-tuple graph that over-approximates all potential paths through the system model. The *Refinement* stage then uses the alphabet refinement algorithm that ensures that the node-tuple graph remains an over-approximation of the potential paths through the system model. For the *FSA extraction* stage, the node-tuple graph to NFA conversion ensures that the unsafe event sequences end at the unique error state that is a non-accepting trap state, the safe event sequences end at one of the accepting states, and the non-terminating event sequences end at one of the non-accepting states. The specialized NFA to DFA conversion needs to appropriately handle any non-determinism that results in an event sequence corresponding to both an unsafe and safe path. In particular, this conversion maps from each set of reachable NFA states to a DFA state. The conversion first checked whether or not the event sequence is unsafe (represented by reaching the NFA's error state). If the set of NFA states does contain the NFA's error state, then the corresponding DFA state will be considered non-safe, in particular unsafe, by mapping to the DFA's error state. If the event sequence is not unsafe, then the conversion checked whether or not the event sequence is safe (represented by reaching any of the NFA's accepting states). If the set of NFA states does not contain the NFA's error state but does contain at least one of the NFA's accepting states, then the corresponding DFA state will be considered safe by making it an accepting state. If the event sequence is not unsafe or safe (represented by not reaching the error state or any of the accepting states), then the corresponding DFA state will be considered non-safe, specifically non-terminating, by making it a non-accepting state. Finally, the *infeasible paths refinement* will ensure that the infeasible event sequences will also be considered non-safe by making these sequences end at the unique error state. In summary, our direct requirement deriver produces derived requirements that disallow all of the non-safe event sequences from either $\mathcal{L}(S) \cap \neg\mathcal{L}(R_S)$ or $\neg\mathcal{L}(S)$ and allow all of the safe event sequences from

$\mathcal{L}(S) \cap \mathcal{L}(R_S)$. This guarantees that the derived requirements are safe and adequately permissive. For the original direct requirement deriver, Giannakopoulou et al. [36] provide the full details about the guarantees concerning the derived requirements. In their case, the derived requirements disallow all of the unsafe event sequences from $\mathcal{L}(S) \cap \neg\mathcal{L}(R_S)$ and allow all of the non-unsafe event sequences from either $\mathcal{L}(S) \cap \mathcal{L}(R_S)$ or $\neg\mathcal{L}(S)$. This guarantees that the derived requirements are safe and most permissive.

The direct requirement deriver has worst-case complexity that is $\mathcal{O}(k \cdot 2^n)$, where k is the number of events in the alphabet Σ_D and n is the number of node-tuples in the node-tuple graph. For this requirement deriver, the FSA extraction stage is the most expensive due to the conversion from a non-deterministic node-tuple graph to a deterministic FSA. In practice, the requirement deriver often does not encounter the exponential blowup during that conversion.

3.5 Learning-Based Requirement Deriver

Given a HIS subject and the derived requirement alphabet Σ_D, the learning-based requirement deriver extends Beyer et. al's interface synthesis method [12] that employs the L^* regular language learning algorithm [6,61] (shortened to L^* *learner* for brevity) and compositional reachability analysis. Our extension is similar to Giannakopoulou and Păsăreanu's extension [35] that employs the L^* learner to iteratively refine the derived requirement represented as an FSA based on counterexample paths generated by a model checker, but differs in the technical details. The derived requirements are guaranteed to be safe but not necessarily adequately permissive as discussed in Section 3.5.3. For the L^* learner, the goal is to learn an unknown regular language U over an alphabet Σ_U and return a minimal deterministic FSA D such that the language $\mathcal{L}(D)$ is equivalent to U.

In more detail, the L^* learner needs to interact with a "minimally adequate teacher" (for brevity shortened here to *teacher*) that is essentially an oracle that is capable of precisely answering two types of queries about U, a *membership query* and an *equivalence query*. For a membership query, the teacher is given an event sequence σ from Σ_U^* and returns true when σ does belong to U and returns false when σ does not belong to U. For an equivalence query, the teacher is given a current FSA D_i and returns true when $\mathcal{L}(D_i)$ is equivalent to U and returns false when $\mathcal{L}(D_i)$ is not equivalent to U along with a counterexample event sequence from the symmetric difference of $\mathcal{L}(D_i)$ and U. If the teacher can precisely answer both the membership and equivalence queries, then the FSA D learned is guaranteed to accept the language U. If not, then the FSA D learned will be an approximation of the language U. In our setting, the direct requirement deriver produces a derived requirement represented as an FSA that accepts the language U that contains all safe event sequences from $\mathcal{L}(S) \cap \mathcal{L}(R_S)$. For this learning-based requirement deriver, our goal is to learn the same language U. In Section 3.5.2, we describe our teacher that employs FLAVERS to answer the membership and equivalence queries. At the end of this section, we will discuss how our teacher provides precise answers for the membership queries but imprecise answers for the equivalence queries. Thus, the final derived requirement D learned may only be an approximation of our language U that accepts all safe event sequences.

Figure 3.4 shows a high-level overview of the learning-based requirement deriver that first performs the *Pre-requisite check* stage. This requirement deriver then uses the L* learner to perform the remaining three stages: *Initialization*, *Conjecture*, and *Counterexample-based refinement*. We first briefly describe each stage in the following paragraphs. In the next sections, we describe the L* learner, our teacher that uses FLAVERS to answer the membership and equivalence queries, and the guarantees provided by the learning-based requirement deriver.

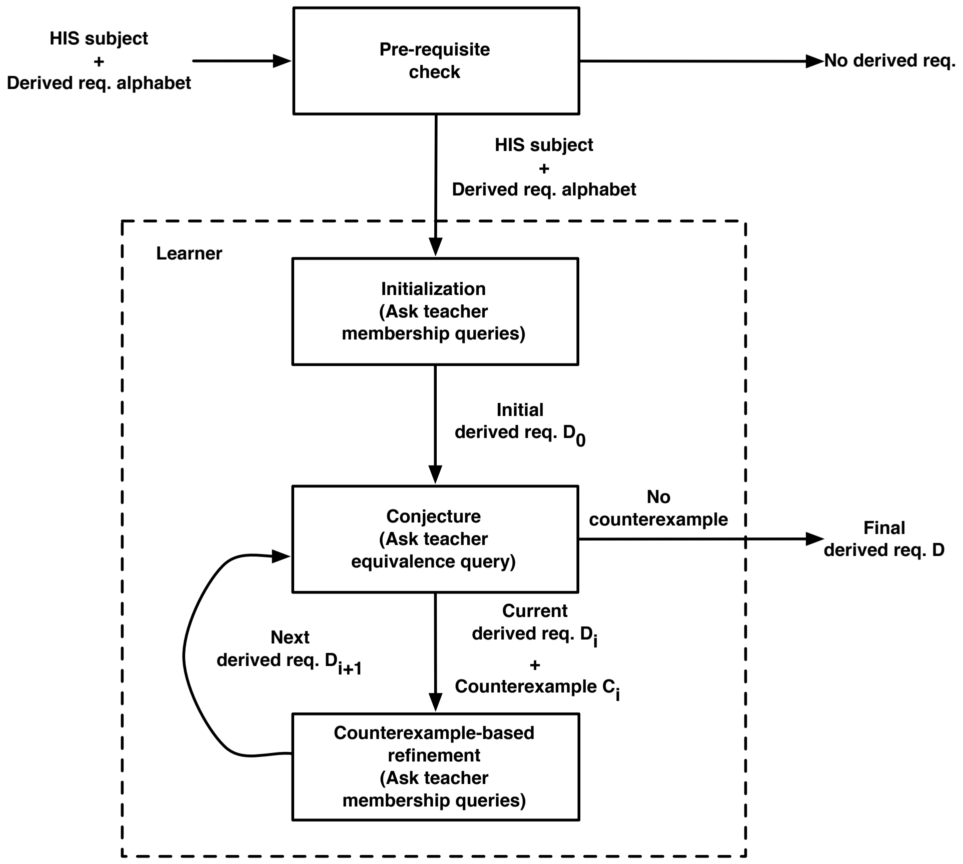

HIS subject + Derived req. alphabet
Pre-requisite check
No derived req.
HIS subject + Derived req. alphabet
Learner
Initialization (Ask teacher membership queries)
Initial derived req. D0
Conjecture (Ask teacher equivalence query)
No counterexample
Final derived req. D
Current derived req. Di + Counterexample Ci
Next derived req. Di+1
Counterexample-based refinement (Ask teacher membership queries)

Figure 3.4: Overview of the learning-based requirement deriver

In Section 3.3, we described the *Pre-requisite check* that determines whether or not a derived requirement is needed. If not, then that determination is reported. If so, then the learner is used to produce the derived requirement that is needed.

The *Initialization* stage creates a very basic initial derived requirement. This stage basically considers an initial set of event sequences that contains the empty event sequence λ and every event sequence of length one from the derived requirement's alphabet. The stage updates the FSA by asking membership queries about the initial set of event sequences. For each event sequence σ in that set, the stage asks a membership query about σ. If the query answer is true, then the initial derived requirement allows the behaviors corresponding to σ. If the query answer is false, the initial derived requirement disallows the behaviors corresponding to σ. The learner next iteratively refines the current derived requirement based on counterexamples generated by the model checker to produce the final derived requirement.

The learner on iteration i first conjectures that the current FSA D_i is equivalent to U by asking the teacher an equivalence query about that FSA. If the current FSA D_i is not equivalent to U, then our teacher tries to find a counterexample in the symmetric difference of $\mathcal{L}(D_i)$ and U, meaning a counterexample in either $U \setminus \mathcal{L}(D_i)$ or else in $\mathcal{L}(D_i) \setminus U$. In our setting, we want to learn the language U that accepts all safe event sequences. If the counterexample is in $\mathcal{L}(D_i) \setminus U$, it is a *safety counterexample* that is allowed by the current derived requirement but violates the system requirement. On the other hand, if the counterexample is in $U \setminus \mathcal{L}(D_i)$, it is a *permissiveness counterexample* that is disallowed by the current derived requirement but satisfies the system requirement. In a similar manner to the *Initialization* stage, the *Counterexample-based refinement* stage updates the FSA by asking the teacher a set of membership queries based on the generated safety or permissiveness counterexample C_i. The next FSA D_{i+1} needs to disallow a safety counterexample or to allow a permissiveness counterexample. After this refinement, the learner then begins another iteration with the current FSA set to the next FSA D_{i+1}. If, on the other other hand, the current FSA D_i is equivalent to U, meaning that the equivalence query did not find a counterexample, then the learner returns FSA D_i as the final derived requirement.

In our discussion about the learning-based requirement deriver, we mentioned that the L^* learner interacts with a teacher that can answer both membership and equivalence queries to learn an unknown language U. If that teacher is minimally adequate, meaning the teacher can precisely answer both the membership and equivalence queries, then the minimal deterministic FSA learned is guaranteed to accept exactly the language U. If the teacher is not minimally adequate, then the FSA learned will accept an approximation of the language U. In our setting, the direct requirement deriver defines the language U to disallow all of the non-safe event sequences and to allow all of the safe event sequences. This means that the derived requirements are guaranteed to be both safe and adequately permissive as discussed in Section 3.4.2. For the learning-based requirement deriver, we define the language U in the same manner as for the direct requirement deriver. Our teacher, however, is not minimally adequate because this teacher performs a less precise static analysis than the direct requirement deriver. In particular, the membership queries provide precise answers but the equivalence queries do not. Because the equivalence queries may not generate all of the permissiveness counterexamples, the derived requirements may not allow all of the safe event sequences. Since the equivalence queries do not generate any counterexamples from $\neg\mathcal{L}(S)$, the requirements may also allow some of the non-terminating and infeasible event sequences. This means that the derived requirements are guaranteed to be safe but not necessarily adequately permissive as further discussed in Section 3.5.3. In particular, the language of the derived requirement $\mathcal{L}(D)$ is guaranteed to be a subset of the language of the safe and most permissive derived requirement but not necessarily the U defined above. Other interface synthesis methods that employ the L^* learning algorithm and a model checker (e.g., [4,35]) are also learning an approximation of the safe and most permissive derived requirement.

3.5.1 L^* Learner

The L^* learner incrementally builds an observation table that records whether or not event sequences from Σ_U^* are in U to represent the FSA. Formally, this table (S, E, T) consists of a set of prefixes S from Σ_U^*, a set of suffixes E from Σ_U^*, and a function T that maps the event sequences seen so far to the answers to their membership queries. The function T maps from $((S \cup (S \cdot \Sigma_U)) \cdot E)$ to either false or true. In the above, we use $P \cdot Q$ to denote the concatenation of two sets of event sequences P and Q, meaning the set of all event sequences $p \cdot q$ for all $p \in P$ and for all $q \in Q$. The learner converts the current observation table to the current FSA D_i with alphabet Σ_U as follows. For each prefix $s \in S$, a new state v_s is added to the FSA. If the prefix s is λ, then state v_s is set as the start state of the FSA. If $T(s)$ is true, then state v_s is added to the accepting states of the FSA. For all $a \in \Sigma_U$, $e \in E$, $s' \in S$, if $T(s \cdot a \cdot e)$ is $T(s' \cdot e)$, then a new transition from state v_s on event a to state $v_{s'}$ is added to the FSA. For this work, we did not implement the original L^* learner proposed by Angluin [6]. To try to reduce the number of membership queries asked, we implemented the L^* learner that incorporates the improvements by Rivest and Schapire [61]. We first describe how to initialize the observation table and then how to refine that table based on a given counterexample. These improvements usually significantly reduce the number of membership queries that need to be asked, which improves the learner's performance in terms of space and time. Lastly, we describe a learner optimization that takes into account characteristics of the unknown language U to further reduce the number of membership queries that need to be asked.

```
1: for all $t \in ((S' \cup (S' \cdot \Sigma_U)) \cdot E')$ do
2:     $T(t) \leftarrow$ Membership-Query(t)
3: end for
```

Figure 3.5: Pseudo-code for the learner's updateT method

For the Initialization stage, the learner initializes both of the S and E sets to $\{\lambda\}$. The learner then updates T by asking the teacher membership queries. Figure 3.5 shows the pseudo-code for the *updateT* method that takes as input a subset S' of S and a subset E' of E and updates the corresponding elements of T to be the answers to the membership queries about those elements. In this case, the learner calls *updateT* with $\{\lambda\}$ and $\{\lambda\}$. For the FSA to be well-formed, the table also needs to be closed, meaning for all $s \in S$, for all $a \in \Sigma_U$, for all $e \in E$, there exists a $s' \in S$ such that $T(s{\cdot}a{\cdot}e)$ is $T(s'{\cdot}e)$. Thus, the learner computes the closure of the table by iterating through all event sequences $s{\cdot}a$ such that there does not exist an $s' \in S$

where $T(s{\cdot}a{\cdot}e) = T(s'{\cdot}e)$. For each such event sequence $s{\cdot}a$, the learner updates S by adding $s{\cdot}a$ and then calls *updateT* with $\{s{\cdot}a\}$ and E.

For the Refinement stage, the learner first updates E by adding a new suffix e' of counterexample cex that "witnesses a difference" between $L(D_i)$ and U. In more detail, the learner splits the counterexample cex_i into the shortest prefix p' and the longest possible suffix e' such that $p'{\cdot}e' \in L(D_i)$ but $p'{\cdot}e' \notin U$ or else $p'{\cdot}e' \notin L(D_i)$ but $p'{\cdot}e' \in U$. The learner then calls *updateT* with S and $\{e'\}$. Lastly, the learner computes the closure of the table as describe in the previous paragraph. The next FSA D_{i+1} is guaranteed to reflect the witnessed difference by containing at least one new state and its transitions. In practice, this means that the Refinement stage often needs to be repeated multiple times for the same counterexample until the next FSA is guaranteed to either disallow the given safety counterexample or allow the given permissiveness one.

```
1: for all $t \in ((S' \cup (S' \cdot \Sigma_U)) \cdot E')$ do
2:     if (There exists a prefix $p$ of $t$ such that $T(p)$ = false) then
3:         $T(t) \leftarrow false$
4:     else
5:         $T(t) \leftarrow$ MEMBERSHIP-QUERY(t)
6:     end if
7: end for
```

Figure 3.6: Pseudo-code for the updateT method that incorporates the prefix-closed optimization

Other researchers (e.g., [10,51]) have observed that the learner's *updateT* method can be optimized when learning an unknown language U that is prefix-closed, meaning that for any event sequence that does not belong to U any extension of that event sequence also does not belong to U. Figure 3.6 shows the pseudo-code for the *updateT* method that incorporates such a prefix-closed optimization to reduce the number of membership queries asked. This method first checks whether or not there exists a prefix of a given event sequence that does not belong to U (line 2). If so, then the method updates T without asking a membership query (line 3). If not, the *updateT* method updates T by asking a membership query (line 5).

3.5.2 FLAVERS-Based L* Teacher

Our teacher has access to the original HIS subject consisting of the system TFG, the system constraint set, and the system requirement property. This teacher also has access to the derived requirement alphabet Σ_D. The teacher uses FLAVERS to answer the membership and equivalence queries. Section 3.5.2.1 first describes the *membership query* that takes as input a given event sequence from Σ_D^*. This query uses FLAVERS to treat the system requirement as the property and the specified event sequence as a constraint to determine whether or not that event sequence is safe (i.e. corresponds to only safe paths). Section 3.5.2.2 then describes the *equivalence query* that takes as input the current derived requirement and uses FLAVERS to search for either safety or permissiveness counterexamples. For the safety counterexample generation, this query uses FLAVERS to treat the system requirement as the property and the current derived requirement as a constraint to search for unsafe paths. For the permissiveness counterexample generation, the query uses FLAVERS to treat the current derived requirement as the property and the system requirement as a constraint to search for safe paths.

3.5.2.1 Membership Query

Since our teacher has assess to the original HIS subject consisting of the system TFG G, the property is the system requirement R_S, and the system constraint set CS, the membership query simply takes as input an event sequence σ and determines whether σ is safe (i.e. does belong to U) or is non-safe (i.e. does not belong to U). For this work, we want to apply the *prefix-closed optimization* described in the learner section but cannot because the node-tuple graphs are usually not prefix-closed. On the other hand, any infeasible path that reaches one or more constraint violations states that are trap states will always remain an infeasible path when extended. In a similar manner, an unsafe path that reaches a property violation state that is a trap state will always remain an unsafe path when extended. Thus we adapt the membership query to return one of the following. If the given event sequence is safe (i.e. corresponds to all safe paths), then IS_PREFIX is returned. For instance, the non-deterministic node-tuple graph shown in Figure 3.2 contains safe event sequence c. If that event sequence is non-safe but can sometimes be extended to an event sequence that is safe (i.e. corresponds to at least one safe path), then IS_POSSIBLE_PREFIX is returned. For instance, the non-deterministic node-tuple graph shown in Figure 3.2 contains non-safe event sequence a that can be extended to safe event sequence a·c but can also be extended to unsafe event sequence a·b. If the event sequence is non-safe and can never be extended to an event sequence that is safe (i.e. corresponds to all infeasible paths or unsafe ones), then IS_NOT_PREFIX is returned. For instance, the non-safe event sequence b can never be extended to a safe event sequence. We also adapt the optimized *updateT* method shown in Figure 3.6 to check if $T(t)$ is IS_NOT_PREFIX (line 2) and set $T(t)$ to IS_NOT_PREFIX (line 3). For instance, the non-deterministic node-tuple graph shown in Figure 3.2 does not contain prefix b and therefore the optimized *updateT* method returned IS_NOT_PREFIX. Thus, the prefix b·c will also return IS_NOT_PREFIX.

Figure 3.7 shows the pseudo-code for the membership query method. This method first checks whether or not the given event sequence is safe (line 3). Specifically, the *state propagation* algorithm is run on a HIS subject consisting of a property that is a copy of the system requirement R_S to check for the existence of unsafe paths, the TFG G, and a constraint set that is the system constraint set CS along with a new constraint C_σ that accepts the event sequence σ. (The *Build-Constraint* method that takes as input a given event sequence σ to be accepted along with whether or not the extensions of that event sequence should also be accepted is described in the next paragraph.) If the verification result is conclusive (in the figure shortened to CON for brevity), then the membership query method returns IS_PREFIX (line 4). If not, then this method next checks whether or not the non-safe event sequence can be extended to be a safe one (line 9), meaning there exists a suffix e from Σ_U^* such that the event sequence $\sigma \cdot e$ is safe. Specifically, the *state propagation* algorithm is run on a HIS subject consisting of a property that is the complement of the system requirement R_S to check for the existence of safe paths instead of unsafe ones, the TFG G, and a constraint set that is the system constraint set CS along with a new constraint C_σ that accepts event sequence σ and any extensions of that event sequence. If the verification result is inconclusive because a counterexample path exists (in the figure shortened to INC-CEXPATH to save space), then the membership query method returns IS_POSSIBLE_PREFIX (line 10), otherwise this query returns IS_NOT_PREFIX (line 12).

```
 1: // Check whether or not the given event sequence $\sigma$ is safe
 2: $C_\sigma \leftarrow$ BUILD-CONSTRAINT($\sigma$, FALSE)
 3: **if** (STATE-PROPAGATION($R_S$, G, CS $\cup$ { $C_\sigma$ }) = CON) **then**
 4:     **return** IS_PREFIX
 5: **else**
 6:     // Check whether or not the non-safe event sequence $\sigma$ can be
 7:     // extended to be a safe event sequence
 8:     $C_\sigma \leftarrow$ BUILD-CONSTRAINT($\sigma$, TRUE)
 9:     **if** (STATE-PROPAGATION($\neg R_S$, G, CS $\cup$ { $C_\sigma$ }) = INC-CEXPATH) **then**
10:         **return** IS_POSSIBLE_PREFIX
11:     **else**
12:         **return** IS_NOT_PREFIX
13:     **end if**
14: **end if**
```

Figure 3.7: Pseudo-code for the teacher's membership query method

The *Build-Constraint* method creates a new constraint that accepts a given event sequence σ and optionally any extensions of that event sequence. Initially, the *Build-Constraint* method creates a new FSA with alphabet Σ_D and a single new state that is set as the start state of the FSA. This method then walks through the event sequence σ creating a new state for each event and then creating a new transition from the previous state to the new state on that event. This method ensures that the FSA accepts event sequence σ by setting the last state created as the single accepting state of the FSA. If the FSA is also supposed to accept the extensions of event sequence σ, then the method makes the last state created into a trap state by adding a transition from the last state to itself on each event in Σ_D.

3.5.2.2 Equivalence Query

As mentioned in the overview of this section, the teacher has assess to the original HIS subject consisting of the system TFG, the system constraints, and the system requirement property. The equivalence query additionally takes as input the current derived requirement. This query determines whether or not the language of the current derived requirement is equivalent to U. If the language of the current derived requirement is equivalent to U, the query returns true. Otherwise, it returns false, along with a counterexample witnessing the inequivalence. The equivalence query first tries to generate a safety counterexample that is allowed by the current derived requirement but violates the system requirement. If a safety counterexample is found, then this query returns false along with that counterexample. If a safety counterexample is not found, then the query tries to generate a permissiveness counterexample that is disallowed by the current derived requirement but satisfies the system requirement. If a permissiveness counterexample is found, then the equivalence query returns false along with that counterexample. If neither a safety nor permissiveness counterexample is found, then this query returns true. In the next section, we discuss why the equivalence query is not precise enough to guarantee that $\mathcal{L}(D)$ is equivalence to U, which allows all safe event sequences and disallows all non-safe event sequences.

In the next two paragraphs, we describe the *safety counterexample generation* method and the *permissiveness counterexample generation* method.

The safety counterexample generation method takes as input the system requirement property R_S, the system TFG G, the system constraint set CS, and the current derived requirement D_i. This method returns a safety counterexample if one is found and NULL otherwise. The method searches for an unsafe path that is allowed by the current derived requirement D_i but violates the system requirement R_S. Specially, the safety counterexample generation method runs the *find path* algorithm, which is one of the FLAVERS verification algorithms described in the background chapter, on a HIS subject consisting of a copy of the system requirement property R_S, the TFG G, and a copy of the the constraint set CS augmented with a copy of the current derived requirement D_i. Since the *state propagation* algorithm does not generate counterexample paths, we use the *find path* algorithm to iteratively generate the counterexample paths. If an unsafe path is not found, then the safety counterexample generation method returns NULL. If an unsafe path is found, then this method first takes the path that is a sequence of node-tuples and projects on the derived requirement alphabet Σ_D to produce the corresponding event sequence from Σ_D. For instance, the path 1, 3, 6, 5 shown in Figure 3.2 projected on a derived requirement alphabet that contains {a, b, c} produces event sequence a·c. On the other hand, the same path projected on a derived requirement alphabet that only contains {c} produces the event sequence c.

Other interface synthesis methods (e.g., [4, 35]) use heuristics to generate permissiveness counterexamples that will produce an approximation to the most permissive interface. This is the approach taken in our work. The permissiveness counterexample generation method takes as input the system requirement property R_S, the system TFG G, the system constraint set CS, and the current derived requirement D_i. This method returns a permissiveness counterexample if one is found and NULL otherwise. Figure 3.8 shows the pseudo-code for the permissiveness counterexample generation method that basically uses a search-based counterexample generation algorithm to heuristically find permissiveness counterexamples. This method first iteratively searches for every potential permissiveness counterexample path t that violates the current derived requirement D_i (line 3). Specifically, the *find path* algorithm is run on a HIS subject consisting of a property that is a copy of the current derived requirement D_i, the TFG G, and the constraint set CS. This is an extension of the permissiveness heuristic originally proposed by Cobleigh et al. [20]. Alternatively, the permissiveness counterexample generation method could first iteratively search for every potential permissiveness counterexample path that violates the current derived requirement D_i and also satisfies the system requirement property. In a similar manner to the membership query method, this method would run the *find path* algorithm on a HIS subject consisting of a property that is a copy of the current derived requirement D_i, the TFG G, and the constraint set CS along with an additional constraint that is a copy of the system requirement R_S. This alternative permissiveness heuristic will be further described in the next chapter. For each potential permissiveness path t, the method checks whether or not that path is null (line 4). If that path is null, then the permissiveness counterexample generation method returns NULL (line 5). If the path is non-null, then this method computes the event sequence σ_t from Σ_D that corresponds to path t (line 9). The method then checks whether or not the event sequence σ_t is safe by asking a membership query (line 10). If so, then the permissiveness counterexample generation method returns the permissiveness counterexample event sequence σ_t (line 11). If not, then this method continues to the next iteration. For instance, the permissiveness counterexample generation method given the simplified node-tuple graph shown in Figure 3.2 could find the potential permissiveness counterexample path 1, 3, 6, 5 (line 3) that corresponds to event sequence a·c (line 9). This event sequence is determined to be safe (line

10) and returned as a permissiveness counterexample (line 11). Alternatively, this method given the node-tuple graph shown in Figure 3.2 could find the potential permissiveness counterexample path 1, 3, 4, 5 (line 3) that corresponds to event sequence a·b (line 9). The method, however, given that same node-tuple graph could find unsafe path 1, 7, 8, 9 that also correspond to event sequence a·b (line 10). Because event sequence a·b corresponds to both a safe path and an unsafe path, that event sequence is determined to not be safe (line 10). Thus, the method continues to the next iteration.

```
1:  // Search for a potential permissiveness counterexample path $t$
2:  // that is disallowed by the current derived req. $D_i$
3:  for all ($t \in$ FIND-PATH($D_i$, $G$, $CS$)) do
4:      if ($t = NULL$) then
5:          return $NULL$
6:      else
7:          // Check whether or not event sequence $\sigma_t$ is safe
8:          // and thus should be allowed by the final derived req.
9:          $\sigma_t \leftarrow$ PROJECT($t, \Sigma_D$)
10:         if (MEMBERSHIP-QUERY($\sigma_t$) = IS_PREFIX) then
11:             return $\sigma_t$
12:         end if
13:     end if
14: end for
```

Figure 3.8: Pseudo-code for the teacher's permissiveness counterexample generation method

3.5.3 Guarantees

In Section 2.5, we discussed that the actual system may contain non-determinism that must be taken into account when determining whether or not a given event sequence is safe or non-safe. The system model may also use system abstractions that introduce non-determinism. The learning-based requirement deriver handles the non-determinism in such a way that the final derived requirement is guaranteed to be safe but not necessarily adequately permissive. In particular, the permissiveness counterexample generation method may not generate some of the actual permissiveness counterexamples. In the next chapter, we will vary the permissiveness counterexample generation method along several dimensions to try to produce more of the actual permissiveness counterexamples to improve the permissiveness of the final derived requirements. The L^* learner [6, 61] guarantees that the final derived requirement is represented as a minimal deterministic FSA. In the following, we first discuss the guarantees about the derived requirements and then discuss the worst-case complexity of the requirement deriver.

For the learning-based requirement deriver, our teacher classifies the event sequences from Σ_D as either safe or non-safe. We defined the unknown language U being learned to disallow the non-safe event sequences and allow the safe ones. To guarantee that the final derived requirement D is equivalent to U, this teacher needs to be minimally adequate, meaning that both the membership and equivalence queries must return precise answers. Our equivalence queries, however, only return precise answers for the unsafe event sequences but imprecise answers for the non-terminating, infeasible, and safe event sequences. This means that the final derived requirement is guaranteed to disallow all unsafe event sequences (i.e. be safe) but may not allow all safe event sequences (i.e. be adequately permissive). Additionally, this means that the requirement may not disallow all of the non-terminating and infeasible event sequences. In what follows, we first describe the guarantees for the membership query and then the guarantees for the equivalence query.

Figure 3.7 shows the pseudo-code for the membership query that determines whether a given event sequence σ from Σ_D is safe or non-safe (line 3). To handle the non-determinism, the *state propagation* algorithm is used to check whether or not that event sequence corresponds to safe paths and no unsafe ones. If so, then this query returns safe (line 4), otherwise the query returns non-safe (line 10 or 12). This means that the membership queries return precise answers.

The equivalence query first uses a safety counterexample generation method and if needed then a permissiveness counterexample generation method. Section 3.5.2.2 describes the safety counterexample generation method that uses the *find path* algorithm to search for a safety counterexample that is allowed by the current derived requirement but violates the system requirement (i.e. is unsafe). This algorithm was designed to generate such unsafe paths but not the non-terminating or infeasible paths. For the non-safe event sequences, the equivalence queries return precise answers for the unsafe event sequences but imprecise answers for the non-terminating and infeasible event sequences. On the other hand, Figure 3.8 shows the permissiveness counterexample generation method that is a conservative but imprecise heuristic. This method also uses the *find path* algorithm (line 3) to search for permissiveness counterexamples that are disallowed by the current derived requirement but satisfy the system requirement. Because the *find path* algorithm can be influenced by the non-determinism, some actual permissiveness counterexamples may not be generated as described below. For the safe event sequences, the equivalence queries return imprecise answers.

In more detail, the permissiveness counterexample generation method first uses the find path algorithm to search for a potential permissiveness counterexample path that is disallowed by the current derived requirement (line 3). To handle the non-determinism, the method then maps that path to its corresponding event sequence (line 9) and checks whether or not that event sequence is safe (line 10). If so, the event sequence is returned as a permissive counterexample (line 11). If not, the event sequence is ruled out as a permissiveness counterexample to guarantee that the derived requirement is safe. For instance in the non-deterministic node-tuple graph shown in Figure 3.2, the potential permissiveness counterexample path 1, 3, 4, 5 that corresponds to event sequence a·b is ruled out because event sequence a·b also correspond to unsafe path 1, 7, 8, 9. For every potential permissiveness counterexample path, the find path algorithm will add each of that path's node-tuples to the visited data structure that stores the node-tuples that have already been explored. This means that a safe path ruled out because of non-determinism can lead to another safe path being ruled out because some of its node-tuples have already been marked as visited. Giannakopoulou and Păsăreanu [35] refer to this as the *state-matching problem*. For instance, the potential permissiveness counterexample path 1, 3, 4, 5 was ruled out (and marked node-tuples 1, 3, 4, and 5 as visited). Thus, the potential permissiveness counterexample path 1, 3, 6, 5 is also ruled out (because the node-tuple 5 was already visited). For FLAVERS, the node-tuple graphs are non-deterministic when the TFG is non-deterministic, the *generate next tuples* function is non-deterministic, or both. The *generate next tuples* function is designed to be non-deterministic but in practice is implemented in a deterministic manner. Thus, this function given the initial tuple and an event sequence σ will produce a unique tuple that is either safe or non-safe. The function therefore prevents the non-deterministic system models from leading to the state-matching problem. On the other hand, the permissiveness counterexample generation method first performs an imprecise search for a potential permissiveness counterexample path that violates the current derived requirement (line 3) and then checks whether or not that path is safe (line 9). If the potential permissiveness counterexample path is unsafe, then that path is ruled out. In a similar manner to the non-deterministic case, such unsafe paths can rule out safe paths because of the state-matching problem. The next chapter will describe alternative permissiveness counterexample heuristics that perform a more precise search for potential permissiveness counterexamples to try to avoid ruling out actual permissiveness counterexamples.

In the worst case, the number of membership queries made by the L^* learner is $O(k \cdot m^2) + m \log l$, where k is the number of events in the alphabet Σ_D, m is the number of states in the minimal deterministic FSA D, and l is the length of the longest counterexample returned by any equivalence query. Since FLAVERS is used to answer the membership and equivalence queries, each query has worst case complexity that is $\mathcal{O}(n^2 \cdot s)$, where n is the number of nodes in the TFG and s is the product of the number of states in the property and the number of states in each of the constraints.

3.6 Derived Requirement Permissiveness Classifier

In the Introduction, we formally defined a derived requirement to be adequately permissive when that requirement allows all safe event sequences. This classifier conservatively checks whether or not a given derived requirement disallows any *potential permissiveness counterexamples* that correspond to safe paths. If not, then that derived requirement is adequately permissive because all potential permissiveness counterexamples are allowed by the requirement. If so, then the derived requirement may or may not be adequately permissive because one or more potential permissiveness counterexamples are disallowed by that requirement.

In the FLAVERS section, we discussed that the *find path* algorithm can be used to generate some of the possible terminating paths through the node tuple-graph but not necessarily all of them. Thus, the derived requirement permissiveness classifier cannot use this algorithm to perform the check for safe paths since some of them may not be generated. On the other hand, the *build node-tuple graph* and *state propagation* algorithms summarize all possible terminating paths through that node-tuple graph but cannot generate any of those paths. This classifier therefore could use either of these algorithms to check for potential permissiveness counterexamples. In the previous chapter, we described how the *build node-tuple graph* algorithm represents the node-tuples as pairs of a TFG node and tuple and explicitly represents the edges among those pairs. On the other hand, the *state propagation* algorithm represents the node-tuples by associating each TFG node with a set of tuples and uses the exiting TFG edges among the TFG nodes. Thus, we chose to use the *state propagation* algorithm because the node-tuple graph representation is more compact in terms of space. Specifically, we use this algorithm to check for potential permissiveness counterexamples by treating the system requirement as a constraint to check for safe paths and by treating the derived requirement as the property to check whether or not any of those safe paths are disallowed by that derived requirement. In more detail, the derived requirement permissiveness classifier runs the *state propagation* algorithm on a HIS subject consisting of the system TFG, the system constraints along with a new constraint that is a copy of the system requirement, and the property is a copy of the derived requirement. This classifier checks whether or not the verification result is CONCLUSIVE. If so, the classifier returns “Is adequately permissive.” If not, the classifier returns “May be adequately permissive.” The permissiveness classifier uses the *state propagation* algorithm to check for potential permissiveness counterexamples in a similar manner to how the teacher’s *permissiveness counterexample generation* method uses the *find path* algorithm to generate the potential permissiveness counterexamples (shown in line 4 of Figure 3.8).

CHAPTER 4
DERIVATION IMPROVEMENTS

For our HIS-based requirement derivation approach, the L^* learner must be provided with a teacher that can answer two different types of queries, membership queries and equivalence queries. Our teacher answers both queries using a model checker. For this work, we consider model checkers such as Spin [49], NuSMV [18], Java Pathfinder, and FLAVERS that provide a counterexample generation algorithm that take as input a system model and one of its properties and then explicitly or symbolically generate the reachability graph to iteratively find counterexamples. Since our preliminary evaluation showed that this approach often did not scale well, we investigated several learning and model checking optimizations. These optimizations improved performance but also influenced the generated counterexamples, impacting the permissiveness of the derived requirements. Thus, we investigated the impact on permissiveness of several counterexample generation heuristics and the interaction of the optimization techniques and counterexample generation approaches. Although many of the optimizations and heuristics have been presented previously, we show how their selective combination affects the derivation results with respect to performance as well as permissiveness.

Various optimizations have been developed for the learning algorithms (e.g., [4,15]) and model checking techniques (e.g., [25,38]). As part of this work, we extended our approach to incorporate several optimizations intended to reduce the amount of space and time needed to perform the requirement derivation. We found, however, that some of these optimizations involving the encoding of variables in the HIS model may interact with the counterexample generation approaches. When these optimizations are applied, the derived requirement may be too restrictive to be useful because not enough permissiveness counterexamples are generated. Thus, we also investigate a general class of permissiveness counterexample generation heuristics that approximate the derived requirements that are most permissive. Our goal is to find one or more combinations of the optimizations and heuristics that have reasonable performance in terms of space and time, but also derive requirements that are permissive enough to be useful. Although we have evaluated these optimizations and heuristics for the particular learning algorithm and model checker employed by our requirement deriver, we believe the results would be applicable to other requirement derivers built with similar approaches. In the next chapter, we will discuss a very preliminary evaluation of another requirement deriver that employs the same learning algorithm and the Java Pathfinder model checker. We first provide more details about the derivation optimizations and the permissiveness counterexample generation heuristics. We then describe our experimental evaluation.

4.1 Derivation Optimizations

For our HIS-based requirement derivation approach, we need to translate the HIS model written in Little-JIL to the FLAVERS representation consisting of a TFG and set of constraints. Since a Little-JIL step's (or procedure's) execution may need to copy input and output parameters, send/receive to/from communication channels, and throw exceptions, a Little-JIL step is typically translated to between 5 and 20 TFG nodes. Additionally, the Little-JIL to FLAVERS translator represents each parameter, exception, or channel as a FLAVERS constraint. For this work, the system properties (e.g., never overdose) specify the intended or unintended interactions between the given component (e.g., pump) and its environment (e.g., perform in-patient surgery). Thus the model checker often needs to model the variables relevant to the properties.

As mentioned in the introduction, our HIS-based requirement derivation approach takes as input a HIS model that is a parallel composition of the component of interest and the environment in which that component is used. At a high level of abstraction, the component defines a set of remote procedures that the environment calls to accomplish its goals. In our HIS models, the remote procedure calls are encoded using communication channels that send and receive messages. Thus, we applied two model checking optimizations to try to ameliorate the state explosion problem. Both optimizations target the sequences of procedure calls to the component made by its environment: we applied program slicing techniques (e.g., [72]) to keep only the portions of the model relevant to those procedure calls, and we used partial order reduction techniques (e.g., [19]) to reduce the number of thread interleavings between the component and its environment. Although these optimizations improved performance by decreasing both the space and time needed, the results for our largest subjects showed the deriver's performance still did not scale well. Thus, we explored the application of three additional model checker optimizations and one learner optimization that affect how the interactions are encoded in the HIS model to further improve the deriver's performance.

Many researchers have investigated model checker optimizations (e.g., [18,19]) and learner optimizations (e.g., [14, 33]). We know from other work (e.g., [9]) that alternative ways of encoding different modeling language features in the system model can have substantial impacts on the size of the model and the performance of the verification algorithm. For this work, we therefore applied three additional model checker optimizations that influence how the interactions are encoded in the HIS model. Two model checker optimizations employ constant propagation and live variable analysis, two common compiler techniques, that affect how the variables relevant to the system property are encoded in the HIS model. The third model checker optimization affects how the channels are encoded in HIS model. We also applied a learner optimization that uses a particular characteristic of the system models to reduce the number of times that the model checker is run to answer the membership queries. In what follows, we will illustrate the four optimizations by applying them to the *setDose* procedure of the pump shown in Figure 2.4. As described in Section 2.3, for each selected component (e.g., *pump*), we declare variables to store the internal state of that component (e.g., *currLib_*). For each of its procedures, we declare variables to store the values of the input parameters (e.g., *setDose_dose*) along with the output parameters or exceptions thrown (e.g., *setDose_DoseAlert*).

The four optimizations will be further described in the following sections. Overall the four optimizations did not significantly interact with each other. The learner optimization and the channel-related model checker optimization improved the deriver's performance and did not impact the permissiveness of the derived requirements. The compiler-based model checker optimizations tended to significantly improve the deriver's performance, especially in terms of space, but could drastically reduce the permissiveness of the derived requirements. Since we need the improved performance provided by all four optimizations, we investigated the ability of the counterexample generation heuristics described in the next section to improve the permissiveness of the derived requirements.

4.1.1 Variable Modeling Alternatives

FLAVERS can track the run-time values of user-specified variables to improve the precision of the system model represented as a TFG and a set of constraints. Since the worst case of the verification algorithm depends on the size of the TFG, the number of constraints, and the size of each constraint, we explored two variable modeling alternatives that differ in how the run-time variable values are tracked in the TFG and constraints to study the impact of both alternatives on the performance of the verification algorithm. In particular, the Little-JIL to FLAVERS translator allows the user to specify the variable modeling alternative on a per variable basis. We next briefly describe both variable modeling alternatives and illustrate each alternative on the *setDose* procedure.

In the first alternative, the variables are modeled as constraints and their run-time values are tracked in the node-tuple graph generated by the verification algorithm. For a given variable (e.g., *setDose_dose*), the Little-JIL to FLAVERS translator employs a template-based approach to construct the appropriate TFG nodes annotated with the events that set (e.g., setDose_dose=LOW) and test (e.g., setDose_dose==LOW) the value of that variable. For instance, the *setDose* procedure shown in Figure 2.4 is translated to the portion of the TFG shown in Figure 2.10 when the user specifies to apply this first alternative to the variables *currLib_*, *currDose_*, *setDose_dose*, and *setDose_DoseAlert*. To illustrate, one potential path through the pseudo-code of the *setDose* procedure consists of statements 1, 5, 8, 9, 10. That path is translated to TFG nodes { 0, 1a, 1b, 5, 8, 9a, 9b, 9c, 9d, 10, 11 } where the *setDose_dose* variable is set at TFG nodes 1a and 1b and then tested at TFG nodes 8, 9a, and 9b. Additionally, the translator constructs an appropriate variable constraint represented as an FSA as described in the FLAVERS section to track the run-time values of that variable during the analysis. For instance, Figure 2.9 shows the variable constraint for the *setDose_dose* variable. So in this first alternative, all the information about the run-time values of variables is tracked in the states of the variable constraints, and the verification algorithm generates the full node-tuple graph by associating nodes with tuples of constraint states. To illustrate, Figure 2.12 shows the portion of the node-tuple graph for the *setDose* procedure where each node-tuple shows the TFG node (top) and the tuple (bottom). In particular, every tuple tracks the states of the variable constraints for *currLib_*, *currDose_*, *setDose_dose*, *setDose_DoseAlert*. The

FLAVERS background section describes how the verification algorithm generates the full node-tuple graph in more detail. We call this first alternative the *analysis-time variable model.*

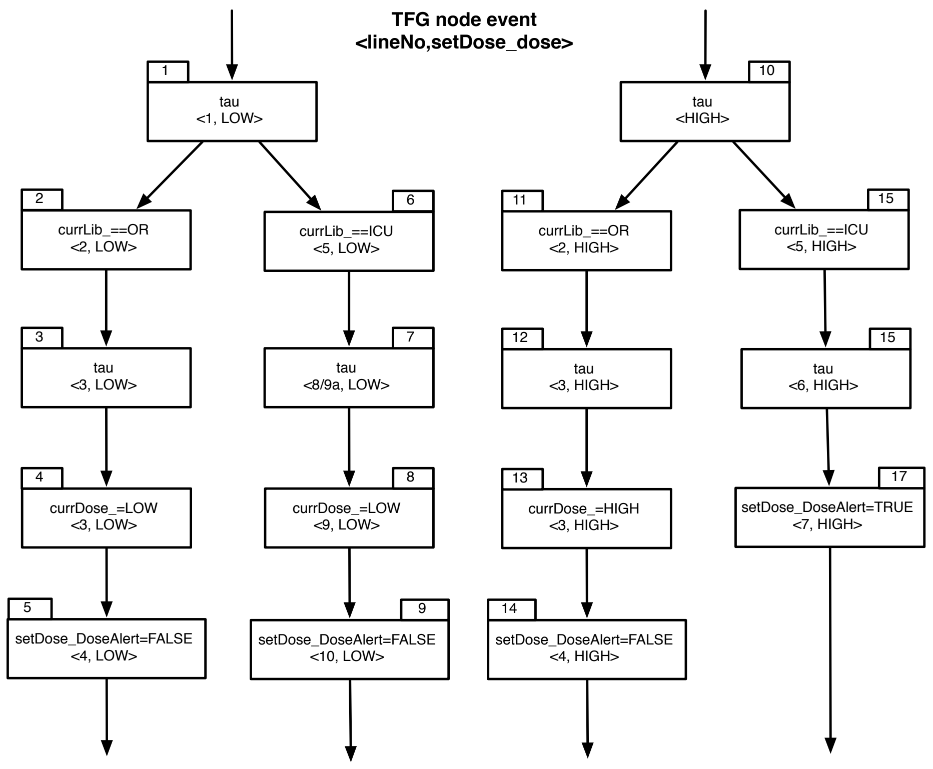


Figure 4.1: Portion of the TFG for the *setDose* procedure constructed when the first alternative for modeling the *currLib_*, *currDose_*, and *setDose_DoseAlert* variables is applied and the second alternative for modeling the *setDose_dose* variable is applied

In the second alternative, selected variables are modeled in the TFG, in particular their run-time values are tracked in the TFG nodes. Like FLAVERS, the INCA model checker [24] first constructs a high-level model of the execution of the system as a graph and then uses that graph as the basis for its verification. But INCA uses constant propagation [1] to encode variable values in the graph and hence does not need to explicitly track those values during the verification algorithm. In FLAVERS, we have implemented this approach by extending the Little-JIL to FLAVERS translator to encode the values of (user-specified) variables in the TFG nodes. Specifically, this translator associates a tuple consisting of the source location (in this case the line number) and the selected variables with each TFG node. This is similar to how the verification algorithms associate tuples of the property and constraints states, including the task and variable constraint states, with the TFG nodes. For instance, Figure 4.1 shows the portion of the TFG for the *setDose* procedure constructed when the user specifies to apply the first alternative to the *currLib_*, *currDose_*, and *setDose_DoseAlert* variables and to apply this second alternative to the *setDose_dose* variable. In this figure, each TFG node is annotated with its event (shown at the top) along with the encoded tuple that consists of the source location value and *setDose_dose* variable value (shown at the bottom). To illustrate, we consider the same path through the pseudo-code of the *setDose* procedure that consists of statements 1, 5, 8, 9, 10. In this figure, that path is translated to TFG nodes { 1, 6, 7, 8, 9, 10 } where the *setDose_dose* variable is set at TFG nodes 1 and 10 and then tested at TFG node 7. For clarity, we added the TFG nodes annotated with τ, but the TFG refinements will later remove such TFG nodes to try to reduce the size of the TFG. The TFG shown in Figure 4.1 essentially corresponds to the node-tuple graph shown in Figure 2.12 where any set or test of the *setDose_dose* variable was relabeled as *tau*. This second alternative eliminates the need for some variable constraints (e.g., dose) and the tracking of their states in the node-tuple graph during the verification algorithm. The second alternative, however, has the potential to increase the size of the TFG to track the variable values in its nodes. We call this second approach the *model-construction-time variable model*.

For this work, we must apply the analysis-time variable model to the channels because they are shared by multiple agents. We also must apply the analysis-time variable model to any parameters and exceptions that are shared by multiple agents. On the other hand, we could apply either variable model to any parameters and exceptions that are local to a single agent. In practice for most systems, the model-construction-time variable model can greatly reduce the space and time needed for the verification algorithm. Thus, we applied the model-construction-time variable model to the parameters and exceptions whenever possible.

4.1.2 Reset Dead Local Variables

A local variable is *live* at a given CFG node if and only if there exists a path to that node where that variable is defined (in our terminology set) and from that node where the variable is used (in our terminology tested) before being redefined. Otherwise that local variable is *dead* (meaning not live). For instance, Figure 2.10 uses the analysis-time variable model for the *setDose_dose* variable that is defined at TFG nodes { 1a, 1b } and used as TFG nodes { 3a, 3b, 3c, 3d, 6, 8, 9a, 9b, 9c, 9d }. Thus, the *dose* variable is live at TFG nodes: { 2, 3a, 3b, 3c, 3d, 5, 6, 8, 9a, 9b, 9c, 9d }. Conceptually, the model checkers need to track all possible values for the live variables but do not need to track the values of the dead variables. Thus some model checkers such as Bandera [25] and INCA incorporate a *reset dead local variables* optimization where each local variable that goes from live to dead is redefined to a single value. This optimization attempts to reduce the number of possible values of the local variables that need to be tracked in the node-tuple graph of the verification algorithm or in the TFG nodes to try to reduce the space and time needed to perform the verification.

For this optimization, the Little-JIL to FLAVERS translator allows the user to specify whether or not each local variable is reset to dead when possible. This translator first performs live variable analysis [1] on the CFGs. Based on the analysis results, each CFG node is then associated with a live variable set. A local variable goes from live to dead when an edge transfers control from a CFG node that contains that variable in its live variable set to another node that does not contain that variable in its live variable set. The translator represents a local variable going from live to dead with the insertion of a new assignment statement where the left hand side is the variable (e.g., *setDose_dose*) and the right hand side is the dead value of that variable, in our case the initial value of the variable (e.g., *LOW*). For instance, Figure 4.2 shows the portion of the TFG for the *setDose* procedure shown in Figure 2.10 where the reset dead local variables optimization was applied to the *setDose_dose* variable. The *setDose_dose* variable goes from live to dead represented by the inserted assignment statements (shown in bold) where the *setDose_dose* variable is set to *LOW* at TFG nodes: { 11, 12, 13 }. We described this optimization at the CFG-level. In practice, we implemented the optimization on the labeled transition systems.

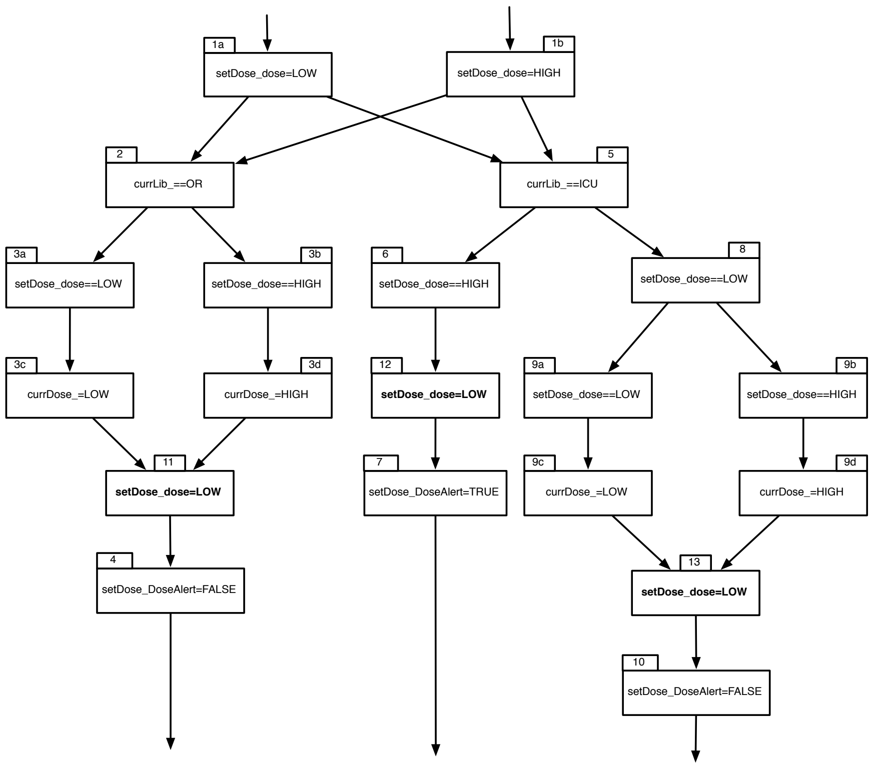


Figure 4.2: Portion of the TFG for the *setDose* procedure shown in Figure 2.10 where the reset dead local variables optimization was applied to the *setDose_dose* variable

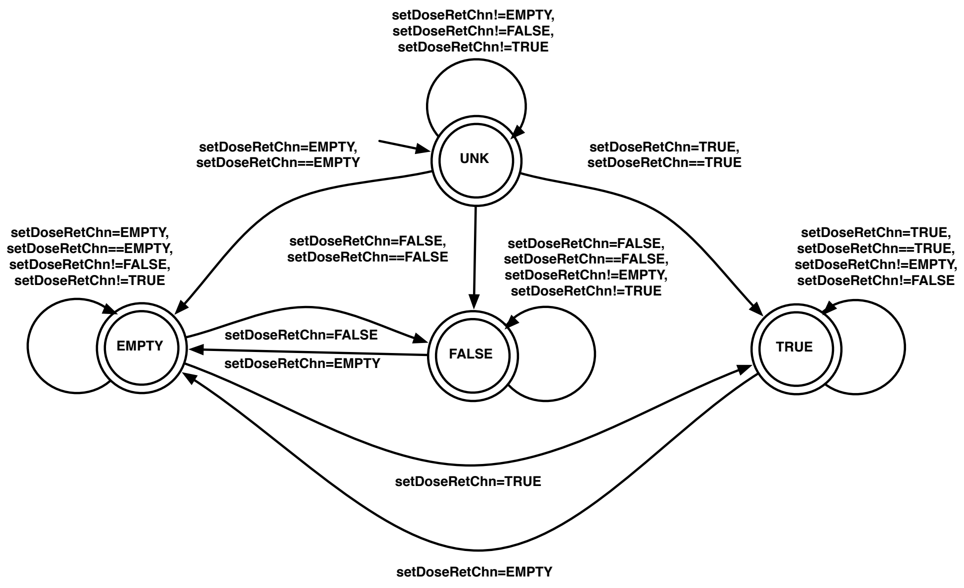
For a given HIS model, each procedure of the selected component has its parameters and exceptions translated to local variables that could have this reset dead local variables optimization applied. For our evaluation, if the reset dead local variables optimization is not applied, then the requirement deriver often exceeds either its space or time bound. This evaluation therefore applied the reset dead local variables optimization whenever possible.

4.1.3 Channel Modeling Alternatives

As mentioned in the background section, we represent the interactions between the selected component and its environment as remote procedure calls. Specifically, we model the remote procedure calls with *channels* in the HIS models written in Little-JIL. Recall that a channel (e.g., *setDoseRetChn*) stores a buffer of messages of a user-specified type (e.g., boolean) and provides two atomic operations, *send* and *receive*, to access the messages. Thus to precisely model the remote procedure calls, the input to the model checker often must encode the channel semantics in the system model. For FLAVERS, we explored two channel modeling alternatives that encode each channel as a single-slot buffer of messages, meaning that channel stores a single message. The message may be set to a special *EMPTY* value or one of the possible values of the user-specified type of the message (e.g., TRUE). The send operation blocks waiting until the channel is not full and then adds a given message to the channel. The receive operation blocks waiting until the channel is not empty and then removes a message from the channel. In both alternatives, the channels are modeled as constraints and the run-time values of the messages in the channels are tracked in the node-tuple graph of the verification algorithm. The first alternative encodes the semantics of the two channel operations in the TFG while the second alternative encodes them in the constraint. We next briefly describe each channel modeling alternative and illustrate its application to the return statement of the *setDose* procedure of the pump.

For the first channel modeling alternative, the Little-JIL to FLAVERS translator encodes the channel semantics by constructing a variable constraint for a given channel (e.g., *setDoseRetChn*) to track the run-time values of the message stored in that channel. Additionally the translator inlines the calls to send to and receive from the channel in the TFG. We call this the *low-level channel model*.

To illustrate, Figure 4.3 shows the variable constraint for the channel *setDoseRetChn*. In this figure, there is a state labeled with the special $UNKNOWN$ value (shortened to UNK), the special $EMPTY$ value, and the remaining states labeled with the other possible values of the message stored in the channel (e.g., TRUE). The transitions are labeled with events for setting the value of the message (e.g., $setDoseRetChn = TRUE$) and testing the value of the message (e.g., $setDoseRetChn == TRUE$). The figure only shows the transitions that may be executed by the inlined calls that send to and receive from the channel.

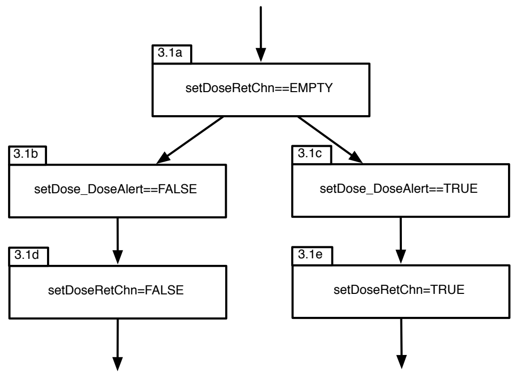


setDoseRetChn!=EMPTY, setDoseRetChn!=FALSE, setDoseRetChn!=TRUE
setDoseRetChn=EMPTY, setDoseRetChn==EMPTY
UNK
setDoseRetChn=TRUE, setDoseRetChn==TRUE
setDoseRetChn=EMPTY, setDoseRetChn==EMPTY, setDoseRetChn!=FALSE, setDoseRetChn!=TRUE
setDoseRetChn=FALSE, setDoseRetChn==FALSE
setDoseRetChn=FALSE, setDoseRetChn==FALSE, setDoseRetChn!=EMPTY, setDoseRetChn!=TRUE
setDoseRetChn=TRUE, setDoseRetChn==TRUE, setDoseRetChn!=EMPTY, setDoseRetChn!=FALSE
EMPTY
setDoseRetChn=FALSE
setDoseRetChn=EMPTY
FALSE
TRUE
setDoseRetChn=TRUE
setDoseRetChn=EMPTY

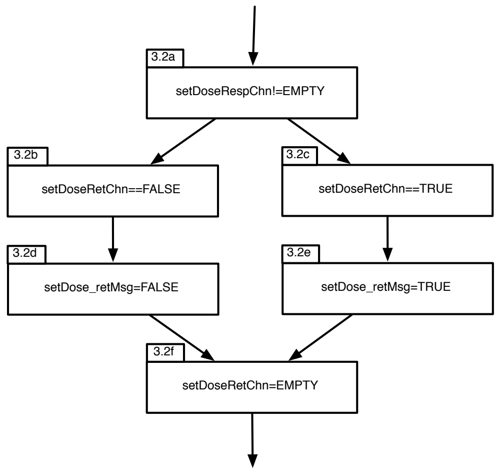
Figure 4.3: Variable constraint for channel *setDoseRetChn* that stores a message of boolean type

For the low-level channel model, the Little-JIL to TFG translator defines a TFG template that inlines the call to the send operation that copies a given input parameter (e.g., *setDose_DoseAlert*) to a specified channel (e.g., *setDoseRetChn*). To illustrate, Figure 4.4a shows the portion of the TFG for the return statement of the *setDose* procedure where this template was applied. The template first tests whether or not the channel is empty (corresponds to TFG node 3.1a). If the channel is not empty, then the send operation will block. If the channel is empty, then this operation will continue on to copy the input parameter to the channel. For each possible value (e.g., FALSE) of the input parameter (e.g., *setDose_DoseAlert*), this template copies that input parameter value to a new message (corresponds to TFG node 3.1b) and then adds that message to the channel (corresponds to TFG node 3.1d). The translator also defines a TFG template that inlines the call to the receive operation that copies a message from a given channel (e.g., *setDoseRetChn*) to a specified output parameter (e.g., *setDose_retMsg*). To illustrate, Figure 4.4b shows the portion of the TFG for the return from the *setDose* procedure where this template was applied. The template first tests whether or not the channel is not empty (corresponds to TFG node 3.2a). If the channel is empty, then the receive operation will block. If the channel is not empty, then this operation will continue on to copy from the channel to the output parameter. For each possible value (e.g., FALSE) of the message in the channel, this template copies that message value (corresponds to TFG node 3.2b) to the output parameter (corresponds to TFG node 3.2d). Lastly, the template removes the message from the channel (corresponds to TFG node 3.2f).

For the second channel modeling alternative, the translator encodes the channel semantics by constructing a channel constraint for a given channel (e.g., *setDoseRetChn*) that both tracks the run-time values of the message stored in that channel and inlines the calls for the send and receive operations for the channel. The translator also constructs TFG nodes annotated with events that correspond to making the calls that send to and receive from the channel. This modeling alternative attempts to reduce the size of the TFG by adding fewer nodes and edges. We call this the *high-level channel model*.



(a) Portion of the TFG for the pump where the low-level send operation was applied to one of the return statements of the *setDose* procedure



(b) Portion of the TFG for the in-patient surgery process where the low-level channel receive operation was applied to the return from the *setDose* procedure

To illustrate, Figure 4.5 shows the channel constraint for the channel *setDoseRetChn*. In each channel constraint (e.g., *setDoseRetChn*), there is a state for the special EMPTY value and a state for every possible value of the type of the message (e.g., TRUE). The state for the EMPTY value is set as the start state and all states that correspond to values of the message are set as accepting states. Additionally, there are transitions labeled with the events that send a message to that channel (e.g., **SND**(*setDoseRetChn*, TRUE)), receive a message from the channel (e.g., **RCV**(*setDoseRetChn*, TRUE)), and test whether or not there is a message (e.g., **isEmpty**(*setDoseRetChn*)). In more detail, the send operation blocks waiting until the channel is empty. This is represented by the implicit transitions from each possible message state to the violation state on the send operations. For example, the FALSE state transitions to the violation state on the event **SND**(*setDoseRetChn*,FALSE). The send operation is permitted when the channel is empty. For each possible message, this is represented by the transition from the EMPTY state to the corresponding message state on the send operation of that message. For example, the EMPTY state transitions to the FALSE state on the event **SND**(*setDoseRetChn*,FALSE). On the other hand, the receive operation blocks waiting until the channel is not empty. This is represented by the implicit transitions from the EMPTY state to the violation state on the receive operations. For example, the EMPTY state transitions to the violation state on the event **RCV**(*setDoseRetChn*,FALSE). The receive operation is permitted when the channel is not empty. For each possible message, this is represented by the transition from the corresponding message state to the EMPTY state on the receive operation of that message. For example, the FALSE state transitions to the EMPTY state on the event **RCV**(*setDoseRetChn*,FALSE).

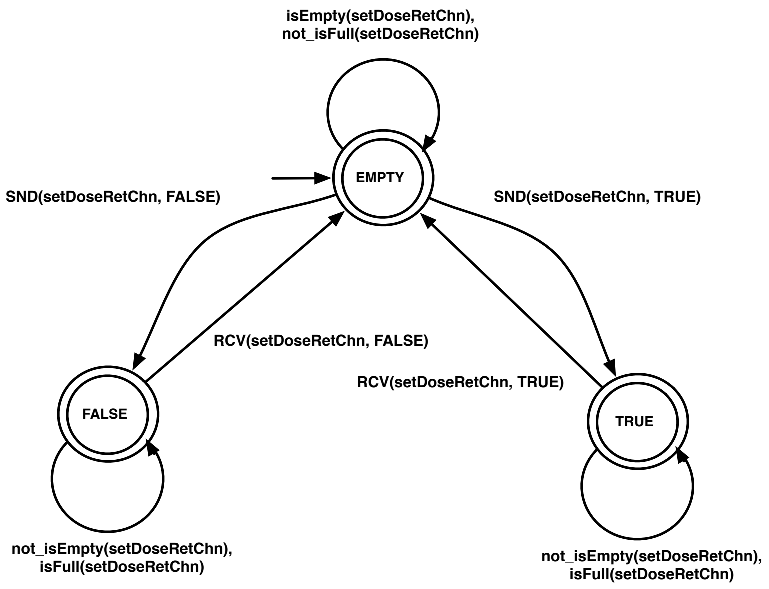
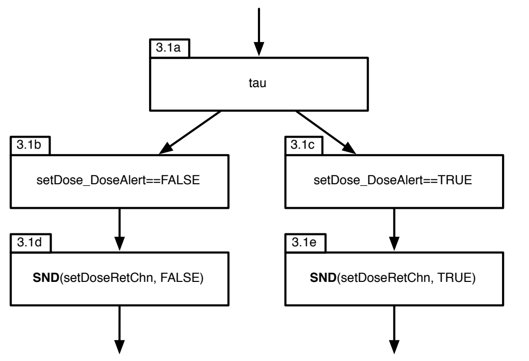
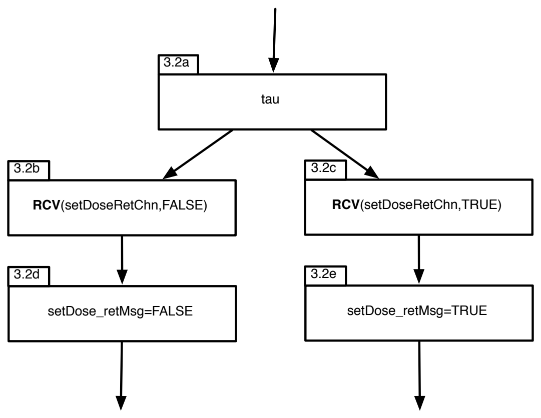

isEmpty(setDoseRetChn),
not_isFull(setDoseRetChn)
EMPTY
SND(setDoseRetChn, FALSE)
SND(setDoseRetChn, TRUE)
RCV(setDoseRetChn, FALSE)
RCV(setDoseRetChn, TRUE)
FALSE
TRUE
not_isEmpty(setDoseRetChn),
isFull(setDoseRetChn)
not_isEmpty(setDoseRetChn),
isFull(setDoseRetChn)

Figure 4.5: Channel constraint for channel *setDoseRetChn* that stores a message of boolean type

For the high-level channel model, the Little-JIL to TFG translator defines a TFG template to make a call to the send operation that copies a given input parameter (e.g., *setDose_DoseAlert*) to a specified channel (e.g., *setDoseRetChn*). To illustrate, Figure 4.6a shows the portion of the TFG for the second return statement of the *setDose* procedure where this template was applied. For each possible value (e.g., FALSE) of the input parameter (e.g., *setDose_DoseAlert*), the template first copies that input parameter value to a new message (corresponds to TFG node 3.1b). This template then makes a call to send that message to the channel (corresponds to TFG node 3.1d). The channel constraint will block the send operation when the channel is full. On the other hand, this constraint will continue on to copy the input parameter to the channel when the channel is empty. The translator also defines a TFG template to make a call to the receive operation that copies a message from the given channel (e.g., *setDoseRetChn*) to a specified output parameter (e.g., *setDose_return*). To illustrate, Figure 4.6b shows the portion of the TFG for the return from the *setDose* procedure where this template was applied. For each possible value (e.g., FALSE) of the message, the template first makes a call to receive that value of the message from the channel (corresponds to TFG node 3.2b). This template then copies that message value to the output parameter (corresponds to TFG node 3.2d). The channel constraint will block the receive operation when the channel is empty. On the other hand, this operation will continue on to copy from the channel to the output parameter when the channel is not empty.



(a) Portion of the TFG for the pump where the high-level send operation was applied to one of the return statements of the *setDose* procedure



(b) Portion of the TFG for the in-patient surgery process where the high-level receive operation was applied to the return from the *setDose* procedure

For this work, we implemented both channel models in FLAVERS. The two channel models could be implemented for other model checkers such as Spin. For our case studies, we compared the performance of the requirement deriver when the low-level and high-level channel models were applied. The low-level channel model generally needed twice the amount of space as the high-level one. Thus, our evaluation always applied the high-level channel model.

4.1.4 Incremental Membership Query

For the learning-based requirement deriver, Section 3.5.1 mentioned that the L* learner asks membership queries during the initialization and also during each iterative refinement. Thus, it is usually beneficial to reduce the number of membership queries, the cost of each membership query, or both to improve the performance of the learning-based requirement deriver. In Section 3.5.1, we described that our L* learner incorporates Rivest and Schapire's improvements to reduce the number of membership queries asked. Additionally, this L* learner incorporates the prefix-closed optimization to further reduce the number of membership queries asked. Figure 3.7 shows the pseudo-code for our teacher's membership query that employs a two-stage strategy to answer the query where each stage needs to run FLAVERS (lines 3 and 9). In the previous sections, we described how to reduce the cost of the membership queries by having each run of FLAVERS incorporate three model checker optimizations. In a complementary manner, this optimization conceptually performs the first stage and then incrementally performs the second stage to reduce the cost of the membership query by only needing one run of FLAVERS.

At a high-level, the *incremental membership query* takes as input a given event sequence σ from Σ_D^* and determines whether or not σ belongs in the unknown language U. The first stage initially checks whether or not event sequence σ is safe. This stage creates a new constraint C_σ with alphabet Σ_D that accepts event sequence σ. The stage then runs the *state propagation* algorithm to determine whether or not all terminating paths that are allowed by C_σ satisfy the system requirement. If so, this stage returns IS_PREFIX. If not, then the second stage checks whether or not event sequence σ that is non-safe can be extended to an event sequence that is safe. This stage modifies the constraint C_σ that accepts σ to also accept any extensions of σ. Since the modified constraint can allow more paths than the original constraint, the stage then continues to run the *state propagation* algorithm to check whether or not there exists a terminating path that is allowed by C_σ that satisfies the system requirement. If so, then this stage returns IS_POSSIBLE_PREFIX . If not, then the stage returns IS_NOT_PREFIX. In what follows, we describe how to implement the incremental membership query by extending the *state propagation* algorithm.

For the incremental membership query, the first stage can use the state propagation algorithm described in the FLAVERS section of the Background chapter. The *state propagation* algorithm employs data flow analysis techniques to verify whether or not all potential terminating paths through a system model satisfy a user-specified property. At a lower-level, this algorithm associates each TFG node with a set of tuples that summarize the paths that can reach that TFG node. The algorithm uses a worklist to start from the initial TFG node that is associated with the initial tuple and then generates the reachable node-tuples. Lastly, the *state propagation* algorithm determines the verification result by examining the final node-tuples. If the verification result is CONCLUSIVE, meaning that there exist final node-tuples that are satisfying and none that are violating, then the first stage determines that the membership query result is IS_PREFIX. If not, then the second stage is incrementally performed by extending the *state propagation* algorithm as follows. After the first stage generates the reachable node-tuples, the second stage modifies the constraint C_σ, which contains a single accepting state that accepts σ, to accept any extension of σ by making that accepting state a trap state. This modification will not affect the set of existing tuples because the states of C_σ remain the same. The modification, however, can lead to the generation of new tuples because the transitions of C_σ have been modified. The second stage then reinitializes the worklist by adding any TFG node to the worklist that is associated with a tuple where constraint C_σ is at its accepting state that was modified to be a trap state. This stage uses the worklist to generate any newly reachable node-tuples. Lastly, the stage reexamines the final node-tuples to determine whether or not there exists a final node-tuple that is satisfying. If so, then the second stage returns IS_POSSIBLE_PREFIX. If not, this stage returns IS_NOT_PREFIX.

4.2 Permissiveness Counterexample Generation Heuristics

In the previous chapter, we described the permissiveness counterexample generation method that specifies a heuristic for generating permissiveness counterexamples that correspond to terminating paths that are disallowed by the current derived requirement but that satisfy the system requirement. Some of the requirement derivation optimizations described in the previous section, however, can negatively impact that heuristic, which can significantly decrease the permissiveness of the derived requirements. Thus, we extend the permissiveness counterexample generation method to specify a general class of heuristics for generating permissiveness counterexamples. Specifically, this method is varied along multiple dimensions suggested by the search-based counterexample generation techniques (e.g., [30]) and the learning-based interface synthesis methods (e.g., [4]). These dimensions can significantly affect the permissiveness of the derived requirements as well as the performance of the the learning-based requirement deriver.

In more detail, the permissiveness counterexample generation method shown in Figure 3.8 iterates through a set of potential permissiveness counterexamples (line 3). The permissiveness of the derived requirement depends on which counterexamples are generated and in what order they are generated. The worst case of the requirement deriver depends on the length of the longest counterexample generated and the cost of the counterexample generation algorithm (in our case the *find path* algorithm). For such search-based counterexample generation algorithms, including the ones provided by FLAVERS, JPF, and Spin, there are two common approaches to vary the search to influence the set of counterexamples generated and the cost of the counterexample generation. The first approach narrows the search by applying additional constraints. The second approach guides the search by parameterizing some or all of the following key data structures and functions: the *is terminating* function, the *worklist* data structure, and the *visited* data structure. The *is terminating* function identifies whether or not a given node-tuple corresponds to a terminating path and hence may lead to the generation of a counterexample. The *worklist* data structure stores the node-tuples to be explored while the *visited* data structure determines whether or not a given node-tuple should be explored (again). For this work, the HIS models contain loops and therefore there may be an infinite number of potential counterexamples to be generated. One important responsibility of the visited data structure is to break out of such loops to generate a finite set of counterexamples. Typically only a small number of unrollings of these loops, however, are considered. Thus, special techniques may be used for the loops to increase the number of unrollings considered.

Figure 4.7 shows the pseudo-code for the extended permissiveness counterexample generation method that varies along the following five dimensions (underlined in the figure). The first dimension varies how the permissiveness counterexample generation method ensures that the permissiveness counterexample generated satisfies the system requirement (lines 2, 6, 13). In particular, the system requirement can be used as an additional constraint to the search (line 2). The next three dimensions vary the search-based counterexample generation algorithm's parameterization of the data structures and function. The second dimension varies the *is terminating* function to affect whether or not the prefixes of the terminating paths are generated. The third dimension varies the *worklist* data structure to specify the order of the node-tuples in the worklist. The fourth dimension varies the *visited* data structure to essentially try to increase branch coverage. On the other hand, the fifth dimension varies the learner (line 8) or the search-based counterexample generation algorithm (line 6) to specify the number of unrollings of the loops that are considered. For each of the five dimensions, we consider different implementation options that affect the set of permissiveness counterexamples generated and the performance of the learning-based requirement deriver in terms of space and time. We will call a combination, involving one option for each of these dimensions, a *permissiveness counterexample generation heuristic*, usually shortened to heuristic. In the remainder of this section, we give some additional details about the extended permissive counterexample generation method's dimensions.

```
1: if ($\underline{IsConstraintBased}$) then
2:     $CS \leftarrow CS \cup \{ R_S \}$
3: end if
4: // Search for a potential permissiveness counterexample path $t$
5: // that is disallowed by the current derived requirement $D_i$
6: for all ($t \in$ Find-Path($D_i$, $G$, $CS$, $\underline{IsTerminating}$, $\underline{Worklist}$, $\underline{Visited}$)) do
7:     if ($t = NULL$) then
8:         return $\underline{\text{Learner-Take-N-Steps-Further}(n)}$
9:     else
10:        // Check whether or not event sequence $\sigma_t$ is safe
11:        // and thus should be allowed by the final derived req.
12:        $\sigma_t \leftarrow$ Project($t$, $\Sigma_D$)
13:        if (Membership-Query($\sigma_t$) = IS_PREFIX) then
14:            return $\sigma_t$
15:        end if
16:    end if
17: end for
```

Figure 4.7: Pseudo-code for the extended permissiveness counterexample generation method that varies along the five dimensions (shown as underlined) that parameterize some of the key helper methods

4.2.1 Ensure the System Requirement is Satisfied

In Figure 3.8, we showed the original permissiveness counterexample generation method that specifies the current derived requirement as the property (line 3) to generate potential permissiveness counterexamples that are disallowed by the current derived requirement. This method then uses the teacher's membership query to ensure that a potential permissiveness counterexample also always satisfies the system requirement (line 10). We call this first option the *query-based* option. In Figure 4.7, we showed the extended permissiveness counterexample generation method that specifies the current derived requirement as the property (line 6) and the system requirement as a constraint (lines 2 and 6). This ensures that any potential permissiveness counterexample found is disallowed by the current derived requirement and also satisfies the system requirement. We call this second option the *constraint-based* option. An additional constraint usually improves the precision of the paths generated. The worst case performance of the search-based counterexample generation algorithm, however, depends on the number of constraints and the number of states in each constraint. In practice, an additional constraint, however, can occasionally improve both the precision and performance. Thus, we investigated both the query-based and constraint-based options to evaluate different tradeoffs between the precision of the counterexamples generated and the cost of the counterexample generation.

4.2.2 Select Terminating Node-Tuples

A model checker determines whether or not all paths through a system model satisfy a given property. Some model checkers such as NuSMV and FLAVERS consider only terminating paths, meaning all of the paths must end at one of the nodes of the reachability graph designated as final ones. Thus, the *only terminating* option returns true when the given node-tuple is one of the final node-tuples. Alternatively, other model checkers such as Spin and JavaPathfinder consider both terminating paths and their prefixes. Thus, the *terminating and prefixes* option returns true for every node-tuple. In general, this significantly increases the size of the set of counterexamples generated, decreases the average length of the counterexamples generated, and increases the cost of the counterexample generation. For this work, we extended the FLAVERS find path algorithm to support both isTerminating options. Since the learner basically explores the event sequences from Σ_D in an iterative deepening order, this usually means that the prefixes of the terminating paths are explored before the terminating paths themselves. Thus, we also extended the permissiveness counterexample generation method to use the IsTerminating options (line 6) to study which option works better with the learner's exploration order.

4.2.3 Order of Node-Tuples in the Worklist

The worklist can affect the set of counterexamples generated, the length of the longest counterexample, and the performance of the counterexample generation. We explored both a *breadth first search (BFS)* and *depth first search (DFS)* worklist option (line 6). The BFS worklist strategy uses a queue to explore the node-tuples in a first in first out order. The search-based counterexample generation algorithm using the BFS worklist strategy will find a shortest counterexample but often has a high cost for the counterexample generation. Alternatively, the DFS worklist uses a stack to explore the node-tuples in a last in first out order. The search-based counterexample generation algorithm using the DFS worklist option generally finds longer counterexamples but often has a lower cost for the counterexample generation.

4.2.4 Determine Previously Visited Node-Tuples

For branches, the commonly used visited data structure often prevents different counterexamples from containing the same node-tuple. Thus, we consider an alternative visited data structure that allows different counterexamples to contain the same node-tuple, which can increase the number of counterexamples generated along with the cost of the counterexample generation.

The *unique visited* option is intended to allow each distinct node-tuple to be visited at most once. This strategy hashes the set of node-tuples that have already been explored. Since the unique visited option may not allow different counterexamples to contain the same node-tuple, the *repeatedly visited* option allows certain node-tuples to be visited multiple times.

For the interface synthesis methods, the system often concurrently executes the component of interest and its environment in their own threads. Such systems are often implemented as a main thread that first allocates the component and environment threads, then forks the allocated threads, and lastly joins them. For model checkers that consider only terminating system executions, different counterexamples should be generated that pass through the node-tuples corresponding to the forks and joins before reaching the final node-tuple. Thus, the *repeatably visited* option allows the concurrency-related node-tuples (i.e. a thread being forked, begun, ended, and joined) to be revisited and the remaining node-tuples to be visited at most once.

4.2.5 Take N Steps Further

Alur et al.'s learning-based interface synthesis method [4] incorporates a *one step further* heuristic to essentially unroll any loops another time to try to augment the set of permissiveness counterexamples found. At a high-level, this heuristic first finds a set of original permissiveness counterexamples. The heuristic then generates a set of new counterexamples by trying to extend each original permissiveness counterexample by "one step further", meaning by each event in the alphabet of the derived requirement. The set of original permissiveness counterexamples is then augmented with each new counterexample that is identified as a permissiveness counterexample. We explored two different take N steps further options that will be described below. Our experimental evaluation considers taking N steps further where N is 0, 1, or 2. In general, a larger N increases the size of the set of counterexamples generated, the average length of the counterexamples generated, and the cost of the counterexample generation.

Alur et al. investigated a *learner-based N steps further* option. This option (line 8) uses the learner's observation table, specifically the set of prefixes, and the alphabet Σ_U to generate the set of new counterexamples as described in the previous paragraph. In the Approach section, we described how the original permissiveness counterexample generation method uses the teacher's membership query to identify whether or not the potential permissiveness counterexamples are permissiveness ones (shown in Figure 3.8 at line 10). In a similar manner, the *learner-based N steps further* option then uses the membership query to identify whether or not the new counterexamples are permissiveness counterexamples.

In a complementary way, the adaptive model checking technique [40] essentially uses iterative deepening DFS to generate counterexamples. The iterative deepening bound, however, must be provided by the analyst. For this work, we also investigated both *BFS N steps further* and *DFS N steps further* options that conceptually first use either BFS or DFS to generate the set of original permissiveness counterexamples and then automatically compute the bound as described below. Both options then switch to using an iterative deepening search with that bound to try to generate the new permissiveness counterexamples.

For the learning-based interface synthesis methods, the bound can be automatically computed as follows. We first run the counterexample generation algorithm to iterate through the set of original permissiveness counterexamples found and then determine the maximum depth of any counterexample. (The depth of a counterexample is the number of node-tuples that correspond to an event in the alphabet of the derived requirement. Both iterative deepening searches annotate every node-tuple with a depth.) We then set the bound to that maximum depth plus N (actually $2N$ for FLAVERS, since a human participant's interaction with an automated component needs to pair a channel send and receive) and again run the counterexample generation algorithm to iterate through the set of new permissive counterexamples found.

CHAPTER 5
DERIVATION EVALUATION

To evaluate such a requirement derivation approach, we extended our requirement derivation toolset to support both the direct and learning-based requirement derivers described in Section 3. For both requirement derivers considered, we incorporated the model checking optimizations described in Section 4.1. For the learning-based requirement deriver, we also incorporated the learning optimizations described in Section 3 and Section 4.1 along with the the permissiveness counterexample generation heuristics described in Section 4.2. We then applied the extended toolset to case studies in two human-intensive domains: healthcare and the election administration. One goal is to compare the direct requirement deriver with the learning-based requirement deriver in terms of permissiveness as well as performance. For the learning-based requirement deriver, another goal is to recommend a combination of the optimizations and one or more permissiveness counterexample generation heuristics that produce derived requirements that are adequately permissive while reducing the space and time needed to perform the requirement derivation. In what follows, we first describe the experimental methodology and then present the experimental results. Additionally, we present a preliminary evaluation of a learning-based requirement deriver that employs the L* learning algorithm and the Java Pathfinder model checker. We extended this requirement deriver to incorporate the *ensure system requirement is satisfied* options, the *worklist* options, and the *take N steps further* options. In conclusion, we discuss our experimental results including possible threats to validity.

5.1 Experimental Methodology

To evaluate our requirement derivation approach, we applied our requirement derivation toolset to a set of case studies. For each case study, we applied the direct and learning-based requirement derivers from both the component and process perspectives. Table 5.1 enumerates the five optimizations applied. In this table, each optimization shows the option used in bold. On our experimental platform, the larger subjects caused the unoptimized requirement derivers to either exceed all available space or to run for multiple days. In the previous chapter, we discussed the model checking and learning optimizations incorporated into the requirement derivers that significantly improved performance in terms of space and time. In the next section, we discuss how we selected the combination of optimization options to be used. In the previous chapter, we also described how to extend our permissiveness counterexample generation method along five dimensions to define a class of heuristics. Each heuristic is a combination of an option for each dimension. Table 5.2 shows the heuristics applied. The direct requirement deriver was run with all 3 model checker optimizations applied. The learning-based requirement deriver was run with all 3 model checker and 2 learner optimizations applied and each of the 80 different heuristics. In what follows, we identify a particular heuristic using the following notation: Learner(IsSystemReqSatisfied,Worklist,Visited,IsTerminating,N). For instance, our preliminary evaluation of the learning-based requirement deriver only used the Learner(Query,BFS,Unique,OnlyTerm,0) heuristic. As mentioned in the Approach chapter, the derived requirements are represented as minimal deterministic FSAs. We automate the comparison between the direct and learning-based requirement derivers using the evaluation metrics described in Section 5.1.1 that quantify the permissiveness of the derived requirements and the performance of the requirement deriver. We also automatically rank the heuristics using the same evaluation metrics.

The case studies are:

Table 5.1: Derivation Optimizations Applied

OPTIMIZATION	TYPE	OPTIONS
Variable modeling alternative	Model checker	Analysis-time, **Model-construction-time**
Reset dead local variables	Model checker	FALSE, **TRUE**
Channel modeling alternative	Model checker	Low-level, **High-level**
Prefix-closed membership query	Learner	FALSE, **TRUE**
Incremental membership query	Learner	FALSE, **TRUE**

- a computerized provider order entry (CPOE) application used in a healthcare facility to manage any orders for the treatment of their patients during a simplified patient evaluation process: an initial version of the CPOE and an extended version of the CPOE that supports a new access data procedure (CPOE-AD)
- a "smart" infusion pump (pump) used to administer medications and fluids during an in-patient surgery process based on usage scenarios described by Avrunin et al. [7]: an initial version of the pump (Pump) and an extended version of the pump that supports a new start procedure (Pump-Start)
- an implantable cardioverter-defibrillator (ICD) [43, 44] used during a cardiac patient care process [32]: an initial version of the ICD and an extended version of the ICD that supports a new write data procedure (ICD-WD)
- an optical scanner (scanner) used for ballot and vote counting during the California Marin County election process [31]
- a direct recording electronic (DRE) application used for ballot marking and counting [60] during the California Yolo County election process [64]

Table 5.2: Derivation Heuristics Applied

DIMENSION	TYPE	OPTIONS
Ensure system requirement is satisfied	Model checker	Query, Constraint
Worklist data structure	Model checker	BFS, DFS
Visited data structure	Model checker	Unique, Repeatable
Is terminating function	Model checker	OnlyTerm, TermAndPre
Take N steps further	Model checker or learner	N is 0, 1, 2

For the three case studies in the healthcare domain, we developed a preliminary version of these case studies and then a scaled up version of the case study by increasing the number of steps and procedures. For the two case studies in the election administration domain, we developed a single version of the case study. In Chapter 2, we described that a HIS subject consists of a system requirement specified as a property (e.g., "never overdose"), a HIS model, and a requirement derivation perspective (either component or process). A HIS model is a composition of a selected component model (e.g., pump) with a process model in which that component will be used (e.g., in-patient surgery process). Table 5.3 summarizes the subjects used in the experiments. In the table, each row gives the name of the human-intensive system, the perspective of the requirement deriver, the number of steps declared and referenced in the system, the number of component procedures, and the number of system requirements (represented as properties) that we used for the requirement deriver. In some cases, a system requirement does not contain enough details about the component and therefore from the process perspective it is not worthwhile to apply the requirement derivers to that system requirement. This is the case for the Pump and PumpS systems. In other cases, a system requirement is already satisfied by the

Table 5.3: Subjects Used in Experiments

SYSTEM	REQUIREMENT DERIVER PERSPECTIVE	# of SYSTEM STEPS	# of COMPONENT PROCEDURES	# of SYSTEM REQ.S
CPOE	Component	33	2	1
CPOE	Process	34	2	1
CPOEAD	Component	44	3	2
CPOEAD	Process	44	3	2
Pump	Component	67	2	2
Pump	Process	54	2	1
PumpS	Component	73	3	2
PumpS	Process	62	3	1
ICD	Component	68	4	2
ICD	Process	96	4	1
ICDWD	Component	79	5	2
ICDWD	Process	96	5	1
Scanner	Component	82	6	2
Scanner	Process	174	6	1
DRE	Component	155	6	1
DRE	Process	223	6	1

process model and therefore a derived component requirement is not needed. This is the case for the ICD, ICDWD, and Scanner systems. Thus, we have a total of 23 subjects. The systems range in size from 33 to 174 steps and have between 2 and 6 procedures.

For each of the 23 subjects, we ran the direct requirement deriver once. We also ran the learning-based requirement deriver with each of the 80 heuristics once. This produced 81 requirement deriver results. As described below, we automatically ranked all of the requirement derivation results by first sorting by the permissiveness of the derived requirements and then sorting by the performance of the requirement derivers.

The requirement derivation toolset is implemented in Java. All experimental runs were on two PCs with dual 3.3 GHz processors and 8 GB of memory, using version 4.4.8 of the Fedora 23 Linux kernel. The toolset was run using OpenJDK version

1.8.0_91. We timed each experimental run with the Linux “time” command. Each experimental run bounded the Java maximum memory at 2 GB and the time at 8 hours.

5.1.1 Evaluation Metrics

One criterion for evaluation of the requirement derivation approach is certainly the performance of the tool: how much space and time are used to derive a requirement. For each experimental run, we recorded the maximum number of node-tuples generated by any single stage of the requirement deriver and the sum of all node-tuples generated during all stages. In more detail, the direct requirement deriver shown in Figure 3.3 consists of the following four main stages: pre-requisite check, composition, refinement, FSA extraction. On the other hand, the learning-based requirement deriver shown in Figure 3.4 consists of the following four stages: pre-requisite check, initialization, conjecture, counterexample-based refinement. This requirement deriver, however, iteratively performs the last two stages and therefore those stages are often repeated multiple times. Additionally, we recorded the overall run-time. As we have discussed however, different combinations of optimizations and permissiveness counterexample generation heuristics can produce quite different derived requirements. So our evaluation must also take the actual derived requirement into account.

There are a number of ways the quality or utility of a derived requirement might be evaluated. In addition to permissiveness, we might be interested in the complexity and understandability of the requirement to measure how useful it will be to component developers or testers. It might be the case that handling some unusual or unimportant corner cases is extremely complex or would require costly hardware support, so that a requirement that allows the component to ignore those cases would be valuable. But such characteristics would be very hard to define and measure objectively. For this evaluation, therefore, we chose to focus solely on the permissiveness of the derived requirements.

But even measuring the permissiveness presents some significant obstacles. As noted earlier, we generally cannot determine the most permissive requirement, so we can only compare the permissiveness of the requirements generated for a particular experimental subject. For this, we begin by using regular language containment—if the language of one derived requirement properly contains the language of another, the first requirement is clearly more permissive. The problem with this is that it only gives a partial order, and we have no good way to compare the permissiveness of two requirements when neither language is contained in the other. We therefore adopted two additional measures as proxies for permissiveness when language containment is not adequate. The first is a measure of the branch coverage of the system model provided by the language of the requirement. The intuition behind this is that a requirement with greater branch coverage is likely to allow more system behaviors. The second is a measure of the size of the minimal deterministic FSA representing the requirement, based on the idea that extra states and transitions are used in the FSA to more finely distinguish between safe and unsafe system behaviors. Both of these measures give total pre-orders.

To evaluate the permissiveness, we start by computing the subset lattice of the languages of the requirements derived for a particular subject. We then assign each requirement a containment permissiveness rank by computing its level in the containment lattice, where the lattice's greatest elements are at level 1. In practice, this metric may be a total ordering or a partial ordering with a single greatest element. If so, then the greatest element is considered the "most" permissive derived requirement. If not, then the other two permissiveness evaluation metrics are used as tie breakers.

The *coverage permissiveness evaluation metric* is a total pre-order that compares the branch coverage value of each derived requirement, essentially meaning the percentage of the reachability graph edges that are permitted by the derived requirement. This metric gives precedence to derived requirements that provide the most coverage. The metric is a coarse approximation of the permissiveness of the derived requirements. We describe how the branch coverage value is computed below.

For FLAVERS, the branch coverage value is the percentage of TFG edges with source nodes that are associated with one or more tuples. To compute the branch coverage value, we first create a new subject that contains the old TFG, the old property, and a new set of constraints that is the union of the old set of constraints and the derived requirement. We then use the state propagation algorithm to perform data flow analysis on the new subject. After the data flow analysis reaches a fixed point, we compute the branch coverage value by first counting the number of TFG edges with source nodes that are associated with one or more tuples and then dividing that count by the total number of TFG edges. For the set of derived requirements, the coverage permissiveness rank is then assigned by sorting their branch coverage values into descending order.

The *FSA size permissiveness evaluation metric* gives precedence to derived requirements that are larger because such requirements often contain extra states and transitions to distinguish the safe behaviors from the unsafe ones. This metric is also a total pre-order that first compares the number of states in each derived requirement that are not the violation state and then compares the number of transitions from any source state in that requirement to any destination state that is not the violation state. The metric is a heuristic that roughly estimates the permissiveness of the derived requirements.

5.1.2 Requirement Deriver Results Ranking

We initially ranked the requirement deriver results by the permissiveness of the derived requirements, using containment of the languages of these requirements to initially sort the requirements into levels. For eight of the subjects, the derived requirements were totally ordered by language containment, and for two more the ordering was not total but had a single maximal element. Within each level, we first sorted the derived requirements in that level by the FSA size metric and then sorted by the coverage metric. We then refined this ranking by performance, using first the maximum number of node-tuples generated on a single iteration and then the wall clock time. We used the maximum node-tuples generated on a single iteration as the primary measure since that determines the space required and space is a more serious limitation for the requirement derivers than time. The top ranked derivation results correspond to requirement derivers that produced the safe and adequately permissive derived requirements with the least amount of space needed.

5.2 Experimental Results

For this evaluation, we define the *best derived requirements* to be both safe and adequately permissive. In the Approach section, we described how the direct requirement deriver is guaranteed to produce the best derived requirements. We also described how our learning-based requirement deriver is only guaranteed to produce a derived requirement that is safe but not necessarily adequately permissive. For this learning-based requirement deriver, we used the permissiveness classifier described in Section 3.6 to identify each derived requirement as either "IS_ADEQUATELY_PERMISSIVE" or "MAY_BE_ADEQUATELY_PERMISSIVE." In the following, we will compare the top ranked derived requirements in terms of permissiveness with the best derived requirement to determine whether or not the permissiveness ranking is accurately identifying the best derived requirement. For the performance, we mentioned that both the direct and learning-based requirement deriver needed to apply all of the optimizations to scale up to the larger subjects. As described above, for each subject, we ranked the 81 requirement deriver results according to the permissiveness of the derived requirement and then by the performance of the requirement deriver. We first briefly describe the optimizations applied and then discuss our observations about the direct requirement deriver and the learning-based requirement deriver using the heuristics.

To justify needing to apply the optimizations for scalability, we ran experiments on all of the combinations of the optimizations where all optimizations were turned off and then one optimization was turned on. These experiments could be run to completion on the smaller subjects but exceeded either the space or time bounds for the larger subjects. To justify applying each individual optimization, we then ran experiments on all of the combinations of the optimizations where all optimizations where turned on and then one optimization was turned off. These experiments showed that each optimization was contributing to the improvement in performance. Specifically, the model checker optimizations affect the performance in terms of both space and time while the learner optimizations only affected the performance in terms of time. In the previous chapter, we also mentioned that the variable-related model checker optimizations may influence the permissiveness of the derived requirements but the remaining model checker optimization and learner optimizations do not.

In Section 3.4, we described the direct requirement deriver that at a high-level of abstraction performs the following three main stages in order: build the full node-tuple graph, refine that graph, convert the graph that may contain non-determinism to a minimal deterministic FSA. For the performance metrics, we treat each main stage as a single iteration. In theory, this requirement deriver may have poor performance because of the blow up that can occur when converting from a non-deterministic FSA to a deterministic one. In practice, the requirement deriver often has reasonable performance when this blow up does not occur. For all 23 of our subjects, the direct requirement deriver had reasonable performance in terms of the maximum number of node-tuples generated and the number of states in the non-deterministic FSA. The maximum node-tuples generated on a single iteration (in this case a single stage) ranged from 82 to 28881 while the number of non-deterministic FSA states ranged from 13 to 6779. For these subjects, the direct requirement deriver needed up to 66% more space than the learning-based requirement deriver but usually needed significantly less overall time. The derived requirements produced are guaranteed to be safe and adequately permissive. In what follows, these best derived requirements will be used to judge the accuracy of the top ranked derived requirements based on the permissiveness evaluation metrics.

On the other hand, the learning-based requirement deriver heuristics varied widely in terms of both the performance of the requirement deriver and the permissiveness of the derived requirements. On the largest subject, for example, 41% of the heuristics exceeded either the space bound of 2 GB or the time bound of 8 hours. For those heuristics that completed within the space and time bounds, the amount of space varied by almost two orders of magnitude. The fastest heuristic took about 1 minute while the slowest heuristic took a little over 5 hours. The heuristics that completed produced 2 different derived requirements whose minimal deterministic FSAs have 39 and 67 states respectively. The heuristics that used the query-based option to ensure that the system requirement is satisfied were almost always worse in terms of both the permissiveness and performance evaluation metrics than similar heuristics that used the constraint-based option.

For the largest subject, 28 of the 40 query-based heuristics exceeded either the space or time bound. For those heuristics that completed within the bounds, both the space and time differed by two orders of magnitude. The heuristics produced two different derived requirements. The largest derived requirement is adequately permissive with a coverage metric of 47% while the smallest requirement is not adequately permissive with a coverage metric of 31%. The containment, coverage, and FSA size metrics are all total orders. The top ranked derived requirement is the same as the best derived requirement as identified by both the direct requirement deriver and the permissiveness classifier. The query-based heuristics when combined with any of the following three individual options, repeatable visited option, terminating and prefixes option, or take 1 (2) steps further option, often increased the permissiveness of the derived requirements but worsened the performance of the requirement deriver. To illustrate, Table 5.4 compares the query-based, BFS heuristics combined with each of those three options. When these heuristics are combined with more than one of the three options, the trends are even more noticeable. For 21 of the 23 subjects, the top permissiveness ranking based on containment, coverage, and FSA size accurately identified the safe and adequately permissive derived requirements. For the remaining two subjects, the top permissiveness ranking identified generalizations of the adequately permissive derived requirements that allow events sequences from $\neg\mathcal{L}(S) \cap \mathcal{L}(R_S)$. To produce the adequately permissive derived requirements, the query-based option must be combined with the following other options. The search-based counterexample generation algorithm must be parameterized with the iterative deepening DFS worklist data structure that takes either 1 or 2 steps further. Additionally, this algorithm must be parameterized with the terminating and prefixes function, the repeatable visited data structure, or both. For 19 of the 23 subjects, these 6 heuristics produce the adequately permissive derived requirements.

For heuristics that used the constraint-based option for the system requirement, the repeatable visited option and *take N steps further* options where N was greater than 0 did not improve permissiveness and often worsened performance. For the largest subject, the Learner(Constraint,DFS,OnlyTerm,Unique,0) heuristic, for example, needed 20% less space than the Learner(Constraint,DFS,OnlyTerm,Repeatable,0)

Table 5.4: Comparison of Query-Based Heuristics in Terms of Permissiveness

QUERY-BASED HEURISTIC	# OF BEST DER. REQ.S
Learner(Query,BFS,Unique,OnlyTerm,0)	4 / 23 (17%)
Learner(Query,BFS,Unique,OnlyTerm,1)	4 / 23 (17%)
Learner(Query,BFS,Unique,OnlyTerm,2)	14 / 23 (61%)
Learner(Query,BFS,Repeatable,OnlyTerm,0)	15 / 23 (65%)
Learner(Query,BFS,Unique,TermAndPre,0)	16 / 23 (70%)

heuristic and took about 10 minutes unlike the Learner(Constraint,DFS,OnlyTerm,Unique,4) heuristic that exceeded the time bound of 8 hours. We therefore discuss further only the 4 heuristics that do apply the constraint-based option but do not apply either the repeatable visited option or a take 1 (2) steps further option, which are:

- Learner(Constraint,BFS,Unique,OnlyTerm,0)
- Learner(Constraint,BFS,Unique,TermAndPre,0)
- Learner(Constraint,DFS,UniqueOnlyTerm,0)
- Learner(Constraint,DFS,Unique,TermAndPre,0)

In what follows, we compare the performance of the learning-based requirement deriver using each of these 4 heuristics and the direct requirement deriver.

For all 23 of the subjects, the 5 requirement derivers found the best derived requirement (in the sense of permissiveness). The performance of these requirement derivers, in both space and time, varied. To illustrate, Figure 5.1 shows the performance of the requirement derivers in terms of the maximum space needed, in particular the maximum number of node-tuples generated on a single iteration, for the 5 largest subjects. For 22 of the 23 subjects, the Learner(Constraint,DFS,Unique,OnlyTerm,0) heuristic had the best performance in terms of the maximum space needed. On the other

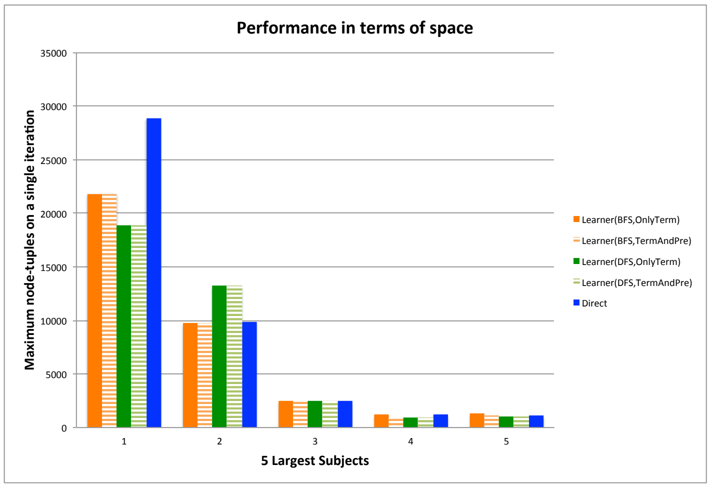


Figure 5.1: Performance of the top 5 ranked requirement derivers in terms of the maximum space on the 5 largest subjects

hand, Figure 5.2 shows the performance in terms of time, specifically the log of the overall run-time in seconds, for the same subjects. This figure shows that the direct requirement deriver had the best performance in terms of time and that the learning-based requirement deriver using the heuristics often decreased the maximum space needed by paying a time penalty. To find the requirement deriver(s) with the best performance in terms of the maximum space needed, we first searched through the requirement deriver results to find the smallest maximum number of node-tuples generated on a single iteration. For each requirement deriver, we then computed the normalized maximum number of node-tuples generated using the smallest maximum number of node-tuples generated. Each requirement deriver with a normalized maximum number of node-tuples generated less than or equal to 110% is identified as having the best performance in terms of the maximum space needed. Table 5.5 shows how often each of the 5 requirement derivers had the best performance in terms of the maximum space needed.

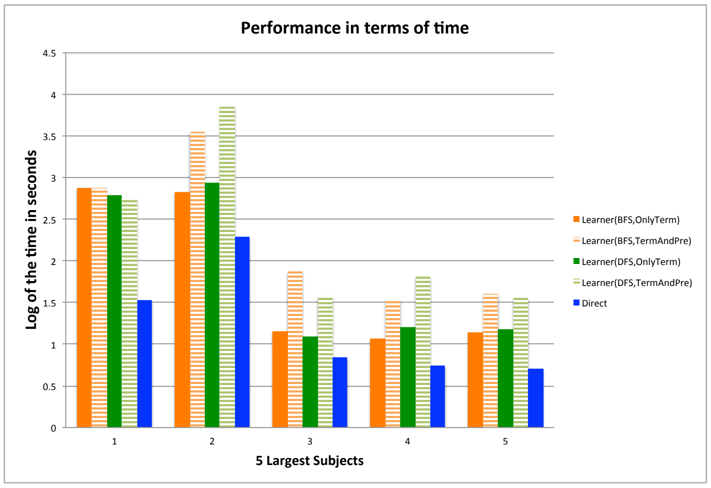


Figure 5.2: Performance of the top 5 ranked requirement derivers in terms of the log of the time on the 5 largest subjects

Table 5.5: Summary of the top 5 ranked requirement derivers in terms of the best performance with regards to maximum space needed

REQUIREMENT DERIVER	# OF SUBJECTS WITH BEST PERFORMANCE
Learner(Constraint,BFS,Unique,OnlyTerm,0)	13 / 23 (57%)
Learner(Constraint,BFS,Unique,TermAndPre,0)	14 / 23 (61%)
Learner(Constraint,DFS,Unique,OnlyTerm,0)	22 / 23 (96%)
Learner(Constraint,DFS,Unique,TermAndPre,0)	8 / 23 (35%)
Direct	9 / 23 (39%)

5.3 Discussion

Based on our experimental results, we recommend the following heuristic: Learner(Constraint,OnlyTerm,DFS,Unique,0). This recommended heuristic produced the best derived requirements and had the best performance in terms of the maximum space for 22 of the 23 subjects. For the second largest subject that did not have the best performance in terms of the maximum space, the recommended heuristic needed about 35% more space than the best performance in terms of maximum space. The direct requirement deriver is guaranteed to produce the best derived requirements.

For all of our subjects, this requirement deriver needed at most 50% more space than the learning-based requirement deriver using the recommended heuristic and needed about 50% less time than that requirement deriver.

It is interesting to note that both the DFS- and BFS-based heuristics worked well. Although further work would be needed to fully understand this, our intuition is that the constraints keep the DFS heuristics from descending too far, while the constraints and the use of the prefixes keep the BFS heuristics from spending too much time finding short counterexamples.

These experimental results appear to contradict those of Alur et al. [4], where the *take N steps further* method improved permissiveness. In fact, the *take N steps further* method did substantially improve the permissiveness of FLAVERS heuristics that use the query-based option, corresponding to the approach of [4]. But the FLAVERS heuristics using the constraint-based option were substantially better, in both performance and permissiveness, than the query-based ones.

For model checking used primarily for verification, only the existence or non-existence of a counterexample is significant. Many other applications of model checking, for such things as test generation, bug finding and understanding, and certainly interface synthesis, do depend on the characteristics of the counterexamples found. Our work demonstrates not only how a number of different aspects of the counterexample generation algorithm may affect those characteristics, but also points to the interactions among these aspects and between these aspects and standard optimizations often applied for model checking.

5.4 Preliminary Evaluation of Java Pathfinder Heuristics

Many of the learning-based requirement derivers incorporate a heuristic to approximate the adequately permissive or most permissive derived requirements (e.g., [4,35]). Specifically, this heuristic tries to generate more permissiveness counterexamples, meaning system executions that are disallowed by the current derived requirement but satisfy the overall system requirement. In the previous sections, we discussed a general class of permissiveness counterexample generation heuristics that varies along several key dimensions. Based on our experimental results for FLAVERS, it is important to carefully tailor the heuristics by selectively combining the the dimensions' options to obtain the best derivation results in terms of the permissiveness of the derived requirements and the performance of the requirement deriver. In what follows, we will describe a smaller experimental evaluation of a learning-based requirement deriver that employs JPF (Java Pathfinder model checker) and a similar class of heuristics. Our goal is to be able to begin to generalize the results about such heuristics.

In related work, Giannakopoulou and Păsăreanu developed a learning-based requirement deriver [35] that takes as input a component model written as a Java class that contains assertions and produces a derived process requirement represented as a minimal deterministic FSA that prevents that class from throwing any assertion violations. Their requirement deriver employs the L^* learning algorithm and the JPF model checker. This JPF learning-based requirement deriver produces a derived process requirement that is guaranteed to be safe but not necessarily adequately permissive. In what follows, we first briefly describe how to translate a FLAVERS subject specification, which consists of a system requirement specified as an FSA and a HIS model written in Little-JIL, to a JPF subject specification, which consists of a Java class that contains assertions. We then describe how we extended their permissiveness counterexample generation method to vary along three of our dimensions to define a general class of JPF heuristics. Lastly, we discuss a very preliminary evaluation of the JPF heuristics that compares the requirement deriver results in terms of the permissiveness of the derived requirements and the performance of the requirement deriver.

From either the process perspective or component perspective, the FLAVERS learning-based requirement deriver applies to concurrent HIS models written in Little-JIL and user-defined system requirements specified as FSAs. From the component perspective, the JPF learning-based requirement deriver applies to sequential component models written in Java. This requirement deriver applies to system requirements that employ the never E property pattern [28] that states that the event E never occurs. In particular, the requirement deriver specifies that event E occurs when any assertion violation is thrown. Because the JPF learning-based requirement deriver only applies from the component perspective, this requirement deriver can potentially be applied to the 14 FLAVERS subjects from the component perspective but not the 7 FLAVERS subjects from the process perspective. From the component perspective, each FLAVERS subject consists of a single system requirement along with a HIS model written in Little-JIL, which is composed of a precise component model and an imprecise process model. On the other hand, each JPF subject will consist of a precise component model written in Java and all of the requirements for that component model that employ the "never E" property pattern specified as assertions. Additionally, the user-defined exceptions thrown will also be specified as assertion violations.

In more detail, we manually translated the 8 component models written as Little-JIL subprocesses that throw user-defined exceptions to Java classes that throw assertion violations. In Little-JIL, the component's subprocess declares local parameters to store the internal state of that component. In the Java class, we translated each such Little-JIL local parameter with a given name and type to a Java field with the same name and type. Additionally, the Little-JIL subprocess declares the component's procedures. In the Java class, we translated each Little-JIL procedure that declares any input parameters, no output parameters, and any exceptions thrown as follows. We declared a Java method with the same input parameters. In particular, we manually inlined the methods with all possible combinations of the input parameters. For instance, the method *setLib* that takes an input parameter named *lib* with enumerated type {ICU, OR} became two methods: *setLib_ICU* and *setLib_OR*. In Section 4.1.1, we described the variable modeling alternatives where the model-construction-time option basically does such inlining of the methods. For each return statement within that method, an assertion violation was not thrown. For each throw statement within the method, an assertion violation was thrown. Thus, there are 8 JPF subjects.

In the previous chapter, we described how to extend our permissiveness counterexample generation method to vary along 5 dimensions to define a class of FLAVERS heuristics. In a similar manner, we extended Giannakopoulou and Păsăreanu's permissiveness counterexample generation method to vary along 3 of the same dimensions to define a class of JPF heuristics. In more detail, their original permissiveness counterexample generation method employed their search-based counterexample generation algorithm (e.g., [41]) parameterized with a DFS worklist data structure, a unique visited data structure, and a terminating and prefixes function. This method does not incorporate Alur's take N steps further dimension. We extended their permissiveness counterexample generation method to vary along the following three of our dimensions: is system requirement satisfied, worklist, and take N steps further. For the *is system requirement satisfied* dimension, the three options are: their constraint, our constraint, or our query. Their search-based counterexample generation algorithm provides a BFS-based worklist and two different DFS-based worklists, one uses a stack and the other uses a query prioritized by longest path. The JPF worklists are implemented slightly differently than the FLAVERS worklists but affect the ordering of the reachability nodes stored in the worklist in a similar manner. For the *take N steps further* dimension, we used our code for the learner's option but not the iterative search-based options. We set N to 0, 1, and 2. For a very preliminary evaluation, we used the 8 JPF subjects to evaluate the 27 JPF heuristics in terms of the permissiveness of the derived requirements and the performance of the requirement deriver.

For both the FLAVERS and JPF learning-based requirement derivers, we used the same experimental platform in terms of the two PCs, Linux operating system, and Linux time command. Since the JPF learning-based requirement deriver is implemented in Java 6, we needed to use version 1.6.0_65 of the Sun JDK and not version 1.8.0_91of the Open JDK. We ran each of the 27 JPF heuristics once. Each experimental run bounded the Java maximum memory at 2 GB and the time at 8 hours. In Section 5.1.1, we described the following three permissiveness evaluation metrics: containment, coverage, and FSA size. For this evaluation, we only collected the containment and FSA size. In that same section, we also described the performance evaluation metrics that quantify the space in terms of the maximum node-tuples generated on a single iteration and the sum of the node-tuples generated on all iterations along with the time in seconds. For JPF, we replaced the number of FLAVERS node-tuples generated with the number of Java instructions executed.

The FLAVERS learning-based requirement deriver was applied to concurrent systems where the procedures may have input parameters and either output parameters or exceptions thrown. On the other hand, the JPF learning-based requirement deriver was applied to sequential systems where the procedures may have input parameters but no output parameters or exceptions thrown. The procedures, however, may have assertion violations thrown. For exceptional situations, the FLAVERS system models throw exceptions, handle those exceptions, and then continue executing. On the other hand, the Java component models throw assertion violations and then stop executing. This means that the FLAVERS heuristics generally need to do substantially more work than the JPF heuristics. To illustrate, 7 of the 8 JPF subjects only slightly varied the maximum number of Java instructions on a single iteration, specifically the maximum changed by at most 2%. These same subjects, however, more widely varied the sum of the number of Java instructions on all iterations. The remaining JPF subject, which is the largest subject, ranged the maximum number of Java instructions from 16,162 to 24,598 while the sum of the Java instructions ranged from 505,585 to 3,372,595. In the previous section, we described that the FLAVERS heuristics that applied the query-based option for the *is system requirement satisfied* dimension widely varied the requirement deriver results in terms of permissiveness of the derived requirements and the performance of the requirement deriver. These FLAVERS heuristics that are query-based needed to apply particular options for the remaining dimensions to produce the best derived requirements, meaning the derived requirement are safe and adequately permissive. For all 8 of the JPF subjects, the JPF heuristics could apply any of the three options for the *is system requirement satisfied* dimension to produce the best derived requirements. For 7 of the 8 subjects, the JPF heuristics that are query-based slightly worsened the performance of the requirement deriver. For the remaining largest subject, the JPF heuristics that are query-based could significantly improve the performance of the requirement deriver. For the *work-list* dimension, the two DFS-based options generally had slightly better performance than the BFS-based option. For the *take N steps further* dimension, every increase in the N value significantly worsened performance. Based on these preliminary results, the best JPF heuristic in terms of permissiveness as well as performance is Learner(TheirConstraint,DFS-Queue,Unique,TermAndPre,0), which differs from the original JPF heuristic of Learner(TheirConstraint,DFS-Stack,Unique,TermAndPre,0).

In practice, both the FLAVERS heuristics that are constraint-based and the JPF heuristics produce the derived requirements that are safe and adequately permissive. Both the FLAVERS and JPF heuristics appeared to favor a DFS-based worklist over a BFS-based one. The FLAVERS constraint-based heuristics and the JPF heuristics did not need to apply the *take N steps further* method to improve permissiveness and applying this method significantly worsened performance. Because our learning-based requirement deriver that employs FLAVERS was applied to concurrent systems while their learning-based requirement deriver that employs Java Pathfinder was applied to sequential systems, the heuristics effect on performance was more noticeable for FLAVERS than for JPF.

5.5 Threats to Validity

For the FLAVERS and JPF evaluations, the fact that the experimental subjects were drawn from only two domains of human-intensive systems is clearly a threat to the external validity of our work. For the direct requirement deriver and learning-based requirement derivers that employ FLAVERS, the small number of systems and their overall system requirements is also clearly a threat to the external validity of this work. For the learning-based requirement deriver that employs JPF, we could only consider the subset of the subjects from the component perspective where the overall system requirement employed the "never E" property pattern. In related work, Beyer et. al [12] discuss how the component model design influences the interface synthesis methods in terms of the performance of the requirement derivers. For our subjects, all of the component models were designed to be *modal*, meaning the component enables different behaviors depending on various settings of the component. For example, a "smart" infusion pump is modal since after the pump is configured with a given drug library that pump will only allow the administration of a pre-defined set of drugs and for each drug there are pre-defined dosing limits.

For the heuristics, it is clearly important to take into account the affect of each dimension of the heuristics and the interactions among the dimensions on the permissiveness of the derived requirements as well as the performance of the requirement deriver. The extent to which our experimental results would generalize to other model checkers, however, is not entirely clear. For the *take N steps further* dimension, the FLAVERS query-based heuristic results appear to be consistent with the SMV heuristic results published by Alur et. al. For that same dimension, the FLAVERS constraint-based heuristic results and the JPF constraint-based heuristic results also seem to be consistent. For the *worklist* dimension, the FLAVERS and JPF heuristic results both seem to favor a DFS-based worklist over a BFS-based one. Based on all of the heuristic results, we expect that other model checkers (e.g., Spin) would produce similar results.

CHAPTER 6
DERIVED REQUIREMENT VIEWS

The derived requirements need to be readily understandable to various stakeholders responsible for developing, certifying, and using the component. Our preliminary investigation of the HIS-based requirement derivation approach, however, illustrated that the derived requirements represented as FSAs often became less understandable as the FSAs increased in complexity. For instance, the largest derived requirement represented as a minimal deterministic FSA has 94 states and 2538 transitions. Since the FSAs provide a very low-level representation of the interface between the selected component and the process, it can be very challenging to understand the FSAs. Thus, we built views of the derived requirements that take advantage of higher-level features such as abstraction and decomposition to decrease the cognitive load placed on the stakeholders. In what follows, we will use the the surgery derived requirement to illustrate the views even though this requirement is not particularly large. The requirement, represented as a deterministic minimal FSA, contains 12 events, 27 states, and 324 transitions. We chose this illustrative example because it is relatively easy to understand and the FSA will fit on a single page.

Each view employs a particular higher-level feature to abstract away or highlight certain aspects of the derived requirements to improve their understandability. In the Background chapter, we mentioned one simple view of the derived requirements represented as FSAs that hide the violation states and their transitions to simplify the FSAs to improve their understandability. If a state does not have an explicit transition on a given event in the alphabet, then there exists an implicit transition on that event to the violation state. We call this the *implicit violation view*. Figure 6.1 shows the implicit violation view of the surgery derived requirement represented as an FSA that contains 12 events, 26 states, and 64 transitions. This figure is to give a sense of the complexity of that FSA and is not meant to clearly show the lower-level details of the FSA. In the following sections, we describe another three views: *procedure abstraction view*, *modal abstraction view*, *safe alternatives view*. Since these four views of the derived requirements complement each other, it is beneficial to apply multiple views to the same derived requirement to further improve understandability of that requirement.

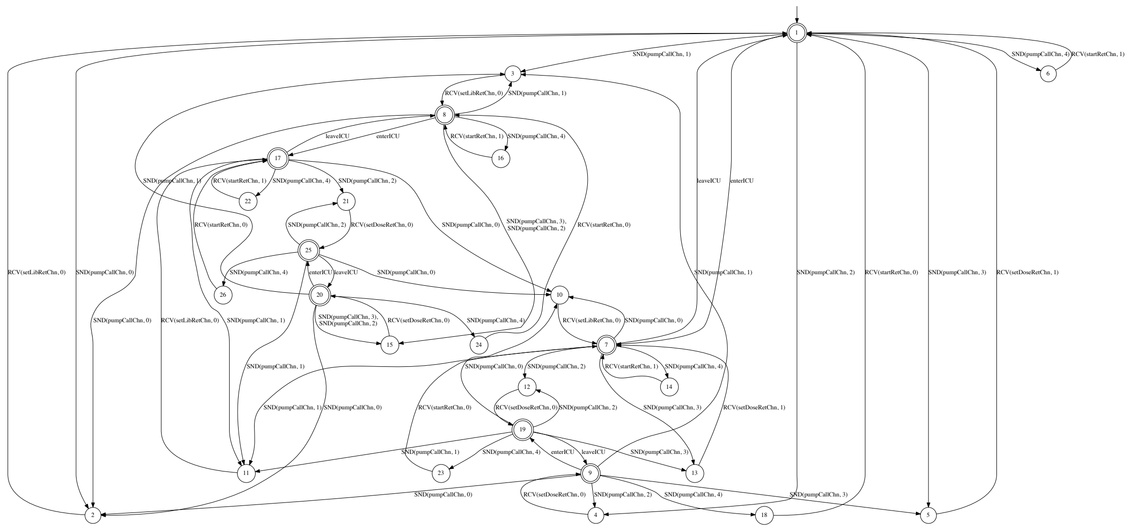


Figure 6.1: Implicit violation view of the surgery derived requirement

6.1 Procedure Abstraction View

In the Background chapter, we described in detail how each HIS model represents the use of a given procedure (e.g., setDose) by pairing a call to that procedure (e.g., **call**$(setDose, HIGH)$) with a return from the procedure (e.g., **return**$(setDose, DoseAlert)$). Additionally, there may be different calling contexts for that procedure represented as the entrance to a given calling context (e.g., $enterICU$), the procedure's use as described above, and then the exit from that calling context (e.g, *leaveICU*). The procedure abstraction view represents each procedures use as the pairing of the call to and return from that procedure surrounded by any calling context. For instance, one concrete event sequence is: *enterICU*, **call**(*setDose*, *HIGH*)), **return**(*setDose*, *DoseAlert*), *leaveICU*. This event sequence would be abstracted to: *enterICU*, setDose(HIGH)_DoseAlert, *leaveICU*.

Procedure Call-Return Event Pair	Procedure Abstraction Event
call(*setLib*, *ICU*), **return**(*setLib*, *OK*)	setLib(ICU)_OK
call(*setLib*, *OR*), **return**(*setLib*, *OK*)	setLib(OR)_OK
call(*setDose*, *LOW*), **return**(*setDose*, *OK*)	setDose(LOW)_OK
call(*setDose*, *LOW*), **return**(*setDose*, *DoseAlert*)	setDose(LOW)_DoseAlert
call(*setDose*, *HIGH*), **return**(*setDose*, *OK*)	setDose(HIGH)_OK
call(*setDose*, *HIGH*), **return**(*setDose*, *DoseAlert*)	setDose(HIGH)_DoseAlert
call(*start*), **return**(*start*, *OK*)	start()_OK
call(*start*), **return**(*start*, *DoseAlert*)	start()_DoseAlert

Table 6.1: Mapping for the surgery derived requirement from call-return event pairs to procedure abstraction events

The procedure abstraction view builder takes as input the derived requirement represented as an FSA, a set of calling context events, and a map from the procedure call and return event pairs to the procedure abstraction events. To illustrate, this builder is given the surgery derived requirement shown in Figure 6.2, the set of calling context events { enterICU, leaveICU }, and the map from the call-return event pairs to procedure abstraction events shown in Table 6.1. The builder produces the procedure abstraction view shown in Figure 2.5. At a high-level, the procedure abstraction view builder creates an output FSA with an alphabet that contains the calling context events and the procedure abstraction events. This builder conceptually uses two event sequence patterns to create the procedure abstraction view. The first event sequence pattern searches for each single transition labeled with a calling context event (e.g., enterICU). Such single transitions in the input FSA are copied to the output FSA.

The second event sequence pattern searches for pairs of call-return transitions defined as follows. The call transition must be labeled with a call event to method m (e.g., $\mathbf{call}(setDose, HIGH)$) and the return transition must be labeled with a return event from method m (e.g., $\mathbf{return}(setDose, DoseAlert)$). Additionally, the target state of the call transition must be the same as the source state of the return transition. For each call-return transition pair in the input FSA, the builder creates a new procedure abstraction transition in the output FSA. That procedure abstraction transition is from the source state of the call transition to the target state of the return transition. The procedure abstraction transition is labeled with the mapping from the pair of the call and return events (e.g., $\mathbf{call}(setDose, HIGH)$, $\mathbf{return}(setDose, DoseAlert)$) to its corresponding procedure abstraction event (e.g., setDose(HIGH)_DoseAlert). For this work, the derived requirements are represented as minimal deterministic FSAs. Since the procedure abstraction view builder usually modifies the inputs FSA's alphabet, states, and transitions, this builder needs to minimize the output FSA.

6.2 Modal Abstraction View

For our case studies, we observed that the HIS components are often *modal*, meaning the component enables different behaviors depending on various settings of the component. For example, a "smart" infusion pump is modal since after the pump is configured with a given drug library that pump will only allow the administration of a pre-defined set of drugs and for each drug there are pre-defined dosing limits. The pump will issue alerts if a drug is not within that set or if for a given drug the entered dosage exceeds the limits for that drug. For the *modal abstraction view*, the derived FSA is decomposed into pieces based on the modes. Each piece describes a given mode's behaviors. For instance, the pump behaviors can be decomposed based on the two modes, one for the ICU and another for the OR. Since stakeholders such as developers, certifiers, and users can now consider each piece separately, the cog-

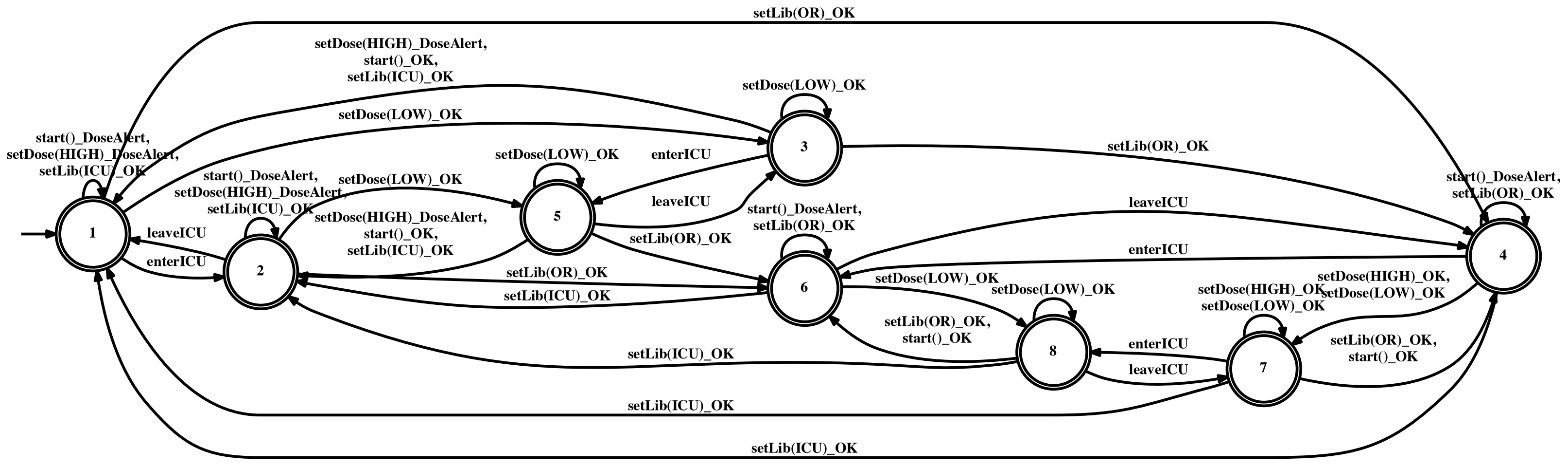
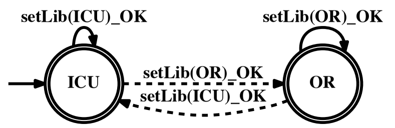

setLib(OR)_OK
setDose(HIGH)_DoseAlert, start()_OK, setLib(ICU)_OK
setDose(LOW)_OK
setDose(LOW)_OK
3
setLib(OR)_OK
start()_DoseAlert, setDose(HIGH)_DoseAlert, setLib(ICU)_OK
start()_DoseAlert, setDose(HIGH)_DoseAlert, setLib(ICU)_OK
setDose(LOW)_OK
setDose(LOW)_OK
enterICU
leaveICU
start()_DoseAlert, setLib(OR)_OK
start()_DoseAlert, setLib(OR)_OK
1
leaveICU
enterICU
2
setDose(HIGH)_DoseAlert, start()_OK, setLib(ICU)_OK
5
setLib(OR)_OK
leaveICU
enterICU
4
setLib(OR)_OK
setLib(ICU)_OK
6
setDose(LOW)_OK
setDose(LOW)_OK
setDose(HIGH)_OK, setDose(LOW)_OK
setDose(HIGH)_OK, setDose(LOW)_OK
setLib(OR)_OK, start()_OK
8
enterICU
leaveICU
7
setLib(OR)_OK, start()_OK
setLib(ICU)_OK
setLib(ICU)_OK
setLib(ICU)_OK

Figure 6.2: Procedure abstraction view of the surgery derived requirement (that was also shown in Figure 2.5)

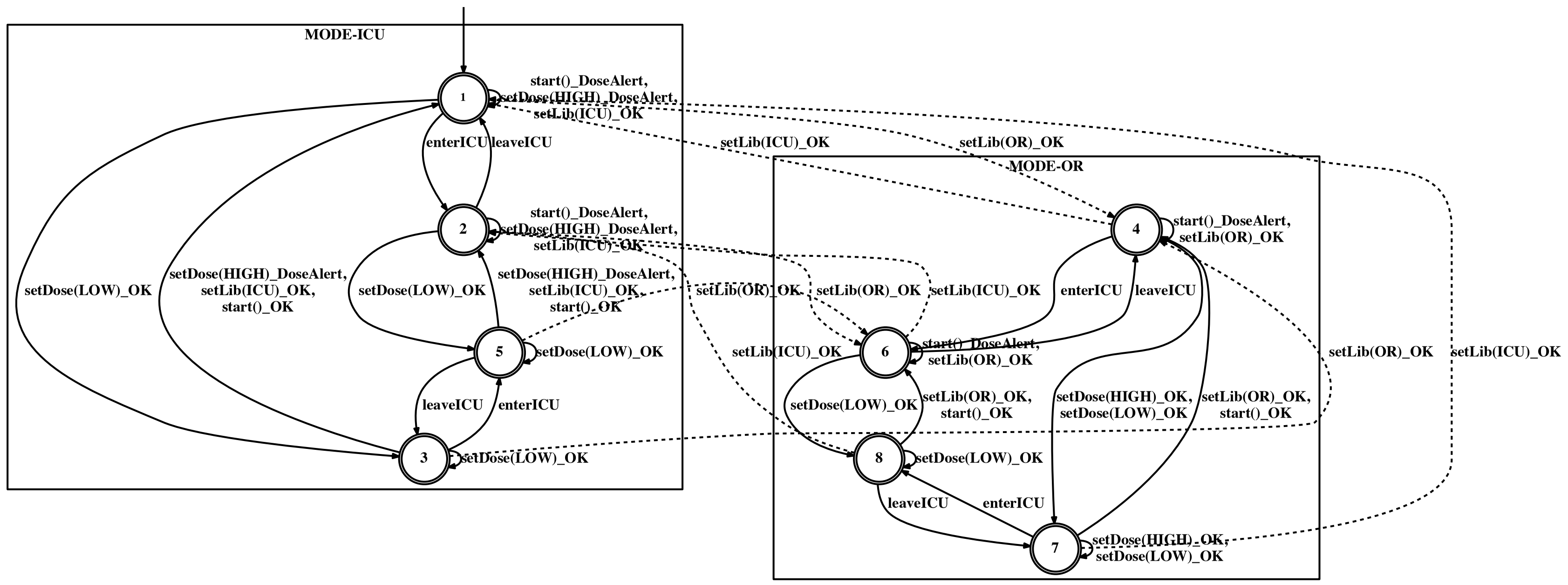
nitive load for the stakeholders should be reduced. We first demonstrate the modal abstraction view on the pump example and then describe our modal abstraction view tool that automatically creates such a view.

For the modal abstraction view, the analyst needs to specify the mode change events that correspond to the procedures that can change the mode within the component model. For the pump example, the mode change events are to configure the pump to use the ICU library (i.e. $setLib(ICU)\_OK$) or the OR library (i.e. $setLib(OR)\_OK$). Based on the mode change events, the derived FSA is hierarchically decomposed into a set of abstracted FSAs. We create one abstracted FSA to show the mode changes and one abstracted FSA for each mode to show that mode's enabled behaviors. For the motivating example, we built the modal abstraction view of the surgery derived requirement shown in Figure 6.1. Figure 6.3a shows the abstracted FSA for the mode changes. Figure 6.3b shows the abstracted FSAs for each mode, in this case the ICU mode shown on the left and the OR mode shown on the right. In these figures, the mode change transitions are shown as dashed lines.

We developed a modal abstraction view tool that takes as input a derived FSA and a mapping from each set of mode change events to its corresponding mode. The sets of mode change events must be disjoint. This tool essentially produces a mapping from every mode to a set of subgraphs for that mode. Each subgraph is represented by the set of states where the device is configured for that mode. This tool iterates through each state s in the FSA. If state s is the violation state or already contained in a subgraph, then the tool continues to the next state. When state s has an incoming transition labeled with a mode change event e, this tool looks up the corresponding mode m for event e and then creates a new subgraph g_m for mode m. First, subgraph g_m is created by finding all states reachable from state s by following outgoing transitions labeled with an event not in the set of mode change events. The tool next updates the output mapping to associate mode m with subgraph g_m. After



(a) Abstracted FSA for the mode changes in the surgery derived requirement



(b) Abstracted FSA for each mode in the surgery derived requirement

Figure 6.3: Modal abstraction view of the surgery derived requirement

iterating through all of the states, this tool does the following special processing. The tool checks whether or not the start state is associated with any mode. If not, then a special NONE mode is created and added. That mode is then associated with the subgraph reachable from the start start by following transitions labeled with events not in the set of mode change events. The modal abstraction view benefits from taking the modes into account because the mode changes can be highlighted and the larger FSA can be decomposed into smaller subgraphs.

In a complementary way, the analyst can use the mode change events to specify the different phases of the process model in which that component will be used. Such phases correspond to the various calling contexts of the procedures of the component. For the pump example, the two phases of the in-patient surgery process model in which the pump will be used are performing the operation in the OR and administering care in the ICU. Thus, the mode (or phase) change events are to move the patient and their attached pump in to the ICU (i.e. *enterICU*) or out from the ICU (i.e. *leaveICU*). Based on the phase change events, the FSA will then be decomposed into one abstracted FSA to show the phase changes and one abstracted FSA for each phase to show how that phase will use the component.

6.3 Safe Alternatives View

From a *component perspective*, our HIS-based requirement derivation approach takes as input a system requirement and a HIS model composed of a model of the component of interest and a very permissive model of the process in which that component will be used. The *derived process requirement* describes the range of processes in which the component can be safely used, in other words a *safety envelope* for the component. Given a particular *process model*, the *safe alternatives view* partitions the behaviors of the derived process requirement into those behaviors that the given process model *recommends* and the other behaviors that are safe alternatives to those recommended behaviors. The safe alternatives view was inspired by the work by Yasmeen and Gunter [77] that defines a safety envelope (in their terminology a *protection envelope*). Since domain experts often want very flexible processes that do not overly constrain the alternative ways to perform their tasks, this view could be used to modify a given process model to increase its flexibility by permitting more of the safe alternatives. We first illustrate the safe alternatives view on the pump example and then describe how to automatically create such a view.

For the safe alternatives view, we highlight the recommended behaviors that exactly follow the given process model and satisfy the system requirement. For the pump example, the recommended process model may state that the nurse should set the drug library once before using the pump. Also we highlight the safe alternative behaviors that vary from the recommended process model but do not violate the system requirement. For the pump example, the nurse may vary from the recommended process model by repeatedly setting the drug library but this repetition would not harm the patient. Figure 6.4 shows the safe alternatives view of the surgery derived requirement that was shown in Figure 6.3b where the recommended behaviors are shown in green and the safe alternative behaviors are shown in orange.

Given a derived process requirement that captures the safe alternative ways to use the component of interest and a particular process model in which that component will be used, we developed a safe alternatives view tool that basically employs data flow analysis to partition the safe alternatives into the alternatives recommended by that process model and those not recommended by it. This tool runs the FLAVERS state propagation algorithm on a new FLAVERS subject that consists of the system TFG that corresponds to the HIS model composed of the component model and the given process model, the constraints that are the system constraints plus the derived process requirement, and the property is the system requirement. During this algorithm, the tool marks all states and transitions of the derived process requirement that are

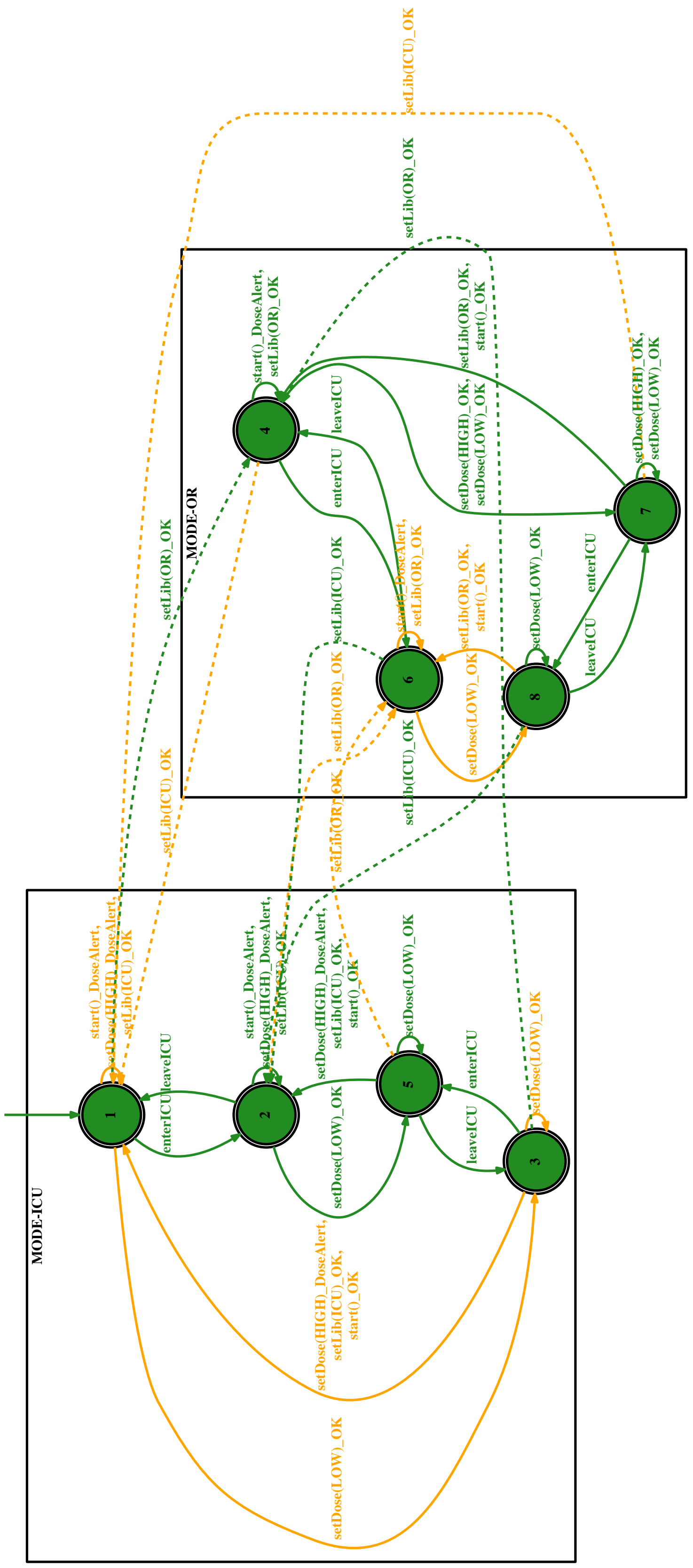


Figure 6.4: Safe alternatives view of the surgery derived requirement

visited by the process model. After the algorithm reaches a fixed point, the visited states and transitions are recommended by the process model while the unvisited ones are not recommended. Based on the visited information, this tool creates the safe alternatives view of the derived process requirement that is annotated with whether or not each state or transition is recommended.

6.4 Evaluation

In Chapter 5, we discussed our evaluation of the optimizations and heuristics that measures the usefulness of the derived requirements in terms of permissiveness. In this chapter, we discuss our evaluation of the views that measures the usefulness of the derived requirements in terms of understandability. For this evaluation, we applied the following four views in order: implicit violation, procedure abstraction, modal abstraction (based on modes or phases), and safe alternatives. We measured the complexity of each FSA using a set of simple evaluation metrics that includes the number of events, states, and transitions in that FSA. For our HIS subjects, there were 23 derived requirements represented as minimal deterministic FSAs. These FSAs were made total by adding a violation state and transitions. From the process perspective, there are 9 derived component requirements. From the component perspective, there are 14 derived process requirements. In these derived requirements, the number of events ranged from 5 to 27, the number of states ranged from 5 to 94, and the number of transitions ranged from 25 to 2,538. Table 6.4 summarizes the applicability of the 4 views to the derived requirements. For each view, we present our proposed rules for when that view is applicable, briefly describe any necessary inputs for the builder for that view, and summarize the evaluation metrics for the built views.

Both the implicit violation and procedure abstraction views could be applied to all 23 of the derived requirements. Since the implicit violation view only removes a single state but often removes many violation transitions, we measure the reduction

Table 6.2: Summary of the applicable views for the derived requirements

VIEW	PERSP.	# of SUBJECTS
Implicit violation	Both	23 / 23 (100%)
Procedure abstraction	Both	23 / 23 (100%)
Modal abstraction	Both	12 / 23 (52%)
Phase abstraction	Comp.	7 / 23 (30%)
Safe alternatives	Comp.	14 / 23 (61%)
ALL	Comp.	7 / 23 (30%)

in the complexity of a given FSA as the number of non-violation transitions divided by the number of all possible transitions. This reduction ranged from 9% to 25% with an average of 16%. Section 6.1 describes the procedure abstraction view that employs component-based development concepts. Specifically, a selected component defines a set of procedures. The process then uses (or calls) those procedures perhaps in different calling contexts. Table 6.3 enumerates the calling context events for the subjects that are from the process perspective. Since the procedure abstraction view pairs each transition labeled with a call event with its matching transition labeled with a return event, this reduces the number of states and transitions. For a given derived requirement represented as an FSA, we measure the reduction in the complexity of that FSA as the number of non-violation states in the procedure abstraction view of that derived requirement divided by the number of non-violation states in the implicit violation view of the same requirement. This reduction ranged from 12% to 67% with an average of 38%.

Section 6.2 describes how the modal abstraction view decomposes a given derived requirement based on either modes or phases. We used the following rules to decide whether or not this view is applicable to a given derived requirement represented as an FSA. If the derived requirement defines modes or phases, the view is applicable. Additionally, we checked whether or not that FSA has more than 3 non-violation states. If so, this view is applied to reduce complexity and hence increases understandability.

Table 6.3: Sets of Calling Context Events

SYSTEM	PERSP.	CALLING CONTEXT EVENTS
Pump, PumpS	Proc.	enterICU, leaveICU
ICD, ICDWD	Proc.	enterHCF, leaveHCF
Scanner, DRE	Proc.	voterArrives, voterLeaves

If not, the view is not applied because it increases complexity and therefore would decrease understandability. Since all of the components that we considered defined 2 or more modes, the modal abstraction view was applicable to 12 of the 23 derived requirements. For the applicable derived requirements, Table 6.4 shows the mapping from mode change events to mode name. The number of modes ranged from 2 to 5. Table 6.4 shows a summary of the modal abstraction views for the applicable derived requirements. In this table, the first column contains the name of the system, the second column contains the derivation perspective, the third column contains the name of the system requirement, and the fourth column contains the number of *non-modal events* (defined to be the events that are not mode change events) over the number of all events in the view. The fifth column contains the maximum number of states in any mode over the number of states in the entire view. On the other hand, the sixth column contains the number of non-modal transitions (defined to be the transitions labeled with non-modal events) over the number of all transitions in the view. The reduction in the number of states ranges from 33% to 71% with an average of 52% while the reduction in the number of transitions ranges from 48% to 96% with an average of 70%.

From the component perspective, the requirement derivers produce the derived process requirements that contain the phases. The modal abstraction view was ap-

Table 6.4: Mapping from Mode Change Events to Mode Name

SYSTEM	PERSP.	MODE MAPPING
Pump, PumpS	Both	{setLib(ICU)_OK}=ICU, {setLib(OR)_OK}=OR
ICD, ICDWD	Both	{activate()_OK}=ACTIVATED, {deactivate()_OK}=DEACTIVATED
Scanner	Comp.	{insertCard(BD)_OK}=PREPOLLING, {insertCard(STARTER)_OK}=POLLING, {insertCard(ENDER)_OK}=POSTPOLLING
Scanner	Proc.	{insertCard(STARTER)_OK}=POLLING, {insertCard(ENDER)_OK}=POSTPOLLING
DRE	Comp.	{enterAccessCode()_OK, changes()_OK}=BALLOTPAGE, castVote()_OK=REVIEWPAGE, {accept()_OK, reject()_OK}=FLAGPAGE
DRE	Proc.	{enterAccessCode()_OK, isFlagPage()_FALSE}=BALLOTPAGE, {castVote()_OK}=REVIEWPAGE, {enterAccessCode()_Alert, isFlagPage()_TRUE}=FLAGPAGE

plicable to 7 of the 9 derived process requirements. For the applicable derived requirements, Table 6.4 shows the mapping from mode (or phase) change events to phase name. The phase change events all correspond to the calling context events shown in Table 6.3. Table 6.4 shows a summary of the modal (or phase) abstraction views for the applicable derived requirements. In this table, the first column contains the name of the system, the second column contains the derivation perspective, the third column contains the name of the system requirement, and the fourth column contains the number of *non-phase events* (defined to the events that are not phase change events) over the number of all events in the view. The fifth column contains the maximum number of states in any phase over the number of states in the entire view. On the other hand, the sixth column contains the number of non-phase transitions (defined to be the transitions labeled with a non-phase event) over the number of all transitions in the view. The reduction in the number of states ranges from

Table 6.5: Summary of Modal Abstraction View Results

SYSTEM NAME	PERSP.	SYSTEM REQ.	# of NON-MODAL EVENTS	# of STATES	# of NON-MODAL TRANS.
Pump	Comp.	alwaysPumpAlerts	6 / 8 (75%)	2 / 4 (50%)	11 / 19 (58%)
PumpS	Comp.	alwaysPumpAlerts	8 / 10 (80%)	4 / 8 (50%)	30 / 46 (65%)
PumpS	Proc.	alwaysPumpAlerts	6 / 8 (75%)	3 / 7 (43%)	20 / 30 (67%)
ICD	Comp.	safeTestMode	7 / 9 (78%)	2 / 4 (50%)	12 / 18 (57%)
ICD	Proc.	neverDeactivateFails	5 / 7 (71%)	5 / 7 (71%)	10 / 15 (67%)
ICDWD	Comp.	safeTestMode	9 / 11 (88%)	2 / 4 (50%)	18 / 22 (73%)
ICDWD	Proc.	neverDeactivateFails	7 / 9 (78%)	5 / 7 (71%)	16 / 21 (76%)
Scanner	Comp.	atMostOneVote	11 / 14 (79%)	3 / 9 (33%)	25 / 52 (48%)
Scanner	Comp.	neverOvervote	13 / 16 (81%)	6 / 15 (40%)	32 / 50 (64%)
Scanner	Proc.	neverOvervote	10 / 12 (83%)	3 / 7 (43%)	43 / 45 (96%)
DRE	Comp.	atMostOneVote	23 / 28 (82%)	6 / 12 (50%)	141 / 155 (91%)
DRE	Proc.	atMostOneVote	21 / 26 (81%)	10 / 14 (71%)	47 / 64 (73%)

Table 6.6: Mapping from Phase Change Events to Phase Name

SYSTEM	PERSP.	SYSTEM REQ.	PHASE MAPPING
Pump, PumpS	Comp.	alwaysNecPumpAlerts	enterICU=insideICU, leaveICU=outsideICU
ICD, ICDWD	Comp.	safeTestMode	enterHCF=insideHCF, leaveHCF=outsideHCF
Scanner	Comp.	atMostOneVote, neverOvervotes	voterArrives=voterVoting, voterLeaves=voterNotVoting
DRE	Comp.	atMostOneVote	voterArrives=voterVoting, voterLeaves=voterNotVoting

50% to 67% with an average of 55% while the reduction in the number of transitions ranges from 78% to 92% with an average of 83%.

At a high-level, our requirement derivers from the process perspective take as input a given recommended process model and produce a derived component requirement that by construction should contain all possible safe alternatives from that process model. The safe alternatives view could be applied to the derived component requirements. This view, however, would consider all of the safe alternatives to be recommended and therefore would not improve understandability. Thus, we do not apply the safe alternatives view to any derived component requirement. On the other hand, the requirement derivers from the component perspective take as input a particular component model and produce a derived process requirement that should contain all possible safe alternatives from all possible recommended process models. Given a particular process model, the safe alternatives view can then be applied to partition all possible safe alternatives into the alternatives recommended by that process model and the other alternatives not recommended by the model. For a given system (e.g., pump), the safe alternatives view builder takes as input the system requirement from the component perspective along with a HIS model composed of the component model from the component perspective and the recommended process model from the process perspective. The HIS model consists of that system requirement, the TFG for

Table 6.7: Summary of Phase Abstraction View Results

SYSTEM NAME	PERSP.	SYSTEM REQ.	# of NON-PHASE EVENTS	# of STATES	# of NON-PHASE TRANS.
Pump	Comp.	alwaysPumpAlerts	6 / 8 (75%)	2 / 4 (50%)	15 / 19 (79%)
PumpS	Comp.	alwaysPumpAlerts	8 / 10 (80%)	4 / 8 (50%)	38 / 46 (83%)
ICD	Comp.	safeTestMode	7 / 9 (78%)	2 / 4 (50%)	14 / 18 (78%)
ICDWD	Comp.	safeTestMode	9 / 11 (82%)	2 / 4 (50%)	18 / 22 (82%)
Scanner	Comp.	atMostOneVote	12 / 14 (86%)	6 / 9 (67%)	43 / 52 (83%)
Scanner	Comp.	neverOvervote	14 / 16 (87%)	7 / 14 (50%)	43 / 49 (88%)
DRE	Comp.	atMostOneVote	26 / 28 (93%)	8 / 12 (67%)	143 / 155 (92%)

that system model, and the set of constraints for the model along with an additional constraint that is the derived process requirement. We applied the safe alternatives view to all 14 of the derived process requirements. Table 6.4 shows a summary of the safe alternatives views for the applicable derived requirements. In this table, the first column contains the name of the system, the second column contains the name of the system requirement, the third column contains the number of events in the view. The fourth column contains the number of non-violation states in the view recommended by the process model over the number of all non-violation states in the view. In a similar manner, the fifth column contains the number of non-violation transitions in the view recommended by that model over the number of all non-violation transitions in the view. For the healthcare HIS models with small to medium sizes, the reduction in the number of transitions ranged from 32% to 100% with an average of 60%. For the election administration HIS models with larger sizes, the reduction in the number of transitions ranged from 15% to 26% with an average of 21%. This illustrates that the recommended process models are reducing the complexity of the process to try to prevent system requirement violations.

Our direct and learning-based requirement derivers that employ the FLAVERS model checker produce derived requirements represented as minimal deterministic FSAs. We applied the following views in order whenever possible: implicit violation, procedure abstraction, modal abstraction, safe alternatives. In Chapter 5, we also described Giannakopoulou and Pasaraneu's learning-based requirement deriver that employs the Java pathfinder model checker to produce derived requirements represented as minimal deterministic FSAs. Their requirement deriver essentially automatically applies the implicit violation and procedure abstraction views to the derived requirements before returning those requirements. In general, these views can be applied to any derived requirements that are represented as FSAs. For this evaluation of the views, we applied each view, whenever possible, to the derived re-

Table 6.8: Summary of Safe Alternatives View Results

SYSTEM NAME	SYSTEM REQ.	# of EVENTS	# of RECOMMENDED STATES	# of RECOMMENDED TRANS.
CPOE	neverLogoutFails	3	2 / 2 (100%)	2 / 3 (67%)
CPOEAD	neverAccessDataFails	5	2 / 2 (100%)	3 / 5 (60 %)
CPOEAD	neverLogoutFails	5	2 / 2 (100%)	3 / 5 (60%)
Pump	alwaysNecPumpAlerts	8	4 / 4 (100%)	12 / 19 (63%)
Pump	noPumpAlerts	6	2 / 2 (100%)	7 / 7 (100%)
PumpS	alwaysNecPumpAlerts	10	8 / 8 (100%)	26 / 46 (57%)
PumpS	noPumpAlerts	8	4 / 4 (100%)	16 / 18 (89%)
ICD	neverDeactivateFails	7	2 / 2 (100%)	4 / 7 (57%)
ICD	safeTestMode	9	3 / 4 (75%)	6 / 18 (33%)
ICDWD	neverDeactivateFails	9	2 / 2 (100%)	5 / 9 (56%)
ICDWD	safeTestMode	11	3 / 4 (75%)	7 / 22 (32%)
Scanner	atMostOneVote	14	5 / 9 (56%)	12 / 52 (23%)
Scanner	neverOvervote	16	7 / 14 (50%)	14 / 49 (29%)
DRE	atMostOneVote	28	7 / 12 (58%)	23 / 155 (15%)

quirements and observed that each view seemed to improve their understandability. Additionally, we applied multiple views, whenever possible, which seemed to further improve their understandability and showed that the views can complement each other. In the conclusions chapter, we describe some possible directions for future work involving the views.

CHAPTER 7
RELATED WORK

In this chapter, we first briefly describe compositional verification approaches that try to improve the scalability of the verification approaches by employing a divide and conquer strategy to decompose the verification of the overall system into the individual verification of each subsystem, subject to an assumption about the behavior of the rest of the system. Some of these approaches have used automated techniques similar to ours to generate the appropriate assumptions. Second, we describe various automated synthesis methods that produce either component or process behavioral models that are represented as automata (e.g., FSAs, statecharts [45], or labeled transition systems). Lastly, we discuss other approaches that determine requirements about the human participants' interactions with components used in HISs.

7.1 Compositional Verification

Compositional verification approaches, e.g., [39, 59], employ a divide and conquer strategy to decompose the verification of the overall system into the individual verification of each subsystem to try to improve the scalability of the verification approach. Assume-guarantee reasoning techniques, e.g, [59], are one widely adopted composition verification approach. The key idea is to provide an assumption about the environments in which each subsystem is guaranteed to be used appropriately. For the assume-guarantee reasoning techniques, the simplest case is when the overall system S is decomposed into two subsystems[1]. To verify that system S, which is

[1]This can be generalized from 2 to N subsystems.

composed of subsystems S_1 and S_2, satisfies R_S, such techniques need to verify both that: 1) S_2 satisfies the assumption A, and 2) for all behaviors of S_1 that satisfy A, S_1 satisfies R_S. For our work, the HIS models are decomposed into a selected component model and a process model in which that component will be used. The derived requirements can be thought of as assumptions. From the process perspective, our approach treats the process model as S_1, the selected component model as S_2, and the derived component requirement as A. On the other hand from the component perspective, this approach treats the component model as S_1, the process model as S_2, and the derived process requirement as A. For the assume-guarantee reasoning techniques, the assumptions are often difficult to provide. Thus, many researchers developed assumption generation methods to support these techniques.

The assumption generation methods may be based on reachability analysis (e.g., [36]), counterexample-guided abstraction refinement (e.g., [33]), or a combination of a learning algorithm and model checker (e.g., [3,17,20]). Such assumption generation methods incorporate various optimizations (e.g., [3], [15]). Like the assumption generation approaches, our approach also considers a decomposition of the overall system model. We separate the overall system model into the component model and the process model in which that component will be used, and then the assumption learned is a derived requirement of the interactions between the component and process. Since the assumption generation methods are performing verification, these methods often stop learning the assumption after encountering the first execution of the system model that violates the given requirement of that system. In our case, this means that the learned assumptions are often not permissive enough to be useful, and so instead we build on the interface synthesis methods.

Conceptually, the interface synthesis methods are performing assumption generation where the component is known but the particular environment in which that component is used is usually unknown. Unlike some of the assumption generation methods, the interface synthesis algorithms do not stop when the first violation is found and thus the interfaces should be permissive enough to be useful requirements. Thus the interface synthesis methods and compositional verification approaches complement each other because the synthesized interfaces can be provided as assumptions to such verifiers.

7.2 Automated Synthesis Methods

For the automated synthesis methods, the system models define the system execution traces to be sequences of interactions between the selected component and the remaining system (most commonly represented by the procedure call/return pairs). A *positive* system execution trace describes the intended system behaviors. Thus, the synthesized behavioral models must allow the positive traces. A *negative* system execution trace, however, describes the unintended system behaviors so the synthesized behavioral models must disallow the negative scenarios. In what follows, we use the terminology from Henzinger et. al [46]. The synthesized behaviorial model is *safe* if no actual negative system execution traces are allowed. The model is *most permissive* if all potential positive system execution traces are allowed. We separate the automated synthesis methods into the *scenario-based development approaches*, the *specification mining methods*, and the *interface synthesis methods*. The scenario-based development approaches elicit, synthesize, and analyze overall system models. On the other hand, the specification mining approaches generalize from a sample set of system execution traces to produce a synthesized interface automaton that captures the most general way to use the component. Alternatively, the interface synthesis methods take as input a (partial) system model along with its requirement(s) and whenever possible produce a synthesized interface automaton that captures the most general way to use the component without violating any system requirements. In the following sections, we provide an overview of the scenario-based development approaches, the specification mining methods, and the interface synthesis methods.

7.2.1 Scenario-based Development Frameworks

The scenario-based development frameworks are centered around *scenarios* that describe either the intended or unintended behaviors of the system under development. (These scenarios are essentially the system execution traces describe above.) Each scenario is a partial system description where the system components interact among themselves often by message passing. The scenarios are typically represented as variations of message sequence charts (MSCs) [62]. The scenario-based development frameworks are often evaluated on HISs, for example HISs in the aeronautics domain or the healthcare domain. Such frameworks iteratively perform three main phases: *elicitation*, *synthesis*, and *analysis*. The elicitation phase tries to help the developer create a "complete" set of scenarios to provide to the synthesis phase. The synthesis phase takes as input a set of scenarios and produces an overall system model. The analysis phase tries to gain assurance about the synthesized system models. Next, we provide further details about the elicitation, synthesis, and analysis phases.

During the elicitation phase, various stakeholders such as customers or developers provide an initial set of scenarios based on insights gained from domain expertise. On later iterations, new scenarios may be manually provided by the stakeholders or may be automatically generated by the framework. Uchitel et al. [70] automatically generate a set of implied scenarios from a given set of scenarios. Various researchers, e.g., [2, 27, 68], automatically generate scenarios from a given set of system requirements or goals and the synthesized system model. Our proposed approach similarly employs the model checker to generate positive and negative execution traces based on the system requirement, the system model, and the current derived requirement.

In more recent work, Shokry [63] proposes using modes and mode-classes during the elicitation and synthesis phases. The modes and mode-classes will be used during the elicitation phase as a coverage metric to ensure that each mode is considered and during the synthesis phase to merge the scenarios into an overall system model that is decomposed into the modes and mode-classes. Alternatively, we post-process the derived requirements to create the modal abstraction views.

The synthesis phase takes as input a set of positive scenarios that describe the intended system behaviors where the known set of system components can be identified based on the given scenarios. This phase produces an overall system model that is a composition of the component behavioral models that allows all of the positive scenarios, e.g., [26, 55, 69, 75]. In later work, the synthesis phase was extended to take as input a set of positive scenarios and a set of *negative scenarios* and produces as output an overall system model that allows the positive scenarios but disallows the negative scenarios, e.g., [26, 71]. To make the synthesized system models more precise, the synthesis phase was further extended to not only take as input the positive and negative scenarios but additionally take as input system requirements or goals, e.g., [2, 27, 68] and then produce as output an overall system model that allows the positive scenarios, disallows the negative scenarios, and satisfies the given system requirements or goals. This last extension is the most similar to our approach. As an alternative way to make the system models more complete and accurate, Makinen and Systä [55] develop a synthesis phase that employs the L* learning algorithm and has the developer take on the role of the teacher to answer queries about the synthesized system model. In a similar manner, our learning-based requirement deriver employs a teacher that uses the FLAVERS model checker to answer the queries about the component or process model.

The analysis phase validates the synthesized system model by applying such techniques as manual reviews, simulation, and static analyses. One static analysis technique commonly employed is model checking. For the manual reviews, the customers and developers need to readily understand the synthesized system models. Systä [66] explores a scenario-based development framework that takes as input scenarios and produces an overall system model represented as a statechart. To improve the understandability of the synthesized statecharts, he develops techniques to collapse the statecharts to reduce their size and to layout the statecharts to improve their readability. Other researchers (e.g., [74]) investigate scenario-based development frameworks that take into account high-level abstractions such as class diagrams or interaction overview diagrams. Thus, the synthesized statecharts can reflect the abstractions with hierarchy, orthogonal regions, and composite states. We similarly created views of the derived requirement FSAs that employ higher-level features such as abstraction and decomposition to improve the understandability of the FSAs.

7.2.2 Specification Mining Methods

Such methods take as input a set of positive system execution traces and generalize from those traces to produce an interface automaton that summarizes all of those traces. This corresponds to our component perspective. The system execution traces are usually either collected during run-time monitoring or extracted from logs. The synthesized interface automata are commonly used for program understanding and bug finding. The specification mining methods were primarily evaluated on C and Java programs ranging from component libraries such as java.util to full applications such as a web-based email system.

These methods basically try to generalize from the sample traces by employing various techniques such as compiler analyses (e.g., [73]), passive learning algorithms (e.g., [5, 54]), and partition refinement techniques (e.g., [11]). But if the synthesized interface automaton is over-generalized then the automaton is not safe. On the other hand, if the synthesized interface automaton is under-generalized then the automaton may not be permissive enough to be useful. Since our derivation approach is being applied to HISs often in critical domains, the derived requirements must be safe so we cannot build on the specification mining methods.

The evaluations of the specification mining methods often compare the FSAs in terms of precision and recall (e.g., [53]). Such methods take as input an expected FSA and then compute the precision and recall of the synthesized FSA against that expected FSA. On the other hand, we compared the derived requirements represented as FSAs in terms of permissiveness as described in Section 5.1.1. Specifically, we quantified the permissiveness using regular language containment, code coverage, and the FSA size in terms of the number of states and transitions. To improve the usefulness of the FSAs in terms of their understandability, some of the specification mining methods annotate the FSAs' transitions with the probabilities of their occurring. These annotations can be used to identify the sequences of transitions that are least or most likely to occur. Alternatively, our safe alternatives view annotates the labels, states, and transitions with whether or not they occur in a given recommended process model.

7.2.3 Interface Synthesis Methods

These methods are solving the submodule construction problem [56] that uses a divide-and-conquer strategy to decompose an overall system into N subsystems. Given an overall system requirement and a model for each of N - 1 of the subsystems, a model for the Nth subsystem is automatically synthesized that ensures that the overall system composed of the N subsystem models satisfies the overall system requirement. Specifically, these methods are solving the submodule construction problem for N is 2 where the given subsystem is for the component and the synthesized subsystem is for the process. This corresponds to our component perspective.

Some interface synthesis methods are automated with such techniques as reachability analysis (e.g, [12]), counterexample guided abstraction refinement (e.g., [4, 46, 65]), or game theory (e.g., [4, 77]). Other methods are based on learning algorithms (e.g., [4,35,65]), as in our approach. All of the model-driven interface synthesis methods guarantee that the synthesized interface is safe. Some of them guarantee that the synthesized interface is also most permissive. In Section 4.2, we described how some of the learning-based interface synthesis methods employ permissive counterexample generation heuristics that guarantee the derived requirements are safe but not necessarily adequately permissive or most permissive. In practice, various evaluations of such methods (e.g., [12]), including our evaluation, show the derived requirements produced are both safe and either adequately permissive or most permissive. Next, we briefly describe two of the learning-based interface synthesis methods that don't employ heuristics and thus have stronger guarantees about the permissiveness of the derived requirements.

Howar, Giannakopoulou, and Rakamaric [50] explore an interface synthesis method that builds on techniques for learning along with static, dynamic, and symbolic analysis. In particular, they employ the L^* learning algorithm that interacts with a teacher that can answer both membership and equivalence queries. Their teacher employs static, dynamic, and symbolic analysis to answer the queries. The symbolic analysis must be provided with the maximum length of any event sequence generated (denoted k). Their goal is for the synthesized interface to be *k-permissive*, meaning the interface is most permissive for all event sequences up to length k. The synthesized interface is guaranteed to be safe, k-permissive, and minimal. In the worst case, the method may ask an exponential number of membership queries. This means that the provided k often needs to be relatively small.

Alternatively, Singh, Giannakopoulou and Păsăreaunu [65] investigate an interface synthesis method that builds on techniques for learning and counterexample guided abstraction refinement. For an infinite-state component, the synthesized interface is created by iteratively refining a MAY predicate abstraction of the component and a MUST predicate abstraction of the component by adding new predicates automatically generated based on counterexamples provided by the model checker. The synthesized interfaces are guaranteed to be safe, most permissive, and minimal. The interface synthesis method, however, only terminates when the infinite-state component has a finite bisimulation quotient that will be computed as the MAY predicate abstraction.

7.3 Determining Requirements about the Human Participants Interactions with Components Used in HISs

All of the approaches discussed below attempt to help the human participants appropriately use the components so that the HIS satisfies its system requirements. In the following, we describe selected approaches proposed in the area of human computer interaction (HCI) and in the area of software engineering.

In the area of HCI research, Gimblett and Thimbleby [37] explore a user interface (UI) model discovery approach. The UI model discovery approach takes into account an interactive component, consisting of a UI and usually an underlying component implementation, but does not take into account the system or component requirement(s). This approach synthesizes a component model that captures the user's knowledge about how to interact with the component. In our approach, a derived process requirement also captures how to interact with a component of interest *without violating a given system requirement*. Gimblett and Thimbleby evaluated their UI discovery approach on two case studies: an air conditioner and an infusion pump. The UI model discovery method instruments the UI to synthesize a component model represented as an FSA. First, the analyst must define the component internal state. For instance for the pump example, a component internal state needs to encode the variables that store whether the pump is on, the entered dose for the infusion, whether an internal error occurred, etc. Next, the synthesis employs a brute-force search to explore all possible component internal states. Since a component may have a large number of internal states, the synthesis often does not scale well.

To improve scalability, Gimblett and Thimbleby mention that the analyst may narrow the focus to a particular aspect of the system. For instance for the pump example, the analyst may only be interested in the keypad entry procedures. The component state should only model variables relevant to that aspect to decrease the number of variables stored. Additionally to improve scalability, they utilize user-defined abstractions to project a given concrete component state into an abstract component state to reduce the number of possible values for each variable stored. For instance, the entered dose that is concretely represented as an integer type could be abstracted as an enumeration type consisting of low and high. Our HIS-based requirement derivation approach incorporates an optimization that only models the variables relevant to the given system requirement. We also compacted the HIS models by utilizing user-defined abstractions selected based on the system requirement. One identified limitation for both the UI model discovery approach and our approach is that the synthesized FSAs are not readily understandable when the FSAs are large and complex.

Combéfis and Pecheur [21] formally define a *full-control abstraction* that basically ensures that for each abstract component state the user knows exactly what *commands* may be sent to the component and how to update the abstract component state based on any *observations* received from the component. In other words, the commands can be thought of as inputs to the component and the observations can be thought of as outputs from the component. The full-control abstraction considers all of the possible behaviors of the component and not only the behaviors relevant to a given system requirement. For our approach, the derived requirements also ensure that for each current state the user knows exactly what actions may be performed *without violating a given HIS requirement.* Combéfis and Pecheur develop a full-control abstraction synthesis method that takes as input a system model and tries to produce as output a full-control abstraction represented as a minimal deterministic automaton. The synthesis is automated with a bisimulation-based algorithm that whenever possible returns a synthesized full-control abstraction automaton and otherwise reports that no full-control abstraction exists.

In later work, Combéfis et al. [22] develop another full-control abstraction synthesis method that employs a learning algorithm and a model checker. Like our approach, the learning-based full-control abstraction synthesis approach adapts the interface synthesis method developed by Giannakopoulou and Păsăraneu [35]. If such a full-control abstraction automaton exists, then the full-control abstraction method will return one, otherwise this method returns a counterexample that illustrates how the component is not full controllable. Additionally, Combéfis et al. discuss how both versions of the full-control abstraction synthesis method can incorporate the modal concept so that the component states track the mode and the transitions change the mode when necessary. They then illustrate how the full-control abstractions can illustrate mode errors where the user believes that the component is in one mode but the component is actually is another mode. For our case studies, we also show how the derived process requirements could help prevent mode errors, e.g., the derived surgery requirement ensures a pump is configured for the appropriate care area before using the pump in that care area. Additionally, we created the modal abstraction view of the derived requirements that emphasize the modes and their behaviors to help prevent mode errors.

In the area of software engineering, C. Gunter et al. [42] explore modeling and analyzing HISs in the healthcare domain. They model the HISs in CSP (Communicating Sequential Processes). Their case study is an automated identification and data capture system employed in a hospital. C. Gunter et al. define a HIS model to be composed of the sub-processes for each human participant's recommended behaviors, e.g., nurse's recommended procedure to identify a particular patient and collect data from a medical device attached to the patient, and each component, e.g., electronic health record or wireless medical device. In contrast, our HIS model is composed of the process model that captures the recommended behaviors and the component models. Additionally, C. Gunter et al. define a *protection envelope* that captures safe deviations from the recommended behaviors. They manually create a protection envelope. Alternatively in our proposed approach, a process requirement can automatically be derived that determines the class of processes in which the component of interest may be safely used without violating a given system requirement. For instance, our automated requirement derivation approach could take as input the "never overdose" system requirement and a HIS model composed of a particular infusion pump implementation and an imprecise process model in which that pump is used. The derived process requirement would restrict the wide range of behaviors of the process to prevent violations of the "never overdose" system requirement. Thus, the requirement could be used to determine the different in-patient surgery processes that safely use the given pump. Additionally, this thesis explored the safe alternatives view that was inspired by the protection envelope to illustrate for a given process model the difference between the recommended behaviors and the safe deviations.

In subsequent work, Yasmeen and E. Gunter [77] investigate a HIS modeling and analysis framework centered around the protection envelope. They develop a synthesis method based on game theory that takes as input a HIS model written as a concurrent game structure and a protection envelope requirement specified in temporal logic and produces as output a synthesized protection envelope represented as an FSA. This method employs game theory techniques to iteratively refine the synthesized protection envelope to disallow negative counterexamples generated by a model checker. Yasmeen and E. Gunter apply this game-based synthesis method to a small case study and report on the synthesized protection envelope automaton but do not report on the synthesis performance in terms of space or time. Unlike the proposed requirement derivation approach, this synthesis approach does not incorporate optimization to reduce the size of the system model or the cost of the requirement derivation algorithm. Therefore, this game-based synthesis method does not seem as if it would scale up well.

In later work, Yasmeen [76] investigates an automated synthesis method that takes as inputs a HIS model that captures the recommended behaviors and a set of system requirements and then produces as output a synthesized protected envelope requirement specified in temporal logic. At a high-level, the method mutates the given HIS model to synthesize the protection envelope requirement. First, Hollnagell's patterns of human behavior deviations [48] are employed to propose a set of mutations to the HIS model, for instance one pattern mutates by omitting an activity and another pattern mutates by repeating an activity. Then, the synthesized protection envelope requirement is essentially the union of the set of proposed mutations that are safe, meaning the mutations prevent violations of the given system requirements. Since the patterns do not cover all possible deviations, the synthesized protection envelope requirement may be missing some safe deviations. Yasmeen implemented the synthesis method and evaluated it on one case study. For the case study, she does show the synthesized protection envelope requirement but does not provide any data about the performance of the synthesis method.

CHAPTER 8
CONCLUSIONS

Human-intensive systems are increasingly prevalent in critical domains such as healthcare. Since such systems are often complex, frequently involving concurrency, non-determinism, and exceptional situations, it is challenging to determine requirements for the components used in these systems that ensure the overall system requirements are still satisfied. For our automated requirement derivation approach, a HIS is modeled as the composition of the selected component model and the process model. The process model describes the recommended ways to achieve the system mission, including how that component should be used. This approach employs interface synthesis methods to produce derived requirements that restrict the interface between the component and process to prevent any violations of the overall system requirements. These derived requirements are represented as minimal deterministic FSAs. The derived requirements are guaranteed to be safe but may or may not be adequately permissive. For this work, we defined adequately permissive to mean that the derived requirement allows all actual system executions that always satisfy the overall system requirement.

Conceptually from the process perspective, this approach takes a process model in which a selected component will be used and produces a derived component requirement that determines the class of such components that can safely be used in the process. Such requirements can be employed to safely develop a new component, modify an existing component, or switch components. In a complementary way from the component perspective, the approach takes a selected component model and produces a derived process requirement that determines the class of processes in which the component can be safely used. These requirements can be employed to safely develop a new process or modify an existing process. In previous work, we investigated such an automated requirement derivation approach that employed an interface synthesis method based on regular language learning algorithms and model checking techniques. This learning-based requirement deriver employs the learning algorithm to iteratively refine the current derived requirement based on counterexamples generated by the model checker. The learning algorithm guarantees that the final derived requirement is represented as a minimal deterministic FSA. This algorithm also guarantees that the derived requirement is safe but not necessarily adequately permissive.

Our preliminary evaluation showed that the performance of the learning-based requirement deriver may not scale well as the HISs increase in complexity. To try to address scalability, we extended this requirement deriver to incorporate several learning and model checking optimizations. These optimizations improved the performance of the requirement deriver. Some of the model checking optimizations, however, could negatively impact the counterexample generation techniques employed by the model checker, decreasing the permissiveness of the derived requirements. We therefore investigated a class of permissiveness counterexample generation heuristics to study their impact on both the performance of the requirement deriver as well as the permissiveness of the derived requirements. Additionally, we investigated a direct interface synthesis method that employs the model checker to build the full reachability graph, refines that graph, and then converts the graph, which is often non-deterministic, to a derived requirement represented as a minimal deterministic FSA. This requirement deriver produces derived requirements that are guaranteed to be safe and adequately permissive. For the direct and learning-based requirement derivers, we created several views of the derived requirements that use higher-level features to abstract away or highlight certain aspects of the requirements to improve their understandability. We evaluated our approach on portions of real-world human-intensive systems in two important domains: healthcare and election administration.

Both the direct and learning-based requirement derivers produced derived requirements represented as minimal deterministic FSAs that provided insights about the component, the process, or both. The direct requirement deriver guarantees that that the derived requirement is safe and adequately permissive. This requirement deriver, however, may blow up converting from a non-deterministic FSA to a deterministic one and exceed all available space. For the direct requirement deriver, our evaluation showed that all of the model checking optimizations should be applied to obtain the best performance. The learning-based requirement deriver guarantees that the derived requirements are safe but not necessarily adequately permissive. The iterative learning tries to significantly reduce the maximum space needed. For the learning-based requirement deriver, our evaluation showed that each of the model checking optimizations should be applied to obtain the best performance in terms of space. Additionally, this evaluation showed that each learning optimization should be applied to obtain the best performance in terms of overall time.

For our HIS model where the component models were modal, both the direct and learning-based requirement derivers could produce the derived requirements that are safe and adequately permissive within a space bound of 2 GB and a time bound of 8 hours. For the learning-based requirement deriver, this evaluation shows that different combinations of the optimizations and permissiveness counterexample generation heuristics can significantly impact the derived requirements' permissiveness and the requirement deriver's performance. The evaluation identified a small number of combinations that seem to provide the best trade-offs in terms of permissiveness and performance. We expect that our results will generalize to different system modeling languages, requirement (or property) specification languages, and model checkers.

Many of the interface synthesis methods represent the derived requirements as minimal deterministic FSAs. We created a library of derived requirement views that use various high-level features to abstract away or highlight certain aspects of the requirements to improve their understandability. This library was shared by the direct and learning-based interface synthesis methods and could be shared by other interface synthesis methods. Additionally, the library allowed the same derived requirement to have different complementary views applied to further improve its understandability.

For a better understanding of the impact of the different interface synthesis methods, optimizations, and permissiveness counterexample generation heuristics on the derivation results in terms of permissiveness and performance, a more extensive evaluation is needed. This evaluation should consider more complex components, more detailed processes, a larger range of properties, and other domains than the two discussed here. Additionally, the evaluation should consider not only how to safely use a single component but also how to safely use multiple components together to achieve that system mission. For instance, a cardiac surgery process often needs to carefully use both a ventilator and a heart lung machine to ensure that the patient continues breathing during the process. The evaluation should also consider other quality metrics for the derived requirements, beyond their size, permissiveness, and understandability. Given the same inputs, the interface synthesis methods often produce different FSAs. Thus, it is also important to consider various FSA comparison techniques such as FSA size, code coverage, language containment, and precision/recall.

For model checking used primarily for verification, only the existence or non-existence of a counterexample is significant. Many other applications of model checking, for such things as test generation, bug finding and understanding, and certainly interface synthesis, do depend on the characteristics of the counterexamples found. Our work demonstrates not only how a number of different aspects of the counterexample generation algorithm may affect those characteristics, but also points to the interactions among these aspects and between these aspects and standard optimizations often applied for model checking. This automated requirement derivation approach employs a learning-based interface synthesis method. As mentioned in the related work chapter, there exist interface synthesis methods based on counterexample guided abstraction refinement that may benefit from the permissiveness counterexample generation heuristics.

In Chapter 6, we discussed our library of derived requirement views that includes: an implicit violation view, a procedure abstraction view, a modal abstraction view, and a safe alternatives view. For the modal abstraction view, we mentioned that the analyst must provide either modes (or phases) to decompose the FSA into subgraphs. To further investigate the modal abstraction view, the view could be extended to decompose each mode into sub-modes. The modal abstraction view currently builds on FSAs. We may be able to build on other higher-level requirement specifications such as modecharts [52], which provide support for transitions among subgraphs in addition to transitions among states. Other views may similarly benefit from converting the FSAs into a higher-level requirement specifications such as statecharts. In the background chapter, we manually annotated the states of the derived requirements with the internal state invariants of the component model. We could automatically compute such annotations by creating a state invariant view based on data flow analysis. Our evaluation of the views used simple quantitative evaluation metrics such as the FSA size in terms of the number of labels, states, and transitions. It would be beneficial to perform an evaluation of the views that uses qualitative evaluation metrics such as human judges scoring the FSAs in terms of understandability.

Although other work (e.g., [16]) has shown that the effort to define and validate human-intensive system models is challenging and time consuming, this effort is worthwhile since these models could also potentially be employed to train medical personnel, be the subject of both static and dynamic analyses, and support guidance when performing the systems in the real-world settings. Such a continuous process improvement framework can be used to gain assurance that these human-intensive systems are being safely developed and safely performed in the real-world settings.

APPENDIX A

BUILD NODE-TUPLE GRAPH ALGORITHM DESCRIPTION

The *build node-tuple graph* algorithm shown in Figure A.1 takes as input a FLAVERS subject and generates the full node-tuple graph along with the verification result. As described in the FLAVERS section, the node-tuple graph is a labeled directed graph $G = (N, E, n_{init}, F, \Sigma_G, L)$ where N is the set of node-tuples, E is the set of edges among the node-tuples, n_{init} is the unique initial node-tuple, F is the set of final node-tuples, Σ_G is the alphabet of system events, and L is a mapping from each node-tuple to either a system event or τ. The verification result is either CONCLUSIVE or INCONCLUSIVE. The *build node-tuple graph* algorithm uses a worklist to store the node-tuples to be expanded. Initially, the set of node-tuples, edges, final node-tuples, and the worklist are all empty. The algorithm first creates the initial node-tuple n_{init} and adds it to the set of node-tuples N and the worklist $Wlist$. Figure A.1 shows the pseudo-code for *build node-tuple graph* algorithm that iteratively generates the reachable node-tuples until the worklist is empty (lines 1 to 16 inclusive) and then determines the verification result by interpreting the final-node tuples (lines 17 to 29 inclusive).

On each iteration, the *build node-tuple graph* algorithm removes a current node-tuple p_{curr} from the worklist Wlist (line 2). This algorithm then iterates through the possible next TFG nodes based on the edges of p_{curr}'s node (line 3). For each next TFG node n_{next}, the algorithm algorithm then iterates through the possible next tuples produced by the tuple generation function on the node n_{next} and the tuple from p_{curr} (line 4). This tuple generation function will prune away any infeasible node-tuples and only return the feasible node-tuples. The *build node-tuple graph* algorithm then checks whether or not the next node-tuple consisting of the next TFG node n_{next} and the next tuple t_{next} is contained is the set of node-tuples N (line 5). If so, then this algorithm gets that existing next node-tuple (line 6). If not, then the algorithm creates a new next node-tuple consisting of n_{next} and t_{next} and adds that next node-tuple to the set of node-tuples N and the worklist $Wlist$ (line 8). The *build node-tuple graph* algorithm then checks whether or not the next node-tuple is a final node-tuple, meaning that the TFG node is the unique final node of the TFG (line 9). If so, then this algorithm adds the next node-tuple to the set of final node-tuples F (line 10). Finally, the algorithm creates an edge from the current node-tuple p_{curr} to the next node-tuple p_{next} and adds that edge to the set of edges E (line 13). The *build node-tuple graph* algorithm then re-checks whether or not the worklist is empty (line 1). If not, then this algorithm continues iterating. If so, then the algorithm stops iterating and determines the verification result by examining the final node-tuples.

The *build node-tuple graph* algorithm iterates through each of the final node-tuples (line 18) looking for violating node-tuples (line 19) and satisfying node-tuples (line 21). If a violating node-tuple is found, then the verification result is "INCONCLUSIVE (Counterexample path exists)" (denoted as INC-CEX at line 20). If no satisfying node-tuples are found, then the verification result is "INCONCLUSIVE (No potential terminating paths exist)" (denoted as INC-NOTUPLES at line 26) and otherwise the verification result is "CONCLUSIVE" (denoted as CON at line 28).

```
while (Wlist is not empty) do
    Remove a current node-tuple $p_{curr}$ from Wlist
    for all (next node $n_{next}$ of the node from $p_{curr}$) do
        for all (next tuple $t_{next}$ generated from $n_{next}$ and the tuple from $p_{curr}$) do
            if (there exists a node-tuple in P with $n_{next}$ and $t_{next}$) then
                Get that node-tuple $p_{next}$
            else
                Create next node-tuple $p_{next}$ and add it to N and Wlist
                if ($p_{next}$ is a final node-tuple) then
                    Add $p_{next}$ to the final node-tuples set F
                end if
            end if
            Add an edge from $p_{curr}$ to $p_{next}$ and add that edge to E
        end for
    end for
end while
$existsSatisfyingNodeTuple \leftarrow FALSE$
for all (final node-tuple $p_{final}$ in F) do
    if ($p_{final}$ is a violating node-tuple) then
        return G along with INC-CEX
    else if ($p_{final}$ is a satisfying node-tuple) then
        $existsSatisfyingNodeTuple \leftarrow TRUE$
    end if
end for
if (not $existsSatisfyingNodeTuple$) then
    return G along with INC-NOTUPLES
else
    return G along with CON
end if
```

Figure A.1: Pseudo-code for the *build node-tuple graph* algorithm

APPENDIX B

LOW-LEVEL DERIVED REQUIREMENT ALPHABET DESCRIPTION

In Section 3.2, we described how the system translator computes the derived requirement alphabet based on the overall system requirement, the requirement deriver perspective, and the HIS model. We presented the high-level derived requirement alphabet where the system events included the component procedure execution events (that pair the component procedure calls with their returns) and the calling context events. In the Little-JIL section, we discussed how the component executions are modeled as remote procedure calls over (communication) channels. Each channel stores messages of a given type. A message can then be sent to that channel or received from the channel. Specifically, each remote procedure call from the process to the select component consists of the following five system activities:

1.1) the process sends a procedure call message to the component on the procedure call channel

1.2) the component receives that procedure call message from the process on the procedure call channel

2) the component executes the procedure

3.1) the component sends a procedure return message to the process on the procedure return channel

3.2) the process receives that procedure return message from the component on the procedure return channel

Thus, the low-level derived requirement alphabet will contain the system events related to the remote procedure calls involving the channels.

In the following, we define the *channel operations* to be the sends to and receives from the channels. Before we can intersect the process alphabet that contains channel operation events with the component alphabet that also contains channel operation events, we need to “complement” the channel operation events. The complement of a send to a given channel is a receive from that channel (e.g., 1.1’s complement is 1.2). In a dual manner, the complement of a receive from a specific channel is a send to that channel (e.g., 1.4’s complement is 1.3). For the illustrative example, the system requirement alphabet is:

- $enterICU$ // Calling context for setDose
- $leaveICU$ // Calling context for setDose
- RCV(pumpCallChn, **call**($setDose, HIGH$))
- SND(setDoseRetChn, **return**($setDose, DoseAlert$))
- SND(setDoseRetChn, **return**($setDose, OK$))

Table B.1 shows the alphabet for the process (on the left) and the component (on the right). From the component perspective, the derived process requirement alphabet is shown on the left. From the process perspective, the derived component alphabet is shown on the right. As mentioned above, the derived component alphabet does not contain all of the system requirement alphabet, specifically the calling context events $enterICU$ and $leaveICU$ are missing.

Table B.1: Alphabets for the process and the component

Process Alphabet	Component Alphabet
SND(pumpCallChn, **call**($setLib, ICU$))	RCV(pumpCallChn, **call**($setLib, ICU$))
SND(pumpCallChn, **call**($setLib, OR$))	RCV(pumpCallChn, **call**($setLib, OR$))
RCV(setLibRetChn, **return**($setLib, OK$))	SND(setLibRetChn, **return**($setLib, OK$))
SND(pumpCallChn, **call**($setDose, LOW$))	RCV(pumpCallChn, **call**($setDose, LOW$))
SND(pumpCallChn, **call**($setDose, HIGH$))	RCV(pumpCallChn, **call**($setDose, HIGH$))
RCV(setDoseRetChn, **return**($setDose, OK$))	SND(setDoseRetChn, **return**($setDose, OK$))
RCV(setDoseRetChn, **return**($setDose, DoseAlert$))	SND(setDoseRetChn, **return**($setDose, DoseAlert$))
SND(pumpCallChn, **call**($start$))	RCV(pumpCallChn, **call**($start$))
RCV(startRetChn, **return**($start, OK$))	SND(startRetChn, **return**($start, OK$))
RCV(startRetChn, **return**($start, DoseAlert$))	SND(startRetChn, **return**($start, DoseAlert$))
enterICU	
leaveICU	

BIBLIOGRAPHY

[1] Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Alrajeh, Dalal, Kramer, Jeff, Russo, Alessandra, and Uchitel, Sebastin. Learning operational requirements from goal models. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 265–275.

[3] Alur, Rajeev, Madhusudan, P., and Nam, Wonhong. Symbolic compositional verification by learning assumptions. In *CAV* (2005), Kousha Etessami and Sriram K. Rajamani, Eds., vol. 3576 of *Lecture Notes in Computer Science*, Springer, pp. 548–562.

[4] Alur, Rajeev, Černý, Pavol, Madhusudan, P., and Nam, Wonhong. Synthesis of interface specifications for Java classes. *SIGPLAN Not. 40*, 1 (2005), 98–109.

[5] Ammons, Glenn, Bodík, Rastislav, and Larus, James R. Mining specifications. In *POPL* (2002), John Launchbury and John C. Mitchell, Eds., ACM, pp. 4–16.

[6] Angluin, Dana. Learning regular sets from queries and counterexamples. *Inf. Comput. 75*, 2 (1987), 87–106.

[7] Avrunin, George S., Clarke, Lori A., Henneman, Elizabeth A., and Osterweil, Leon J. Complex medical processes as context for embedded systems. *SIGBED Rev. 3*, 4 (2006), 9–14.

[8] Avrunin, George S., Clarke, Lori A., Osterweil, Leon J., Christov, Stefan C., Chen, Bin, Henneman, Elizabeth A., Henneman, Philip L., Cassells, Lucinda, and Mertens, Wilson. Experience modeling and analyzing medical processes: Umass/baystate medical safety project overview. In *Proceedings of the 1st ACM International Health Informatics Symposium* (New York, NY, USA, 2010), IHI '10, ACM, pp. 316–325.

[9] Avrunin, George S., Corbett, James C., Dwyer, Matthew B., Păsăreanu, Corina S., and Siegel, Stephen F. Comparing finite-state verification techniques for concurrent software.

[10] Berg, Therese, Jonsson, Bengt, Leucker, Martin, and Saksena, Mayank. Insights to Angluin's learning. *Electron. Notes Theor. Comput. Sci. 118* (Feb. 2005), 3–18.

[11] Beschastnikh, Ivan, Brun, Yuriy, Schneider, Sigurd, Sloan, Michael, and Ernst, Michael D. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (New York, NY, USA, 2011), ESEC/FSE '11, ACM, pp. 267–277.

[12] Beyer, Dirk, Henzinger, Thomas, and Singh, Vasu. Algorithms for interface synthesis. In *CAV* (2007), Lecture Notes in Computer Science, Springer, pp. 4–19.

[13] Cass, Aaron G., Lerner, Barbara Staudt, Sutton, Jr., Stanley M., McCall, Eric K., Wise, Alexander, and Osterweil, Leon J. Little-JIL/Juliette: a process definition language and interpreter. In *ICSE '00: Proc. of the 22nd Int. Conf. on Software Eng.* (New York, NY, USA, 2000), ACM, pp. 754–757.

[14] Chaki, Sagar, Clarke, Edmund, Sharygina, Natasha, and Sinha, Nishant. Verification of evolving software via component substitutability analysis. *Form. Methods Syst. Des. 32*, 3 (June 2008), 235–266.

[15] Chaki, Sagar, and Strichman, Ofer. Optimized l*-based assume-guarantee reasoning. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems* (Berlin, Heidelberg, 2007), TACAS'07, Springer-Verlag, pp. 276–291.

[16] Chen, Bin. Improving processes using static analysis techniques, Doctoral Thesis, Department of Computer Science, University of Massachusetts, Amherst, MA 01003, September 2010.

[17] Chen, Yu-Fang, Farzan, Azadeh, Clarke, Edmund M., Tsay, Yih-Kuen, and Wang, Bow-Yaw. Learning minimal separating dfa's for compositional verification. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,* (Berlin, Heidelberg, 2009), TACAS '09, Springer-Verlag, pp. 31–45.

[18] Cimatti, Alessandro, Clarke, Edmund M., Giunchiglia, Enrico, Giunchiglia, Fausto, Pistore, Marco, Roveri, Marco, Sebastiani, Roberto, and Tacchella, Armando. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification* (London, UK, UK, 2002), CAV '02, Springer-Verlag, pp. 359–364.

[19] Clarke, Edmund M., Grumberg, Orna, and Peled, Doron A. *Model Checking.* The MIT Press, 1999.

[20] Cobleigh, Jamieson M., Giannakopoulou, Dimitra, and Păsăreanu, Corina S. Learning assumptions for compositional verification. In *TACAS '03: Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (New York, NY, USA, 2003), vol. 2619 of *Lecture Notes in Computer Science*, Springer-Verlag Berlin Heidelberg, pp. 331–346.

[21] Combéfis, S., and Pecheur, C. A bisimulation-based approach to the analysis of human-computer interaction. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (New York, 2009), ACM.

[22] Combéfis, Sébastien, Giannakopoulou, Dimitra, Pecheur, Charles, and Feary, Michael. Learning system abstractions for human operators. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering* (New York, NY, USA, 2011), MALETS '11, ACM, pp. 3–10.

[23] Conboy, Heather M., Avrunin, George S., and Clarke, Lori A. Process-based derivation of requirements for medical devices. In *Proceedings of the 1st ACM International Health Informatics Symposium* (New York, NY, USA, 2010), IHI '10, ACM, pp. 656–665.

[24] Corbett, James C., and Avrunin, George S. Using integer programming to verify general safety and liveness properties. *Form. Methods Syst. Des. 6*, 1 (Jan. 1995), 97–123.

[25] Corbett, James C., Dwyer, Matthew B., Hatcliff, John, Laubach, Shawn, Păsăreanu, Corina S., Robby, and Zheng, Hongjun. Bandera: Extracting finite-state models from Java source code. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering* (New York, NY, USA, 2000), ACM Press, pp. 439–448.

[26] Damas, Christophe, Lambeau, Bernard, Dupont, Pierre, and van Lamsweerde, Axel. Generating annotated behavior models from end-user scenarios. *IEEE Trans. Softw. Eng. 31*, 12 (Dec. 2005), 1056–1073.

[27] Damas, Christophe, Lambeau, Bernard, and van Lamsweerde, Axel. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2006), SIGSOFT '06/FSE-14, ACM, pp. 197–207.

[28] Dwyer, Matthew B., Avrunin, George S., and Corbett, James C. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering* (Los Alamitos, CA, USA, 1999), IEEE Computer Society Press, pp. 411–420.

[29] Dwyer, Matthew B., Clarke, Lori A., Cobleigh, Jamieson M., and Naumovich, Gleb. Flow analysis for verifying properties of concurrent software systems. *ACM Transactions on Software Engineering and Methodology (TOSEM) 13*, 4 (2004), 359–430.

[30] Dwyer, Matthew B., Person, Suzette, and Elbaum, Sebastian. Controlling factors in evaluating path-sensitive error detection techniques. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2006), SIGSOFT '06/FSE-14, ACM, pp. 92–104.

[31] election office, Marin County. Marin county pollworker guides, http://www.co.marin.ca.us/depts/rv/main/pollworkers/guides.html.

[32] Fu, Kevin. Research notes about implantable medical devices, 2006.

[33] Gheorghiu, Mihaela, Giannakopoulou, Dimitra, and Păsăreanu, Corina S. Refining interface alphabets for compositional verification. In *TACAS* (2007), Orna Grumberg and Michael Huth, Eds., vol. 4424 of *Lecture Notes in Computer Science*, Springer, pp. 292–307.

[34] Giannakopoulou, Dimitra. Personal correspondence, 2016.

[35] Giannakopoulou, Dimitra, and Păsăreanu, Corina S. Interface generation and compositional verification in JavaPathfinder. In *FASE '09: Proc. of the 12th Int. Conf. on Fundamental Approaches to Software Eng.* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 94–108.

[36] Giannakopoulou, Dimitra, Păsăreanu, Corina S., and Barringer, Howard. Assumption generation for software component verification. In *ASE '02: Proceedings of the 17th IEEE International Conference on Automated Software Engineering* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 3–12.

[37] Gimblett, Andy, and Thimbleby, Harold. User interface model discovery: towards a generic approach. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems* (New York, NY, USA, 2010), EICS '10, ACM, pp. 145–154.

[38] Godefroid, Patrice, and Wolper, Pierre. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Form. Methods Syst. Des. 2*, 2 (Apr. 1993), 149–164.

[39] Graf, Susanne, and Steffen, Bernhard. Compositional minimization of finite state systems. In *IN PROC. 2ND INTERNATIONAL CONFERENCE OF COMPUTER-AIDED VERIFICATION* (1991), pp. 186–196.

[40] Groce, Alex, Peled, Doron, and A, Mihalis Yannakakis. Adaptive model checking. In *TACAS '02: Proceedings of the Eighth International Conference on Tools and Algorithms for Construction and Analysis of Systems* (2002), vol. 2280 of *Lecture Notes in Computer Science*, Springer-Verlag Berlin Heidelberg, pp. 357–370.

[41] Groce, Alex, and Visser, Willem. Model checking java programs using structural heuristics. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2002), ISSTA '02, ACM, pp. 12–21.

[42] Gunter, Elsa L., Yasmeen, Ayesha, Gunter, Carl A., and Nguyen, Anh. Specifying and analyzing workflows for automated identification and data capture. In *HICSS '09* (2009), pp. 1–11.

[43] Halperin, Daniel, Heydt-Benjamin, Thomas S., Fu, Kevin, Kohno, Tadayoshi, and Maisel, William H. Security and privacy for implantable medical devices. *IEEE Pervasive Computing 7*, 1 (2008), 30–39.

[44] Halperin, Daniel, Heydt-Benjamin, Thomas S., Ransford, Benjamin, Clark, Shane S., Defend, Benessa, Morgan, Will, Fu, Kevin, Kohno, Tadayoshi, and Maisel, William H. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 129–142.

[45] Harel, David. Statecharts: A visual formalism for complex systems, 1987.

[46] Henzinger, Thomas A., Jhala, Ranjit, and Majumdar, Rupak. Permissive interfaces. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2005), ACM, pp. 31–40.

[47] Holley, L. Howard, and Rosen, Barry K. Qualified data flow problems. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages* (New York, NY, USA, 1980), POPL '80, ACM, pp. 68–82.

[48] Hollnagel, Erik. The phenotype of erroneous actions. *Int. J. Man-Mach. Stud. 39*, 1 (July 1993), 1–32.

[49] Holzmann, Gerard J. The model checker SPIN. *IEEE Transactions on Software Engineering 23* (1997), 279–295.

[50] Howar, Falk, Giannakopoulou, Dimitra, and Rakamarić, Zvonimir. Hybrid learning: Interface generation through static, dynamic, and symbolic analysis. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2013), ISSTA 2013, ACM, pp. 268–279.

[51] Hungar, Hardi, Niese, Oliver, and Steffen, Bernhard. Domain-specific optimization in automata learning. In *CAV* (2003), Warren A. Hunt Jr. and Fabio Somenzi, Eds., vol. 2725 of *Lecture Notes in Computer Science*, Springer, pp. 315–327.

[52] Jahanian, Farnam, and Mok, Aloysius K. Modechart: A specification language for real-time systems. *IEEE Transactions Software Engineering 20*, 12 (Dec. 1994), 933–947.

[53] Krka, Ivo, Brun, Yuriy, and Medvidovic, Nenad. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, ACM, pp. 178–189.

[54] Lo, David, and Khoo, Siau-Cheng. Smartic: towards building an accurate, robust and scalable specification miner. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2006), SIGSOFT '06/FSE-14, ACM, pp. 265–275.

[55] Mäkinen, Erkki, and Systä, Tarja. MAS : an interactive synthesizer to support behavioral modelling in UML. In *Proceedings of the 23rd International Conference on Software Engineering* (Washington, DC, USA, 2001), ICSE '01, IEEE Computer Society, pp. 15–24.

[56] Merlin, Philip, and Bochmann, Gregor V. On the construction of submodule specifications and communication protocols. *ACM Trans. Program. Lang. Syst. 5*, 1 (1983), 1–25.

[57] Naumovich, Gleb, and Avrunin, George S. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. *SIGSOFT Softw. Eng. Notes 23*, 6 (Nov. 1998), 24–34.

[58] Naumovich, Gleb, Avrunin, George S., and Clarke, Lori A. Data flow analysis for checking properties of concurrent Java programs. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (New York, NY, USA, 1999), ACM, pp. 399–410.

[59] Pnueli, Amir. In transition from global to modular temporal reasoning about programs. In *Logic and Models of Concurrent Systems* (New York, NY, USA, 1984), K. Apt, Ed., vol. 13, Springer-Verlag, pp. 123–144.

[60] Proebstel, Elliot, Riddle, Sean, Hsu, Francis, Cummins, Justin, Oakley, Freddie, Stanionis, Tom, and Bishop, Matt. An analysis of the Hart Intercivic DAU eslate. In *EVT '07: Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop* (2007), USENIX Press.

[61] Rivest, R. L., and Schapire, R. E. Inference of finite automata using homing sequences. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing* (New York, NY, USA, 1989), ACM, pp. 411–420.

[62] Sector, ITU Telecommunication Standardization. ITU-T recommendation Z.120 - message sequence charts (MSC'96), May 1996.

[63] Shokry, Hesham. Towards behavior elaboration and synthesis using modes. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2010), FSE '10, ACM, pp. 349–352.

[64] Simidchieva, Borislava I., Marzilli, Matthew S., Clarke, Lori A., and Osterweil, Leon J. Specifying and verifying requirements for election processes. In *dg.o '08: Proceedings of the 2008 international conference on Digital government research* (2008), Digital Government Society of North America, pp. 63–72.

[65] Singh, Rishabh, Giannakopoulou, Dimitra, and Păsăreanu, Corina. Learning component interfaces with may and must abstractions. In *Proceedings of the 22nd international conference on Computer Aided Verification* (Berlin, Heidelberg, 2010), CAV'10, Springer-Verlag, pp. 527–542.

[66] Systä, Tarja. Static and dynamic reverse engineering techniques for Java, PhD thesis, Dept. of Computer and Information Sciences, University of Tampere.

[67] Tan, Jianbin, Avrunin, George S., Clarke, Lori A., Zilberstein, Shlomo, and Leue, Stefan. Heuristic-guided counterexample search in flavers. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2004), SIGSOFT '04/FSE-12, ACM, pp. 201–210.

[68] Uchitel, Sebastian, Brunet, Greg, and Chechik, Marsha. Behaviour model synthesis from properties and scenarios. In *Proceedings of the 29th international conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 34–43.

[69] Uchitel, Sebastian, and Kramer, Jeff. A workbench for synthesising behaviour models from scenarios. In *Proceedings of the 23rd International Conference on Software Engineering* (Washington, DC, USA, 2001), ICSE '01, IEEE Computer Society, pp. 188–197.

[70] Uchitel, Sebastian, Kramer, Jeff, and Magee, Jeff. Detecting implied scenarios in message sequence chart specifications. In *In ACM Proceedings of the joint 8th ESEC and 9th FSE* (2001), ACM Press, pp. 74–82.

[71] Uchitel, Sebastian, Kramer, Jeff, and Magee, Jeff. Negative scenarios for implied scenario elicitation. *SIGSOFT Softw. Eng. Notes 27*, 6 (Nov. 2002), 109–118.

[72] Weiser, Mark. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (Piscataway, NJ, USA, 1981), ICSE '81, IEEE Press, pp. 439–449.

[73] Whaley, John, Martin, Michael C., and Lam, Monica S. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 2002), ISSTA '02, ACM, pp. 218–228.

[74] Whittle, Jon, and Jayaraman, Praveen K. Synthesizing hierarchical state machines from expressive scenario descriptions. *ACM Trans. Softw. Eng. Methodol. 19*, 3 (Feb. 2010), 8:1–8:45.

[75] Whittle, Jon, and Schumann, Johann. Generating statechart designs from scenarios. In *Proceedings of the 22nd international conference on Software engineering* (New York, NY, USA, 2000), ICSE '00, ACM, pp. 314–323.

[76] Yasmeen, Ayesha. Formalizing operator task analysis, Doctoral Thesis, Department of Computer Science, University of Illinois at Urbana-Campaign, Urbana, IL, June 2011.

[77] Yasmeen, Ayesha, and Gunter, Elsa L. Automated framework for formal operator task analysis. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, ACM, pp. 78–88.