2016

An Incremental Approach to Identifying Causes of System Failures using Fault Tree Analysis

Huong T. Phan
University of Massachusetts

AN INCREMENTAL APPROACH TO IDENTIFYING CAUSES OF SYSTEM FAILURES USING FAULT TREE ANALYSIS

A Dissertation Presented

by

HUONG PHAN

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 2016

College of Information and Computer Sciences

AN INCREMENTAL APPROACH TO IDENTIFYING CAUSES OF SYSTEM FAILURES USING FAULT TREE ANALYSIS

A Dissertation Presented

by

HUONG PHAN

Approved as to style and content by:

DEDICATION

To my parents.

ACKNOWLEDGMENTS

I would like to thank my dissertation advisors, Professors Lori Clarke and George Avrunin, for their guidance and especially their encouragement.

I also would like to thank the committee members, Professors Leon Osterweil, Matt Bishop, and Wayne Burleson, for their feedback and support.

This work would not be possible without the initial work laid out by Bin Chen, a former lab member, or without hours of discussion with current as well as former lab members, Heather Conboy and Stefan Christov.

I am forever grateful for Bonnie Nasca and her husband Philip Nasca for giving me unconditional love and support in my early time at UMass. They moved away but we still remain family.

My time in Amherst was made enjoyable by my photography friends and especially salsa friends. Thank you, friends.

Last but not least, I want to express by deep gratitude toward my family – my parents, my sister's family, my daughter, and my partner Sophal Khun – for all their love and support.

ABSTRACT

AN INCREMENTAL APPROACH TO IDENTIFYING CAUSES OF SYSTEM FAILURES USING FAULT TREE ANALYSIS

MAY 2016

HUONG PHAN

B.I.T., QUEENSLAND UNIVERSITY OF TECHNOLOGY

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

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

This work presents a systematic, incremental approach to identifying causes of potential failures in complex systems. The approach builds upon Fault Tree Analysis (FTA), but enhances previous work to deliver better results. FTA has been applied in a number of domains to determine what combinations of events might lead to a specified *undesired event* that represents a system failure. Given an undesired event, FTA constructs a *fault tree (FT)* and computes its *cut sets*, the sets of events that together could cause the undesired event. Such cut sets provide valuable insights into how to improve the design of the system being analyzed to reduce the likelihood of the failure. Manual FT construction can be tedious and error-prone. Previous approaches to automatic FT construction are limited to systems modeled in specific modeling languages and often fail to recognize some important causes of failures. Also, these approaches tend to not provide enough information to help users understand how the events in a cut set could lead to the specified undesired event and, at the same time, often provide too many cut sets to be helpful, especially when systems are large and complex.

Our approach to identifying causes of potential system failures is incremental and consists of two phases that support selective exploration. In the first phase, a high-level FT, called the *initial FT*, is constructed based on the system's data and control dependence information and then the initial FT's cut sets, called the *initial cut sets*, are computed. In the second phase, users select one initial cut set for more detailed analysis. In this detailed analysis, additional control dependence information is incorporated and error combinations are considered to construct a more detailed FT, called the *elaborated FT*, that focuses on the chosen initial cut set. The cut sets of the elaborated FT, called the *elaborated cut sets*, are then computed, and concrete *scenarios* are generated to show how events in each of those elaborated cut sets could cause the specified undesired event. Our approach is applicable to any system model that incorporates control and data dependence information. The approach also improves the precision of the results by automatically eliminating some inconsistent and spurious cut sets.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Any complex system is subject to failures. The likelihood of failures, however, can often be reduced by improving the system design. To do so, one must understand how a failure might happen, what the causes could be. One approach to identifying causes of potential failures is to use Fault Tree Analysis (FTA), a deductive, top-down analytical technique used in a variety of industries [12]. In FTA, an undesired event representing the system failure is specified, and the system model is analyzed to find ways that the specified undesired event could occur. FTA includes two parts: constructing a *fault tree (FT)* and analyzing the FT. A FT is a graphical model that depicts the combinations of events that together could cause the specified undesired event, which is at the top of the FT. Hereafter we use *top event* to refer to the undesired event representing the system failure being analyzed. Once the FT is constructed, different quantitative and qualitative analyses can be applied. To identify causes of the system failure, one uses a qualitative analysis that computes the FT's *cut sets*, which are combinations of events that might cause the top event. Straightforward Boolean algebra can be applied to automatically compute a FT's cut sets.

Manual FT construction, however, is tedious and error prone. Many have attempted to automate FT construction for various types of systems [3,4,14,21,22,24]. There are still some problems with these previous approaches. They are language dependent; each approach is limited to systems that are modeled in a specific language, e.g., ADA programs [21], Pascal programs [14], UML models [19,24,34], SysML models [22], and Little-JIL process models [3, 4]. They do not appear to fully exploit control dependence information, hence producing incomplete FTs. In some cases, published papers show little evidence of evaluation on real complex systems. Further, the cut sets that are computed often do not provide users with enough information to understand how all the events in a cut set could happen in a system execution and lead to the occurrence of the top event. At the same time, these approaches may provide too many cut sets to be helpful, especially when systems are large and complex. Finally, these previous approaches do not seem to deal with false positives — inconsistent and spurious cut sets[1].

In an attempt to address the above problems, we investigated a two-phase approach that lets users incrementally gain deeper understandings of causes of system failures. Given a system model and an undesired event representing a system failure, the first phase uses data and control dependence information from the model to automatically construct, or *derive*, a high-level FT, called the *initial FT*, whose top event is the undesired event. Also in this first phase, the cut sets from the initial FT, called the *initial cut sets*, are computed and presented to users. In the second phase, users can then select one initial cut set as the basis for more detailed analysis. In this detailed analysis, with the focus on the chosen initial cut set, additional control dependence information is incorporated and error combinations are considered to elaborate the initial FT to result in a more detailed FT, called the *elaborated FT*. Its cut sets, called the *elaborated cut sets*, are then computed, and concrete *scenarios* for each elaborated cut set are generated. A scenario is a system execution path that contains all events in an elaborated cut set and shows how the events in the cut set

[1]A cut set is said to be inconsistent if it is impossible for all of the events in the cut set to occur in one system execution. A cut set is said to be spurious if all system executions, each of which contains all events in the cut set and does not contain all events in any other cut set, turn out to not result in the top event.

could lead to the top event. After cut sets from FTs are computed, some inconsistent and spurious cut sets are automatically identified and removed.

We implemented and evaluated our approach by applying it to models of four non-trivial human-intensive systems from the medical and election domains. As part of larger projects for modeling and analyzing human-intensive systems, these models had been created with help from domain experts. These models are defined in Little-JIL, a process modeling language [33], which provides sufficient control and data flow information to support our approach. The terms *process* and *system* therefore are used interchangeably from this point forward. By applying our approach to Little-JIL process models, we show that the approach is generic and applicable to any language that includes data and control flow information.

The contributions of this work are:

1. An improved approach to identifying causes of system failures using FTA that:
    - is systematic in that it is based on data and control dependence, and thus can be implemented using standard data and control flow analysis;
    - identifies more cut sets through exploiting additional control dependence information;
    - supports selective incremental exploration of an initial high-level FT, thus allows users to focus their effort and gain a deeper understanding on specific areas of interest;
    - improves the precision of the results (initial and elaborated cut sets) by automatically removing some types of inconsistent and spurious cut sets.
2. Experimental results that demonstrate some of the strengths and weaknesses of our approach.

The rest of the thesis is organized as follows. Chapter 2 presents the background on FTA, data and control dependence, the process modeling language Little-JIL, and related work on variations of FTs and other automated FT construction work. Chapter 3 describes our two-phase approach based on data and control flow analyses. This chapter focuses on the FT derivation for the two phases, the removal of spurious cut sets, and the generation of scenarios. The discussion in this chapter is independent of the system modeling language used. Chapter 4 shows the evaluation of our approach using four systems modeled in Little-JIL. Finally, Chapter 5 presents our conclusions and discusses potential future work.

CHAPTER 2
BACKGROUND AND RELATED WORK

The first section of this chapter presents a brief history of Fault Tree Analysis (FTA) and an introduction to elements of a fault tree (FT). Since our FT derivation builds upon data and control flow analyses, the second and third sections provide the necessary background of those areas. The forth section describes the Little-JIL process definition language. And the last section discusses FTA-related work.

2.1 Fault Tree Analysis

Fault Tree Analysis (FTA) was first developed in 1962 at Bell Laboratories by H.A. Watson in a contract with US Air Force to study the Minuteman Launch Control System. Since then the value of FTA has been well recognized and FTA has been adopted by various industries, such as aerospace, nuclear power, chemical processing and automotive [12]; and more recently FTA has been used in health care and security [3, 16, 32]. Lately, it has been adopted experimentally in reasoning about the possibility of failures in election processes [25, 28].

FTA is a deductive, top-down analytical technique that begins with a specification of an *undesired event* representing a system failure. "The launch vehicle fails to ignite at launch" and "no propellant is supplied to the thruster when the arming command is initiated" are two examples of system failures from the aerospace industry [30].

With FTA, one first postulates the possibility of a particular system failure, and then attempts to find out which events in the system could combine to cause the actual occurrence of the failure, represented by an undesired event. Given the un-

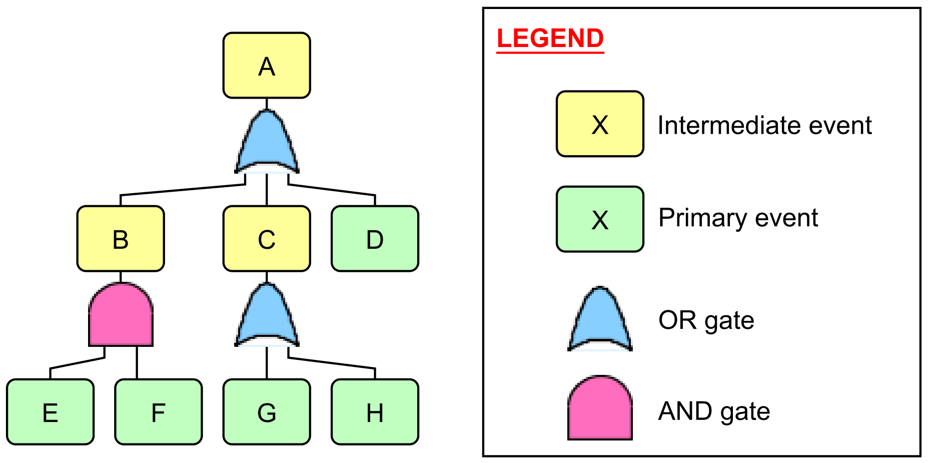


Figure 2.1. Fault Tree Example.

desired event, FTA produces a *fault tree (FT)*, a graphical model of all the various combinations of events that could lead to the undesired event.

Events and *gates* are the basic elements of a FT. The event at the root (top) of the tree is the undesired event, hereafter referred to as the *top event*. The logical relationships of the events are shown by *gates*. A gate connects one or more *input events* (typically drawn below the gate) to a single *output event* (typically drawn above the gate); the gate is called the *input gate* of the output event. In a FT, the *primary events* are the causal basic events; they are at the bottom of the tree and are not further developed; they do not have input gates. On the other hand, the *intermediate events*, including the top event, are events that have input gates.

Figure 2.1 shows an example of a FT with the top event A, intermediate events B and C, and primary events D, E, F, G and H.

We consider two types of gates:

- AND gates specify that the output event occurs only if all input events occur (inputs are assumed to be independent), e.g., in Figure 2.1, B occurs only if both E and F occur.

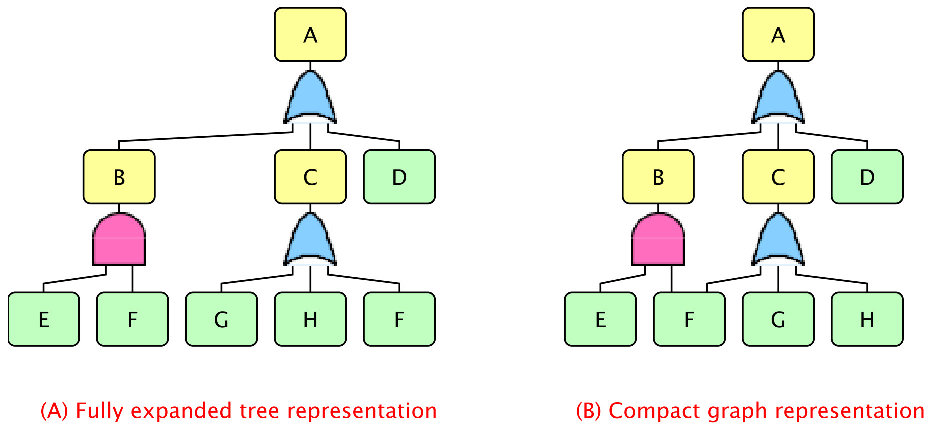

A
B
C
D
E
F
G
H
F
(A) Fully expanded tree representation
A
B
C
D
E
F
G
H
(B) Compact graph representation

Figure 2.2. Fault Tree's Compact vs. Fully Expanded Representation.

- OR gates specify that the output event occurs only if at least one of the input events occurs, e.g., in Figure 2.1, A occurs only if at least B or C or D occurs.

It is not uncommon to see an event being a contributing cause of more than one other event. For example, in Figure 2.2(A), we see that the event F is an input event of B's input gate; F is also an input event of C's input gate. We can compactly represent the FT as shown in Figure 2.2(B). In this compact representation, the FT is not really a tree as defined traditionally where there is exactly one path from the root to any vertex. While the compact graph representation is smaller in size, it produces the same cut sets as its fully expanded counterpart does. Therefore we choose the compact graph representation and still use the term *fault tree*.

Once a FT is constructed, its cut sets and minimal cut sets can be computed using Boolean algebra. A *cut set* is a set of primary events whose occurrence together could cause the top event to occur. A cut set is considered *minimal* if, when any of its events is removed from the set, the resulting set is no longer a cut set. To compute cut sets from a FT, each gate in the FT is first translated into a Boolean equation. Consider the FT example in Figure 2.2(B), the Boolean equations are shown as follows.

- A's input gate: $A = B + C + D$

- B's input gate: $B = E \cdot F$
- C's input gate: $C = F + G + H$

Given these equations, we can apply Boolean algebra to get rid of all intermediate events, except for the top event, and obtain an equation whose left hand side is the top event and the right hand side is a formula of primary events in *Conjunctive Normal Form*. Each clause in this formula corresponds to a cut set, which contains all events in that clause. The final equation computed for the example FT in Figure 2.2(B) is

$$A = E \cdot F + F + G + H + D.$$

From this equation we get five cut sets: $\{E, F\}$, $\{F\}$, $\{G\}$, $\{H\}$, and $\{D\}$. Further Boolean "minimization" produces four minimal cut sets $\{F\}$, $\{G\}$, $\{H\}$, and $\{D\}$. The cut set $\{E, F\}$ is not minimal because removing E from it, we have $\{F\}$ which is also a cut set.

A cut set indicates a potential system vulnerability, which might be a flaw or weakness in the system's design, implementation, or operation and management, that could potentially allow a system failure to occur. A cut set with one element represents a *single point of failure*. The probability of a system failure occurring could be calculated if sufficient information about the probabilities of events in the cut sets is available.

In our approach, we analyze systems that can be modeled as control flow graphs (CFGs) annotated with data flow information. We define this modeling formalism more precisely in the next section (Section 2.2). With system models of this type, we target system failures that can be characterized as the arrival of an incorrect, unsuitable data value as an input to a particularly critical vertex on the CFG or as the production of an incorrect data value as an output of a vertex. Some examples of such failures are "the wrong type of blood is transfused to a patient", or "the voter casts votes using an incorrect type of ballot". And while minimal cut sets have traditionally been used to obtain an estimate of reliability for complex systems, we also pay attention to cut sets (i.e., not necessarily minimal) because we want to present to users all possible scenarios about how the failure could arise.

2.2 Data Dependence

Our FTA approach builds upon data and control dependence information. To define data dependence, we first introduce the Reaching Definitions Problem in this section. The Reaching Definitions Problem is used as a basis for our Immediate Data Dependence Problem, described later in Section 3.2. Even though the discussions in this background section are about computer programs, they are applied to any systems, including processes that involve human participants, that can be presented by control flow graphs.

The Reaching Definitions Problem

The Reaching Definitions Problem introduced by Allen [1] is a well-known data flow problem to answer the question: which variable definitions reach a given point in a program?

A program is represented by a *control flow graph (CFG)*. A CFG is a directed and connected graph $G = (V, E, \mathbf{start}, \mathbf{end})$, where V is the set of vertices, E is the set of edges, **start** is the single entry vertex, and **end** is the single exit vertex. For any vertex v in V there exists a path from **start** to v and from v to **end**. Each vertex represents a program unit (which could be a statement, a block of code, or a procedure, depending on the desired level of granularity of analysis). As with a typical graph structure, associated with each vertex v there are the sets $\mathit{Pred}(v)$, and $\mathit{Succ}(v)$, which include the immediate predecessor and immediate successor vertices of v respectively. To annotate the CFG with data flow information, for each vertex v, let $Def(v)$ and $Use(v)$ be the sets that include the variables defined and referenced at v respectively. Let Σ be the set of all variables in the program.

In the CFG representation of the program, we assume that a vertex v with more than one successor has predicates associated with its outgoing edges. An edge that has a predicate associated with it is called a *guarded edge*. Let $(u \to v)$ denote an edge from vertex u to vertex v in the CFG. If $(u \to v)$ is a guarded edge, we use $p(u, v)$ to denote the predicate on that edge. Readers might be familiar with CFGs that have decision vertices represented as diamond shapes; each decision vertex has two outgoing edges, one labeled `T` and one labeled `F` (e.g., the vertex c_1 in Figure 2.3(A) with the condition $x > 0$). In our CFG representation, the decision is explicitly associated with the edges as in the example in Figure 2.3(B), we have the following edge predicates: $p(c_1, s_1) = x > 0$, and $p(c_1, s_2) = \neg(x > 0)$.

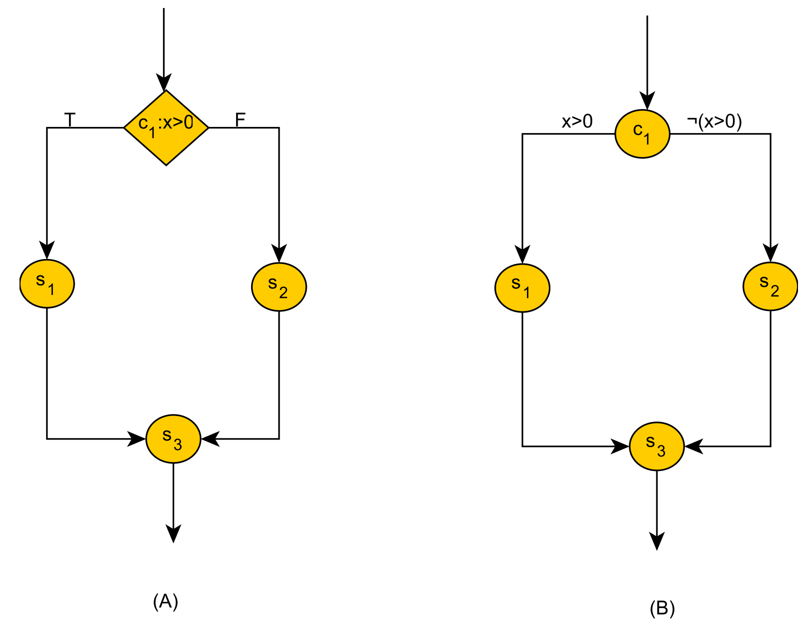


Figure 2.3. CFG With Decision Vertex (A) Converted To CFG With Edge Predicates (B).

We also assume that the CFG representation does not contain loops. This restriction is discussed later in section 3.2.1.

In the Reaching Definitions Problem, for the purpose of analysis, if variable b is defined at vertex v, a label b_v is created and that label is used to represent the definition of b at v. A definition b_v reaches a program point or vertex u if b_v occurs on some path in the CFG from **start** to u and is not followed by any other definition of b on this path. For every variable b, a definition $b_{\textbf{start}}$ reaches **start**. If definition $b_{\textbf{start}}$ reaches a use of b, it suggests that b has to be an input variable to the program, or else it is a potential use before definition.

At each vertex v, we have the following information:

The set of definitions that v generates

$$GEN_v = \{b_v \mid b \in \mathit{Def}(v)\},$$

and the set of definitions that v can *kill* (i.e. redefine)

$$KILL_v = \{b_u \mid b \in \mathit{Def}(v),\ u \in V\}.$$

We use IN_v to denote the set of definitions that reach v on some path through the CFG, and OUT_v to denote the set of definitions available right after v is executed. Intuitively, OUT_v contains the definitions that either are generated at v or reach v and are not redefined at v.

$$IN_v = \begin{cases} \{b_{\textbf{start}} \mid b \in \Sigma\} & \text{if } v = \textbf{start} \\ \bigcup\limits_{u \in \mathit{Pred}(v)} OUT_u & \text{otherwise} \end{cases}$$

$$OUT_v = GEN_v \cup \Big(IN_v \setminus KILL_v\Big)$$

The computation of IN_v and OUT_v is shown below.

In the terms described by Kildall [17], Reaching Definitions is an instance of a distributive data flow framework defined by:

- The semi-lattice $(L, \wedge)$ in which
  - L, the domain of values, consists of all subsets of the set of definitions; and
  - $\wedge$, the binary *meet operation*, is set union $\cup$. Consequently, the partial ordering on L is the set containment $\supseteq$.
- The *transfer function*, $f : V \times L \to L$

$$f(v, x) = \mathit{GEN}_v \cup \Big(x \setminus \mathit{KILL}_v\Big)$$

  that has the homomorphism property (also called the *distributivity* property)

$$\begin{aligned} f(v, x) \wedge f(v, y) &= \mathit{GEN}_v \cup (x \setminus \mathit{KILL}_v) \cup \mathit{GEN}_v \cup (y \setminus \mathit{KILL}_v) \\ &= \mathit{GEN}_v \cup ((x \cup y) \setminus \mathit{KILL}_v) \\ &= f(v, x \wedge y) \end{aligned}$$

Kildall provides an iterative algorithm, which he calls a "global analysis algorithm", that starts with an approximation and iteratively updates the "current approximate pool of optimizing information" associated with each vertex until no more changes can be made. Kildall shows that the algorithm applied to a distributive data flow analysis framework yields a unique maximal fixed point solution, independent of the order in which vertices are visited. Cooper [11] gave a more efficient yet easy to understand work-list algorithm by focusing the iteration on regions in the graph where information is changing. The algorithm begins by initializing the sets at all vertices and constructing an initial work-list. It then repeats the process of removing a vertex from the work-list and updating its data-flow information. If the update changes the data-flow information at the vertex, then all of the vertices that depend on the changed information are added to the work-list.

The work-list approach for the Reaching Definitions Problem is shown in Algorithm 2.1, starting with the “approximation” ($IN_v = \varnothing$ for all $v \neq$ **start**, $IN_{\textbf{start}} = \{b_{\textbf{start}} \mid b \in \Sigma\}$) and converging to the maximal fixed point solution.

```
input  : CFG, GEN, KILL
output: IN, OUT
foreach v $\neq$ start do
    // initialize OUT on the assumption IN(v) = $\varnothing$ for all v
    OUT$_v$ $\longleftarrow$ GEN$_v$;
IN$_{start}$ $\longleftarrow$ $\{b_{\textbf{start}} \mid b \in \Sigma\}$;
worklist $\longleftarrow$ {start } ;
while worklist is not empty do
    Remove vertex v from worklist;
    // Recompute the sets at vertex v
    IN$_v$ $\longleftarrow$ $\bigcup_{u \in Pred(v)}$ OUT$_u$;
    oldOUT $\longleftarrow$ OUT$_v$;
    OUT$_v$ $\longleftarrow$ GEN$_v$ $\cup$ $\left(IN_v \setminus KILL_v\right)$;
    if OUT(v) $\neq$ oldOUT then
        // recompute the IN and OUT sets for successors of v
        Add each successors of v to worklist;
```

Algorithm 2.1: Compute Reaching Definitions.

Once the IN and OUT sets converge to the maximal fixed point solution, the definitions reaching vertex v are:

$$REACH(v) = IN_v.$$

If $b_u \in REACH(v)$ then v is said to be *data flow dependent* on u [18].

Example:

In Figure 2.4, x is defined at vertex 1 and vertex 7 , and it is used at vertex 8 to define z. Vertices 2 and 3 define Boolean variables p and q respectively. There is another variable y defined at vertex 6. The edges $(2 \rightarrow 3)$, $(2 \rightarrow 7)$, $(3 \rightarrow 4)$, and $(3 \rightarrow 5)$ are guarded with predicates p, $\neg p$, q, and $\neg q$ respectively. In the figure, each

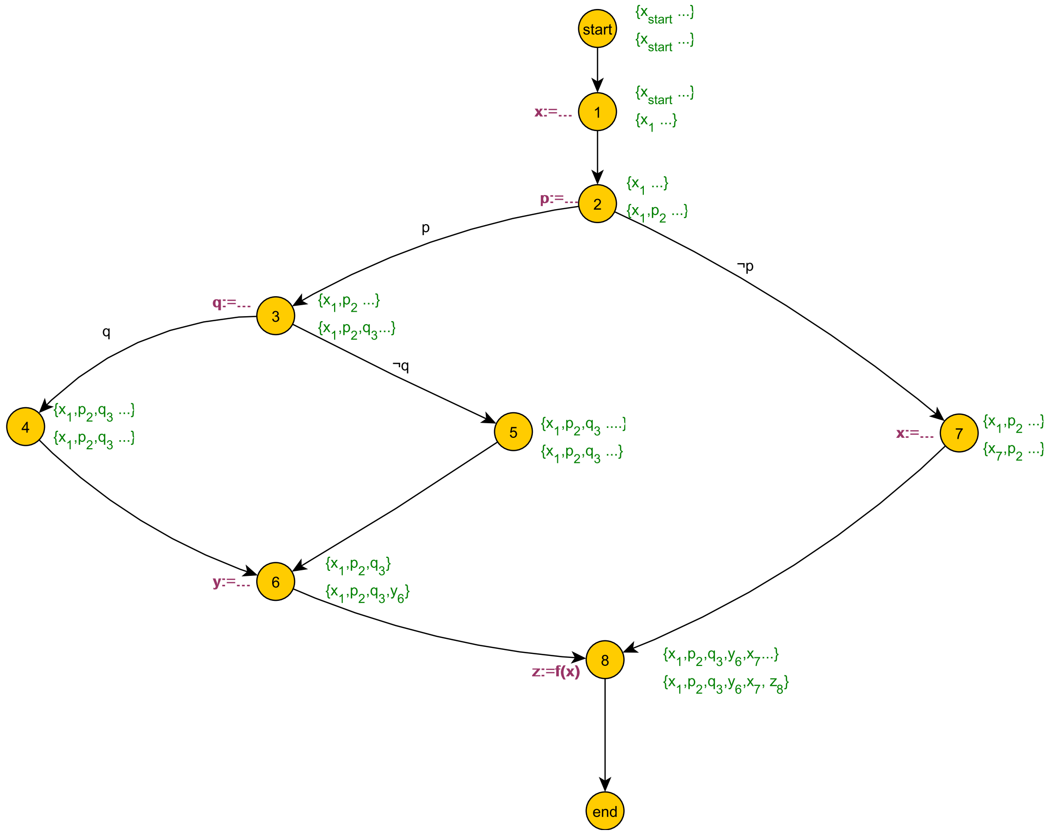


Figure 2.4. CFG Showing Reaching Definitions.

processing	IN	OUT	work-list
start	$\{p_{\textbf{start}}, q_{\textbf{start}}, x_{\textbf{start}}, y_{\textbf{start}}, z_{\textbf{start}}\}$	$\{p_{\textbf{start}}, q_{\textbf{start}}, x_{\textbf{start}}, y_{\textbf{start}}, z_{\textbf{start}}\}$	1
1	$\{p_{\textbf{start}}, q_{\textbf{start}}, x_{\textbf{start}}, y_{\textbf{start}}, z_{\textbf{start}}\}$	$\{x_1, p_{\textbf{start}}, q_{\textbf{start}}, y_{\textbf{start}}, z_{\textbf{start}}\}$	2
2	$\{x_1, p_{\textbf{start}}, q_{\textbf{start}}, y_{\textbf{start}}, z_{\textbf{start}}\}$	$\{x_1, p_2, q_{\textbf{start}}, y_{\textbf{start}}, z_{\textbf{start}}\}$	3,7
3	$\{x_1, p_2, q_{\textbf{start}}, y_{\textbf{start}}, z_{\textbf{start}}\}$	$\{x_1, p_2, q_3, y_{\textbf{start}}, z_{\textbf{start}}\}$	7,4,5
7	$\{x_1, p_2, q_{\textbf{start}}, y_{\textbf{start}}, z_{\textbf{start}}\}$	$\{x_7, p_2, q_{\textbf{start}}, y_{\textbf{start}}, z_{\textbf{start}}\}$	4,5,8
4	$\{x_1, p_2, q_3, y_{\textbf{start}}, z_{\textbf{start}}\}$	$\{x_1, p_2, q_3, y_{\textbf{start}}, z_{\textbf{start}}\}$	5,8,6
5	$\{x_1, p_2, q_3, y_{\textbf{start}}, z_{\textbf{start}}\}$	$\{x_1, p_2, q_3, y_{\textbf{start}}, z_{\textbf{start}}\}$	8,6
8	$\{x_7, p_2, q_{\textbf{start}}, y_{\textbf{start}}, z_{\textbf{start}}\}$	$\{x_7, p_2, z_8, q_{\textbf{start}}, y_{\textbf{start}}\}$	6
6	$\{x_1, p_2, q_3, y_{\textbf{start}}, z_{\textbf{start}}\}$	$\{x_1, p_2, q_3, y_6, z_{\textbf{start}}\}$	8
8	$\{x_1, p_2, q_3, y_6, x_7, q_{\textbf{start}} y_{\textbf{start}}, z_{\textbf{start}}\}$	$\{z_8, x_1, x_7, p_2, q_3, q_{\textbf{start}}, y_6, y_{\textbf{start}}\}$	

Table 2.1. Reaching Definitions Example. Showing the progress of the work-list algorithm.

CFG vertex is annotated with two sets on two lines: on the first line is the *IN* set of the vertex, and on the second line is the *OUT* set. Table 2.1 shows the progress of repeatedly updating the IN and OUT sets until reaching a fixed point.

2.3 Control Dependence

We use control dependence information in deriving FTs. This section describes the necessary background information about control dependence.

- A vertex v is said to *post-dominate* a vertex t if all paths to the **end** vertex of the CFG starting at t must go through v.

- A vertex v is said to be *control dependent* on an edge $(u \to t)$ if

1. v post-dominates t, and

2. v does not post-dominate u.

Intuitively, this means that if the control flows from u to t along $(u \to t)$, it will eventually reach v; however, the control may reach **end** from u without passing through v. Thus, u is a "decision-point" that influences the execution of v.

The Control Dependence Problem has been well studied in the literature. There are different algorithms to compute the control dependence relation. Ferrante et al. [13] provide an algorithm to compute control dependence given the post-dominator tree of the CFG. The post-dominator tree in a CFG is the dominator tree in the *reverse CFG*.

The *reverse CFG* of a CFG $G = (V, E, \mathbf{start}, \mathbf{end})$ is a graph $G' = (V, E', \mathbf{end}, \mathbf{start})$. The two graphs G and G' have the same set of vertices V, however, in the reverse CFG, **end** is the entry vertex and **start** is the exit vertex, and all the edges are reversed, i.e. $(u \to v) \in E \iff (v \to u) \in E'$ for all u and v such that $u \neq \mathbf{start}$ and $v \neq \mathbf{end}$. Besides the reversed edges, an auxiliary edge $(\mathbf{end} \to \mathbf{start})$ is added to the reverse CFG.

Next we show how to compute the dominator tree of any CFG. To compute the post-dominator tree of a CFG, one just has to compute the dominator tree for its reverse CFG. Note that, in this section, we do not require a CFG to have the requirements about edge predicates as stated in the previous section. The notion of control dependence do not depend on edge predicates.

Computing the dominator tree of a CFG $G = (V, E, \mathbf{start}, \mathbf{end})$

A vertex u is said to *dominate* a vertex v if all paths from **start** of the CFG to v must go through u.

u is said to *strictly dominate* v if u dominates v and u does not equal v.

The *immediate dominator* or *idom* of a vertex v is the unique vertex that strictly dominates v but does not strictly dominate any other vertex that strictly dominates v. Every vertex, except **start**, has a unique immediate dominator.

A *dominator tree* is a directed graph where each edge $(u \to v)$ means that u immediately dominates v. Because each vertex has a unique immediate dominator, that directed graph is a tree. The **start** vertex is the root of the tree.

The set of dominators of a vertex v, denoted Dom_v, is given by the solution to the following set of equations:

$$Dom_v = \Big(\bigcap_{u \in \text{Pred}(v)} Dom_u \Big) \cup \{v\}.$$

In fact, the problem of computing the relation *Dom*, called the *Dominators* problem, is another instance of a distributive data-flow analysis framework mentioned above, defined by

- The semi-lattice $(L, \wedge)$ where
  - L, the set of values to be propagated, is the power set of vertices in the CFG,
  - $\wedge$, the meet operator, is set intersection $\cap$.
- The distributive transfer function $f : V \times L \to L$ is $f(v, x) = x \cup \{v\}$.

The dominator of the **start** vertex is the **start** vertex itself. The set of dominators for any other vertex v is the intersection of the sets of dominators for all predecessors u of v. The vertex v is also in the set of dominators for v. Algorithm 2.2 shows the algorithm, essentially adapted from Kildall's [17]. As shown above, this algorithm when terminates provides the maximal fixed point solution for Dominators.

Once the full dominance relation is computed, we build the dominator tree top-down with the root as the **start** vertex as shown in Algorithm 2.3.

```
input  : CFG
output: Dom relation
// Dominator of the start vertex is the start itself
$Dom_{start} \leftarrow \{start\}$; // For all other vertices, set all vertices as the
   dominators
foreach $v \neq start$ do
    $Dom_v \leftarrow V$;
// Iteratively eliminate vertices that are not dominators
while there are still changes in any $Dom_v$ do
    foreach $v \neq start$ do
        $Dom_v \leftarrow \big(\bigcap_{u \in Pred(v)} Dom_u\big) \cup \{v\}$;
```

Algorithm 2.2: Compute dominance relation *Dom*.

```
Procedure buildSubTree(v,R)
    // v is the vertex being processed,
    // R is the remained vertices
    foreach r ∈ R do
        Remove v from Dom(r);
        if r is then dominated only by itself then
            Add r as a child to v;
            Remove r from R;
    foreach r just added as a child to v do
        buildSubTree(r,R);
Procedure buildDomTree()
    buildSubTree(start, V);
```

Algorithm 2.3: Compute dominator tree.

Figure 2.5, on the left hand side, shows the reverse CFG of the original CFG in Figure 2.4, augmented with an edge from **end** to **start**. The reverse CFG is annotated with the *Dom* set for each vertex. On the right hand side, Figure 2.5 shows the dominator tree for that reverse CFG, so in fact, the tree is the post-dominator tree of the original CFG.

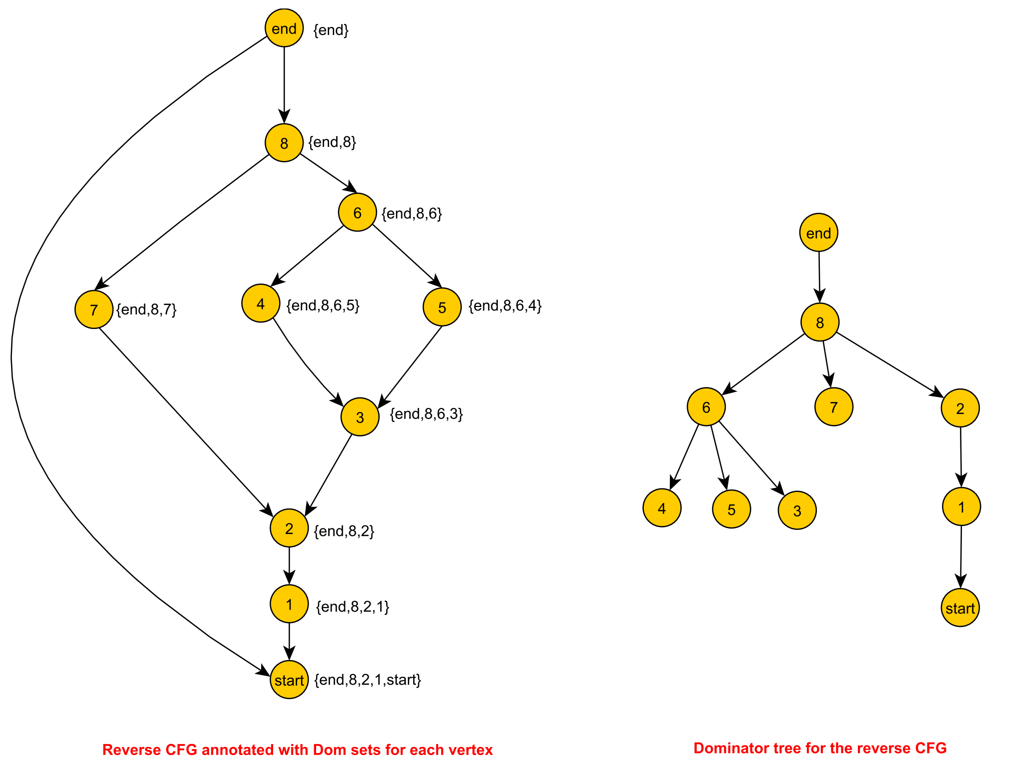


Figure 2.5. Reverse CFG And Its Corresponding Dominator Tree.

Computing Control Dependence

Given the post-dominator tree, we can determine control dependence by examining certain CFG edges. Let S consist of all edges $(u \to v)$ in the CFG such that v is not an ancestor of u in the post-dominator tree (i.e., v does not post-dominate u). We note that each $(u \to v)$ in S is a guarded edge. Let w denote the least common ancestor of u and v in the post-dominator tree. Ferrante et al. [13] show that all vertices in the post-dominator tree on the path from w to v, including v but not w are control dependent on the edge $(u \rightarrow v)$.

In our example, $S = \{(2 \rightarrow 3), (2 \rightarrow 7), (3 \rightarrow 4), (3 \rightarrow 5)\}$. Table 2.2 shows the control dependence relation.

Edge	Least Common Ancestor	Control dependent vertices
$(2 \rightarrow 3)$	8	6,3
$(2 \rightarrow 7)$	8	7
$(3 \rightarrow 4)$	6	4
$(3 \rightarrow 5)$	6	5

Table 2.2. Control Dependence Example.

2.4 The Process Modeling Language Little-JIL

We implement and evaluate our approach by applying it to models of four real-world human-intensive systems from the medical and election domains. These models are defined in Little-JIL, a process modeling language. Therefore, this section provides an introduction to Little-JIL. Details about the language can be found in [33].

Little-JIL is used to model processes. A process informally is a sequences of tasks carried out to achieve a certain outcome. Each Little-JIL process model carries control and data flow information, therefore they are suitable to be used as system models in our approach. As mentioned earlier, we use the terms *process* and *system* interchangeably.

Little-JIL represents a process as a hierarchy of *steps* carried out by *agents* that may be humans, hardware devices, or software applications. A Little-JIL model consists of activity diagrams showing the hierarchical decomposition of steps, a specification of the *artifacts* manipulated by the steps, and a specification of the agents and *resources* needed to perform the steps.

A step is the basic building block of Little-JIL models. A Little-JIL step is a specification of a unit of work assigned to an agent in the process. A step may be decomposed into *sub-steps* (children). A *leaf step* has no sub-steps and its behavior depends entirely on its assigned agent. A *non-leaf step*'s behavior consists of the behaviors of its sub-steps and their order of execution. Every Little-JIL process model has a *root step* that represents the entire process. This step is decomposed as far as necessary to describe the process.

For example, a model reflecting the election process used by Yolo County, California[1] consists of the root step "conduct election", which is decomposed into sub-steps representing activities such as the preparations made before the election day, the conduct of the election at a single precinct, the counting of ballots, and the post-election canvass. These sub-steps in turn are decomposed into steps at a more specific level of detail.

Figure 2.6 shows the part of the elaboration of "issue ballot and get vote"—one of the steps in the Yolo County election process. Voters are checked for their eligibility before being issued a ballot. If the voter is verified as eligible, a regular ballot is issued, otherwise a provisional ballot (which is not examined unless the election is close) is issued. In the Little-JIL model, the step "issue ballot and get vote" has two sub-steps, namely "issue ballot" and "cast vote". The sub-steps are executed sequentially (denoted by the blue arrow $\rightarrow$ on the step bar). The step "issue ballot" in turn has three sub-steps, "verify registered", "verify not-voted" and "issue regular", which are also executed sequentially.

Each step may contain a specification for pre-requisites that must be satisfied before an agent can begin the work and post-requisites to check that the work was completed correctly (not present in this example).

[1]http://www.yoloelections.org/

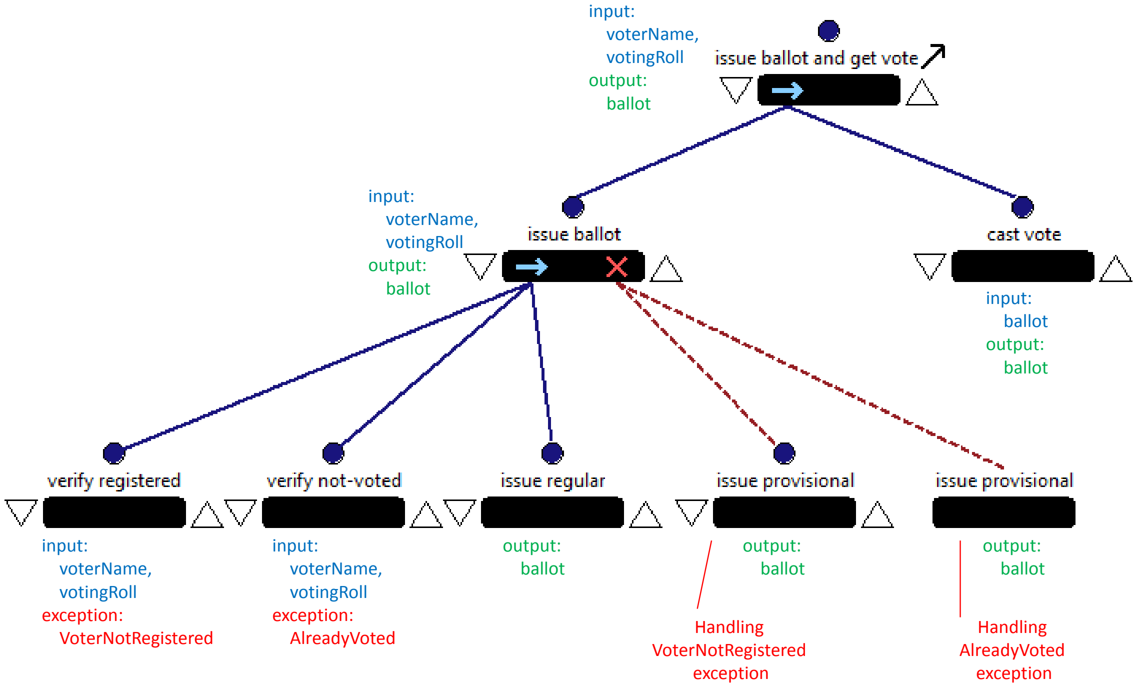


Figure 2.6. Little-JIL Process Model Of “Issue ballot and get vote”.

The execution of a Little-JIL step is modeled as a progress through several states. Step execution begins in the `POSTED` state during which the execution of the step is assigned to an agent. The execution then proceeds to the `STARTED` state when the agent begins performing the step. Eventually the step enters either the `COMPLETED` state (normal execution) or the `TERMINATED` state (the execution ends with an exception).

A step may also specify how to handle exceptions that are reported during the execution of its descendant steps. An exception handler can be a Little-JIL step that is capable of defining an arbitrarily complex response to an exception, or can be a *simple handler* that specifies only how execution should continue after the occurrence of the exception. In the example in Figure 2.6, the steps "verify registered" and "verify not-voted" might throw exceptions of types `VoterNotRegistered` and `AlreadyVoted` respectively; the red cross × on the "issue ballot" step bar connects to the "issue provisional" step that handles `VoterNotRegistered` exceptions; the red cross × also connects to a *reference* of "issue provisional" (similar to invoking a predefined procedure in a procedural programming language) which handles `AlreadyVoted` exceptions. In this example, the handlers are specified so that the sub-steps throwing the exception will be terminated, the handlers are executed and the process execution moves to the `COMPLETED` state of "issue ballot".

Each Little-JIL step has an artifact declaration that defines the artifacts it is accessing or providing. Artifacts are generally passed through the coordination hierarchy between steps and their sub-steps. For example, the `ballot` artifact is output from the step "issue regular" to its parent step "issue ballot"; so is the `ballot` artifact output from the step "issue provisional". The `voterName` and `votingRoll` artifacts are input into the step "issue ballot" from its parent "issue balot and cast vote", then passed down from "issue ballot" to its sub-steps "verify registered" and "verify not-voted".

A Little-JIL process model also includes agent specifications. Each step specifies the kind of agent that is to be assigned to the step to be responsible for the execution of that step[2].

Little-JIL has been used to model different processes for different purposes. Conboy et al. [10] used a Little-JIL process model as the underlying model for a smart checklist to mitigate the risk of stroke in cardiac surgeries. Many different analyses have been performed on Little-JIL process models, namely finite state verification [3, 29], online deviation detection [5], resource scheduling [27], automatic requirement derivation [9], automatic failure mode and effect analysis [31], fault tree analysis [3, 25], etc.

Given a Little-JIL process model, we can generate an appropriate CFG to be used in our FTA approach. Section 4.1.1 describes how to generate a CFG from a Little-JIL process model.

2.5 Related Work

This section discusses related work including the variations of FTs and what they are used for, and the various approaches to automatic FT construction.

2.5.1 Variations of Fault Trees

Attack trees, introduced by Schneider [26] are similar to fault trees in the sense that they are both hierarchical logic diagrams in which one event is represented as a logical combination of lower-level events. Moore et al. [23] used attack trees and attack patterns to model attacks for the purpose of documentation. Lazarus [20] created a catalog of election attacks in the form of a single attack tree, attempting to

[2]Note that, while "cast vote" is a leaf step performed by an agent of type `Voter`, the step "issue ballot" is not a leaf step, it composes of sub-steps which can be performed by the same or different agent of type `ElectionOfficial`. The agent specification is not modeled in this example.

provide a threat model and a quantitative threat evaluation that are reusable across different jurisdictions.

Helmer et al. [15] used augmented Software Fault Trees (SFTs), attack trees with temporal order, to model intrusions. In their models, the root node represents the intrusion and a minimal cut set contains events to be monitored to detect intrusions. Their SFTs are then automatically converted to colored Petri nets for intrusion detection systems.

2.5.2 Other Automated Fault Tree Generation Work

Many software tools, commercial as well as open-source, facilitate the manual construction of FTs. When FTs become large, which they typically do, manual construction, even with such tool support, may be error-prone and time-consuming.

There have been attempts to generate FTs automatically, for example from source code written in programming languages. In [21], Leveson et al. proposed to use FTs to guide analysts to identify errors that cause an Ada program to produce incorrect outputs. The incorrect output is represented as the FT's top event. Templates, one for each kind of Ada statement, are used to elaborate immediate events to construct the full FT. Friedman [14] also developed a template-based tool to construct FTs given a Pascal program and a software-caused failure (post condition).

In [24], Pai and Bechta showed an algorithm to automatically derive FTs from UML models. They pointed out that the disadvantage of their approach is the semi-formal nature of UML. Because of UML's ambiguous semantics, their FTs whose derivation is based on one interpretation of the semantics, might be incorrect.

Zhao and Petriu [34] also generated FTs from UML models. They used a rule-based transformation language ATL and created rules to map elements of UML models (sequence diagrams, use case diagrams, and composite structure diagrams) to elements of FTs. Their top events, as well as other events in their FTs, are simply

"X fails", where X could be the whole system, a hardware component, a software component, or a use case. Our approach considers more types of events that involve data and hardware/software/human behaviors.

Lauer et al. [19] synthesized FTs from different types of UML models (software/hardware architecture diagrams and application models built on top of those architectures). They focused on the separation of application independent and application dependent views of the system. They claimed that in doing so various different system concepts could be investigated with minimal re-modeling effort, thus could be used at the early design stages.

In [22] Mhenni et al. presented a methodology to automatically generate FTs from SysML system models. They represented a SysML Internal Block Diagram (IBD) as a directed multi-graph, then used a graph traversal algorithm and identified *block design patterns* in the IBD to generate sub-FTs and combined them all into a generic FT. To produce a specific FT for an undesired top event, information from separate FMEA (Failure Mode and Effect Analysis) was used to refine the generic FT. Our approach starts from the top event, then traverse backward on the system's flow graph to identify errors, so the derived FT is specific for the given top event; we do not generate a generic FT. Also, our approach exploits control flow information in system models to construct FTs; their SysML IBDs provide only data flow information.

All of these approaches to automatic FT construction [19, 22, 24, 34] target system models of types different from what our approach does. Our approach targets system models that are control flow graphs with data flow information so that we can exploit the data and control dependence to generate FTs.

Closest to our approach, and what our work is built upon, is the FTA by Chen et al. [3, 4]. They developed a framework that supports the automatic derivation of FTs from rigorously defined process models. In that framework, templates are used to elaborate FT events. More templates can be added to allow new types of events. Given the specification of an undesired event representing a process failure, they use control and data flows to trace back through the process model to identify the original causes that could lead to the failure. They implemented and evaluated the framework for processes modeled in Little-JIL. We already described the language in the previous section of this chapter. Our approach is also template-based, and we also evaluate our approach by applying it to processes modeled in Little-JIL. Their approach, however, does not fully consider control dependence information and thus fails to recognize certain causes of failures in a process model. In our approach, we exploit both data and control dependence to construct FTs. In addition, they claimed that their approach was generic and applicable to any process definition languages as long as they incorporate sufficient data and control flow information, but they only showed how the approach worked for Little-JIL. We explicitly show that our approach is generic in the sense that it works on CFGs and most languages, including Little-JIL, can be translated to CFG.

CHAPTER 3
APPROACH

In this chapter, we present our approach to identifying possible causes of potential system failures using Fault Tree Analysis (FTA). This approach is incremental; it allows users to selectively explore possible causes by initially presenting the users with cut sets for a high-level, and thus smaller, fault tree (FT); the users can then select one cut set for more detailed analysis, culminating in elaborated cut sets and concrete scenarios which show how events in an elaborated cut set could lead to the specified system failure. Our approach also attempts to produce precise results by automatically eliminating some inconsistent and spurious cut sets.

The organization of this chapter is as follows:

- Section 3.1 presents the overall architecture of the approach.
- Section 3.2 shows the simple templates used in the first phase of analysis to derive the initial FT.
- Section 3.3 describes the first phase of the approach by applying it to a small model of an "Issue ballot and get vote" system. The section shows the derivation of the initial FT and then shows the need for incremental selective exploration of the FT's cut sets. This example "Issue ballot and get vote" is used in the following sections to keep illustrating the approach.
- Section 3.4 presents how to extract the part of the initial FT that focuses on a selected cut set.

- Section 3.5 describes the detailed templates used in the second phase of analysis to derive the elaborated FT that focuses on the selected cut set.
- Section 3.6 shows the elimination of inconsistent and spurious cut sets.
- Finally, section 3.7 explains the generation of concrete scenarios.

3.1 The Overall Approach

In this work, as mentioned in section 2.1, a system failure is defined to be an input or output variable of a specific vertex in the system's control flow graph (CFG) receiving an incorrect value. Predefined templates exploit the system's control and data flows to trace back through the CFG to identify the causes that could lead to the specified failure. Starting from the top event that represents the system failure, the templates are applied iteratively to develop intermediate events until all intermediate events have been fully developed, resulting in a FT whose leaves are all primary events. The templates are categorized into *simple* and *detailed* templates. The simple templates are used to derive a FT at the high level, called the *initial FT*. It is only when users select a specific cut set from the initial FT that the detailed templates are used to produce an *elaborated FT*, as described later in this chapter.

Figure 3.1 shows the overall approach. First, given the system model and the undesired event representing the system failure, simple templates (section 3.2) are used to derive the initial FT and then the initial cut sets from the initial FT are automatically computed. In this phase of analysis, we only consider single errors and do not consider combinations of errors. For example, a system execution unit's[1] output being incorrect could be caused by its input being incorrect (data dependence), or the execution incorrectly reaching the unit (control dependence), or the unit being

[1]A system execution unit corresponds to a vertex in the system's CFG.

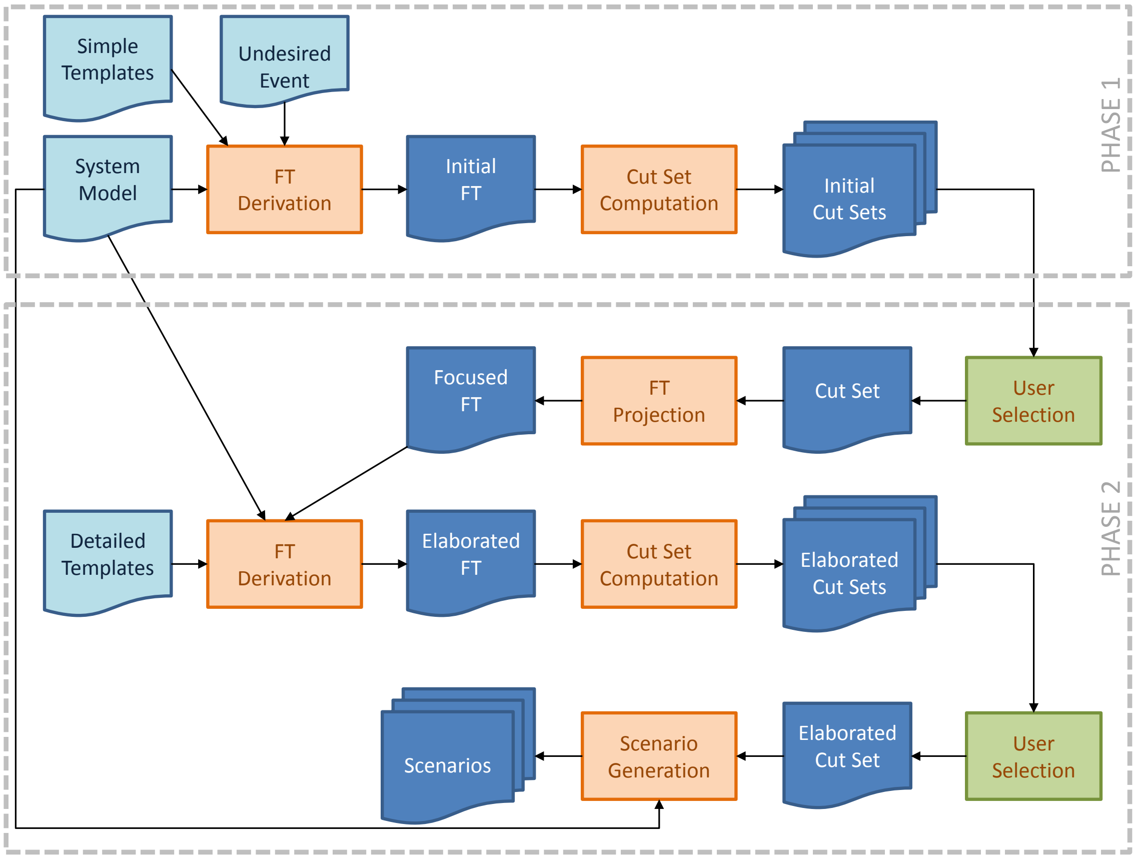


Figure 3.1. The Incremental Approach.

performed incorrectly, or any combination of those three reasons. In this phase, error combinations are not considered.

Once the initial cut sets are computed from the initial FT, users can select one cut set to focus on. This approach allows users to *zoom in* on a specific cut set by:

- creating a projection of the initial FT, called the *focused FT*, that keeps only events relevant to this initial cut set (section 3.4);
- elaborating the focused FT using detailed templates to derive a more detailed FT, called *elaborated FT*, which focuses on the chosen initial cut set (section 3.5); and then
- automatically computing the cut sets of the elaborated FT, called the *elaborated cut sets*. When computing cut sets for the initial FT as well as the elaborated FT, we automatically identify some inconsistent and spurious cut sets and remove them from the result (section 3.6).

Users can then select one of the elaborated cut sets to explore even further. We generate concrete *scenarios* that show the ways that the events in the chosen elaborated cut set can occur and how they can then lead to the top event (section 3.7).

The next section presents the simple templates used in the first phase of this approach.

3.2 Deriving Initial Fault Trees Using Simple Templates

Based on data and control dependence information, we develop simple templates that can be used in the first phase of the analysis to derive the initial FT. Section 3.2.1 presents such templates which are called the *DF* templates. This work builds upon the FTA Framework by Bin Chen [3], in which he also developed templates for deriving FTs for Little-JIL process models. To leverage the existing implementations by Chen, we decide to "re-use" some of the templates made by Chen, but generalize them so that they are independent of system modeling languages. These templates are called the *SP* templates and are described in section 3.2.2. The proof that the two template sets, DF and SP, are equivalent is presented in Appendix A.

3.2.1 DF Set: Templates To Derive Initial Fault Trees Based On Data Flow Analysis

Data flow dependence, in the form of Reaching Definitions, and control dependence are described in the Background Chapter (sections 2.2 and 2.3 respectively). To derive FTs in our approach, we need extra information, therefore we extend the Reaching Definitions Problem to the Path-sensitive Reaching Definitions Problem (PRDP) to answer the question: which variable definitions reach a given point (vertex) in a system's CFG, and which paths do the definitions take to reach that point? In this section, we first describe the PRDP, and we then show how the data dependence information derived by solving PRDP, together with control dependence information, are used to develop the DF Template Set.

3.2.1.1 Path-sensitive Reaching Definitions Problem (PRDP) And Immediate Data Dependence

First of all, we require CFGs to be acyclic, i.e., containing no loops. A CFG that contains loops has an infinite number of paths. Our FT construction method constructs events in the FT by walking along paths in the CFG; so a CFG with infinitely many paths could result in a FT with infinitely many events. To deal with loops, techniques such as unrolling to a given bound can be used.

As mentioned in the background section 2.2, the function $\mathit{Use}\colon V \to \mathcal{P}(\Sigma))$ associates to each vertex the set of variables referenced at the vertex. We now extend the domain of the function to include the edge predicates such that, $\mathit{Use}(p(u, v))$ is the set of all variables referenced by the predicate $p(u, v)$. In the example in Figure 2.3(B), there are the following edge predicates: $p(c_1, s_1) = x > 0$, and $p(c_1, s_2) = \neg(x > 0)$. Therefore, $\mathit{Use}(p(c_1, s_1)) = \mathit{Use}(p(c_1, s_2)) = \{x\}$.

A vertex in the CFG might have multiple inputs and outputs, but not every output is dependent on or influenced by all of the inputs. In such cases, a CFG can always be normalized so that each vertex has at most one output and zero or more inputs, and if the vertex has an output, that output is a function of all of the inputs.

Let $\underline{P}$ be the set of paths in the CFG $G = (V, E, \mathbf{start}, \mathbf{end})$. Each path P in $\underline{P}$ is a sequence of vertices $\langle v_1, v_2, ..., v_n\rangle$ with $n \geq 1$ such that, for all i with $1 \leq i < n$, there is an edge from v_i to v_{i+1}.

The set of path-sensitive definitions is

$$\mathcal{D} = \{(b, \langle v_1, v_2, ..., v_n\rangle) \mid b \in \Sigma, \langle v_1, v_2, ..., v_n\rangle \in \underline{P}, b \in \mathit{Def}(v_1), b \notin \mathit{Def}(v_i), 1 < i \leq n\}.$$

Each element $(b, \langle v_1, v_2, ..., v_n\rangle)$ of $\mathcal{D}$ implies that b is defined at v_1 and might be propagated to v_n via $\langle v_1, v_2, ..., v_n\rangle$ which is a definition-clear path with respect to b. Since we assume the CFG is acyclic and Σ is finite, we can see that $\mathcal{D}$ is also finite.

At each vertex v in the CFG, GEN_v is the set of path-sensitive definitions generated by v

$$\mathit{GEN}_v = \{(b, \langle v\rangle) \mid b \in \mathit{Def}(v)\},$$

and KILL_v is the set of path-sensitive definitions that v might kill

$$KILL_v = \{(b, P) \mid b \in Def(v), P \in \underline{P}\}.$$

To solve the PRDP to answer the question “which variable definitions reach a given point (vertex) in a system’s CFG, and which paths do the definitions take to reach that point”, we employ Cooper’s work-list algorithm as in the traditional Reaching Definitions Problem, but with the following IN and OUT sets:

- IN_v contains the path-sensitive definitions that reach v:

$$IN_v = \begin{cases} \{(b, \langle \mathbf{start} \rangle) \mid b \in \Sigma\} & \text{if } v = \mathbf{start} \\ \bigcup_{x \in OUT_u, u \in Pred(v)} g(v, x) & \text{otherwise} \end{cases}$$

  in which function $g : V \times \mathcal{D} \longrightarrow \mathcal{P}(\mathcal{D})$ is defined as follows:

$$g(v, b, \langle v_1, v_2, ..., v_n \rangle) = \begin{cases} \{(b, \langle v_1, v_2, ..., v_n, v \rangle)\} & \text{if } (v_n \to v) \in E \\ \{\} & \text{otherwise.} \end{cases}$$

- OUT_v contains the path-sensitive definitions that are available after the execution of v:

$$OUT_v = GEN_v \cup (IN_v \setminus KILL_v).$$

We can see that the PRDP problem is also an instance of a distributive data-flow framework with the semi-lattice $(\mathcal{P}(\mathcal{D}), \cup)$ and the transfer function $f : V \times \mathcal{P}(\mathcal{D}) \longrightarrow \mathcal{P}(\mathcal{D})$ as follows:

$$f(v, X) = GEN_v \cup \left(\bigcup_{x \in X} g(v, x) \setminus KILL_v \right).$$

It’s easy to see that f is distributive:

$$f(v, X \cup Y) = f(v, X) \cup f(v, Y).$$

Therefore, Cooper's work-list algorithm [11], which essentially is the Kildall's iterative algorithm [17][2], applied to the PRDP problem, is guaranteed to produce the unique maximal fixed point solution where the IN and OUT sets for each vertex converge. At that point, the IN_v set contains the path-sensitive definitions that can reach v.

Example:

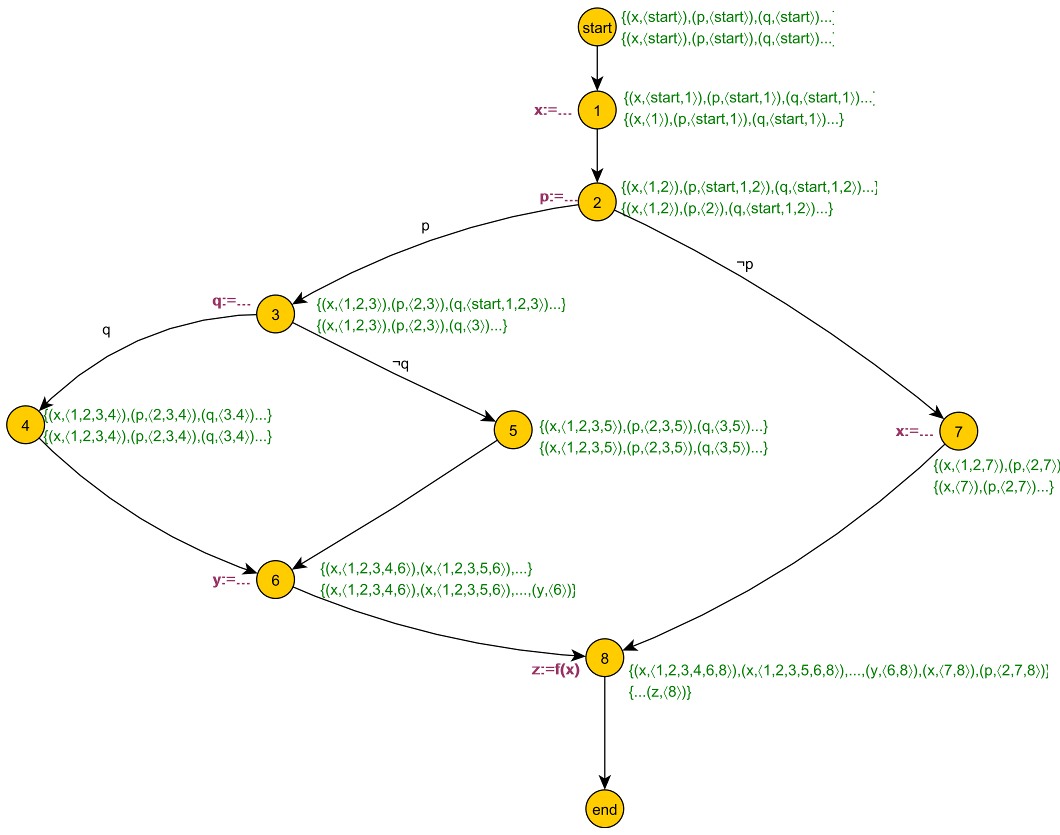

start
{(x,⟨start⟩),(p,⟨start⟩),(q,⟨start⟩)...}
{(x,⟨start⟩),(p,⟨start⟩),(q,⟨start⟩)...}
x:=...
1
{(x,⟨start,1⟩),(p,⟨start,1⟩),(q,⟨start,1⟩)...}
{(x,⟨1⟩),(p,⟨start,1⟩),(q,⟨start,1⟩)...}
p:=...
2
{(x,⟨1,2⟩),(p,⟨start,1,2⟩),(q,⟨start,1,2⟩)...}
{(x,⟨1,2⟩),(p,⟨2⟩),(q,⟨start,1,2⟩)...}
p
¬p
q:=...
3
{(x,⟨1,2,3⟩),(p,⟨2,3⟩),(q,⟨start,1,2,3⟩)...}
{(x,⟨1,2,3⟩),(p,⟨2,3⟩),(q,⟨3⟩)...}
q
¬q
4
{(x,⟨1,2,3,4⟩),(p,⟨2,3,4⟩),(q,⟨3.4⟩)...}
{(x,⟨1,2,3,4⟩),(p,⟨2,3,4⟩),(q,⟨3,4⟩)...}
5
{(x,⟨1,2,3,5⟩),(p,⟨2,3,5⟩),(q,⟨3,5⟩)...}
{(x,⟨1,2,3,5⟩),(p,⟨2,3,5⟩),(q,⟨3,5⟩)...}
x:=...
7
{(x,⟨1,2,7⟩),(p,⟨2,7⟩)...}
{(x,⟨7⟩),(p,⟨2,7⟩)...}
y:=...
6
{(x,⟨1,2,3,4,6⟩),(x,⟨1,2,3,5,6⟩),...}
{(x,⟨1,2,3,4,6⟩),(x,⟨1,2,3,5,6⟩),...,(y,⟨6⟩)}
z:=f(x)
8
{(x,⟨1,2,3,4,6,8⟩),(x,⟨1,2,3,5,6,8⟩),...,(y,⟨6,8⟩),(x,⟨7,8⟩),(p,⟨2,7,8⟩)}
{...(z,⟨8⟩)}
end

Figure 3.2. CFG Showing Path-sensitive Reaching Definitions.

[2]Both are mentioned in the Background section 2.2.

We use the same CFG example as described in the background section 2.2 but this time we will show the Path-sensitive Reaching Definitions information. In this CFG (Figure 3.2), $\Sigma = \{p, q, x, y, z\}$, x is defined at vertex 1 and vertex 7 , and it is used at vertex 8 to define z. Vertices 2 and 3 define Boolean variables p and q respectively. There is another variable y defined at vertex 6. The edges $(2 \to 3)$, $(2 \to 7)$, $(3 \to 4)$, and $(3 \to 5)$ are guarded with predicates p, $\neg p$, q, and $\neg q$ respectively. In the figure, each CFG vertex is annotated with two sets on two lines: on the first line is the *IN* set of the vertex, and on the second line is the OUT set.

Table 3.1 shows the progress of iteratively updating the *IN* and OUT sets until they converge. Below are some noteworthy example iterations:

- Showing the computation of IN_v: Suppose we want to process vertex 1 , whose immediate predecessor is **start**, and we already have

$$OUT_{\textbf{start}} = \{(b, \langle \textbf{start} \rangle) \mid b \in \Sigma\}$$

  from previous processing. Therefore, $IN_1 = \{(b, \langle \textbf{start}, 1 \rangle) \mid b \in \Sigma\}$.

- Showing the computation of IN_v: Suppose we want to process vertex 6 whose immediate predecessors are 4 and 5, and we already have $(x, \langle 1, 2, 3, 4 \rangle) \in OUT_4$ and $(x, \langle 1, 2, 3, 5 \rangle) \in OUT_5$. So the definitions coming into vertex 6 are both x being defined at vertex 1 through different paths, one through the path $\langle 1, 2, 3, 4 \rangle$, and one through the path $\langle 1, 2, 3, 5 \rangle$. We do not combine these two; we keep them separately so that the end results really show different ways a definition can reach a vertex. Therefore, both $(x, \langle 1, 2, 3, 4, 6 \rangle)$ and $(x, \langle 1, 2, 3, 5, 6 \rangle)$ are elements of IN_6.

- Showing the computation of OUT_v: Have a look at vertex 7,

$$IN_7 = \{(x, \langle 1, 2, 7 \rangle), (p, \langle 2, 7 \rangle)\} \cup \{(b, \langle \textbf{start}, 1, 2, 7 \rangle) \mid b \in \Sigma, b \neq x, b \neq p\}$$

processing	IN	OUT	work-list
start	$\{(p, \langle \textbf{start} \rangle), (q, \langle \textbf{start} \rangle), (x, \langle \textbf{start} \rangle), (y, \langle \textbf{start} \rangle), (z, \langle \textbf{start} \rangle)\}$	$\{(p, \langle \textbf{start} \rangle), (q, \langle \textbf{start} \rangle), (x, \langle \textbf{start} \rangle), (y, \langle \textbf{start} \rangle), (z, \langle \textbf{start} \rangle)\}$	1
1	$\{(p, \langle \textbf{start}, 1 \rangle), (q, \langle \textbf{start}, 1 \rangle), (x, \langle \textbf{start}, 1 \rangle), (y, \langle \textbf{start}, 1 \rangle), (z, \langle \textbf{start}, 1 \rangle)\}$	$\{(x, \langle 1 \rangle), (p, \langle \textbf{start}, 1 \rangle), (q, \langle \textbf{start}, 1 \rangle), (y, \langle \textbf{start}, 1 \rangle), (z, \langle \textbf{start}, 1 \rangle)\}$	2
2	$\{(x, \langle 1, 2 \rangle), (p, \langle \textbf{start}, 1, 2 \rangle), (q, \langle \textbf{start}, 1, 2 \rangle), (y, \langle \textbf{start}, 1, 2 \rangle), (z, \langle \textbf{start}, 1, 2 \rangle)\}$	$\{(p, \langle 2 \rangle), (x, \langle 1, 2 \rangle), (q, \langle \textbf{start}, 1, 2 \rangle), (y, \langle \textbf{start}, 1, 2 \rangle), (z, \langle \textbf{start}, 1, 2 \rangle)\}$	3,7
3	$\{(p, \langle 2, 3 \rangle), (x, \langle 1, 2, 3 \rangle), (q, \langle \textbf{start}, 1, 2, 3 \rangle), (y, \langle \textbf{start}, 1, 2, 3 \rangle), (z, \langle \textbf{start}, 1, 2, 3 \rangle)\}$	$\{(q, \langle 3 \rangle), (p, \langle 2, 3 \rangle), (x, \langle 1, 2, 3 \rangle), (y, \langle \textbf{start}, 1, 2, 3 \rangle), (z, \langle \textbf{start}, 1, 2, 3 \rangle)\}$	7,4,5
7	$\{(p, \langle 2, 7 \rangle), (x, \langle 1, 2, 7 \rangle), (q, \langle \textbf{start}, 1, 2, 7 \rangle), (y, \langle \textbf{start}, 1, 2, 7 \rangle), (z, \langle \textbf{start}, 1, 2, 7 \rangle)\}$	$\{(x, \langle 7 \rangle), (p, \langle 2, 7 \rangle), (q, \langle \textbf{start}, 1, 2, 7 \rangle), (y, \langle \textbf{start}, 1, 2, 7 \rangle), (z, \langle \textbf{start}, 1, 2, 7 \rangle)\}$	4,5,8
4	$\{(q, \langle 3, 4 \rangle), (p, \langle 2, 3, 4 \rangle), (x, \langle 1, 2, 3, 4 \rangle), (y, \langle \textbf{start}, 1, 2, 3, 4 \rangle), (z, \langle \textbf{start}, 1, 2, 3, 4 \rangle)\}$	$\{(q, \langle 3, 4 \rangle), (p, \langle 2, 3, 4 \rangle), (x, \langle 1, 2, 3, 4 \rangle), (y, \langle \textbf{start}, 1, 2, 3, 4 \rangle), (z, \langle \textbf{start}, 1, 2, 3, 4 \rangle)\}$	5,8,6
5	$\{(q, \langle 3, 5 \rangle), (p, \langle 2, 3, 5 \rangle), (x, \langle 1, 2, 3, 5 \rangle), (y, \langle \textbf{start}, 1, 2, 3, 5 \rangle), (z, \langle \textbf{start}, 1, 2, 3, 5 \rangle)\}$	$\{(q, \langle 3, 5 \rangle), (p, \langle 2, 3, 5 \rangle), (x, \langle 1, 2, 3, 5 \rangle), (y, \langle \textbf{start}, 1, 2, 3, 5 \rangle), (z, \langle \textbf{start}, 1, 2, 3, 5 \rangle)\}$	8,6
8	$\{(x, \langle 7, 8 \rangle), (p, \langle 2, 7, 8 \rangle), (q, \langle \textbf{start}, 1, 2, 7, 8 \rangle), (y, \langle \textbf{start}, 1, 2, 7, 8 \rangle), (z, \langle \textbf{start}, 1, 2, 7, 8 \rangle)\}$	$\{(z, \langle 8 \rangle), (x, \langle 7, 8 \rangle), (p, \langle 2, 7, 8 \rangle), (q, \langle \textbf{start}, 1, 2, 7, 8 \rangle), (y, \langle \textbf{start}, 1, 2, 7, 8 \rangle)\}$	6
6	$\{(q, \langle 3, 4, 6 \rangle), (q, \langle 3, 5, 6 \rangle), (p, \langle 2, 3, 4, 6 \rangle), (p, \langle 2, 3, 5, 6 \rangle), (x, \langle 1, 2, 3, 4, 6 \rangle), (x, \langle 1, 2, 3, 5, 6 \rangle), (y, \langle \textbf{start}, 1, 2, 3, 4, 6 \rangle), (y, \langle \textbf{start}, 1, 2, 3, 5, 6 \rangle), (z, \langle \textbf{start}, 1, 2, 3, 4, 6 \rangle), (z, \langle \textbf{start}, 1, 2, 3, 5, 6 \rangle)\}$	$\{(y, \langle 6 \rangle), (q, \langle 3, 4, 6 \rangle), (q, \langle 3, 5, 6 \rangle), (p, \langle 2, 3, 4, 6 \rangle), (p, \langle 2, 3, 5, 6 \rangle), (x, \langle 1, 2, 3, 4, 6 \rangle), (x, \langle 1, 2, 3, 5, 6 \rangle), (z, \langle \textbf{start}, 1, 2, 3, 4, 6 \rangle), (z, \langle \textbf{start}, 1, 2, 3, 5, 6 \rangle)\}$	8
8	$\{(y, \langle 6, 8 \rangle), (y, \langle \textbf{start}, 1, 2, 7, 8 \rangle), (q, \langle 3, 4, 6, 8 \rangle), (q, \langle 3, 5, 6, 8 \rangle), (q, \langle \textbf{start}, 1, 2, 7, 8 \rangle), (p, \langle 2, 7, 8 \rangle), (p, \langle 2, 3, 4, 6, 8 \rangle), (p, \langle 2, 3, 5, 6, 8 \rangle), (x, \langle 7, 8 \rangle), (x, \langle 1, 2, 3, 4, 6, 8 \rangle), (x, \langle 1, 2, 3, 5, 6, 8 \rangle), (z, \langle \textbf{start}, 1, 2, 3, 4, 6, 8 \rangle), (z, \langle \textbf{start}, 1, 2, 3, 5, 6, 8 \rangle) (z, \langle \textbf{start}, 1, 2, 7, 8 \rangle)\}$	$\{(z, \langle 8 \rangle), (y, \langle 6, 8 \rangle), (y, \langle \textbf{start}, 1, 2, 7, 8 \rangle), (q, \langle 3, 4, 6, 8 \rangle), (q, \langle 3, 5, 6, 8 \rangle), (q, \langle \textbf{start}, 1, 2, 7, 8 \rangle), (p, \langle 2, 7, 8 \rangle), (p, \langle 2, 3, 4, 6, 8 \rangle), (p, \langle 2, 3, 5, 6, 8 \rangle), (x, \langle 7, 8 \rangle), (x, \langle 1, 2, 3, 4, 6, 8 \rangle), (x, \langle 1, 2, 3, 5, 6, 8 \rangle),$	

Table 3.1. Path-sensitive Reaching Definitions Example. Showing the progress of the work-list algorithm.

and x is re-defined here, thus,

$$(x, \langle 1, 2, 7 \rangle) \in \mathit{KILL}_7$$

$$\mathit{GEN}_7 = \{(x, \langle 7 \rangle)\}$$

$$\therefore \mathit{OUT}_7 = \{(x, \langle 7 \rangle), (p, \langle 2, 7 \rangle)\} \cup \{(b, \langle \mathbf{start}, 1, 2, 7 \rangle) \,|\, b \in \Sigma, b \neq x, b \neq p\}.$$

Immediate Data Dependence

We define the *immediate data dependence* of variable b at vertex v to be:

- all the definitions that reach v and are used at v to define b in case $b \in \mathit{Def}(v)$, or
- all the definitions of b that reach v in case $b \notin \mathit{Def}(v)$.

Instead of keeping the path information as a sequence of vertices from the vertex where the variable is defined, we keep only the first vertex of the path and the guarded edges along that path, which are sufficient to re-construct the path if need be. We define a 1:1 mapping e as follows:

$$e : \mathcal{D} \longrightarrow L \times \omega(E)$$
$$(b, \langle v_1, v_2, ..., v_n \rangle) \longmapsto (b_{v_1}, \langle (v_{i_1} \rightarrow v_{i_1+1}), (v_{i_2} \rightarrow v_{i_2+1}) ... (v_{i_k} \rightarrow v_{i_k+1}) \rangle)$$

where

- L is a set of labels of the form b_v with $b \in \Sigma$ and $v \in V$,

- $\omega(E)$ is the set of sequences of edges,
- For all j such that, $1 \leq j \leq k$: $(v_{i_j} \to v_{i_j+1})$ is a guarded edge, and
- For all m such that, $1 \leq m \leq n$ and $m \notin \{i_1, i_2, ..., i_k\}$: $(v_m \to v_{m+1})$ is NOT a guarded edge.

Intuitively, $e(b, P)$ is a tuple of a definition of b at the first vertex of the path P and the set of all the guarded edges on that path.

The immediate data dependence of variable b at vertex v is then defined formally as follows:

$$IDD(b, v) = \begin{cases} \{e(a, P) \mid a \in Use(v),\ (a, P) \in IN(v)\} & \text{if } b \in Def(v) \\ \{e(b, P) \mid (b, P) \in IN(v)\} & \text{if } b \notin Def(v) \end{cases}$$

Since we assume a vertex has at most one output, we can write $IDD(v)$ in place of $IDD(b, v)$ in case $b \in Def(v)$. In such cases, $IDD(v)$ is the set of all definitions that reach v and used at v to define the only output b.

In the above example, x is being used at 8 to define z, and

$$IN_8 = \{(x, \langle 1, 2, 3, 4, 6, 8\rangle), (x, \langle 1, 2, 3, 5, 6, 8\rangle), (x, \langle 7, 8\rangle), ..., (y, \langle 6, 8\rangle), ...\},$$

therefore $IDD(z, 8)$ or $IDD(8) = \{(x_1, \langle(2 \to 3), (3 \to 4)\rangle), (x_1, \langle(2 \to 3), (3 \to 5)\rangle), (x_7, \langle\rangle)\}$ — only keep the definitions of x, and the path information includes the vertex there x is defined and the set of guarded edges from that vertex to vertex 8.

Failure-influencing labels and variables

Let D be a binary relation over the set of labels L such that

$$D(a_u, b_v) \Leftrightarrow \exists P \in \omega(E) : (a_u, P) \in IDD(b, v).$$

Informally, we say a_u directly influences the value of b at vertex v. The transitive closure of D on L is the relation D^+ in which $D^+(a_u, b_v)$ denotes that a_u either directly or indirectly influences the value of b at vertex v.

Recall that a system failure is defined to be an input or output variable of a specific vertex in the system's CFG receiving an incorrect value. For example, "b output from v is incorrect", or "b input to v is incorrect". We call v the *failure vertex*. If $D^+(a_u, b_v)$, we call a_u a *failure-influencing label* and a a *failure-influencing variable*.

3.2.1.2 Deriving Fault Trees using Immediate Data Dependence and Control Dependence Information

Our approach to FT construction is independent of the system modeling language used as long as the language incorporates sufficient control and data flow information, so that the CFG extracted from the system model complies with the requirements of CFGs we have mentioned thus far, including having unique **start** and **end** vertices, being acyclic, having edge predicates on outgoing edges of vertices with more than one outgoing edges, having *Def* associated with each vertex, having *Use* associated with each vertex and each edge predicate[3].

In our approach, a system failure (top event) is specified as either an output from a vertex being incorrect, or a variable input to a vertex being incorrect.

To derive the FT, we start with the top event as an intermediate event and repeatedly apply appropriate templates to elaborate intermediate events until all the leaves of the resulting FT are primary events. Each template elaborates an intermediate event and results in a *partial FT*. The intermediate event being elaborated is then

[3]We also assume that the checking of edge predicates are done correctly; our approach does not consider faults caused by incorrect evaluation of edge predicates. If one wants to consider such faults, one can add to the CFG an extra vertex whose output is the evaluation of the predicate. See further discussion in section 3.2.1.8.

replaced with the partial FT. Any intermediate events in the partial FT are then elaborated using appropriate templates.

So given the CFG, the immediate data dependence, the control dependence information, and the top event, Algorithm 3.1 is used to derive the initial FT.

```
input  : CFG, IDD, CD, topEvent
output: FT
visitedEvents ⟵ {};
worklist ⟵ {topEvent };
while worklist is not empty do
    Remove event e from worklist;
    if visitedEvents does not contain e then
        Add e to visitedEvents;
        Find appropriate template (elaboration procedure) t for e;
        partial-fault-tree ⟵ elaborate e using t;
        Replace e with the partial-fault-tree;
        foreach leaf event $e'$ in the partial-fault-tree do
            if $e'$ is not a primary event then
                Add $e'$ to worklist;
// at this point topEvent is a fully derived FT
Return topEvent;
```

Algorithm 3.1: FT Derivation.

As stated in the Background section 2.1, one FT event can be an input event of different gates. In the process of elaborating immediate events using templates, we might encounter an intermediate event that is a leaf of the partial FT from elaborating an event e_1, and it is also a leaf of the partial FT from elaborating an event e_2. In this algorithm, the intermediate event is only elaborated once thanks to the use of the set `visitedEvents`.

The remainder of this section shows the simple templates to derive the initial FTs based on the immediate data dependence and control dependence information.

3.2.1.3 Template DF-1: "b output from v is incorrect"

Let v be a vertex in the CFG, and b be an output of v, i.e., $b \in Def(v)$. The value of b output from v is considered incorrect only if any of the followings happens:

1. Any of the inputs a into v is incorrect[4], we call this a *data dependence error*,

2. The execution incorrectly reaches v, we call this a *control dependence error*,

3. The computation at v is done incorrectly, we call this an *agent error*.

By "the execution incorrectly reaches v", we mean that on the execution path from **start** to v, the execution takes an incorrect branch because of some error (one of the three types mentioned above), while if taking the correct branch, the execution might not reach v at all or the execution reaches v carrying a different input to v. Our approach aims to identify such errors. We discuss this more fully when describing Template DF-5 in section 3.2.1.7 below.

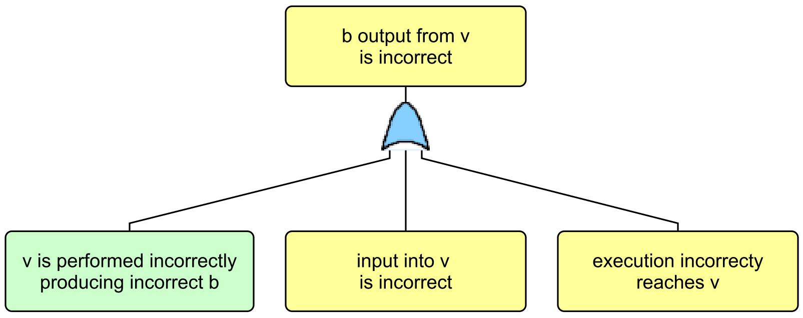

b output from v is incorrect
v is performed incorrectly producing incorrect b
input into v is incorrect
execution incorrecty reaches v

Figure 3.3. Template DF-1 For Event Type *"b output from v is incorrect"* When $b \in Def(v)$.

The FT template DF-1 in Figure 3.3 reflects the above mentioned relationship. In that template, the event *"v is performed incorrectly producing incorrect b"*, depicted by the green box, is a primary event; it is not further elaborated. In the context of process model analysis, it can be interpreted as an agent's error where the agent can be a human, software, or a hardware device performing step v. The other events, *"input into v is incorrect"* and *"execution incorrectly reaches v"* will be further elaborated as discussed in the next sections 3.2.1.4 and 3.2.1.7 respectively.

[4]We assume that each vertex has at most one output and the output depends on all of the inputs.

Algorithm B.1 in the Appendix shows the pseudo-code of the procedure to elaborate the intermediate event *"b output from v is incorrect"*. The input is the intermediate event to be elaborated. The algorithm creates a new gate for the intermediate event, elaborates the gate and produces as the result the partial fault tree according to the template. The algorithm's output is the list of any newly generated intermediate events of the partial fault tree.

3.2.1.4 Template DF-2: "input to v is incorrect"

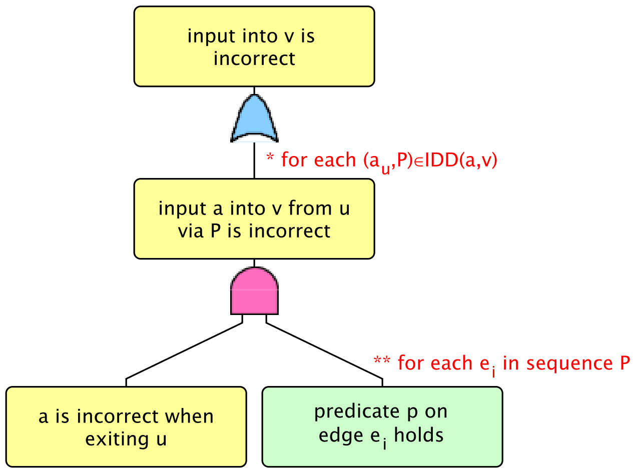

input into v is incorrect
* for each (a_u,P)∈IDD(a,v)
input a into v from u via P is incorrect
** for each e_i in sequence P
a is incorrect when exiting u
predicate p on edge e_i holds

Figure 3.4. Template DF-2 For Event Type *"input to v is incorrect"* When $b \in Def(v)$. The symbol * indicates that one event is added for each $(a_u, P) \in IDD(b, v)$ and the symbol ** indicates that one event is added for each e_i in the sequence $P = \langle e_1, \ldots, e_n \rangle$.

Using immediate data dependence information, the event *"input into v is incorrect"* can be elaborated as in Template DF-2 in Figure 3.4. For each $(a_u, P) \in IDD(v)$, there is one event *"input a into v from u is incorrect"*, which in turn is a combination of *"a is incorrect when exiting u"* and all the predicates on the guarded edges of P (guiding the control from u to v) being true. Note that here we use *"a is incorrect when exiting u"* instead of *"a output from u is incorrect"* to take care of the cases that u is the **start** vertex or a is not an output of u. Template DF-3 (see section 3.2.1.5) is in place to take care of these different cases.

In this elaboration, the event *"predicate p on edge e_i holds"* is a primary event, while the event *"a is incorrect when exiting u"* is an intermediate event which is elaborated using Template DF-3 below.

Algorithm B.2 in the Appendix shows the pseudo-code of the procedure to elaborate the intermediate event *"input into v is incorrect"*.

3.2.1.5 Template DF-3: "b is incorrect when exiting v"

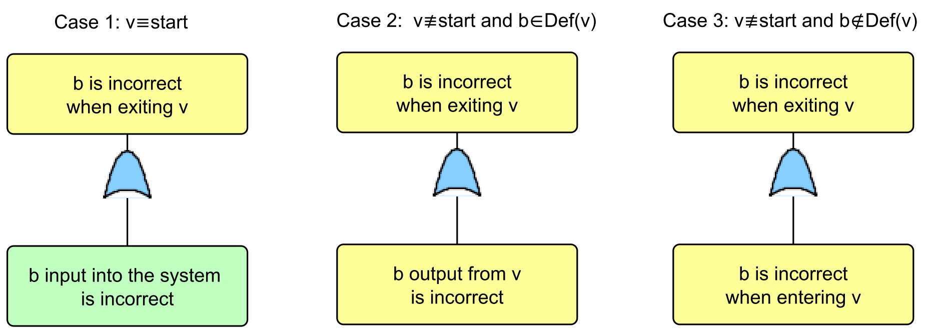

Case 1: v≡start
b is incorrect when exiting v
b input into the system is incorrect
Case 2: v≢start and b∈Def(v)
b is incorrect when exiting v
b output from v is incorrect
Case 3: v≢start and b∉Def(v)
b is incorrect when exiting v
b is incorrect when entering v

Figure 3.5. Template DF-3 For Event Type *"b is incorrect when exiting v"*.

Figure 3.5 shows the elaboration of the event type *"b is incorrect when exiting v"*. It depends on whether v is the **start** vertex and whether b is an output of v.

- Case 1: v is the **start** vertex. In this case, b is defined outside of the system, it is just an input into the system, therefore, the event *"b is incorrect when exiting v"* is elaborated into the primary event *"b input to the system is incorrect"*[5].

- Case 2: v is not the **start** vertex and b is an output of v. In this case, the execution of v does affect the value of b. Thus, the event *"b is incorrect when exiting v"* is elaborated into the intermediate event *"b output from v is incorrect"*, which can be further elaborated using Template DF-1 (see section 3.2.1.3).

[5]In Template DF-3, each OR gate has only one input event; it basically means the output event occurs if the input event occurs. AND gates could also be used here since there is only one input event. We choose to use OR gates.

- Case 3: v is not the **start** vertex and b is not an output of v. In this case, the execution of v does not affect the value of b, thus b must have been incorrect when entering v. Thus, the event *"b is incorrect when exiting v"* is elaborated into the intermediate event *"b is incorrect when entering v"*, which can be further elaborated using Template DF-4 (see section 3.2.1.6).

Algorithm B.3 in the Appendix shows the pseudo-code of the procedure to elaborate the intermediate event *"b is incorrect when exiting v"*.

3.2.1.6 Template DF-4: "b is incorrect when entering v"

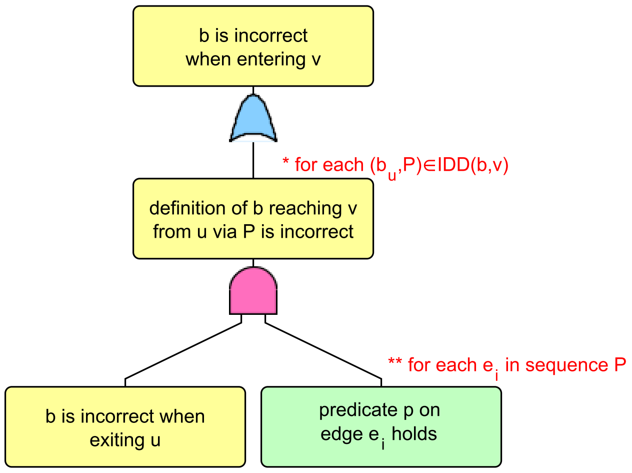

b is incorrect when entering v
* for each (b_u,P)∈IDD(b,v)
definition of b reaching v from u via P is incorrect
** for each e_i in sequence P
b is incorrect when exiting u
predicate p on edge e_i holds

Figure 3.6. Template DF-4 For Event Type *"b is incorrect when entering v"*. The symbol * indicates that one event is added for each $(b_u, P) \in IDD(b, v)$ and the symbol ** indicates that one event is added for each e_i in the sequence $P = \langle e_1, \ldots, e_n \rangle$.

In order to elaborate the event *"b is incorrect when entering v"*, we also use the data dependence information $IDD(b, v)$ as in template DF-2. Template DF-4 shown in Figure 3.6 is essentially the same as template DF-2 shown in Figure 3.4.

Algorithm B.4 in the Appendix shows the pseudo-code of the procedure to elaborate the event *"b is incorrect when entering v"*.

3.2.1.7 Template DF-5: "execution incorrectly reaches v"

We use the control dependence information computed for each CFG vertex v to elaborate the event *"execution incorrectly reaches v"*.

Recall that $CD(v)$ denotes the set of edges that v is control dependent on:

$$CD(v) \overset{\text{def}}{=} \{(u \to t) \,|\, v \text{ is control dependent on } (u \to t)\}.$$

So, for each $(u \to t)$ in $CD(v)$ there are two cases that allow the execution to incorrectly reach v from u via the edge $(u \to t)$ as shown in template DF-5 in Figure 3.7:

1. The execution incorrectly reaches u first, and then from there reaches v through edge $(u \to t)$ with predicate $p(u, t)$;

2. The value of the predicate $p(u, t)$ deciding the path from u to v is incorrect.

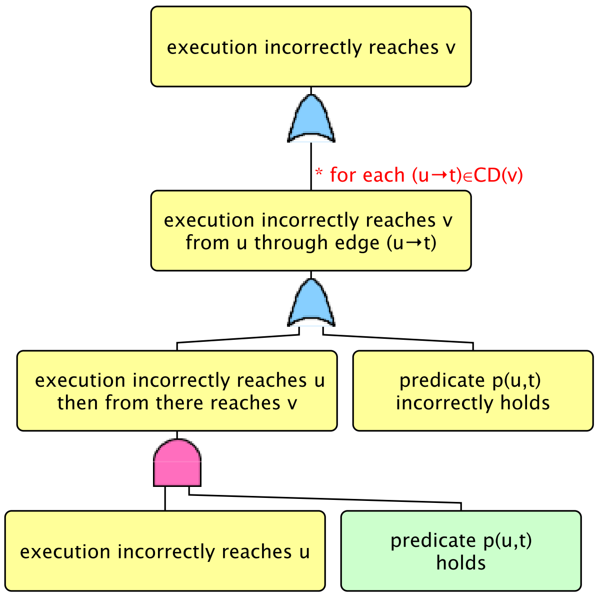

execution incorrectly reaches v
* for each (u→t)∈CD(v)
execution incorrectly reaches v from u through edge (u→t)
execution incorrectly reaches u then from there reaches v
predicate p(u,t) incorrectly holds
execution incorrectly reaches u
predicate p(u,t) holds

Figure 3.7. Template DF-5 For Event Type *"execution incorrectly reaches v"*. The Kleene star * indicates one event is added for each $(u \to t) \in CD(v)$.

Algorithm B.5 in the Appendix shows the pseudo-code of the procedure to elaborate the intermediate event *"execution incorrectly reaches v"*.

3.2.1.8 Template DF-6: "predicate $p(u, t)$ incorrectly holds"

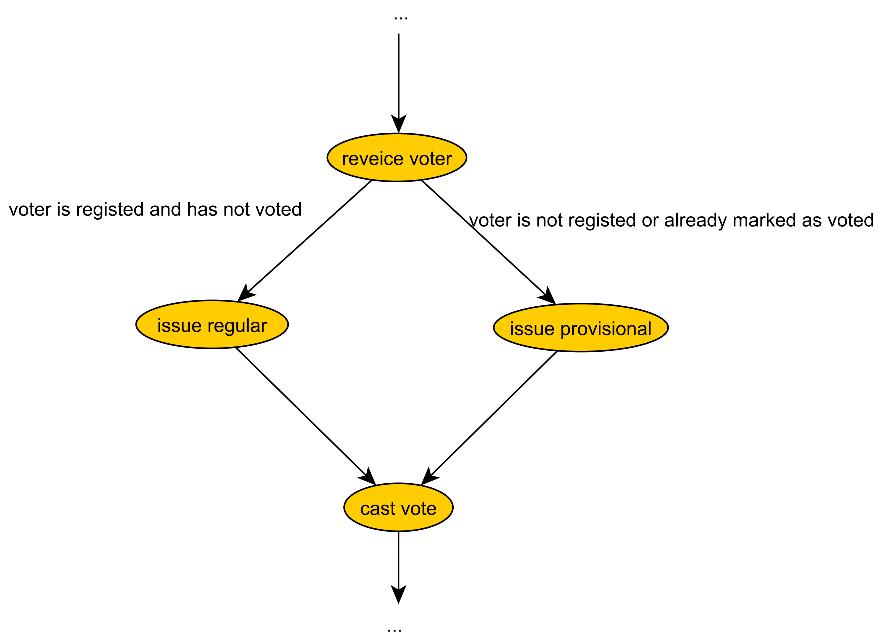

...
reveice voter
voter is registed and has not voted
voter is not registed or already marked as voted
issue regular
issue provisional
cast vote
...

Figure 3.8. An Example CFG Of "Issue ballot and get vote".

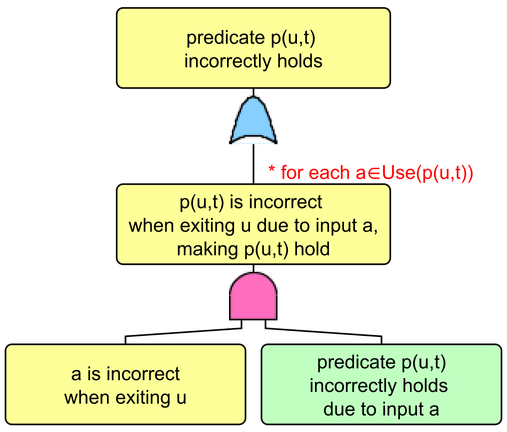

predicate p(u,t) incorrectly holds
* for each a∈Use(p(u,t))
p(u,t) is incorrect when exiting u due to input a, making p(u,t) hold
a is incorrect when exiting u
predicate p(u,t) incorrectly holds due to input a

Figure 3.9. Template DF-6 For Event Type *"predicate p(u,t) incorrectly holds"*. The Kleene star * indicates one event is added for each $a \in Use(p(u, t))$.

We assume that in the system model, the edge predicates (on guarded edges) always get evaluated correctly. We do not take predicate evaluations into account as places for potential errors. If a predicate could be evaluated incorrectly, it has to be treated as a variable that is output from a vertex. For example, in the CFG in Figure 2.3(B), the predicate $p(c_1, c_1)$ is $x > 0$; we assume the evaluation of $x > 0$ is always correct, thus the value of $p(c_1, c_1)$ is only incorrect if x is incorrect. Another example is in the election process as described in the Background section 2.4, to issue an appropriate ballot (regular or provisional) the election official has to check for the voter's eligibility. If the voter is registered and has not marked as voted, the election official issues a regular ballot, otherwise the election official issues a provisional ballot. A CFG as shown in Figure 3.8 reflects such control flow. However, we assume that the predicate evaluations are always done correctly, therefore if we use that CFG, the errors in the predicate evaluations will not be identified. It is possible that the election official does the check incorrectly (intentionally or not). So in order for such error to be identified using our approach, the predicate has to be treated as output variables (Boolean variables `registered` output from 'verify register' and `notVoted` output from 'verify not-voted') as shown later in section 3.3.

With that assumption, we see that the predicate $p(u, t)$ incorrectly holds only if any variable used in the predicate is incorrect. Figure 3.9 shows template DF-6 to elaborate the event *"predicate $p(u, t)$ incorrectly holds"*. The elaboration pseudo-code is given in Algorithm B.6 in the Appendix.

3.2.2 SP Set: Simple Templates To Derive Initial Fault Trees

In this section, we show another set of templates as an equivalent way to generate the initial fault tree. In the DF template set, for data dependence errors, the templates (DF-2 and DF-4) use the IDD information to go directly to the places in the CFG where the variables-in-question are defined; these places could be many steps away from the place where the variable is being used. In contrast to using IDD as in the DF template set, the templates SP-2 and SP-4 in the SP set do not use IDD information, they instead go back through the CFG, one step at the time. We call this set of templates the *SP* set[6].

The control dependence information usage in this SP set is still the same as in the DF-set.

There are two reasons for using the templates described in this section:

1. Using this set of templates, we do not have to compute the immediate data dependence information ahead of time. It is an advantage when the system failure of interest is associated with a vertex "close" to the beginning of the process; in this case, the part of the CFG to be explored is much smaller than the entire CFG.

2. We can leverage the previous implementation of FTA on Little-JIL process models, developed by Bin Chen [3]. Chen uses templates that trace the data flow on the CFG, one vertex back at a time.

We prove that this SP set of templates generates FTs that are equivalent to those produced by the templates in the DF set described earlier in section 3.2.1. By that we mean given the same top event, the FT derived using the SP template set has the same cut sets as the FT derived using the DF template sets. The induction proof is presented in Appendix A.

Table 3.2 shows the difference between the two sets of templates. The SP set also uses some templates from the DF set, namely DF-1, DF-3, DF-5, DF-6 but it has its own templates SP-2 and SP-4 to replace DF-2 and DF-4 respectively.

3.2.2.1 Template SP-2: "input to v is incorrect"

Template SP-2 (Figure 3.10) is self-explanatory. We just add one event *"a is incorrect when entering v"* for each input $a \in \mathit{Use}(v)$.

[6]SP stands for Simple.

Template in DF set	Event	Template in SP set	Different templates between two sets
DF-1	b output from v is incorrect	DF-1	No
DF-2	**input to v is incorrect**	**SP-2**	**Yes**
DF-3	b is incorrect when exiting v	DF-3	No
DF-4	**b is incorrect when entering v**	**SP-4**	**Yes**
DF-5	execution incorrectly reaches v	DF-5	No
DF-6	predicate p(u,t) is incorrect	DF-6	No

Table 3.2. Templates In DF Set And SP Set.

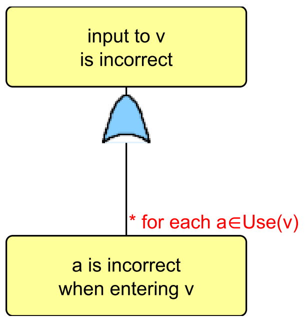


Figure 3.10. Template SP-2 For Event Type *"input to v is incorrect"*. The Kleene star * indicates one event is added for each $a \in Use(v)$.

3.2.2.2 Template SP-4: "b is incorrect when entering v"

The value of b does not change when the control flows from vertex $u \in Pred(v)$ to v. If there is a predicate $p(u, v)$ on the edge $(u \rightarrow v)$, then we include that condition on our FT template SP-4, as shown in Figure 3.11. Note that this is a *simple* template in which we focus on one single error at a time. Therefore the event *"predicate $p(u, v)$ holds"* is considered a primary event and there is no further elaboration. Later in the analysis, if users choose to elaborate on this part of the FT, such events will be converted to intermediate events and thus will be further elaborated.

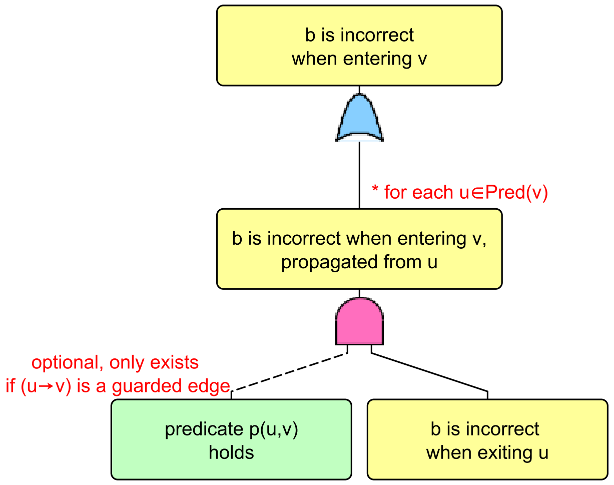

b is incorrect when entering v
* for each u∈Pred(v)
b is incorrect when entering v, propagated from u
optional, only exists if (u→v) is a guarded edge
predicate p(u,v) holds
b is incorrect when exiting u

Figure 3.11. Template SP-4 For Event Type *"b is incorrect when entering v"*. The Kleene star * indicates one event is added for each $u \in Pred(v)$.

Error events

In a cut set, only events of the following types are considered *error events*:

- v is performed incorrectly producing incorrect a,
- a input to the system is incorrect.

Events of other types are not considered error events. For example, "v produces incorrect b due to input a" is rather a consequence event, or "predicate $p(u, t)$ holds" is a boolean condition.

3.3 "Issue ballot and get vote" Example

This section describes an example of analyzing a small model of an "Issue ballot and get vote" (IB) system, from showing the derivation of the initial FT using the SP template set to showing the need of incremental selective exploration of the cut sets. This example system model is used in the following sections to keep illustrating our approach.

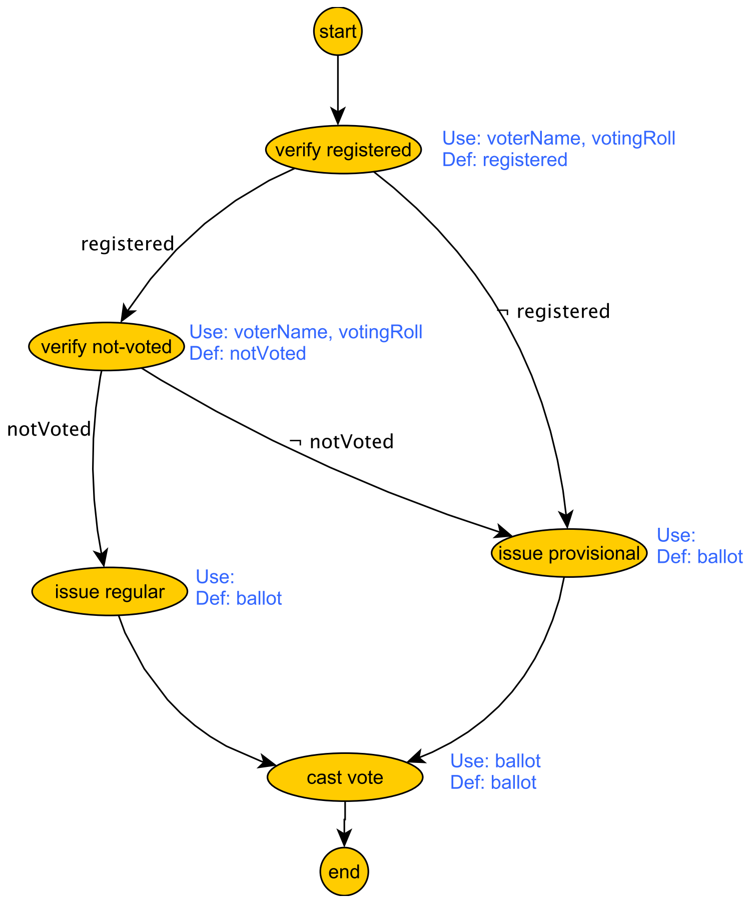


Figure 3.12. CFG Annotated With Data Flow Information Of "Issue ballot and get vote" System.

Figure 3.12 shows a CFG representation of "Issue ballot and get vote", a very much simplified portion of an election system that we already described in the background section 2.4. When voters go to their polling place to vote, they are checked for eligibility. In this model, the voter has to go through two checks. The first is to verify that the voter is registered by checking if the voter name is in the voting roll (represented by vertex 'verify registered'). So the inputs to this vertex are `voterName` and `votingRoll`, and the output is a Boolean `registered` indicating whether the voter is registered or not. Both `voterName` and `votingRoll` are inputs to (this part of) the system, they are not defined within the system. If the voter passes the first check, then the second check 'verify not voted' is done to verify that the voter has not [already] voted. The inputs to this vertex are also `voterName` and `votingRoll`. The outputs include: (1) a Boolean `notVoted` indicating whether the voter has not voted (passes the check) or already voted (fails the check); and (2) `votingRoll` updated with the information that the voter has voted.

If either of the checks fails, the control flows to 'issue provisional' — the voter is given a provisional ballot, which is typically not examined unless the election is close. If the voter passes both checks, the system control flows to 'issue regular' — the voter is given a regular ballot.

The voter, after being issued an appropriate ballot, casts the vote (the control flows to 'cast vote').

Let us consider a system failure *"*`ballot` *input into 'cast vote' is incorrect"*. It could be an eligible voter casting his/her vote using an provisional ballot, or an ineligible voter casting his/her vote using a regular ballot. The failure can be translated to *"*`ballot` *is incorrect when entering 'cast vote'"* so we can apply our templates.

We apply the simple templates in the SP set described in section 3.2.2 to derive the initial FT. Starting at the top event *"*`ballot` *is incorrect when entering 'cast vote'"*, template SP-4 is applied to elaborate the event, resulting in two new intermediate events *"*`ballot` *is incorrect when exiting 'issue regular'"* and *"*`ballot` *is incorrect when exiting 'issue provisional'"*. Note that since there are no predicates on the edges from 'issue regular' and 'issue provisional' to 'cast vote', each AND gate has only one input event (Figure 3.13).

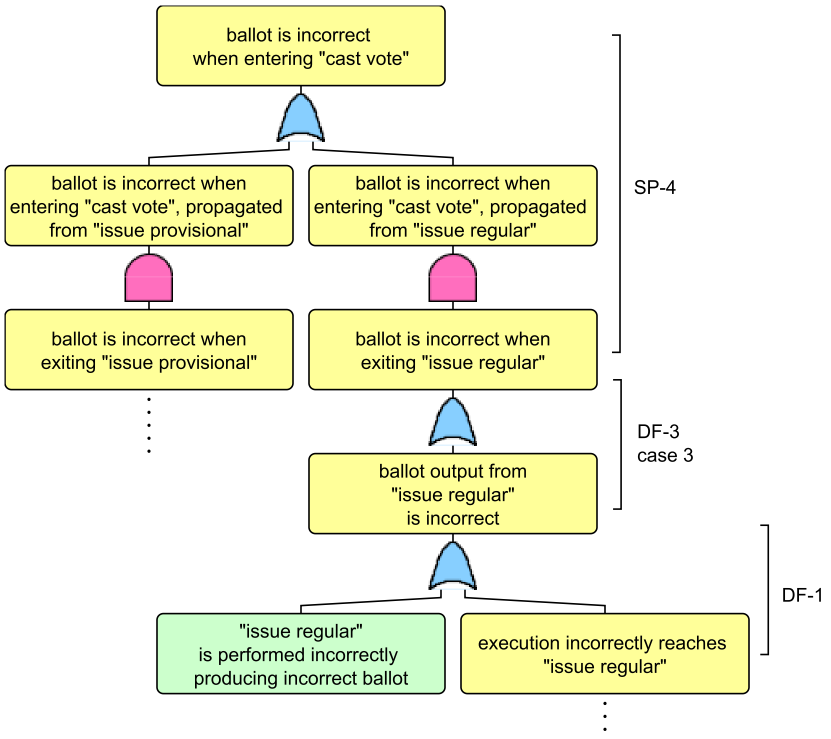

ballot is incorrect when entering "cast vote"
ballot is incorrect when entering "cast vote", propagated from "issue provisional"
ballot is incorrect when entering "cast vote", propagated from "issue regular"
SP-4
ballot is incorrect when exiting "issue provisional"
ballot is incorrect when exiting "issue regular"
DF-3 case 3
ballot output from "issue regular" is incorrect
DF-1
"issue regular" is performed incorrectly producing incorrect ballot
execution incorrectly reaches "issue regular"

Figure 3.13. Applying Simple Templates to "Issue ballot and get vote" System Model Given the Failure "`ballot` *is incorrect when 'cast vote' starts*".

Template DF-3-case-3 is applied to elaborate the event *"`ballot` is incorrect when exiting 'issue regular'"* since `ballot` is an output of 'issue regular', and 'issue regular' is not the **start** vertex. This results in the intermediate event *"`ballot` output from 'issue regular' is incorrect"*, which is further elaborated using template DF-1. Since 'issue regular' does not have any input, there are only two cases: one is *"'issue regular' is performed incorrectly and produces incorrect* `ballot`*"*, and the other is *"execution incorrectly reaches 'issue regular'"*.

We keep applying the templates in the SP-set to elaborate intermediate events until all the leaves of the resulting FT are primary events[7]. Figure 3.14 shows the resulting initial FT. Note that the FT is in fact not exactly a tree structure as we already mentioned in the Background section 2.1; there are several FT events each of which is an input event to more than one gate.

Once the FT is derived, Boolean algebra is applied to compute its cut sets as described in the Background section 2.1. There are 18 cut sets in this example, 3 of which are found to be spurious using our technique discussed later in section 3.6. The remained 15 cut sets are all minimal with one or two error events. Besides obvious single-event cut sets such as

CS-1:

- 'issue regular' is performed incorrectly producing incorrect ballot

there are more complicated cut sets, e.g.,

[7]We could of course apply the templates in the DF set and would get an equivalent initial fault tree. In order to do that we would have to compute in the IDD information before applying the DF templates.

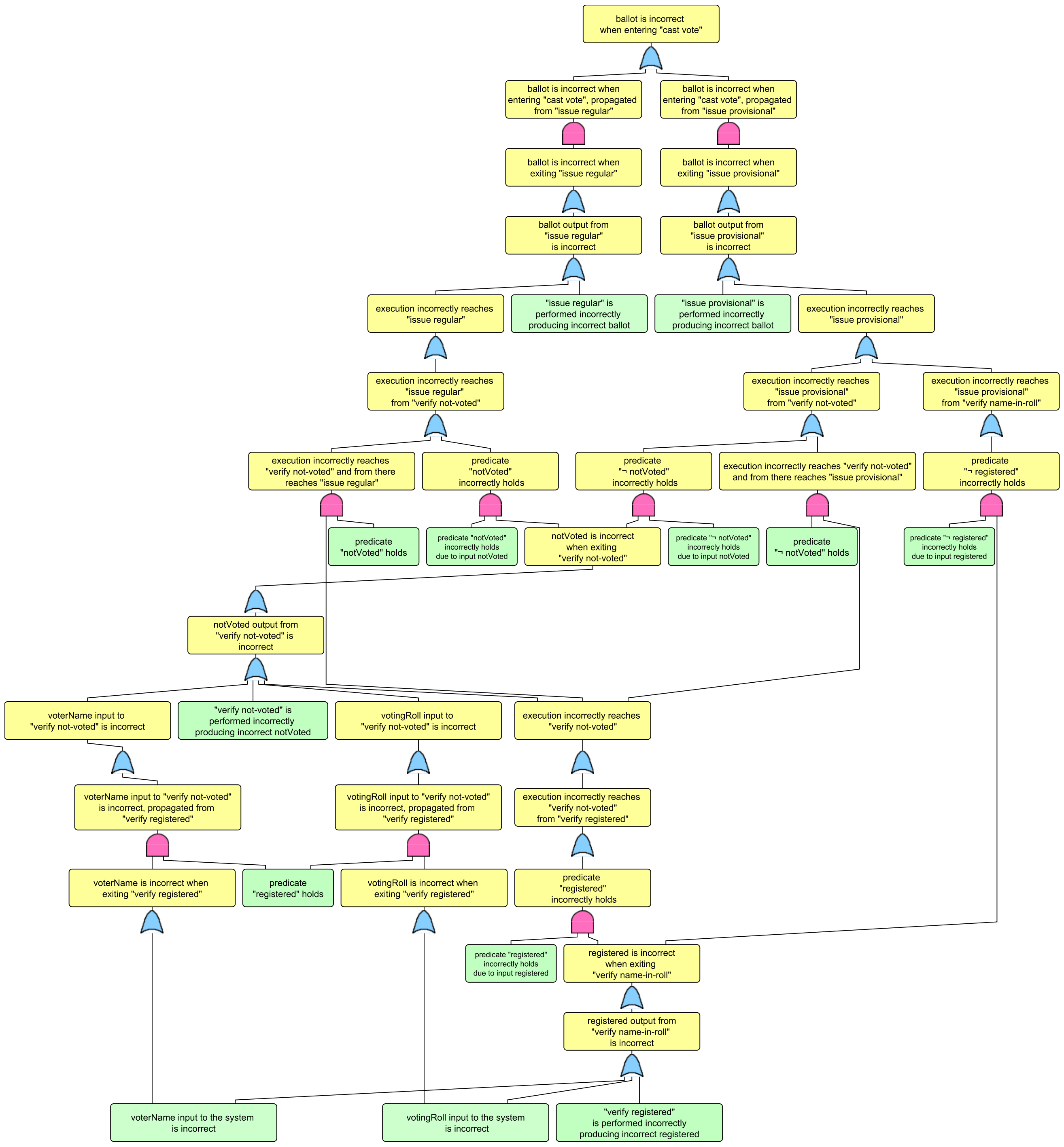

ballot is incorrect when entering "cast vote"
ballot is incorrect when entering "cast vote", propagated from "issue regular"
ballot is incorrect when entering "cast vote", propagated from "issue provisional"
ballot is incorrect when exiting "issue regular"
ballot is incorrect when exiting "issue provisional"
ballot output from "issue regular" is incorrect
ballot output from "issue provisional" is incorrect
execution incorrectly reaches "issue regular"
"issue regular" is performed incorrectly producing incorrect ballot
"issue provisional" is performed incorrectly producing incorrect ballot
execution incorrectly reaches "issue provisional"
execution incorrectly reaches "issue regular" from "verify not-voted"
execution incorrectly reaches "issue provisional" from "verify not-voted"
execution incorrectly reaches "issue provisional" from "verify name-in-roll"
execution incorrectly reaches "verify not-voted" and from there reaches "issue regular"
predicate "notVoted" incorrectly holds
predicate "¬ notVoted" incorrectly holds
execution incorrectly reaches "verify not-voted" and from there reaches "issue provisional"
predicate "¬ registered" incorrectly holds
predicate "notVoted" holds
predicate "notVoted" incorrectly holds due to input notVoted
notVoted is incorrect when exiting "verify not-voted"
predicate "¬ notVoted" incorrectly holds due to input notVoted
predicate "¬ notVoted" holds
predicate "¬ registered" incorrectly holds due to input registered
notVoted output from "verify not-voted" is incorrect
voterName input to "verify not-voted" is incorrect
"verify not-voted" is performed incorrectly producing incorrect notVoted
votingRoll input to "verify not-voted" is incorrect
execution incorrectly reaches "verify not-voted"
voterName input to "verify not-voted" is incorrect, propagated from "verify registered"
votingRoll input to "verify not-voted" is incorrect, propagated from "verify registered"
execution incorrectly reaches "verify not-voted" from "verify registered"
voterName is incorrect when exiting "verify registered"
predicate "registered" holds
votingRoll is incorrect when exiting "verify registered"
predicate "registered" incorrectly holds
predicate "registered" incorrectly holds due to input registered
registered is incorrect when exiting "verify name-in-roll"
registered output from "verify name-in-roll" is incorrect
voterName input to the system is incorrect
votingRoll input to the system is incorrect
"verify registered" is performed incorrectly producing incorrect registered

Figure 3.14. Initial FT Derived From "Issue ballot and get vote" System Model Given The Top Event *"ballot is incorrect entering 'cast vote'"*.

CS-2:

- ‘verify not-voted’ is performed incorrectly producing incorrect `notVoted`
- predicate ‘notVoted’ incorrectly holds due to input `notVoted`

One can understand from this cut set CS-2 that the Boolean `notVoted` has the value `true`, but this value is incorrect because of the incorrect performance of the ‘verify not-voted’ step, implying that the voter is in fact already listed in the voting roll as having voted. But it is not immediately understandable how the system execution reaches the step ‘verify not-voted’ in the first place to produce the incorrect value of `notVoted` and how this incorrect value of `notVoted` is related to the hazard. Our new approach allows users to *zoom in* on a specific cut set by

- creating a projection of the initial FT that keeps only events relevant to this cut set, resulting in what we call the *focused FT*;
- applying detailed templates to this projected FT to derive a more detailed FT, called *elaborated FT*; and then
- automatically computing the new cut sets of the elaborated FT and generating concrete scenarios showing ways that the events in the cut sets can occur and how they can then lead to the failure.

The next section discusses the projection of a FT given a cut set to derive the focused FT.

3.4 Deriving The Focused Fault Tree

Given a FT and one of its cut sets, the focused FT is a subtree of the original FT, and the set of all the leaves of the subtree is exactly the given cut set.

For example, given the FT in Figure 2.1 and a cut set $\{E, F\}$, the focused FT is the subtree as shown in Figure 3.15(B) — Figure 3.15(A) shows the original FT with the irrelevant parts grayed out, so the reader has can see easily that the focused FT is a subtree of the original; Figure 3.15(B) shows that subtree alone, only laid out differently due to the graph layout algorithm of the yEd tool we use[8].

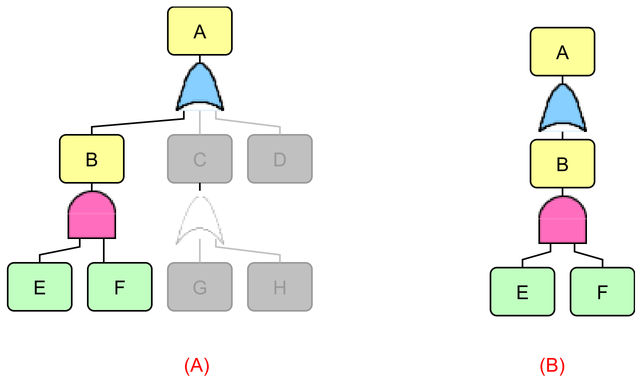


Figure 3.15. Focused Fault Tree Example.

To derive the focused FT, we first mark all the primary events that are contained in the given cut set as *relevant*. We traverse the tree from the root, recursively visit each event's input gate and each gate's input events. After all of the input events of a gate are visited, if (1) the gate is an OR gate and one of its events is relevant, or (2) the gate is an AND gate and all of its input events are relevant, then mark the gate as relevant. After an event's input gate is visited, if the gate is marked as relevant, then mark the event as relevant. Once all the marking is done, we can get rid of all the gates and events that not marked as relevant. The remainder is the focused FT with respect to the given cut set. By construction, the events and gates that are not marked relevant are not relevant to showing how the primary events in the cut sets are combined (via the Boolean gates) to lead to the top event.

[8]http://www.yWorks.com

Figure 3.16 shows the focused FT given the cut set CS-2. This focused FT is the projection of the initial FT derived from "Issue ballot and get vote" model given the failure *"`ballot` is incorrect when 'cast vote' starts"*, focusing on the cut set CS-2.

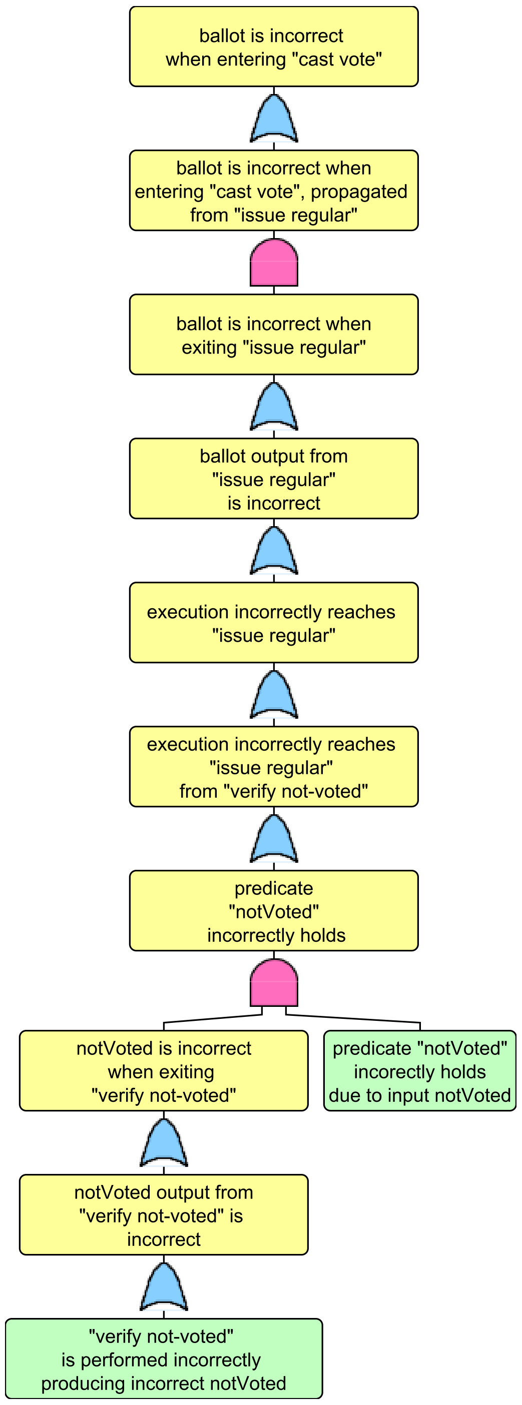

ballot is incorrect when entering "cast vote"
ballot is incorrect when entering "cast vote", propagated from "issue regular"
ballot is incorrect when exiting "issue regular"
ballot output from "issue regular" is incorrect
execution incorrectly reaches "issue regular"
execution incorrectly reaches "issue regular" from "verify not-voted"
predicate "notVoted" incorrectly holds
notVoted is incorrect when exiting "verify not-voted"
predicate "notVoted" incorectly holds due to input notVoted
notVoted output from "verify not-voted" is incorrect
"verify not-voted" is performed incorrectly producing incorrect notVoted

Figure 3.16. Focused Fault Tree Given Cut Set CS-2.

With the cut set CS-2, the focused FT reveals that because `notVoted` is incorrectly true, the system execution incorrectly reaches 'issue regular', therefore the output `ballot` from that vertex is considered incorrect. But the user might then want to understand better how the execution can ever reach 'verify not-voted' in the first place. We facilitate the understanding by applying our new detailed templates to derive the focused-elaborated FT. The next section discusses the elaboration of the focused FT using detailed templates.

3.5 Zooming In On An Initial Cut Set With Detailed Templates

In this second phase of the analysis, we elaborate the focused FT to an elaborated FT (see the flow chart in Figure 3.1.) The key idea for this elaboration is threefold: (1) considering how the execution reaches the original source of error, (2) elaborating events that are considered primary in the initial FT, and (3) considering the combinations of errors.

First, in deriving the initial FT, the SP templates allow users to trace back through the control flow graph to where a data variable b first gets an incorrect value, let's say vertex v; it could be because an input a into v is incorrect, so the elaboration proceeds further back to where a first gets the incorrect value; it could be because the value of b input to the system is incorrect (if v is the **start** vertex); it could be because the execution incorrectly reaches v so output b from v is considered incorrect; or it could be because v is performed incorrectly, therefore producing an incorrect value of b. For the latter case, no more further elaboration is done in deriving the initial FT. In elaborating the focused FT, when we encounter the situation where v is performed incorrectly producing incorrect b, we are interested in how the execution reaches v as well.

Second, the event "predicate $p(u, t)$ holds" which is considered a primary event in the initial FT, is now considered an intermediate event in the elaborated FT. It is elaborated by applying a new template.

Third, in deriving the initial FT, the SP templates consider only single errors as input events to an OR gate, causing its output event. In elaborating the focused FT, we consider combinations of errors, that is, sometimes two or more error events occur together causing the output event.

To achieve the above three goals, we develop a new set of templates called DT — detailed template set. Below, we describe how this DT template set is different from the SP set: what SP templates can be used in the DT set, what SP templates have to be replaced by their DT counterparts, and what detailed templates are newly added to the DT set.

Table 3.3 shows the difference between the two sets of templates SP (simple) and DT (detailed). The DT set also uses some templates from the SP set, namely DF-3 and SP-4 but it has its own templates DT-1, DT-2 and DT-6 to replace DF-1, SP-2 and DF-6 respectively. In addition, the DT set also has two new templates: DT-7 to elaborate events of type *"predicate p(u,t) holds"* and DT-8 to elaborate *"execution correctly reaches v"*. Both of those events are not applicable in deriving the initial FT using the simple templates but they appear in elaborating the focused FT as shown later in this section.

Template in SP set	Event	Template in DT set	Different templates between two sets
DF-1	**b output from v is incorrect**	**DT-1**	**Yes**
SP-2	**input to v is incorrect**	**DT-2**	**Yes**
DF-3	b is incorrect when exiting v	DF-3	No[9]
SP-4	b is incorrect when entering v	SP-4	No[10]
DF-5	execution incorrectly reaches v	DF-5	No[11]
DF-6	**predicate $p(u,t)$ incorrectly holds**	**DT-6**	**Yes**
N/A	**predicate $p(u,t)$ holds**	**DT-7**	**Yes**
N/A	**execution correctly reach v**	**DT-8**	**Yes**

Table 3.3. Templates In DT Set And SP Set.

[9]There is no error combination in DF-3.

[10]SP-4 (Figure 3.11) elaborates the event of type *"b is incorrect when entering v"* by listing all possible events of type *"b is incorrect when exiting u"* where $u \in Pred(v)$. In one failure-enabling scenario, the data item b can only flow to v from one predecessor u, it cannot be the case that *"b is incorrect when exiting u_1"* and *"b is incorrect when exiting u_2"* at the same time, $u_1, u_2 \in Pred(v)$, to make *"b is incorrect when entering v"*. There cannot be a combination of errors here.

[11]DF-5 (Figure 3.7) elaborates the event of type *"execution incorrectly reaches v"*. The two single errors being considered in this templates are (1) conjunction of (1a) the execution incorrectly reaches u and (1b) predicate $p(u,t)$ holds, and (2) the control-dependent edge predicate $p(u,t)$ incorrectly holds. To consider error combinations, we obviously would want to add another input event as a

3.5.1 Template DT-1: “b output from v is incorrect”

As mentioned in the simple template section 3.2.1, the value of b coming out of v is considered incorrect only if any of the followings happens:

1. Any input into v is incorrect (data dependence error);
2. The execution incorrectly reaches v (control dependence error); or
3. v is performed incorrectly (agent error).

In this phase of analysis, we consider not only single errors but also combinations of errors, therefore any combination of the three above should be listed as a possible cause of the event *“b output from v is incorrect”*. Figure 3.17 shows the elaboration of *“b output from v is incorrect”*, listing all possible combinations of errors. Note that there is a special case where *“agent error”* is a conjunction of *“v is performed incorrectly producing incorrect b”* and *“execution correctly reaches v”*. The latter event is added, so that the final results (cut sets of the elaborated FT) provide necessary information to construct concrete scenarios — execution paths in the system model — showing how the failure could arise.

All the intermediate events in the template with generic names, such as “data dependence error”, etc. are specific to the elaboration of *“b output from v is incorrect”*. We index those events to separate them from those coming from the elaboration of any other events of the same type *“b′ output from v′ is incorrect”*.

conjunction of (1a) and (2). However, template DT-7 elaborates the event (1b) *“predicate $p(u,t)$ holds”* as shown in Figure 3.20, resulting in the conjunction of (1a) and (2) to be an input event of *“execution incorrectly reaches v”*. So there is no need to replace template DF-5.

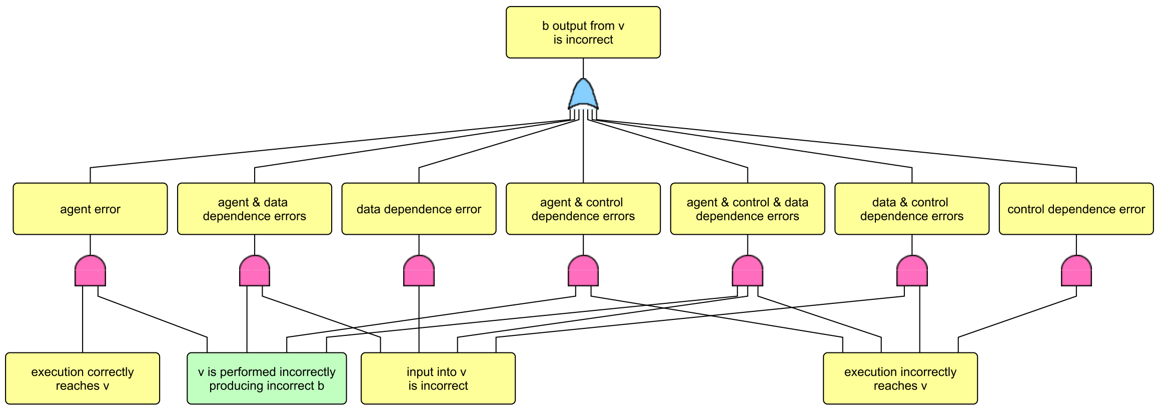

b output from v is incorrect
agent error
agent & data dependence errors
data dependence error
agent & control dependence errors
agent & control & data dependence errors
data & control dependence errors
control dependence error
execution correctly reaches v
v is performed incorrectly producing incorrect b
input into v is incorrect
execution incorrectly reaches v

Figure 3.17. Template DT-1 For Event Type *"b output from v is incorrect"*.

3.5.2 Template DT-2: "input to v is incorrect"

To elaborate this event, we have to iterate through all the possible combinations of input variables into v, which are all possible subsets of $Use(v)$, in other words, all the elements in $\mathcal{P}(Use(v))$ — the power set of $Use(v)$ (see Figure 3.18).

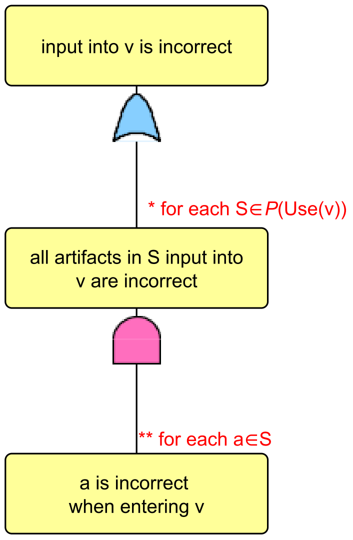
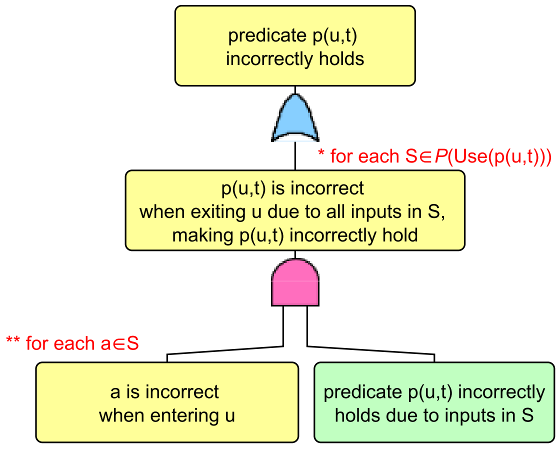

input into v is incorrect
* for each S∈P(Use(v))
all artifacts in S input into v are incorrect
** for each a∈S
a is incorrect when entering v

Figure 3.18. Template DT-2 For Event Type *"input into v is incorrect"*. The symbol * indicates that one event is added for each S, a subset of $Use(v)$, or $S \in \mathcal{P}(Use(v))$. The symbol ** indicates that one event is added for each artifact $a \in S$.

3.5.3 Template DT-6: "predicate $p(u,t)$ incorrectly holds"

Similar to DT-2 in section 3.5.2, to elaborate this event, we also have to iterate through all the possible combinations of variables input to u to compute predicate $p(u,t)$, which are all possible subsets of $Use(p(u,t))$, in other words, all the elements in $\mathcal{P}(Use(p(u,t)))$ — the power set of $Use(p(u,t))$ (see Figure 3.19).

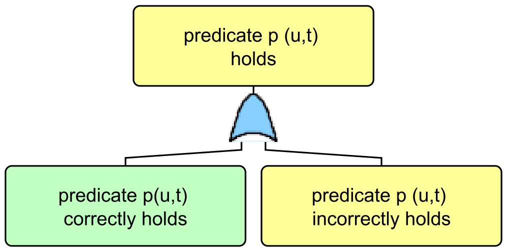


predicate p(u,t) incorrectly holds
* for each S∈P(Use(p(u,t)))
p(u,t) is incorrect when exiting u due to all inputs in S, making p(u,t) incorrectly hold
** for each a∈S
a is incorrect when entering u
predicate p(u,t) incorrectly holds due to inputs in S

Figure 3.19. Template DT-6 For Event Type *"predicate $p(u,t)$ incorrectly holds"*. The symbol * indicates that one event is added for each S, a subset of $Use(p(u,t))$, or $S \in \mathcal{P}(Use(p(u,t)))$. The symbol ** indicates that one event is added for each artifact $a \in S$.

3.5.4 Template DT-7: "predicate $p(u,t)$ holds"

This template (Figure 3.20) is self-explanatory. The event *"predicate $p(u,t)$ holds"* is a disjunction of the intermediate event *"predicate $p(u,t)$ incorrectly holds"* and the primary event *"predicate $p(u,t)$ correctly holds"*.



predicate p (u,t) holds
predicate p(u,t) correctly holds
predicate p (u,t) incorrectly holds

Figure 3.20. Template DT-7 For Event Type *"predicate $p(u,t)$ holds"*.

3.5.5 Template DT-8: "execution correctly reaches v"

This template (Figure 3.21) does not explore or trace back causes of any error. It basically serves the purpose of constructing the execution path in the process model, which helps generate concrete scenarios of a cut set. In this template, we iterate through all control dependent edges $(u \to t)$ of v, i.e. $(u \to t) \in CD(v)$. To have the execution correctly reach v, we must have the execution correctly reaches u and the predicate on the edge $(u \to t)$ has to correctly hold.

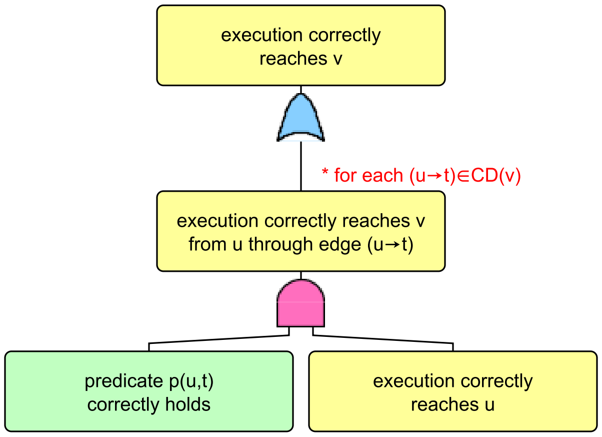

execution correctly reaches v
* for each (u→t)∈CD(v)
execution correctly reaches v from u through edge (u→t)
predicate p(u,t) correctly holds
execution correctly reaches u

Figure 3.21. Template DT-8 For Event Type *"execution correctly reaches v"*. The symbol * indicates that one event is added for each $(u \to t) \in CD(v)$.

3.5.6 Elaborating The Focused Fault Tree

Unlike deriving the initial FT using the simple templates in which we start with one top event (representing the system failure), in this phase of analysis, to derive the elaborated FT, we start with the focused FT, not just a single event. Note that in the initial FT, all the primary events are of one of the following types:

- a input to the system is incorrect (from template DF-3-case-1);
- predicate $p(u, t)$ holds (from templates SP-4 and DF-5);
- v is performed incorrectly producing incorrect b (from template DF-1).

The focused FT is the projection of the initial FT, keeping only events relevant to the chosen initial cut set. Its primary events are thus also of the types mentioned above.

In order to elaborate this focused FT, first, we have to make some of the primary events become intermediate events so that we can elaborate them. Since the first event type *"a input to the system is incorrect"* deals with the data item being incorrect from outside of the scope of the system, no further elaboration is possible. The second event type *"predicate $p(u, t)$ holds"* can be further elaborated into a disjunction of the primary event *"predicate $p(u, t)$ correctly holds"* and the intermediate event *"predicate $p(u, t)$ incorrectly holds"* using template DT-7 as shown in Figure 3.20.

The last primary event *"v is performed incorrectly producing incorrect b"* only appears in template DF-1 (Figure 3.3), which means it only "stems" from the event *"b output from v is incorrect"*. To elaborate the event *"v is performed incorrectly producing incorrect b"*, we must consider it as a single error as well as consider it as a part of an error combinations as listed in template DT-1 (section 3.5.1). Therefore, we convert the stem in the focused FT into a partial FT as shown in Figure 3.22. In the figure, the top part is template DF-1. We are interested in only the stem with the input event *"v is performed incorrectly producing incorrect b"*. The other two are kept in the figure, but grayed out, so that readers easily see that the stem comes from template DF-1. The bottom part of the figure is in fact template DT-1 and we keep only the parts related to the agent error; the gray parts have nothing to do with the agent error.

Note that given the system model and the failure, the detailed templates could be applied from the beginning without having to applying the simple templates in the first phase and then the detailed templates in the second. The result would be one fault tree whose set of cut sets is the union of all elaborated cut sets of all

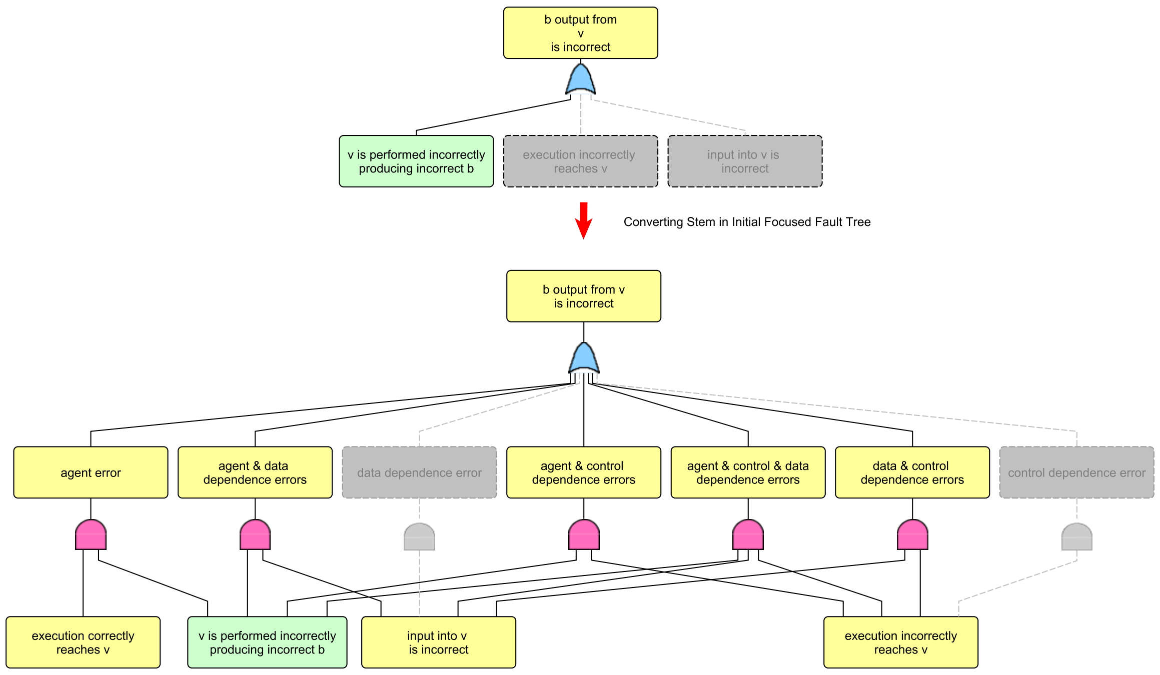

b output from v is incorrect
v is performed incorrectly producing incorrect b
execution incorrectly reaches v
input into v is incorrect
Converting Stem in Initial Focused Fault Tree
b output from v is incorrect
agent error
agent & data dependence errors
data dependence error
agent & control dependence errors
agent & control & data dependence errors
data & control dependence errors
control dependence error
execution correctly reaches v
v is performed incorrectly producing incorrect b
input into v is incorrect
execution incorrectly reaches v

Figure 3.22. Converting A Stem In The Focused Fault Tree.

of the elaborated FTs when applying the two-phase approach. Using the two-phase approach, however, allows users to selectively explore the possible causes of the failure.

3.6 Removing Inconsistent And Spurious Cut Sets

Our FT construction is based on templates, which more or less explore the local errors around the FT events being elaborated. Once the elaborated FT is constructed, the results incorporate information at a global scale, and the local exploration of templates might result in some inconsistent and spurious outcomes.

A cut set is said to be inconsistent if it is impossible for all of its events to happen in one system execution. For example, assuming the value of P does not change during a system execution, P and $\neg P$ cannot happen in the same execution, making any cut set containing both P and $\neg P$ inconsistent. We can automatically identify such cut sets and remove them from the result.

A cut set is said to be spurious if all system executions, each of which contains the events in this cut set and not all the events in any other cut set, turn out to not result in the top event. We develop an algorithm to identify and eliminate certain types of spurious cut sets. Below is a motivating example.

One cut set of the initial FT example shown in the previous section (Figure 3.14) is:

Cut set S:

- `voterName` input to the system is incorrect
- predicate '`registered`' incorrectly holds due to input `registered`[a]
- predicate '¬ `notVoted`' holds

[a]The variable `registered` is an input to the predicate '`registered`'.

Figure 3.23 shows, on the left hand side, the focused FT of cut set S. The focused FT shows that, since `voterName` input to the system is incorrect, it makes the value of `registered` incorrectly *true*, thus the predicate '`registered`' incorrectly holds. That means, the correct value of `registered` has to be *false*. Because the predicate '`registered`' incorrectly holds, the execution goes to 'verify not-voted'. In this scenario, `notVoted` has the value *false*, leading the execution to 'issue provisional' and then 'cast vote'[12]. This execution path is marked as red on the CFG on the right hand side of Figure 3.23.

So, if the error (*"`voterName` input to the system is incorrect"*), which leads to `registered` having the incorrect value *true*, was caught, then `registered` would have had the correct value *false* and the execution would go from 'verify registered'

[12]Of course there are other scenarios in which `notVoted` has the value *true*, either correctly or incorrectly, but we are considering the scenario in which `notVoted` has the value *false*.

to ‘issue provisional’ and then “cast vote”. This execution path is marked as dashed green on the CFG on the right hand side of Figure 3.23.

Both execution paths (one according to the cut set, and one with the error being caught) end up at the same vertex ‘issue provisional’. Consider the case that person A is not registered in his/her correct name; A then impersonates a registered voter B, goes to the polling place, gives B’s name (thus `voterName` input to the system is incorrect), is verified as registered, and then fails the ‘verify not-voted’ check because B has already voted in this election, so A ends up getting a provisional ballot. So this ineligible voter A does not get a regular ballot, but a ballot he or she should have gotten if everything had been done correctly. This cut set is considered spurious.

We would like to identify cut sets of this sort and eliminate them. We have developed an algorithm, shown as Algorithm 3.2, that can correctly identify some spurious cut sets. It is based on the fact that, on the execution path, if at the first branching vertex v that has two successors w_1 and w_2 and there is an event in the cut set that says “predicate $p(v, w_1)$ incorrectly holds”, then we know that $(v \to w_1)$ is the incorrect branch, thus $(v \to w_2)$ is the correct branch. Let P be the path in the CFG from w_2 to either the next branching vertex or the vertex right before **end**, formally as follows:

$$P = \langle w_2, w_3, ..., w_n \rangle$$

with $n \geq 2$, $\forall i$ such that $1 \leq i < n, (w_i \to w_{i+1}) \in E$, $w_n \neq \textbf{end}$, and either $(w_n \to \textbf{end}) \in E$ or $\exists x, y \in V, x \neq y : (w_n \to x) \in E \wedge (w_n \to y) \in E$.

Let V_P be the set of all vertices in P, then if the incorrect execution from v to w_1 somehow ends up at any $w \in V_P$ without encountering a failure-influencing label[13], it

[13]i.e., encountering a vertex u whose output is a and that a_u is a failure-influencing label

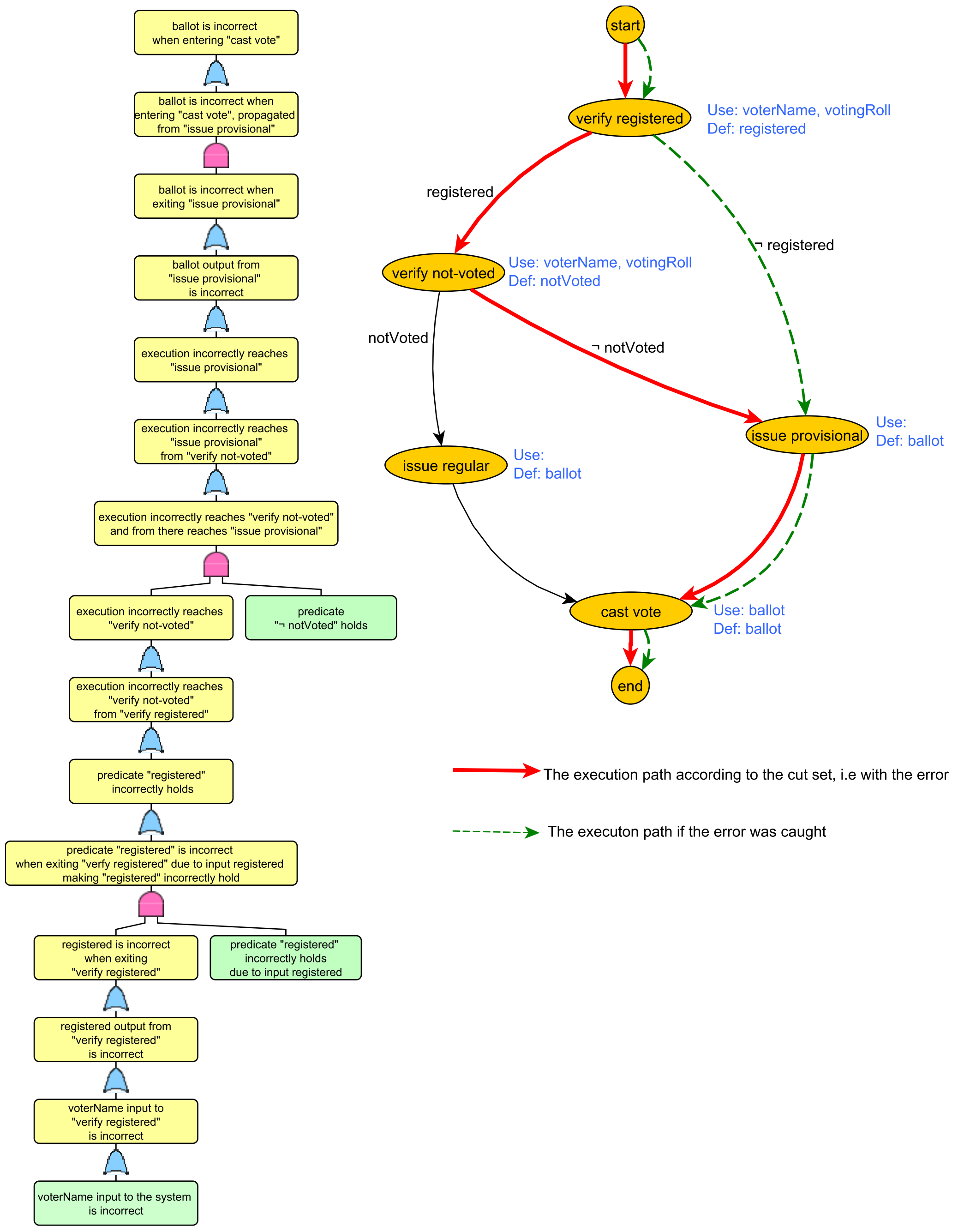

ballot is incorrect when entering "cast vote"
ballot is incorrect when entering "cast vote", propagated from "issue provisional"
ballot is incorrect when exiting "issue provisional"
ballot output from "issue provisional" is incorrect
execution incorrectly reaches "issue provisional"
execution incorrectly reaches "issue provisional" from "verify not-voted"
execution incorrectly reaches "verify not-voted" and from there reaches "issue provisional"
execution incorrectly reaches "verify not-voted"
predicate "¬ notVoted" holds
execution incorrectly reaches "verify not-voted" from "verify registered"
predicate "registered" incorrectly holds
predicate "registered" is incorrect when exiting "verfy registered" due to input registered making "registered" incorrectly hold
registered is incorrect when exiting "verify registered"
predicate "registered" incorrectly holds due to input registered
registered output from "verify registered" is incorrect
voterName input to "verify registered" is incorrect
voterName input to the system is incorrect
start
verify registered
Use: voterName, votingRoll
Def: registered
registered
¬ registered
verify not-voted
Use: voterName, votingRoll
Def: notVoted
notVoted
¬ notVoted
issue provisional
Use:
Def: ballot
issue regular
Use:
Def: ballot
cast vote
Use: ballot
Def: ballot
end
The execution path according to the cut set, i.e with the error
The executon path if the error was caught

Figure 3.23. Spurious Result Example.

```
Function isSpurious()
    input  : CFG, cutset– the cut set of interest
    output: spurious– true if cutset is spurious, false otherwise
    // Initialization
    correctPlaces ⟵ ∅ ;
    reachCorrectNotModData ⟵ false;
    noErrSinceCorrect ⟵ true;
    current ⟵ start;
    while current ≠ end do
        // advance to the next vertex in the CFG
        if current has only one successor then
            current ⟵ the only successor;
        else
            use events with predicates in cutset to determine the next vertex, called next;
            if reachCorrectNotModData ∧ (cutset contains error events at current )
            then
                noErrSinceCorrect ⟵ false;
            if (correctPlaces = ∅) ∧ (cutset contains event “predicate on
            (current → next) incorrectly holds”) ∧ (current has only two successors)
            then
                correctPlaces ⟵ getCorrectPlaces(the other successor that is not
                next) ;
                modifiedDataFromBranch ⟵ false;
            current ⟵ next;
        if (correctPlaces ≠ ∅) ∧ (current has output) then
            modifiedDataFromBranch ⟵ true;
        if (current ∈ correctPlaces ) ∧ (modifiedDataFromBranch = false) then
            reachCorrectNotModData ⟵ true;
    spurious ⟵ reachCorrectNotModData ∧ noErrSinceCorrect;
    return spurious;
Function getCorrectPlaces(v)
    input  : v
    output: correctPlaces
    correctPlaces ⟵ ∅ ;
    while v ≠ end do
        correctPlaces ⟵ correctPlaces +{v } ;
        if v has only one successor then
            v ⟵ the successor ;
        else
            return correctPlaces;
    return correctPlaces;
```

Algorithm 3.2: Determine Spurious.

is safe to say that even taking the incorrect branch, the outcome is the same[14]. So if the outcome at w is the same, whether taking the incorrect branch ($v \rightarrow w_1$) or the correct branch ($v \rightarrow w_2$) to w, and from w to the end of the execution, no more error is made to lead to the top event, then the cut set is decided to be spurious.

In the algorithm, we call V_P the `correctPlaces`. We use `noErrSinceCorrect` to keep track whether the execution from `correctPlaces` to the end encounters any more error events. Once the `correctPlaces` has been found, we also use `modifiedDataFromBranch` to keep track whether the execution from the branching point v has encountered any failure-influencing label. We use `reachCorrectNotModData` to keep track whether or not the execution has reaches the `correctPlaces` without encountering any failure-influencing label along the way.

This algorithm is sound, but not complete. We cannot guarantee that when the algorithm returns *false*, the input cut set is not spurious.

[14]When the execution takes the incorrect branch from v to w_1, even if the execution ends up at $w \in V_P$ but the execution has changed some data which affects the value of the variable at the failure vertex, we cannot guarantee that the outcome is the same as when the execution takes the correct branch from v to w_2.

3.7 Generating Scenarios

Given a cut set, we can generate a scenario showing how the failure may arise. We call the vertex mentioned in the top event the *failure vertex* v_f. A scenario is an execution path in the CFG, from **start** to v_f, that contains all events in the cut set.

By the way our FTs are constructed, the primary events can only be of the following types:

- E1: b input to the system is incorrect,
- E2: v is performed incorrectly and produces incorrect b,
- E3: predicate $p(u, t)$ holds.

We want to find a path in the CFG from **start** to v_f that includes all vertices mentioned in the E2 events, called *E2 vertices*, and satisfies all the predicates mentioned in E3 events, called *E3 predicates*, of the cut set. E1 events are simply put at the **start** vertex.

It's noted that each E3 event uniquely identifies an edge $(u \rightarrow t)$. When searching for scenarios, we want to make sure that, at the decision-making vertex u, the control flow has to take the edge $(u \rightarrow t)$ and not any other outgoing edges emanating from u. Therefore, we may ignore all of these other outgoing edges emanating from u from the CFG.

The remaining problem is simply finding a path between **start** and v_f that contains the E2 vertices in the cut set. We employ a best-first-search algorithm to solve this problem.

For this search problem, each search state is a structure that contains the following information: the CFG vertex v that this node corresponds to, and the goal status g that contains the remaining E3 vertices that we still need to cover in order to satisfy the goal.

Let S be the set of all E2 vertices of the given cut set. The initial state has no parent so the CFG vertex it corresponds to is the **start**, and the goal status g is S. The goal state must have the failure vertex v_f as the corresponding CFG vertex, and the goal status g must be $\emptyset$.

We use a `worklist` to store search states to be examined and a `visited` list to store all states that have been evaluated and will not be looked at again. We also use `parent` to map a search state to its parent state in order to reconstruct the CFG path after reaching the goal state. The initial state has no parent.

Algorithm 3.3 shows the algorithm that returns a scenario given the cut set. Note that this algorithm is a best-first-search algorithm. In order to get all possible scenarios, one can apply a typical breath-first-search algorithm on the CFG to get all the paths from `start` to v_f and select all the paths that contain the all of the E2 vertices of the given cut set.

```
input  : cut set, CFG, $v_f$
output: scenario
// pre-process the CFG
foreach E3 event "predicate $p(u,t)$ holds" in the cut set do
    Delete all edges $\{(u \rightarrow t') \mid t' \neq t\}$ from the CFG;
// the initial state
current.v ⟵ start;
current.g ⟵ S;                                // all the E2 vertices in the cut set
parent [current] ⟵ null;
visited ⟵ $\emptyset$;
worklist ⟵ {current} ;
while worklist is not empty do
    current ⟵ the search state from worklist with fewest vertices in its goal status g;
    Remove current from worklist;
    Add current to visited;
    if current.v $= v_f \wedge$ current.g $= \emptyset$ then // reach the goal
        // trace back from current using recorded parent
        // initialize scenario, which is a sequence of vertices
        scenario ⟵ ⟨current.v⟩ ;
        while current.v $\neq$ start do
            current ⟵ parent [current ] ;
            // add the current vertex to head of the scenario sequence
            scenario ⟵ ⟨current.v⟩+ scenario;
        Return scenario;
    Compute successors of current;
    // successor of current is s such that
    // (current.v, s.v) is an edge in CFG
    // and s.g = current.g − s.v
    foreach successor s of current do
        if s is not in worklist then
            Add s to worklist;
            parent [s] ⟵ current;
```

Algorithm 3.3: Generate Scenario.

CHAPTER 4
EVALUATION

In this work, we designed and implemented a two-phase Fault Tree Analysis approach that automatically identified causes of potential failures in complex systems and allows users to incrementally explore those causes. To evaluate this approach, we applied it to several non-trivial real-world human-intensive processes: *Issue Ballot and Get Vote* (IB), *Count Votes* (CV), *Chemotherapy* (CM), *In-patient Blood Transfusion* (BT). These systems have been subjects of other projects to model and analyze human-intensive processes [7, 8, 28]. As mentioned earlier, although they are called processes, they are indeed systems with control and data flow information, suitable subjects for our approach.

Both IB and CV processes are sub-processes of the larger election process in Yolo County, California. The CM and BT processes are also real-world processes that were employed by Baystate Medical Center, a hospital in Springfield, Massachusetts, at the time the above-mentioned projects were conducted. As those projects' outcomes, the process models were available to be used in this work. In those projects, analysts had shadowed performers of the real processes, and interviewed the performers and domain experts who could also be the process performers to elicit the process models. The analysts, with the help from the domain experts, had then reviewed, evaluated, and corrected the process models to make sure the models accurately and concisely represent the the processes. The processes were modeled in Little-JIL, a modeling language with expressive and well-defined semantics so that the resulting models can be analyzed rigorously.

We chose these processes because, first, their process models were readily available for analysis so we did not have to elicit the process models. Second, these processes are important real-life processes. Third, these processes incorporate non-trivial data and control flows, therefore they are good vehicles for demonstrating and evaluating our approach.

By applying the two-phase FTA approach to the four processes with corresponding failure specifications, we have found that:

1. Using the detailed templates led to scenarios that completely specified the paths through the process that could result in the failures. This finding confirms the usability of the approach which is to help users better understand possible causes of process failures.

2. Some spurious cut sets could be automatically identified and therefore eliminated. However, among the four processes, we only discovered spurious cut sets in the IB process. Our algorithm did not find spurious cut sets in the other processes.

3. The FTA result size, in terms of the numbers of initial cut sets, of elaborated cut sets and and of minimal cut sets (MCSs), depended on multiple factors, which include the overall number of failure-influencing variables, the number of such variables input to a step, the number of steps that have multiple failure-influencing inputs, and the complexity of the control flow to the step that corresponds to the top event.

4. Using all available control and data dependence information resulted in a larger number of MCSs for the domain experts to examine. We discovered MCSs that were not found using Chen's approach. The larger number of MCSs, however, could be overwhelming to the users. We suggest in this chapter some techniques that could be used to make the FTA result more manageable.

In the remainder of this chapter, we first describe how a Little-JIL process model is translated to a CFG suitable to be used in our FTA approach. We then give a brief introduction to the four selected processes. After that, we present our findings in more details. Finally, we discuss the impact on FT derivation of some limitations of Little-JIL.

4.1 Applying The Approach To Little-JIL Process Models

Our FTA approach works on system models represented as CFGs with data flow information. To apply our approach to a Little-JIL process model, first we have to translate the Little-JIL model to a CFG.

4.1.1 Translating A Little-JIL Model To A Control Flow Graph

As mentioned in the Background Chapter (section 2.4), Little-JIL represents a process as a hierarchy of steps. A step is the basic building block of a Little-JIL model. A step represents a unit of work in the process and may be decomposed into sub-steps. Every Little-JIL model has a root step that represents the entire process. This step is decomposed as far as necessary to describe the process. The execution of a Little-JIL step is modeled as progress through several states. Step execution begins in the `POSTED` state during which the execution of the step is assigned to an agent. The execution then proceeds to the `STARTED` state, when the agent begins performing the step. Eventually the step enters either the `COMPLETED` state (normal execution) or the `TERMINATED` state (the execution ends with an exception).

In the CFG representation $G = (V, E, \textbf{start}, \textbf{end})$ of a Little-JIL model, each CFG vertex is a state of a step in the Little-JIL model. There is an edge from u to v if and only if the step state u may immediately precede v according to the Little-JIL process definition. We can determine whether step state u may immediately precede step state v based on the Little-JIL semantics. The CFG **start** vertex is the `POSTED` state of the root step, because it is the first state of any process execution. Since the execution can end at the `COMPLETED` or `TERMINATED` state of the root step, we add an extra **end** vertex, and an edge from each `COMPLETED` or `TERMINATED` state of the root step to this **end** vertex. Figure 4.1 shows the CFG corresponding to the Little-JIL model of the IB ("Issue ballot and get vote") process described in section 2.4.

We note that in the CFG for a Little-JIL model, each vertex represents a state of a step rather than an execution unit. Those step states (vertices) cannot produce or update values for any artifact (data item). In a Little-JIL model, only leaf steps can produce or update values of its output artifacts. So to make it easier to understand the adaptation of our FTA approach to Little-JIL process models, for each leaf step s, we introduce an intermediate vertex "perform s" to represent the actual execution of the step s, and only these vertices have *Def* and *Use* information[1]. The addition of the new vertex is as follows:

- add one new vertex called "perform s";
- add outgoing edges for "perform s" to the existing immediate successors of "s `STARTED`";
- remove all outgoing edges of "s `STARTED`"; and
- add one edge from "s `STARTED`" to "perform s";
- add input artifacts of s to the *Use* of "perform s";
- add output artifacts of s to the *Def* of "perform s".

. Figure 4.2 shows an example for such addition for a leaf step s which has one exception e.

[1]We show later in Section 4.1.2.3 that in the actual implementation, we do not need to add this type of vertex.

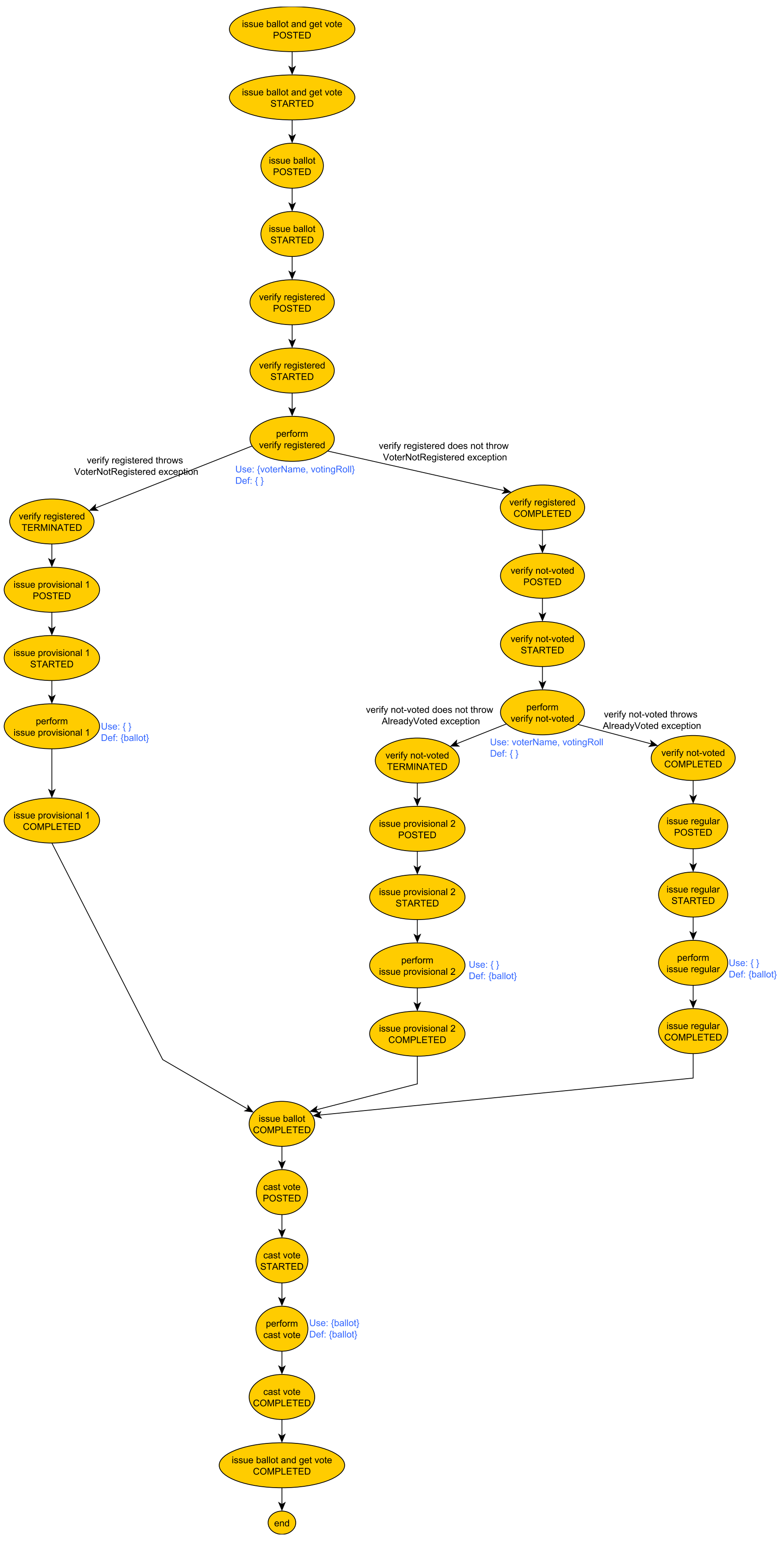

issue ballot and get vote POSTED
issue ballot and get vote STARTED
issue ballot POSTED
issue ballot STARTED
verify registered POSTED
verify registered STARTED
perform verify registered
Use: {voterName, votingRoll}
Def: { }
verify registered throws VoterNotRegistered exception
verify registered does not throw VoterNotRegistered exception
verify registered TERMINATED
issue provisional 1 POSTED
issue provisional 1 STARTED
perform issue provisional 1
Use: { }
Def: {ballot}
issue provisional 1 COMPLETED
verify registered COMPLETED
verify not-voted POSTED
verify not-voted STARTED
perform verify not-voted
Use: voterName, votingRoll
Def: { }
verify not-voted does not throw AlreadyVoted exception
verify not-voted throws AlreadyVoted exception
verify not-voted TERMINATED
issue provisional 2 POSTED
issue provisional 2 STARTED
perform issue provisional 2
Use: { }
Def: {ballot}
issue provisional 2 COMPLETED
verify not-voted COMPLETED
issue regular POSTED
issue regular STARTED
perform issue regular
Use: { }
Def: {ballot}
issue regular COMPLETED
issue ballot COMPLETED
cast vote POSTED
cast vote STARTED
perform cast vote
Use: {ballot}
Def: {ballot}
cast vote COMPLETED
issue ballot and get vote COMPLETED
end

Figure 4.1. CFG Of The Little-JIL Process Model Of “Issue ballot and get vote”.

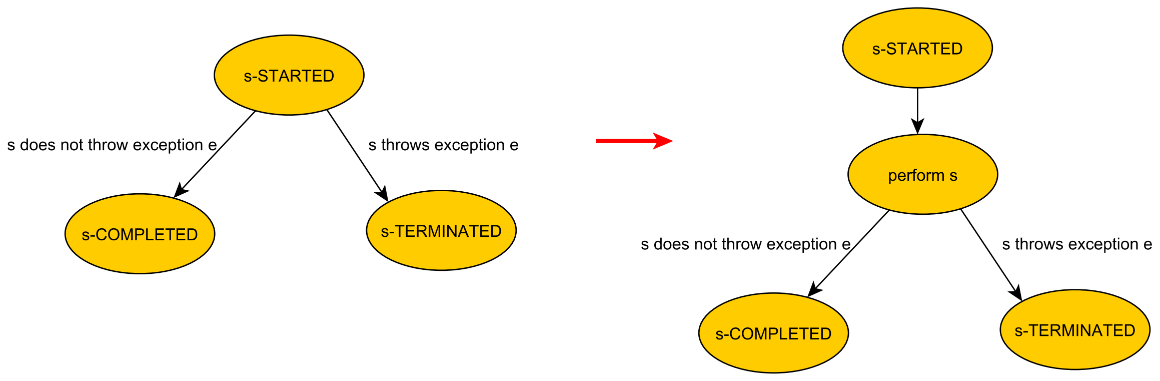


Figure 4.2. Adding "perform *s*" Vertex In The CFG For Each Leaf Step *s*.

In the CFG of the IB process (Figure 4.1), there are six vertices of type "perform *s*" (e.g., "perform verify registered", "perform verify not-voted") corresponding to six leaf steps in the Litte-JIL model of the process; they are annotated with *Def* and *Use* information.

As we require the CFG to be acyclic, we first have to "unroll" and "bound" the Little-JIL model. The recursive steps and steps with unbounded cardinality are unrolled to *step instances* up to bounds given by the analyst. Note that *unbounded cardinality* is a Little-JIL language feature that allows the agent of the step to decide how many times the step should be repeated. *Unbounded cardinality* does not mean the step is repeated infinitively. For example, in the "Count Votes" process as shown later in this chapter (Figure 4.11), the step "count votes for precinct" has unbounded cardinality (depicted by the + symbol) used to indicate the step is executed once for each precinct. An example of infinite recursive execution is: step S throws an exception e, which is handled by step H, and the exception handling specification specifies that S is restarted after e is handled by H. Thus the loop $S \rightarrow H \rightarrow S \rightarrow H \rightarrow ...$ could be infinite. There is no infinite recursion in the four process models in our evaluation.

The details of process unrolling are given in section 3.2.2 of Chen's PhD thesis [3]. The process analyst is asked to designate upper bounds for recursive steps as well as steps with unbounded cardinalities. Given these bounds, a process unrolling preprocessor unrolls the input Little-JIL process model into a unrolled version of the process that contains a finite number of step instances. In addition, a step may be used multiple times in a Little-JIL process via *step references*. The process unrolling turns each step reference into a step instance. As a result, the unrolled process model becomes a tree of step instances. From this point forward, we use "process" to refer to the unrolled Little-JIL process model, and we use "step" and "step instance" interchangeably to refer to a step instance in the unrolled Little-JIL process model.

In the Little-JIL model of the IB process, 'issue provisional' is defined to handle the exception `VoterNotRegistered` thrown by 'verify registered'; and a reference to 'issue provisional' is defined to handle the exception `AlreadyVoted` thrown by 'verify notvoted'. Once the unrolling of the Little-JIL process model is done, 'issue provisional' and 'issue provisional' reference are two separate independent step instances. We use indices to distinguish them — 'issue provisional 1' and 'issue provisional 2' as shown in the CFG (Figure 4.1). Note that the Algorithm 3.2 to determine the spuriousness of a cut set has to consider 'issue provisional 1' and issue provisional 2' as the same step.

If a leaf step s is specified with an exception, named e, it means that from the `STARTED` state of the step, the execution can either go to `COMPLETED` state or `TERMINATED` state of the step, depending on whether e is thrown or not. With the addition of the vertex "perform s" for the leaf step s, we have edges from this vertex to its immediate successors guarded with predicates as follows:

- from "perform s" to "s `COMPLETED`": the edge predicate is "s does not throw exception e",
- from "perform s" to "s `TERMINATED`": the edge predicate is "s throws exception e".

In case the leaf step s is specified to have more than one exception $e_1, e_2, ..., e_n$, we can duplicate the "s `TERMINATED`" state and add predicates on the edges as follows:

- the guarded edge from "perform s" to "s `COMPLETED`" has the edge predicate "s does not throw any exception", which is equivalent to

$$\bigwedge_i (s \text{ does not throw } e_i),$$

- a guarded edge from "perform s" to "s `TERMINATED` i" with the edge predicate "s throws exception e_i".

Little-JIL allows a step to throw more than one exception in one execution, but here we assume that a step can throw at most one exception in one execution. To handle multiple exception throwing in a single execution, more vertices and edges should be added to the CFG; we leave this for future work.

In the IB process example, Figure 4.1 shows four predicates on four edges, two outgoing from "perform verify registered", and two from "perform verify not-voted".

We treat these predicates specially when customizing the FTA templates for Little-JIL as described later in section 4.1.2.

4.1.2 Applying Simple Templates To Little-JIL Models

Most vertices in the Little-JIL CFG have the form "s−{`STATE`}" in which {`STATE`} is one of the possible step states. At those vertices, an artifact's value does not change between entering and exiting the vertices. Therefore, an event of type *"a is incorrect when exiting s−{STATE}"* will be elaborated using the Template DF-3-case-3 as shown in Figure 4.3 (A). To make the fault tree smaller and easier to understand without changing the final result (cut sets), we can substitute the the sub-tree (Figure 4.3 (A)) by a single event *"a is incorrect when s is {STATE}"*.

So from this point forward, when we elaborate an event of type *"a is incorrect when s is* $\{STATE\}$*"*, it means we elaborate the event *"a is incorrect when entering s-*$\{STATE\}$*"*.

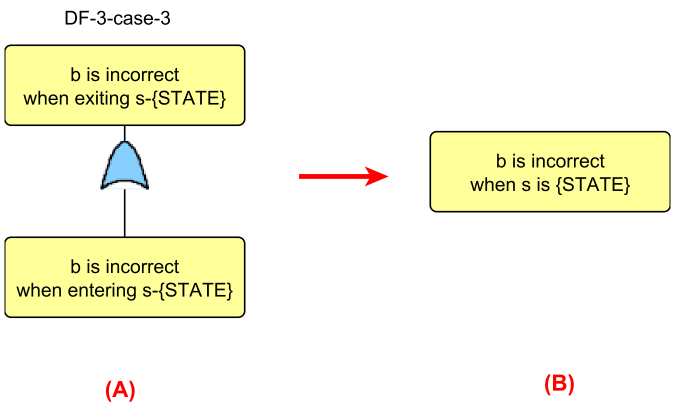

DF-3-case-3
b is incorrect when exiting s-{STATE}
b is incorrect when entering s-{STATE}
b is incorrect when s is {STATE}
(A)
(B)

Figure 4.3. Simplifying DF-3-case-3 When Applying to Little-JIL.

In our generic approach, a system failure is specified to be an incorrect variable input to or output from a specific execution unit of the system (each execution unit is a vertex in the system's CFG). With Little-JIL models, we also specify a failure to be an artifact input to or output from a specific step. But since a step does not correspond to a vertex in the Little-JIL model's CFG, we have to translate the failure specification differently.

For example, the failure is "b input to step s is incorrect" is translated to "b is incorrect when s is `STARTED`"; the failure is "b output from step s is incorrect" is translated to "b is incorrect when s is `COMPLETED`".

To derive the initial FT, simple templates in the SP set are applied. And to derive the elaborated FT, detailed templates in the DT set are applied as described in the Approach Chapter. Most of the applications are straight forward, however, we make some adjustments or create new templates to better suit Litte-JIL. In the remainder of this section, we describe the adjustments and the new Little-JIL FTA templates. It is important to note that all these variations could be expressed in terms of different ways of translating from Little-JIL to the CFG.

4.1.2.1 Elaborating "b is incorrect when s-`POSTED`" When s Is Not The Root Step

We want to present this template because it shows the use of Little-JIL's parameter bindings. As described in the Background Chapter, each Little-JIL step has an artifact declaration that specifies the artifacts it is accessing or providing. Artifacts are generally passed through the coordination hierarchy between steps and their substeps using *parameter binding*. In the "Issue ballot and get vote" example, the `ballot` artifact is an output parameter of the step 'issue regular'; it is bound to the `ballot` parameter of the parent step 'issue ballot'; so is the `ballot` output parameter of the step 'issue provisional'. Parameters of different steps might have the same name, ('issue ballot', 'issue regular', 'issue provisional' and 'cast vote' all have a parameter called `ballot`), therefore we use 'parameter name @ step name' to distinguish them.

We use a graph, called *Parameter Binding Graph* (PBG), to capture the bindings among different parameters. It is a directed graph $G_p = (V_p, E_p)$ where V_p is the set of parameters in the process, and E_p is the set of edges. There is an edge from p_1 to p_2 if and ony if there is a parameter binding indicating that p_1's value is passed to p_2. So in G_p, a definition-free path from p_1 to p_2 is a path in G_p from p_1 to p_2, along which none of the parameters except p_1 and p_2 is an output parameter of a leaf step. Recall that only leaf steps in Little-JIL are allowed to change the value of its output parameter. Thus, a definition-free path from p_1 to p_2 indicates that the value of p_1 is passed to p_2.

Figure 4.4 gives the PBG for the "Issue ballot and get vote" process.

Go back to elaborating "b is incorrect when s-`POSTED`" when s is not the root step. Given that s is not the root step (which would make s-`POSTED` the **start** vertex

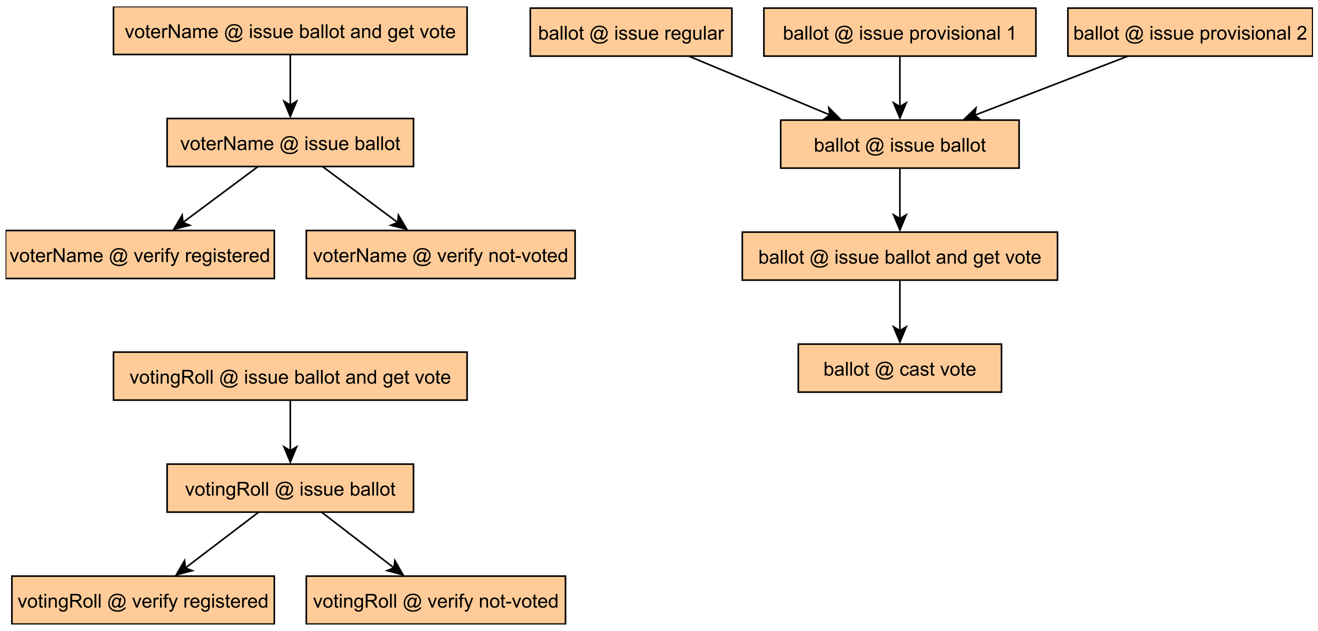

voterName @ issue ballot and get vote
voterName @ issue ballot
voterName @ verify registered
voterName @ verify not-voted
votingRoll @ issue ballot and get vote
votingRoll @ issue ballot
votingRoll @ verify registered
votingRoll @ verify not-voted
ballot @ issue regular
ballot @ issue provisional 1
ballot @ issue provisional 2
ballot @ issue ballot
ballot @ issue ballot and get vote
ballot @ cast vote

Figure 4.4. Parameter Binding Graph (PBG) Of "Issue ballot and get vote".

of the CFG), applying the simple template SP-4, we basically have the elaboration as shown in Figure 4.5. In this elaboration, if there exists a parameter of s', named a, and there is a from a to b in the PBG, then $b' \equiv a$, otherwise $b' \equiv b$. In case there is no such a, the template is exactly the same as SP-4. In case there is such a parameter a, i.e., a's value is passed to b, saying "b is incorrect when $s' - \{\texttt{STATE}\}$" is equivalent to saying "a is incorrect when $s' - \{\texttt{STATE}\}$".

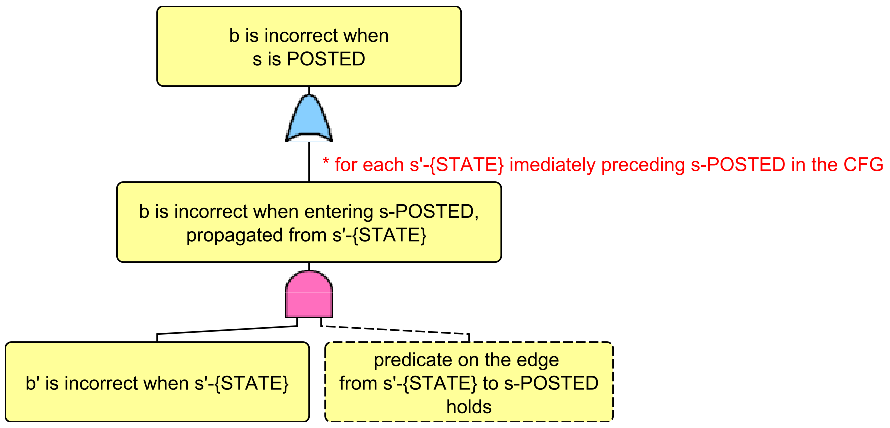

b is incorrect when s is POSTED
* for each s'-{STATE} imediately preceding s-POSTED in the CFG
b is incorrect when entering s-POSTED, propagated from s'-{STATE}
b' is incorrect when s'-{STATE}
predicate on the edge from s'-{STATE} to s-POSTED holds

Figure 4.5. Applying SP-4 To Elaborate *"b is incorrect when s is POSTED"*.

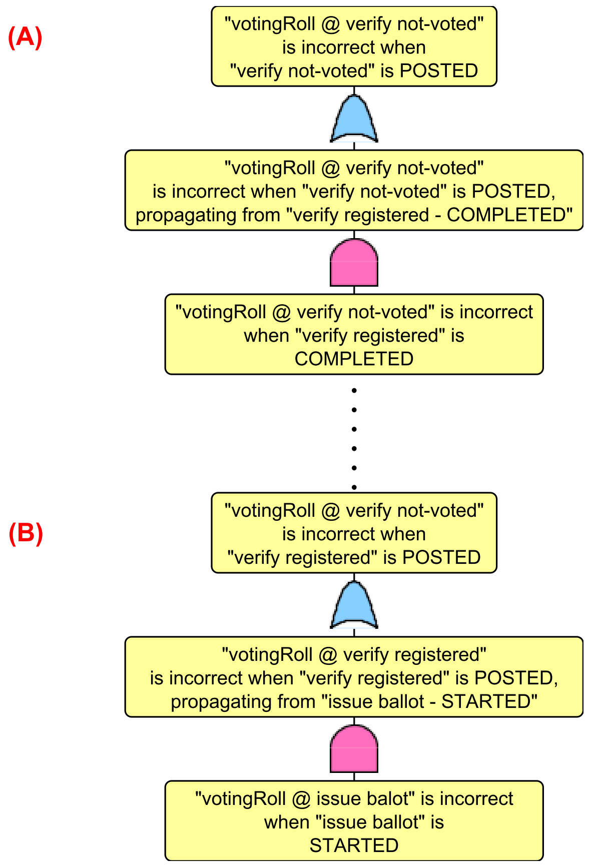

(A)
"votingRoll @ verify not-voted" is incorrect when "verify not-voted" is POSTED
"votingRoll @ verify not-voted" is incorrect when "verify not-voted" is POSTED, propagating from "verify registered - COMPLETED"
"votingRoll @ verify not-voted" is incorrect when "verify registered" is COMPLETED
(B)
"votingRoll @ verify not-voted" is incorrect when "verify registered" is POSTED
"votingRoll @ verify registered" is incorrect when "verify registered" is POSTED, propagating from "issue ballot - STARTED"
"votingRoll @ issue balot" is incorrect when "issue ballot" is STARTED

Figure 4.6. Example Of Applying SP-4 To *"$\mathfrak{b}$ is incorrect when s is* `POSTED`*"*.

In Figure 4.6 (A) showing the example of applying SP-4 to *"'votingRoll @ verify not-voted' is incorrect when 'verify not-voted* `is` *POSTED'"*, we see that 'verify not-voted - `POSTED`' is immediately preceded by 'verify registered - `COMPLETED`'. Since there is no parameter of 'verify register' being passed into 'votingRoll @ verify not-voted', the elaboration results in the new intermediate event *"'votingRoll @ verify not-voted' is incorrect when 'verify registered* `is` *COMPLETED'"*.

Further elaboration arrives at the event *"'votingRoll @ verify not-voted' is incorrect when 'verify registered* `is` *POSTED'"*. Also applying template SP-4 (Figure 4.6 (B)), this time we see that 'verify registered - `POSTED`' is immediately preceded by 'issue ballot - `STARTED`'. Since the parameter 'votingRoll @ issue ballot' is bound to 'votingRoll @ verify not-voted' (see the PBG in Figure 4.4), the elaboration results in the new intermediate event *"'votingRoll @ issue ballot' is incorrect when 'issue ballot* `is` *COMPLETED'"*.

4.1.2.2 Elaborating *"b is incorrect when s-POSTED"* When s Is The Root Step

When s is the root step, "s-`POSTED`" is the **start** vertex of the CFG.

Obviously if b is an input parameter of s, then the event *"b is incorrect when s-POSTED"* must be caused by *"b input to the process is incorrect"*, similar to DF-3-case-1. We decide to not elaborate the event in this case. It is easily understood that *"b is incorrect when the root step is posted"* means *"b input to the process is incorrect"*.

If b is not an input parameter of s, the event *"b is incorrect when s-POSTED"* is marked as *impossible*. The impossible events are pruned from the fault tree after the whole fault tree is derived.

4.1.2.3 Elaborating "b is incorrect when s is `COMPLETED`"

In Little-JIL, only leaf steps can manipulate their own parameters, non-leaf steps can only pass parameters. Thus the elaboration of *"b is incorrect when s is* ***`COMPLETED`****"* depends on whether s is a leaf step, and whether b is the output artifact of s.

In the case s is a leaf step and b is the output artifact of s, we can see that b's value can be propagated from only "perform s", on the condition that s must not throw any exception[2]. Template SP-4 is used to elaborate *"b is incorrect when s is* ***`COMPLETED`****"* as shown in Figure 4.7 (A). In Figure 4.7, we use circled numbers to mark the fault tree events. For the purpose of brevity, we use those numbers to refer to the events.

Applying Template SP-4 to event $\textcircled{1}$ yields two intermediate events — $\textcircled{3}$ *"b is incorrect when exiting 'perform s' "* and $\textcircled{4}$ *"s does not throw any exception"*. We discuss how to elaborate the latter event later in the next section. Elaborating the former event is straightforward, using Template DF-1. Templates SP-2, SP-4, and DF-3-case-3 are then applied to yield the fault tree as we have in Figure 4.7 (A).

We can compact the fault tree in Figure 4.7 (A) into the fault tree in Figure 4.7 (B) to make the tree smaller without compromising the result. First of all, since b is the output of s, the value of b can be propagated to "s-`COMPLETED`" only from "perform s", therefore there is only one event input to the gate of event $\textcircled{1}$. So we can collapse $\textcircled{1}$ and $\textcircled{2}$ into $\textcircled{12}$ in Figure 4.7 (B). Event $\textcircled{13}$ is just event $\textcircled{3}$ with a different name[3].

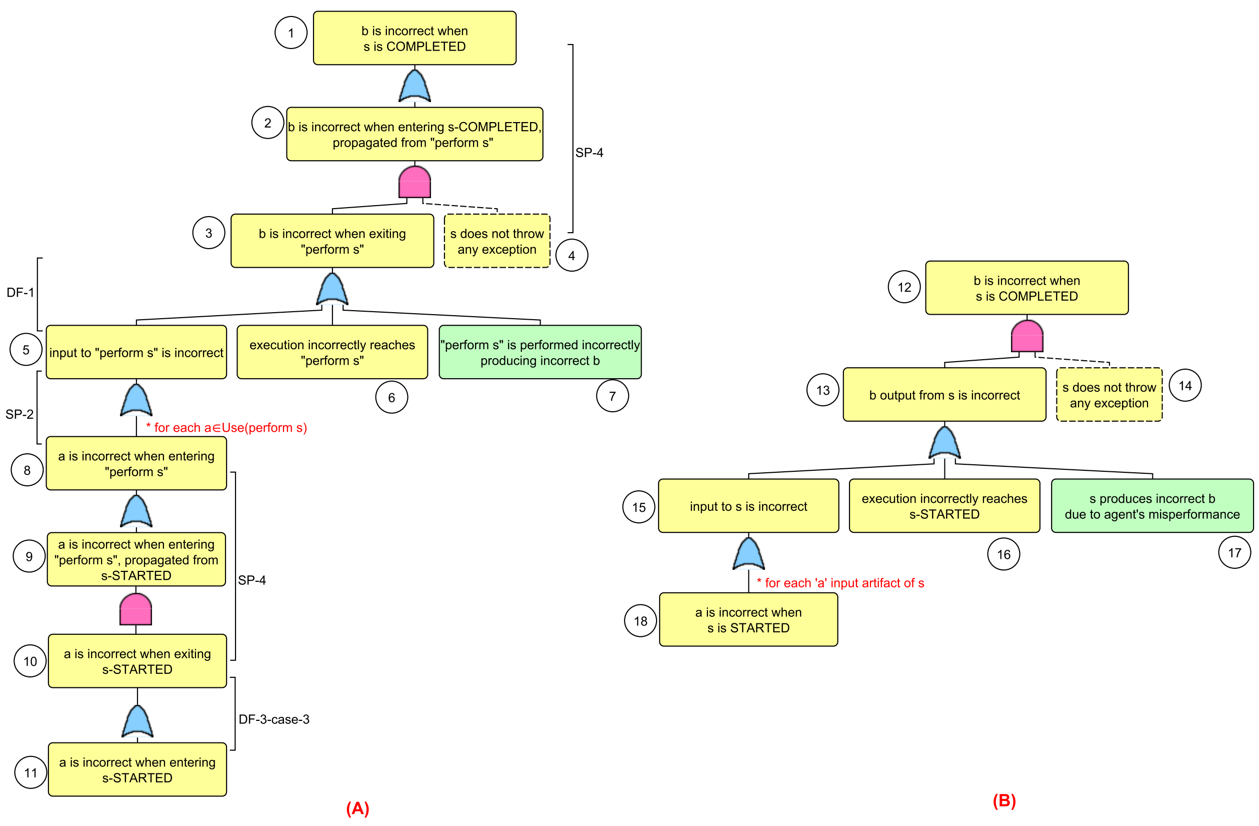
By definition, *Use*("perform s") contains all and only the input artifacts of the step s; "s-`STARTED` is the only vertex immediately precedes "perform s"; and for the

[2]According to the copy-in-copy-out semantics of parameter passing in Little-JIL, if a leaf step is terminated, the value of the parameter that it changes will not be copied out.

[3]As mentioned before, the addition of the vertex "perform s" is just for the purpose of easy understand the approach's application to Little-JIL. The actual implementation does not have this vertex addition.

simplification desceribed above (Figure 4.3), we can collapse events ⑧⑨⑩⑪ into the event ⑱. Lastly, the event ⑰ is just ⑦ with slightly different name.

We call such an elaboration in Figure 4.7 (B) *Template LS-1*.



1
b is incorrect when s is COMPLETED
2
b is incorrect when entering s-COMPLETED, propagated from "perform s"
SP-4
3
b is incorrect when exiting "perform s"
s does not throw any exception
4
DF-1
5
input to "perform s" is incorrect
execution incorrectly reaches "perform s"
"perform s" is performed incorrectly producing incorrect b
6
7
SP-2
* for each a∈Use(perform s)
8
a is incorrect when entering "perform s"
9
a is incorrect when entering "perform s", propagated from s-STARTED
SP-4
10
a is incorrect when exiting s-STARTED
DF-3-case-3
11
a is incorrect when entering s-STARTED
(A)
12
b is incorrect when s is COMPLETED
13
b output from s is incorrect
s does not throw any exception
14
15
input to s is incorrect
execution incorrectly reaches s-STARTED
s produces incorrect b due to agent's misperformance
16
17
* for each 'a' input artifact of s
18
a is incorrect when s STARTED
(B)

Figure 4.7. Template LS-1 For Event Type *"b is incorrect when s is* `COMPLETED`*"* When *b* Is An Output Artifact Of Leaf Step *s*.

For example, in the "Issue ballot and get vote" Little-JIL process, at some point when deriving the fault tree, we encounter the event *"'ballot @ issue regular' is incorrect when 'issue regular' is* `COMPLETED`*"*. Since 'ballot @ issue regular' is the output artifact of the leaf step 'issue regular', we apply Template LS-1 to elaborate the event as shown in Figure 4.8. Note that the step 'issue regular' does not have any input, so this elaboration yields only one intermediate event *"execution incorrectly reaches 'issue regular-*`STARTED`*'"*.

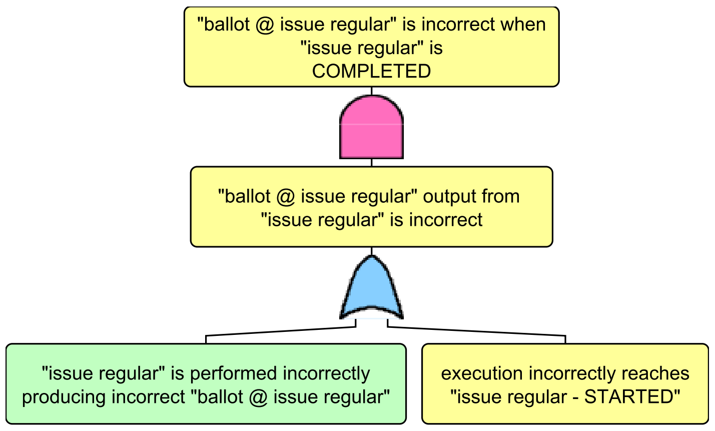

"ballot @ issue regular" is incorrect when "issue regular" is COMPLETED
"ballot @ issue regular" output from "issue regular" is incorrect
"issue regular" is performed incorrectly producing incorrect "ballot @ issue regular"
execution incorrectly reaches "issue regular - STARTED"

Figure 4.8. Elaborating *"'ballot @ issue regular' is incorrect when 'issue regular' is* `COMPLETED`*"* Using Template LS-1.

In cases that s is not a leaf step or b is not an output of s, the elaborations are straight forward using the templates in the SP set.

4.1.2.4 Template LS-2: s incorrectly throws exception e

A Little-JIL step's specification can contain exceptions the step may throw during its execution. A non-leaf step may throw exceptions propagated from its sub-steps. The decision of whether an exception is thrown from a leaf step depends totally on the agent performing the step. Then the leaf step incorrectly throwing exception might be caused by:

- one of the inputs into the step being incorrect,
- the agent's misperformance.

Figure 4.9 shows the template LS-2 to elaborate the event *"s incorrectly throws exception e"*. Figure 4.10 shows an example of applying the template to elaborate the event *" 'verify not-voted' incorrectly throws* ***`AlreadyVoted`****"* when deriving the initial FT for the "Issue ballot and get vote" Little-JIL process model.

A similar template for elaborating *"s incorrectly does not throw exception e"* is shown in the Appendix C.1.

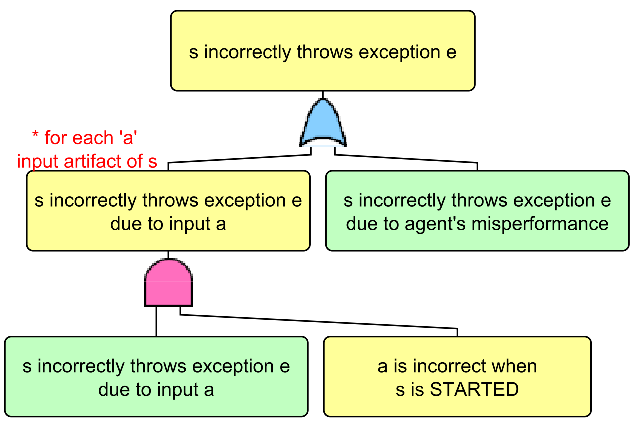

s incorrectly throws exception e
* for each 'a' input artifact of s
s incorrectly throws exception e due to input a
s incorrectly throws exception e due to agent's misperformance
s incorrectly throws exception e due to input a
a is incorrect when s is STARTED

Figure 4.9. Template LS-2 For Event Type *"s incorrectly throws exception e"*.

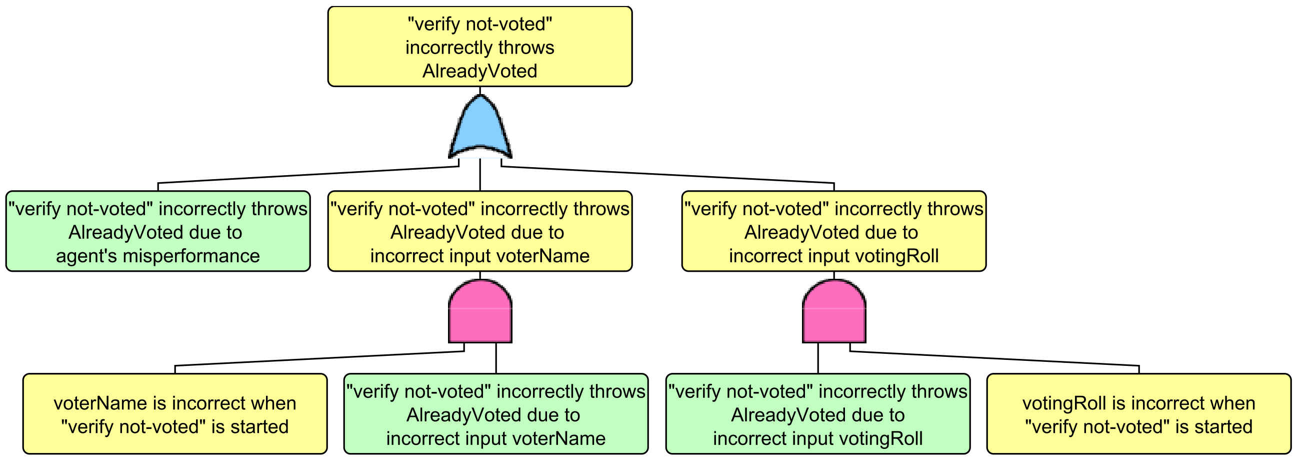

"verify not-voted" incorrectly throws AlreadyVoted
"verify not-voted" incorrectly throws AlreadyVoted due to agent's misperformance
"verify not-voted" incorrectly throws AlreadyVoted due to incorrect input voterName
"verify not-voted" incorrectly throws AlreadyVoted due to incorrect input votingRoll
voterName is incorrect when "verify not-voted" is started
"verify not-voted" incorrectly throws AlreadyVoted due to incorrect input voterName
"verify not-voted" incorrectly throws AlreadyVoted due to incorrect input votingRoll
votingRoll is incorrect when "verify not-voted" is started

Figure 4.10. Applying LS-2 to Elaborate *" 'verify not-voted' incorrectly throws `AlreadyVoted`"*.

4.2 Case Studies On Four Little-JIL Processes

We evaluated the approach on four different processes which were modeled in Little-JIL. In this section, we briefly describe the processes, then present our findings.

4.2.1 Four Little-JIL Processes Used In The Approach's Evaluation

We used models of four different real-world processes in our approach's evaluation. These process models are the result of extensive process elicitation done by analysts; they shadowed the performers of the processes and interviewed domain experts to capture the real-world processes as closely as possible. These process models have been used in other analyses and have been described in published papers (see [2, 6, 7, 25].)

The first process is the "Issue ballot and get vote" (IB) process that we use from the beginning to illustrate the approach. It involves the voter authentication, the issue of an appropriate ballot, and the casting of votes. The failure is that a voter casts his/her votes with an incorrect ballot.

The second process we used in our evaluation is the "Count votes" (CV) process, also in the election domain. Like the IB process model, the CV process model is also based on the real-world election process at Yolo County, California. To count the votes, election officials first count votes in each precinct then add each precinct's result into the total counts, then perform a random audit (manual recount of 1% of ballot to ensure consistency), and then, finally, if no exceptions are raised, report final vote totals to Secretary of State. A recount of all ballots is performed if the random audit raises any exception. The failure being analyzed is "the incorrect vote totals are reported to Secretary of State". Figure 4.11 shows the top level of the Little-JIL model of the CV process.

The third and forth processes are in the medical domain: "Chemotherapy" (CM) and "In-patient blood transfusion" (BT) processes.

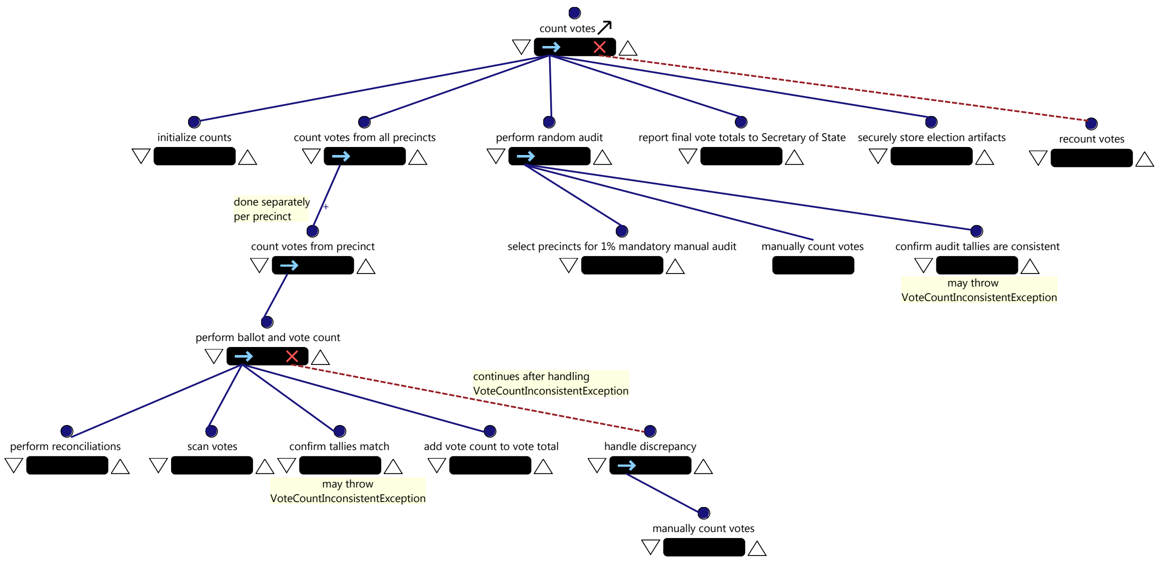

count votes
initialize counts
count votes from all precincts
perform random audit
report final vote totals to Secretary of State
securely store election artifacts
recount votes
done separately
per precinct
+
count votes from precinct
select precincts for 1% mandatory manual audit
manually count votes
confirm audit tallies are consistent
may throw
VoteCountInconsistentException
perform ballot and vote count
continues after handling
VoteCountInconsistentException
perform reconciliations
scan votes
confirm tallies match
add vote count to vote total
handle discrepancy
may throw
VoteCountInconsistentException
manually count votes

Figure 4.11. Little-JIL Process Model Of "Count votes".

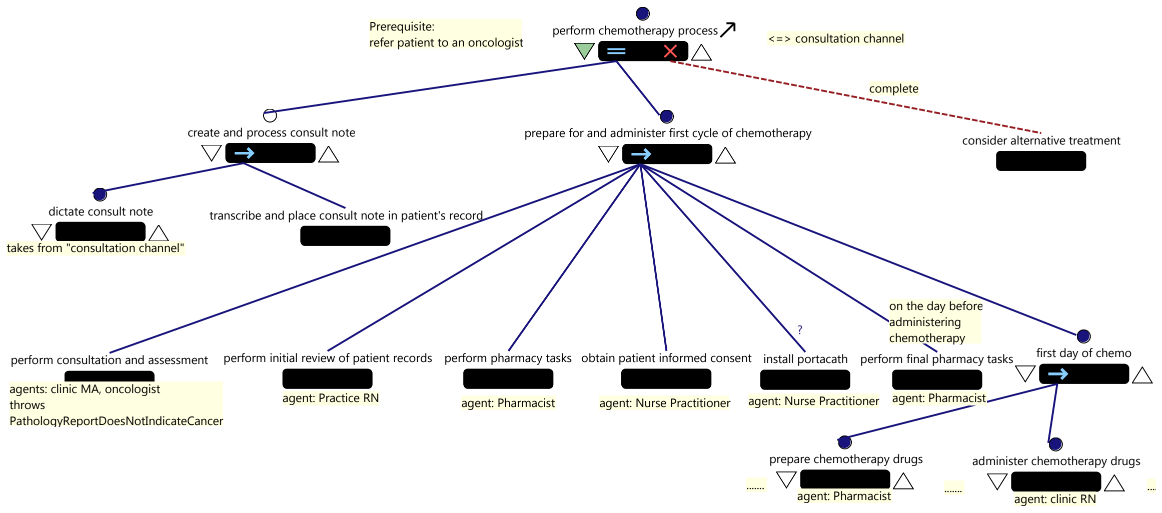

Prerequisite:
refer patient to an oncologist
perform chemotherapy process
<=> consultation channel
complete
create and process consult note
prepare for and administer first cycle of chemotherapy
consider alternative treatment
dictate consult note
takes from "consultation channel"
transcribe and place consult note in patient's record
perform consultation and assessment
agents: clinic MA, oncologist
throws
PathologyReportDoesNotIndicateCancer
perform initial review of patient records
agent: Practice RN
perform pharmacy tasks
agent: Pharmacist
obtain patient informed consent
agent: Nurse Practitioner
?
install portacath
agent: Nurse Practitioner
on the day before
administering
chemotherapy
perform final pharmacy tasks
agent: Pharmacist
first day of chemo
prepare chemotherapy drugs
agent: Pharmacist
administer chemotherapy drugs
agent: clinic RN

Figure 4.12. Little-JIL Process Model Of "Chemotherapy".

The CM process model (Figure 4.12) covers multiple phases of the chemotherapy process, which includes diagnosing the patient and ordering chemotherapy; thorough review of the treatment plan and medication orders by a medical assistant, a nurse, and a pharmacist; conducting an informational/teaching session with the patient and obtaining an informed consent form; preparing chemotherapy drugs, performed by a pharmacist and pharmacy technicians; and assessing the patient and administering the drugs, performed by a nurse. The failure to be analyzed is "incorrect drugs are administered to the patient".

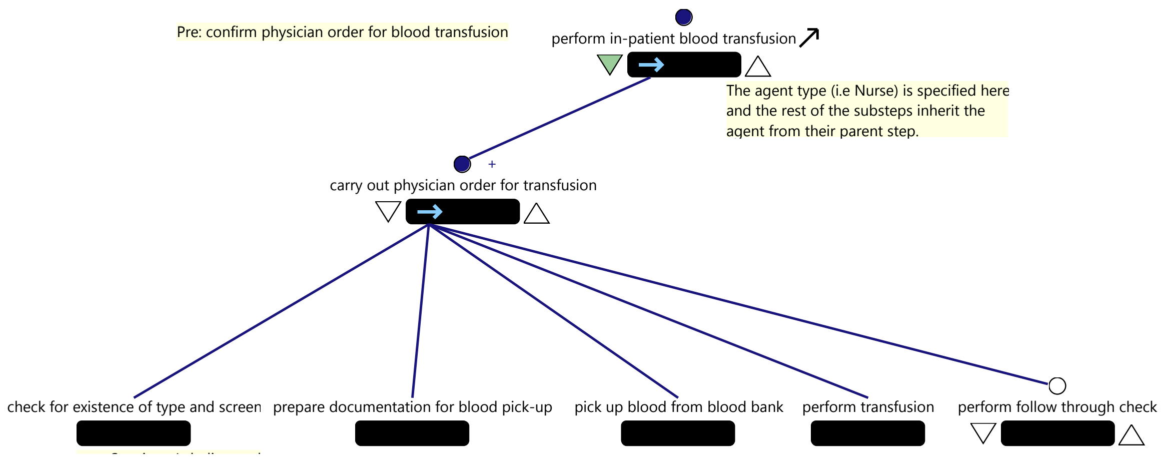

Pre: confirm physician order for blood transfusion
perform in-patient blood transfusion
The agent type (i.e Nurse) is specified here and the rest of the substeps inherit the agent from their parent step.
+
carry out physician order for transfusion
check for existence of type and screen
prepare documentation for blood pick-up
pick up blood from blood bank
perform transfusion
perform follow through check

Figure 4.13. Little-JIL Process Model Of "In-patient blood transfusion".

The BT model defines the blood transfusion process, which includes the following: the nurse confirming there is an order from the physician to transfuse blood to a patient, verifying the patient's ID, getting the blood product from the blood bank, and administering the blood product to the patient. The failure to be analyzed is "incorrect blood product is administered to the patient".

All of the above processes are defined in Little-JIL and translated to appropriate CFGs to be used with our approach. The four processes vary in size (number of steps, thus number of CFG vertices and edges) and number of artifacts as shown later.

For each case study, we applied the simple templates to generate the initial FT, computed the initial cut sets and MCSs. For each initial cut set, we applied the detailed templates to generate the elaborated FT, computed its cut sets and MCSs. We then collected all elaborated FTs (some of them were duplicates as expected) and removed duplications, so the result was unique elaborated cut sets; we also computed the elaborated MCSs from those unique elaborated cut sets.

4.2.2 Findings

By applying the two-phase approach of FTA to the four process models with corresponding failure specifications, we have found that:

1. Using the detailed templates led to scenarios that completely specified the paths through the process that could result in the failure.

2. Some spurious cut sets could be automatically identified and therefore eliminated.

3. The size of the results of FTA, in terms of the numbers of initial cut sets, of elaborated cut sets and and of MCSs depended on multiple factors, including the overall number of failure-influencing variables, the number of such variables input to a step, the number of steps that have multiple of such inputs, and the complexity of the control flow to the failure steps.

4. As expected, using all available control and data dependence information resulted in a larger number of MCSs for the domain experts to examine. We discovered MCSs that were not found before using Chen's approach. These MCSs fell into two categories: in one category, the MCSs elaborated on MCSs previously found using Chen's approach, detailing how the process execution could reach the error-making steps; in the second category, the MCSs represented failure causes in which some steps were executed while they should not have been executed. MCSs in the second category were found by our approach by exploiting control dependence information in process models. The larger

number of MCSs, however, could be overwhelming to users. We suggested four techniques that could be used to make the FTA result more manageable.

In this section, we describe the above findings in more details.

4.2.2.1 Using the detailed templates led to scenarios that completely specified the paths through the process that could result in the failure.

As described in the Approach Chapter, we first generated the initial FT for each case, computed its initial cut sets and elaborated each initial cut set to view its elaborated FT and elaborated cut sets. An example of such incremental exploration of the IB process was described earlier (section 3.3).

Below is another example, taken from the CV process. In the first stage, the initial FT was derived, its cut sets were computed. One of the initial cut sets is:

CS-1:

- 'recount votes' produces incorrect `recountedVoteTotals` due to agent's misperformance

The step 'recount votes' is not on the normal process execution path. If the vote counting and auditing went without any exception, 'recount votes' would not be executed. This initial cut set (which was also produced as an MCS using Chen's approach) did not explain how the execution could get to 'recount votes'. So in the second phase, we derived the elaborated FT using the detailed template. The elaborated cut sets showed multiple ways how the execution could first reach 'recount votes', then the agent performing 'recount votes' produced an incorrect output; all of these led to the failure *"*`totalTallies` *input to 'report vote totals to Secretary of State' is incorrect"*. One such a way to reach 'recount votes' and produce incorrect vote counts is shown in the following elaborated cut set:

CS-1-1:

- 'confirm audit tallies are consistent' incorrectly throws `VoteCountInconsistentException` due to agent's misperformance
- 'recount votes' produces incorrect `recountedVoteTotals` due to agent's misperformance

It is not easy to see that in order for the step 'recount votes' to be executed, the step 'confirm audit tallies are consistent' has to throw the exception `VoteCountInconsistentException`, either correctly or incorrectly (in the above example, the exception was thrown incorrectly due to the agent's misperformance). In the case the exception is thrown incorrectly, we, however, could not automatically infer from the Little-JIL process definition what caused the exception to be thrown. The users might only able to infer from the exception name, `VoteCountInconsistentException`, that the vote counts were somehow inconsistent, therefore the exception was thrown. This Little-JIL language feature, leaving the decision of throwing exception to the agent performing the step, is a strength (allowing abstraction), but also a weakness (inhibiting automatic inference); we discuss this matter more in Section 4.4.

Using this incremental approach with the simple templates in the first phase and the detailed templates in the second phase did not only produce a more accurate result (an error has to happen in certain conditions in order to cause the failure), but also appeared to provide better understanding of the real causes of the failure.

4.2.2.2 Some spurious cut sets could be automatically identified and therefore eliminated

The spurious cut set identification algorithm introduced in section 3.6 was able to identify spurious cut sets in only one of the four cases — the IB process. Table 4.1 shows the result.

Table 4.1. Evaluation Result: Spurious Cut Set Identification.

Process	# cut sets before elimination	# identified spurious cut sets	percentage
IB	429	96	22%
CV	57505	0	0%
CM	183	0	0%
BT	355	0	0%

In the IB process case study, 96 out of 429 unique elaborated cut sets were found to be spurious, according to the algorithm, i.e., 22% reduction in the number of cut sets. All of the identified spurious cut sets in the IB process were conforming to the pattern described in the example in section 3.6.

In other case studies, because of the nature of the processes, there were no cut sets fitting the "profile" of spuriousness that the algorithm aims to identify, that is: in the execution path corresponding to the events in the cut set[4], at the first branching vertex, taking the correct branch definitely leads to a specific step S without encountering any failure-influencing labels, while taking the incorrect branch also definitely leads to S without encountering any failure-influencing labels along the way; and from S to the end of the execution, no further error occurs. Generally, if a process has multiple checks with two possible outcomes: one outcome is when all the checks pass, and the other is when any of the checks fails, then there will be spurious cut sets identified by the algorithm.

4.2.2.3 The result size depended on multiple factors

We consider several measures of the result size: the numbers of initial cut sets, of initial MCSs, of unique elaborated cut sets, and of elaborated MCSs. In addition to the four process models mentioned above, we created a variation of the "Count

[4]The path that contains all of the events in the cut set.

votes" process, called CV2, by removing the artifact `coverSheet` from all the steps. The purpose is to compare the result size when there is a difference in the number of failure-influencing variables and everything else is the same.

Table 4.2 shows the result sizes of all of the case studies. The charts in Figure 4.14 show the comparisons in terms of process models' CFG sizes (numbers of vertices and of edges) and the FTA result sizes (numbers of initial MCSs and of elaborated MCSs).

In the IB process, there are two places where the process's CFG branches out, corresponding to the two Little-JIL steps in the process that might throw exceptions — 'verify registered' and 'verify not-voted' — hence the number of edges is one more than the number of vertices (see Figure 4.1 and Figure 2.6). Similarly, for the CV and CV2 processes, each also has two such steps — 'confirm tallies match' and 'confirm audit tallies are consistent' — hence the number of edges is one more than the number of vertices (see Figure 4.11). The CM process has four and the BT process has more than 20 such steps.

We observe that the size of the CFG in terms of numbers of vertices and edges does not seem to have a much clear effect on the FTA result size. The CM process' CFG has more vertices and edges than the CFG of the IB or the CV processes, but the CM's FTA result is smaller than the FTA result of the other processes. The BT process had a much larger CFG compared to the rest of the processes but its FTA result was much smaller, especially compared to the CV processes. On the other hand, BT had fewer failure-influencing variables: `bloodProduct`, `bloodTypeAndScreen`, and `bloodProductDocument`; only one step had two of those inputs ("release blood product from blood bank" with inputs `bloodProductDocument` and `bloodTypeAndScreen`). CV had five failure-influencing variables: `tallies`, `total-Tallies`, `coverSheet`, `paperTrail`, and `voteRepository`; many steps had multiple of those artifacts as inputs. The number of failure-influencing variables in the process appears to play a large role in the FA result size.

Table 4.2. Evaluation Result: Result Size Comparison.

Process	CFG		# failure-influencing variables	Result			
	# vertices	# edges		# initial CSs	# initial MCSs	# elaborated CSs	# elaborated MCSs
IB	26	27	3	15	15	333	25
CM	60	63	2	10	9	183	19
BT	315	339	3	10	10	345	74
CV	48	49	5	29	29	57505	357
CV2	48	49	4	27	27	12615	220

To investigate this further, we compared CV and CV2, we saw that by having one fewer failure-influencing variable (`coverSheet`), CV2's FTA result was reduced by 7% in the number of initial MCSs (27 vs. 29), and by 38% in the number of elaborated MCSs (220 vs. 357).

There are multiple factors that seem to affect the FTA result size:

1. The overall number of failure-influencing variables,

2. The complexity of the control flow to the failure steps, how many paths there are to the failure steps or size of the control dependence set (CD) of the failure steps,

3. The number of failure-influencing variables input to a step,

4. The number of steps that have multiple failure-influencing inputs.

Future research would be needed to analyze the actual effects of these factors and maybe other factors on the FTA result size.

4.2.2.4 Using all available control and data dependence information resulted in a larger number of MCSs for the domain experts to examine

As explained the in the Approach Chapter, the new two-phase approach exploits available control and data flow information to provide more details to the FTA result,

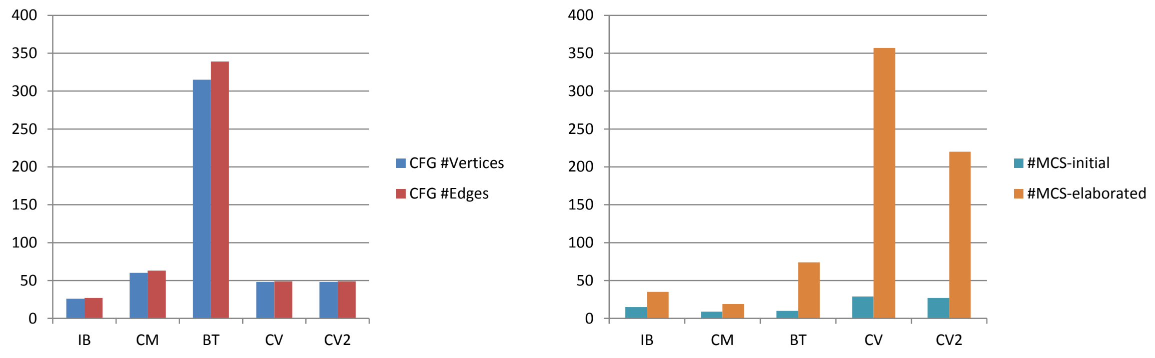


Figure 4.14. Evaluation Result: Result Size Comparison.

Table 4.3. Evaluation Result: More MCSs To Examine.

Process	Chen's Approach	2-phase Approach	
	# MCSs	# initial MCSs	# elaborated MCSs
IB	2	15	35
CM	2	9	19
BT	5	10	74
CV	7	29	357
CV2	6	27	220

especially the control dependence information. Table 4.3 and Figure 4.15 show that in each case study our new FTA approach provided many more initial MCSs and elaborated MCSs than Chen's approach did.

We obviously discovered MCSs that were not identified using Chen's approach. For example, in the IB process, Chen's approach only found two MCSs, both were single-points-of-failure: one was { *"Step 'issue regular ballot' produces wrong* `ballot`*"*}, and the other was { *"Step 'issue provisional ballot' produces wrong* `ballot`*"*}. Those two MCSs were identified as two of the cut sets from our initial FT. We elaborated each of those two initial cut sets, such that the resulting elaborated cut sets showed how the process execution reached the error steps ('issue regular ballot' in the first initial cut set, and 'issue provisional ballot' in the second initial cut set). Our final MCSs were extracted from the elaborated cut sets. Our approach identified, in addition to those

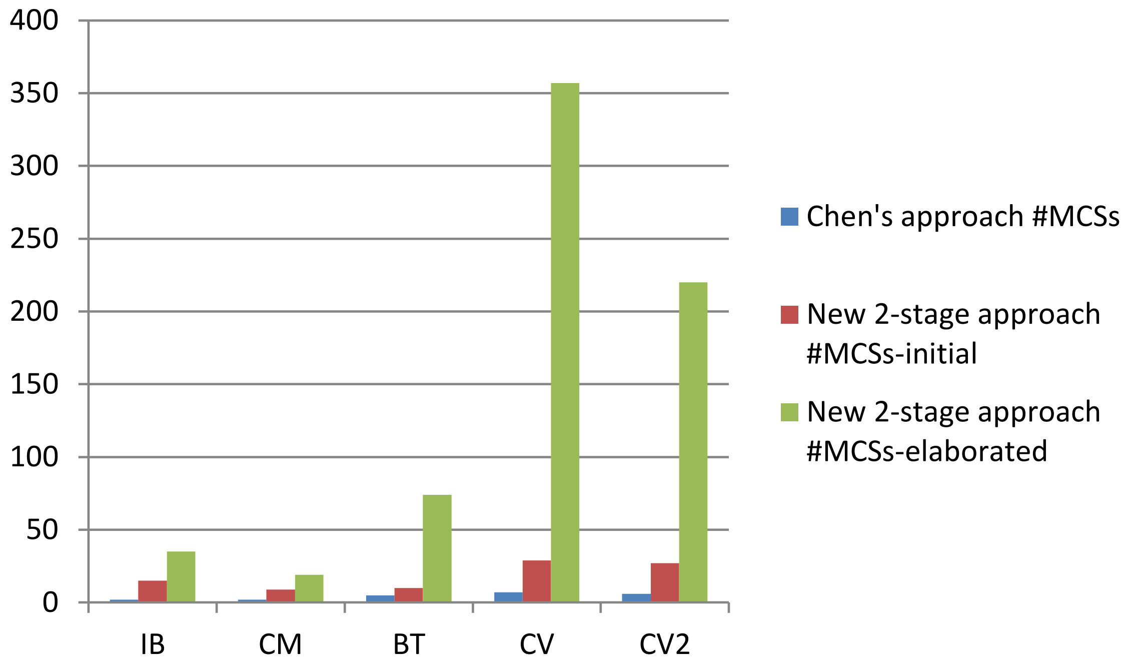
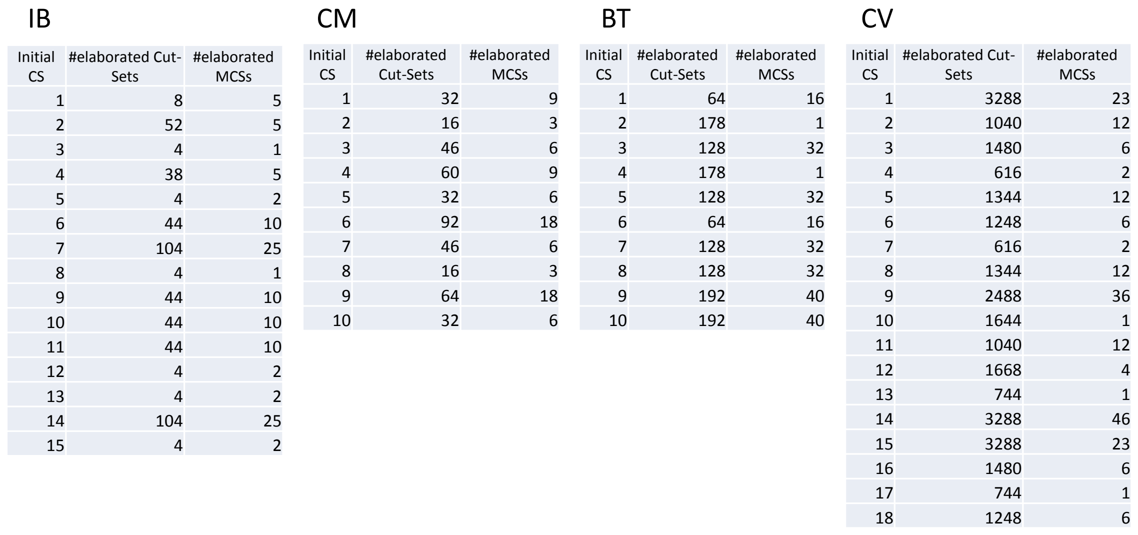


Figure 4.15. Evaluation Result: Comparing With Chen's Approach.



IB

Initial CS	#elaborated Cut-Sets	#elaborated MCSs
1	8	5
2	52	5
3	4	1
4	38	5
5	4	2
6	44	10
7	104	25
8	4	1
9	44	10
10	44	10
11	44	10
12	4	2
13	4	2
14	104	25
15	4	2

CM

Initial CS	#elaborated Cut-Sets	#elaborated MCSs
1	32	9
2	16	3
3	46	6
4	60	9
5	32	6
6	92	18
7	46	6
8	16	3
9	64	18
10	32	6

BT

Initial CS	#elaborated Cut-Sets	#elaborated MCSs
1	64	16
2	178	1
3	128	32
4	178	1
5	128	32
6	64	16
7	128	32
8	128	32
9	192	40
10	192	40

CV

Initial CS	#elaborated Cut-Sets	#elaborated MCSs
1	3288	23
2	1040	12
3	1480	6
4	616	2
5	1344	12
6	1248	6
7	616	2
8	1344	12
9	2488	36
10	1644	1
11	1040	12
12	1668	4
13	744	1
14	3288	46
15	3288	23
16	1480	6
17	744	1
18	1248	6

Figure 4.16. Evaluation Result: Initial CSs and Corresponding Numbers of Elaborated CSs and MCSs.

two initial cut sets, many more initial cut sets and then elaborated MCSs that exposed the causes of the failure, including errors made by agents in the authentication steps (‘verify registered’, or ‘verify not-voted’), or errors in the artifacts (`voterName`, or `votingRoll`), etc.

In the CM process, Chen’s approach found only two MCSs: one was {*“Step ‘prepare chemotherapy drugs’ produces wrong* ***`chemo drug`****”*}, and the other was { *“Step ‘create chemotherapy orders’ produces wrong* ***`chemo orders`****”, “Step ‘perform initial review of patient record’ does not throw exception* ***`PracticeRNFindsProblemWithOrders`****”, “Step ‘perform pharmacy tasks’ does not throw exception* ***`PharmacistFindsProblemWithOrders`****”*}. In addition to those initial cut sets and their elaborated MCSs, we found others, here is an example:

- ‘confirm pathology report indicates cancer’ incorrectly does not throw PathologyReportDoesNotIndicateCancer due to agent’s misperformance
- ‘perform initial review of patient record’ correctly does not throw PracticeRNFindsProblemWithOrders
- ‘perform pharmacy tasks’ correctly does not throw PharmacistFindsProblemWithOrders

The above MCS exposed the possible causes of the system failure in which the error made by the oncologist who performed the step ‘confirm pathology report indicates cancer’[5] (incorrectly not throwing the PathologyReportDoesNotIndicateCancer exception) eventually led to the execution of the step ‘administer chemotherapy drugs’, which should have not been executed in the first place, hence the failure.

Having more MCSs (and even more cut sets) in the result, however, appeared to be overwhelming, especially in the case of the CV process. Figure 4.16 shows, for

[5]Step “confirm pathology report indicates cancer” is a sub-step of “perform consultation and assessment” shown in the CM’s Little-JIL model in Figure 4.12.

each process, the initial cut sets (CSs) and their corresponding numbers of elaborated CSs and MCSs. For example, in IB process, the elaboration of the initial cut set #1 resulted in 8 elaborated CSs, 5 of which were MCSs. In CV process, the number of elaborated CSs of a single initial CS went up to more than 3,000 which seems very unmanageable.

To make the FTA result more accessible to users, we suggest the following:

1. Allow users to specify events that do not have to be considered because they are regarded as too unlikely to occur. We call them *unlikely* events.

2. Allow even more incremental exploration by parameterizing the depth of fault tree elaboration.

3. Group cut sets into equivalence classes.

4. Present only MCSs, and show CSs up on request.

Below we discuss each of the above suggestions in more detail.

4.2.2.4.1 Allow specification of unlikely events. Domain experts might specify some events that are unlikely to happen, so they can be removed from consideration. For example, in the CV case study, if the event *"coverSheet input to process is incorrect"* was specified as unlikely, the FTA result size could be substantially reduced (by almost 80% in the number of unique elaborated CSs, about 40% in the number of elaborated MCSs, see Table 4.4).

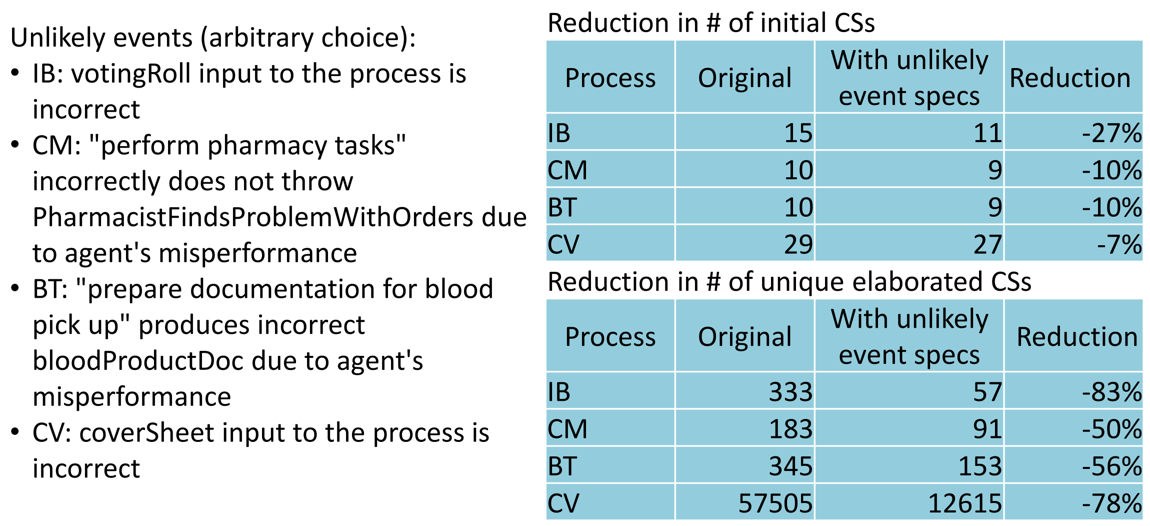
We applied some specification of unlikely events, one for each case study, as shown in Figure 4.17. The unlikely events were chosen randomly[6]. The result sizes were reduced considerably in all cases. Further experiments that specify other randomly

[6]We did not know in advance how often those events appeared in the cut sets before the unlikely-event specification. Obviously, the more cut sets the event appears in, the more the number of cut sets is reduced when the event is specified to be unlikely.

Table 4.4. Specifying Unlikely Events To Reduce Result Size.

Process	# initial CSs	# unique elaborated CSs	Across all initial CSs		
			Average # of elaborated CSs	Average # of elaborated MCSs	Max # of elaborated MCSs
CV (original)	29	57505	9263	45	180
CV' (with unlikely event specs)	27	12615	2019	28	108
CV' vs. CV	*-7%*	*-78%*	*-78%*	*-39%*	*-40%*

chosen events to be unlikely should be carried out to verify the effectiveness of this suggestion.


Unlikely events (arbitrary choice):

- IB: votingRoll input to the process is incorrect
- CM: "perform pharmacy tasks" incorrectly does not throw PharmacistFindsProblemWithOrders due to agent's misperformance
- BT: "prepare documentation for blood pick up" produces incorrect bloodProductDoc due to agent's misperformance
- CV: coverSheet input to the process is incorrect

Reduction in # of initial CSs

Process	Original	With unlikely event specs	Reduction
IB	15	11	-27%
CM	10	9	-10%
BT	10	9	-10%
CV	29	27	-7%

Reduction in # of unique elaborated CSs

Process	Original	With unlikely event specs	Reduction
IB	333	57	-83%
CM	183	91	-50%
BT	345	153	-56%
CV	57505	12615	-78%

Figure 4.17. Specifying Unlikely Events To Reduce Result Size In All Cases.

4.2.2.4.2 Allow even more incremental exploration by parameterizing the depth of fault tree elaboration. Recall the algorithm for elaborating a fault tree: we iteratively apply appropriate templates to elaborate all existing leaf events that are not primary events until all leaf events are primary. In each iteration, we increase one more level of elaboration. We call the number of iterations the *elaboration depth*. We speculated that by limiting the elaboration depth, the numbers of CSs and MCSs in the result were not as many compared to those in unlimited-depth elaboration. To confirm the speculation, we performed the FT elaboration with various depths on the CV process. The result is shown in Figure 4.18. There were 29 CSs in the initial FT. The graph shows the number of elaborated CSs for each of the 29 initial CSs in three cases, varied by the elaboration depths: Full (unlimited depth), Depth=3 and Depth=1.

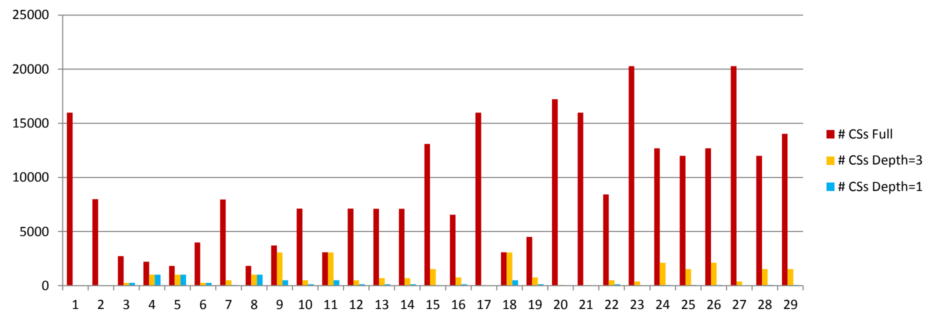


Figure 4.18. Parameterizing Elaboration Depth To Control Result Size.

4.2.2.4.3 Group cut sets into equivalence classes. Cut sets can be partitioned into groups given an equivalence relation. Here is an example of an equivalence relation: Two cut sets are equivalent if they have the same set of error events. Recall that events in cut sets are primary events and some of them are not error events:

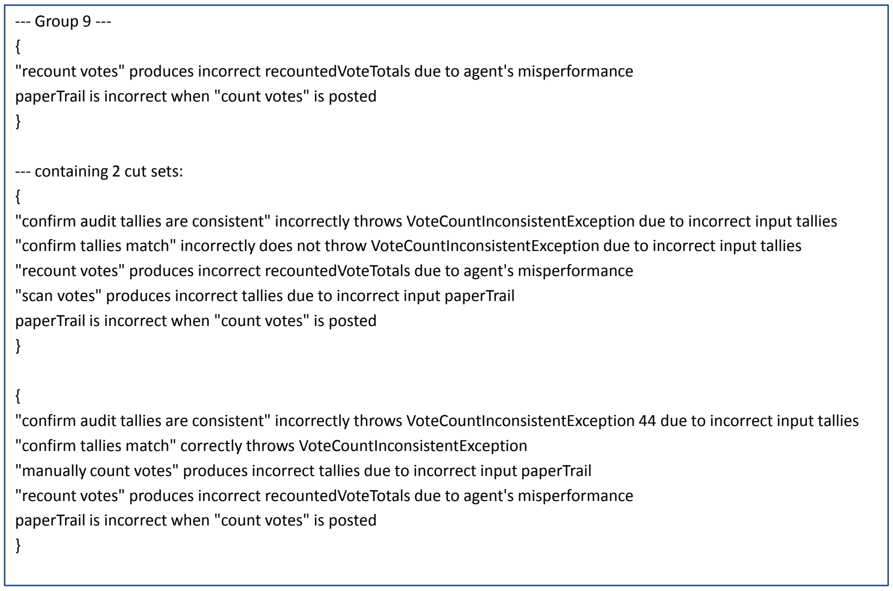
- Branch condition (predicate p correctly holds). In Little-JIL, we have: Step S correctly throws / does not throw exception E.
- Consequence event (v produces incorrect b due to input a). In Little-JIL, we have: Step S produces incorrect b due to input a; Step S incorrectly throws/ does not throw exception E due to input a.

Error events in cut sets are of the following types:

- v is performed incorrectly producing incorrect b. In Little-JIL: Step S produces incorrect b due to agent's misperformance; Step S incorrectly throws / does not throw exception E due to agent's misperformance.

- a input to the system is incorrect. In Little-JIL: a is incorrect when ROOT step is POSTED.

Figure 4.19 shows an example in the CV case study of two cut sets belonging to the same group because they have the same set of error events.



```
--- Group 9 ---
{
"recount votes" produces incorrect recountedVoteTotals due to agent's misperformance
paperTrail is incorrect when "count votes" is posted
}

--- containing 2 cut sets:
{
"confirm audit tallies are consistent" incorrectly throws VoteCountInconsistentException due to incorrect input tallies
"confirm tallies match" incorrectly does not throw VoteCountInconsistentException due to incorrect input tallies
"recount votes" produces incorrect recountedVoteTotals due to agent's misperformance
"scan votes" produces incorrect tallies due to incorrect input paperTrail
paperTrail is incorrect when "count votes" is posted
}

{
"confirm audit tallies are consistent" incorrectly throws VoteCountInconsistentException 44 due to incorrect input tallies
"confirm tallies match" correctly throws VoteCountInconsistentException
"manually count votes" produces incorrect tallies due to incorrect input paperTrail
"recount votes" produces incorrect recountedVoteTotals due to agent's misperformance
paperTrail is incorrect when "count votes" is posted
}
```

Figure 4.19. Grouping Cut Sets.

We experimented with grouping on the CV process. Table 4.5 shows the result across all the initial cut sets. On average, the number of cut set groups (203 CS groups) is only 2.2% of the number of elaborated CSs (9263 CSs), and the number of MCS groups (15 MCS groups) is only 34% of the number of MCSs (45 MCSs). We noted that with this grouping, each group could correspond to more than one scenario (the example above). So even though the grouping allowed users to focus on error events, elaboration on the grouping to reveal all corresponding scenarios is necessary to provide users with better understanding of the vulnerabilities.

Table 4.5. Grouping CSs And MCSs In CV Process.

	# elaborated CSs	# elaborated MCSs	# CS groups	# MCS groups
AVG	9263	45	203	15
MAX	20284	180	320	44
MIN	1832	1	104	1

4.2.2.4.4 Present only MCSs, and show CSs up on request. As seen in Figure 4.16, the numbers of MCSs in the result are less overwhelming. So if we present the result by showing only MCSs, and present CSs only upon user's request, it might be a better user experience. Another suggestion is to present all CSs, but sort the CSs by their minimality and then by the number of error events in each CS. This allows users to easily focus their effort on what might be more important.

4.3 Summary

In summary, we found that the approach was promising but inherited several limitations and thus needed more improvement. First of all, we saw that using the detailed templates led to scenarios that completely specified the paths through the process that could result in the failure. Second, some spurious cut sets could be automatically identified and therefore eliminated. However, it was quite unexpected that among four case studies, the approach only identified spurious cut sets in only one case (IB process). We have not yet identified (even manually) spurious cut sets in other processes. One should investigate more if spurious cut set elimination is desirable. Third, we found that the number of elaborated CSs (and MCSs) depended on multiple factors, not necessarily on the CFG size. The most likely factor was the number of failure-influencing variables overall and per step. And finally, we found that using all available control and data dependence information resulted in a much larger number of MCSs for the domain experts to examine. We made some suggestions to make result less overwhelming to users.

4.4 Little-JIL's Abstraction Posing Limits On Our Fault Tree Derivation

The Little-JIL definition language highlights the use of abstractions, but as a consequence, it limits our FT derivation as follows.

First, a leaf step in Little-JIL definition is only defined by its interface, including the inputs, outputs, agent, and possible exceptions. The step specification does not define how the outputs are computed from the inputs. Therefore we have to assume each output depends on all of the inputs. Thus, leaf steps that do not satisfy this assumption will cause the derived fault tree to contain incorrect sub-trees. We were able to annotate leaf steps with output-input dependence information, so that the derived fault trees were more accurate. However, it is desirable to have a feature in the language to specify the output-input dependence information.

Second, in Little-JIL, when a step is declared to have an exception, the decision of whether or not to throw the exception during the process execution depends on the agent performing that step. For example, the step 'confirm audit tallies are consistent' is defined with an exception VoteCountInconsistentException. This Little-JIL language feature allows procedural abstraction; the details of how it is decided that the exception has to be thrown are omitted. However, as mentioned in Section 4.2.2.1, this feature also inhibits automatic inference of the fault tree events of type *"step S correctly throws exception E"*, which usually are the cases when something goes wrong thus the process has to deviate from its normal execution path. With no further details provided in the process definition about which cases the exception should be thrown, we cannot automatically infer those cases and further derive the fault tree, we have to treat those events *"step S correctly throws exception E"* as primary events.

CHAPTER 5
CONCLUSIONS

Fault Tree Analysis (FTA) has been widely used in various industries to study failures of complex systems, either after the fact or as a preventive measure. We are interested in the latter — identifying causes of potential system failures — so that modifications can be made to the system designs to reduce the possibility of such failures.

FTA is a deductive, top-down analysis technique that aims to determine the various combinations of hardware and software faults and human errors that could cause a specified undesired event representing a system failure. Given a specified undesired event, FTA produces a *fault tree (FT)* and computes its *cut sets* — combinations of events that could lead to the occurrence of the undesired event at the top of the tree, referred to as the *top event*.

Straightforward Boolean algebra can be used to compute a FT's cut sets, but the construction of the FT itself is still an on-going problem. As manual FT construction is time consuming and prone to errors, work has been done to automate the FT construction for various system models. Previous approaches to automatic FT construction, however, are mostly language dependent; they are limited to systems modeled in specific languages. Some of these types of models do not carry data or control dependence information, or even if they do (Pascal, ADA, Little-JIL), control dependence information is not fully exploited, therefore FTs produced by the respective FT construction approaches are incomplete. There is little evidence that these approaches have been applied to non-trivial systems; only simple systems were shown in the papers presenting the approaches. Once a FT is constructed and its cut sets are computed, the cut sets alone often do not provide enough information to show users how events in a cut set together can cause the top event. At the same time, there might be too many cut sets to be useful, especially when systems are large and complex. Moreover, previous FTA approaches do not seem to attend to false positives — cut sets that in fact do not cause the top event.

In this thesis, we investigated a systematic two-phase approach to identifying causes of potential system failures using FTA as follows:

- Phase 1: Given a system model and an undesired event representing a system failure, we first use data and control dependence information from the system model to automatically derive a high-level FT, called the *initial FT*, whose top event is the undesired event; and then present users with the cut sets, called the *initial cut sets*, from the initial FT.

- Phase 2: Users can then select one initial cut set for more detailed analysis. This analysis considers additional control dependence information and combinations of possible errors to generate an *elaborated FT* that focuses on the initial cut set of interest. This second phase produces as its final result *concrete scenarios*, each of which is a system execution path that contains all the events in a cut set of the elaborated FT, showing how the events in the cut set could lead to the top event.

System models used in this approach are control flow graphs (CFGs) and since most system modeling languages can be translated to CFG, therefore this approach is generally applicable.

To evaluate this approach, we applied it to several systems in the medical and election domains. These systems have been subjects of other projects [7, 8, 28] to model and analyze human-intensive processes. Although being called "processes", they are in fact "systems" in our approach. The systems are modeled in Little-JIL, a modeling language with expressive and well-defined semantics so that the resulting models can be analyzed rigorously. Even though these system models are relatively small, they are useful because they were carefully defined with the help from domain expert in order to closely reflect the real-world systems.

By applying the two-phase FTA approach to the four systems with corresponding system failures, we found that:

1. Using the detailed templates led to scenarios that completely specified the paths through the system model that could result in the failure. This finding confirms the usability of the approach which is to help users better understand the causes of the failures.

2. Some spurious cut sets could be automatically identified and therefore eliminated. However, among the four systems, we only discovered spurious cut sets in one system. Our algorithm did not find spurious cut sets in the other systems. The reason is the other three systems do not have the control flows that fit the "profile" of spuriousness that the algorithm is aiming to identify.

   More work can be done in this area to develop more refined algorithms to identify different types of spurious cut sets.

3. There were multiple factors that affected the number of cut sets in the FTA results. Those factors included the overall number of failure-dependent data items, the number of such data items input to a control flow graph's vertex, the number of vertices that have multiple failure-dependent inputs, and the complexity of the control flow to the failure vertices.

4. Using all available control and data dependence information resulted in a larger number of cut sets for the domain experts to examine. We discovered cut sets

that were not found before in the previous Chen's approach. The larger number of cut sets, however, could be overwhelming to users.

We suggested and performed some preliminary exploration of some techniques that could be used to make the FTA result more manageable:

1. Allow users to specify events that do not have to be considered because they are regarded as too unlikely to occur,

2. Allow even more incremental exploration by parameterizing the depth of fault tree elaboration,

3. Group cut sets into equivalence classes,

4. Present only minimal cut sets, and show cut sets up on request.

Future work should include more thorough evaluation of the above suggested techniques and more investigation into other techniques to help manage the easily-overwhelming FTA results.

Other areas for future work include: considering systems with concurrency and examining other techniques to manage loops in system models. In case of using loop unrolling, the user would have to choose a bound, and that choice may not be easy to make. We unrolled loops twice in our evaluation and the future work should also investigate other bounds and compare the results.

APPENDIX A

PROOF OF EQUIVALENCE BETWEEN DF AND SP TEMPLATE SETS

Two fault trees T_1 and T_2 are said to be *equivalent* if and only if they have the same set of cut sets. We write $T_1 \cong T_2$.

Given a fault tree T, a subtree of T is a tree consisting of an event in T and all of its descendants in T.

Suppose we have two fault trees T and T' that are identical, except for the elaboration of an event E in both T and T'. The subtree in T with top event E is called T_E, while the subtree in T' with top event E is called T'_E. It is easy to see that if $T_E \cong T'_E$ then $T \cong T'$.

In deriving a fault tree T given a top event H, a template is applied to elaborate an intermediate event E of a specific type, generating a fault tree T_E whose top event is E. Each resulting fault tree T_E is a subtree of current version T (still being elaborated).

The two sets of templates DF and SP are the same except for two templates: DF-2 and SP-2 for the event *"input into v is incorrect"*, DF-4 and SP-4 for the event *"b is incorrect when entering v"*. Therefore, in order to prove that the two sets generate equivalent fault trees, we only need to show the following statements are correct:

> **S1**(v): If the elaboration of *"input into v is incorrect"* using DF set generates fault tree $T1_{DF}(v)$ and the elaboration of *"input into v is incorrect"* using SP set generates fault tree $T1_{SP}(v)$ then $T1_{DF}(v) \cong T1_{SP}(v)$.
>
> **S2**(b,v): If the elaboration of *"b is incorrect when entering v"* using DF set generates fault tree $T2_{DF}(b, v)$ and the elaboration of *"b is incorrect when entering v"* using SP set generates fault tree $T2_{SP}(b, v)$ then $T2_{DF}(b, v) \cong T2_{SP}(b, v)$.

A.1 Proof of S1(v)

Let's look at DF-2 (Fig. 3.4, page 43) to elaborate an event of type *"input into v is incorrect"*. Basically the template iterates through all possible definitions that reach v and are used as input in v.

Every time DF-2 is applied, a new intermediate event of type *"a is incorrect when exiting u"* for each input $a \in \mathit{Use}(v)$ is created. Note that by the construction of

DF-2, either u is the not **start** vertex (implying a is defined at u) or u is the **start** vertex. If u is not the **start** vertex, DF-3-case-2 (Fig. 3.5, page 3.5) will be applied generating another event of type *"input into u is incorrect"* and then DF-2 is applied again.

Only when we reach the event of type *"a is incorrect when exiting u"* and u is the **start** vertex, then DF-3-case-1 can be applied, generating the primary event *"a input into the system is incorrect"*, the fault tree elaboration terminates.

Informally, the template application along one branch of the fault tree, starting from *"input into v is incorrect"* has this formulation $\boxed{\text{DF-2 - (DF-3-case-2 - DF-2)* - DF-1}}$ (the Kleene star * indicates 0 or more times).

Let $L(v)$ be the maximum number of times that DF-2 has to be applied along one tree branch before the elaboration terminates. For example, in Figure A.1, $L(u_1) = 1$.

We are going to prove S1(v) by induction. We first prove that S1(v) is correct for all v with $L(v) = 1$.

A.1.1 Proof of S1(v) - Base case: $L(v) = 1$

Case 1: There is only one element in *IDD*(v)

Suppose (a_u, P) is the only element in *IDD*(v) — definition of a at u reaches v via the path guided by P and a is used at v.

Suppose the path is $u_0 \rightarrow u_1 \rightarrow ... \rightarrow u_n$ where $u_0 \equiv u$ and $u_n \equiv v$[1]. And along that path, there are m guarded edges $(u_{k_1} \rightarrow u_{k_1+1}), ..., (u_{k_m} \rightarrow u_{k_m+1})$ with $0 \leq m \leq n$, $0 \leq k_i < n$. Without loss of generality, we assume $k_i < k_j$ if $i < j$. Let p_i denote the predicate on the guarded edge $(u_i \rightarrow u_{i+1})$.

In this case (Base Case 1), since (a_u, P) is the only element in *IDD*(v), we can infer that v has only one input a, and there is only one path guided by P from u to v. Since $L(v) = 1$, u has to be the **start** vertex. Recall that $u_0 \equiv u$ and $u_n \equiv v$. So

[1]We use $v_1 \equiv v_2$ to denote v_1 and v_2 are the same vertex.

in this Base Case 1 ($L(v) = 1$ and there is only one element in $IDD(v)$), u_0 is the **start** vertex.

A.1.1.0.5 Case 1.1: $n = 1$ and $(u_0 \rightarrow u_1)$ is not a guarded edge: In this case, there is no predicate on the edge $(u_0 \rightarrow u_1)$.

- Using the DF template set: Applying DF-2 then DF-3-case-1 will generate the fault tree $T1_{DF}(v)$ as in the left hand side of Figure A.1.
- Using the SP template set: Applying SP-2, SP-4, then DF-3-case-1 will generate the fault tree $T1_{SP}(v)$ as in the right hand side of Figure A.1.

In each fault tree in Figure A.1, the event *"predicate p_0 holds"* is left there to show full template DF-2, but it is shaded gray because $(u_0 \rightarrow u_1)$ is unguarded, there is no such predicate.

Both of these trees have only one cut set, that is {*"a input to the system is incorrect"*}.

A.1.1.0.6 Case 1.2: $n = 1$ and $(u_0 \rightarrow u_1)$ is a guarded edge: In this case, there is predicate p_0 on the edge $(u_0 \rightarrow u_1)$.

- Using the DF template set: Applying DF-2 then DF-3-case-1 will generate the fault tree $T1_{DF}(v)$ as in the left hand side of Figure A.2.
- Using the SP template set: Applying SP-2, SP-4 and then DF-3-case-1 will generate the fault tree $T1_{SP}(v)$ as in the right hand side of Figure A.2.

Both of these trees have only one cut set, that is {*"a input to the system is incorrect", "p_0 holds"*}.

A.1.1.0.7 Case 1.3: $n > 1$:

- Using the DF template set: Applying DF-2 then DF3-case-1 will generate the the fault tree $T1_{DF}(v)$ as in the left hand side of Figure A.3.

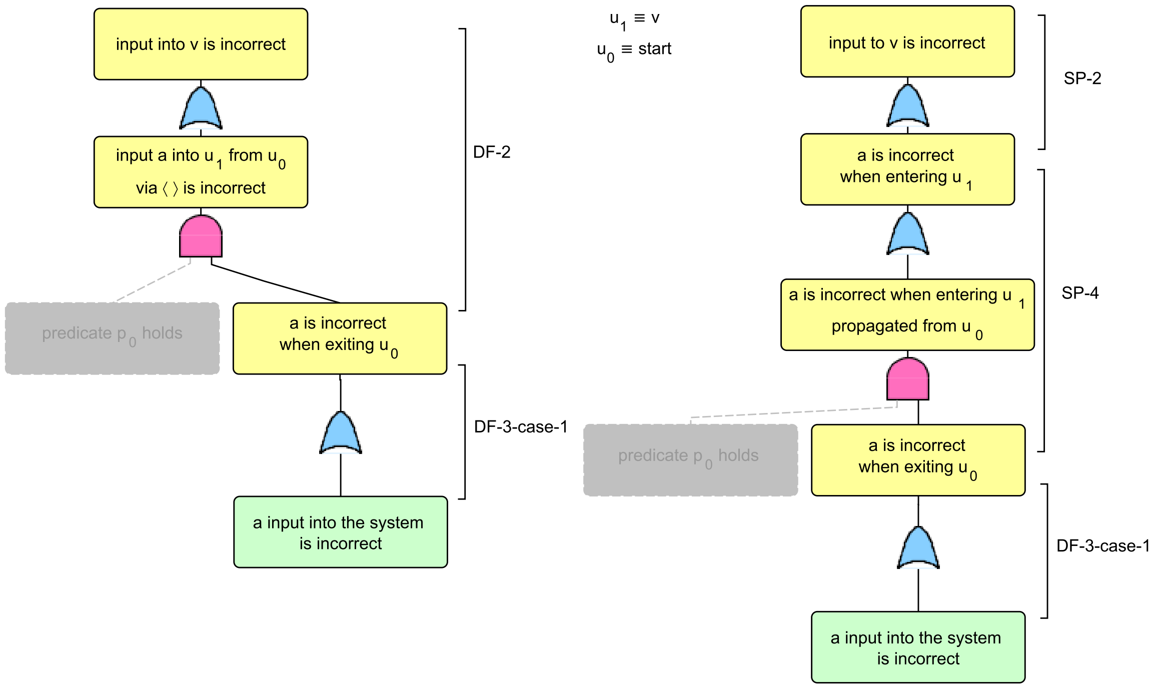

input into v is incorrect
input a into u_1 from u_0 via ⟨ ⟩ is incorrect
predicate p_0 holds
a is incorrect when exiting u_0
a input into the system is incorrect
DF-2
DF-3-case-1
$u_1 \equiv v$
$u_0 \equiv$ start
input to v is incorrect
a is incorrect when entering u_1
a is incorrect when entering u_1 propagated from u_0
predicate p_0 holds
a is incorrect when exiting u_0
a input into the system is incorrect
SP-2
SP-4
DF-3-case-1

Figure A.1. Base Case 1.1. Elaboration of event *"input to v is incorrect"* with an input a used at v and the definition of a from **start** reaches v via unguarded edge $(u \rightarrow v)$. LHS: $T1_{DF}(v)$. RHS: $T1_{SP}(v)$.

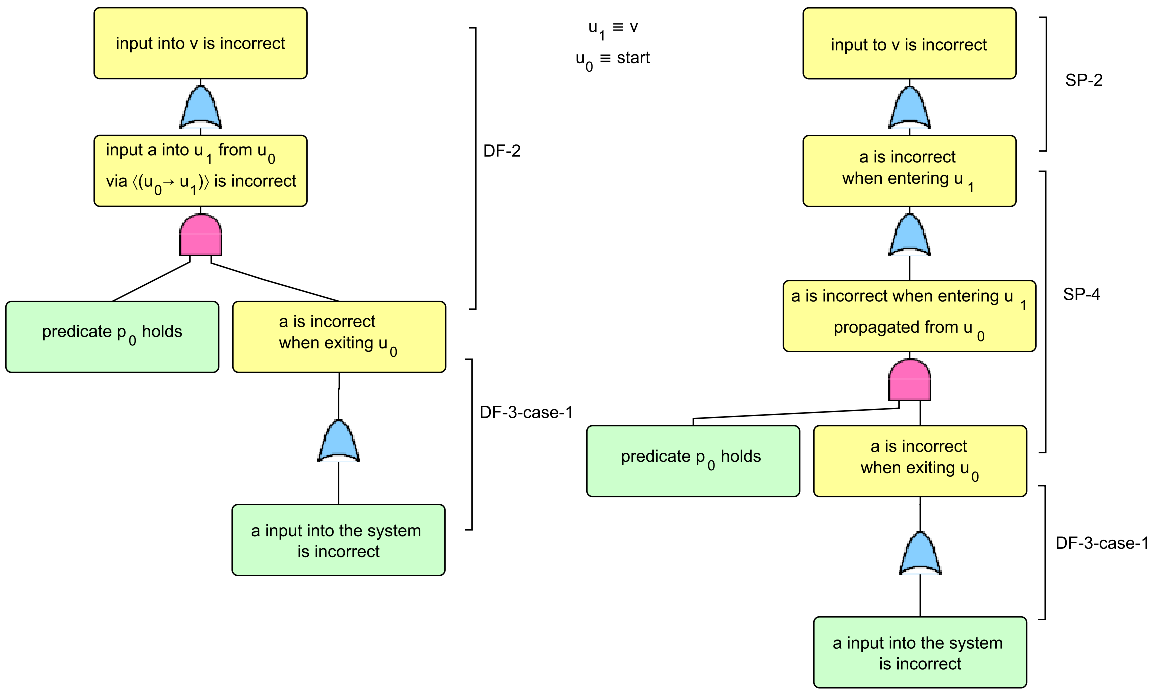

input into v is incorrect
input a into u 1 from u 0 via ⟨⟨(u 0→ u 1)⟩⟩ is incorrect
predicate p 0 holds
a is incorrect when exiting u 0
a input into the system is incorrect
DF-2
DF-3-case-1
u 1 ≡ v
u 0 ≡ start
input to v is incorrect
a is incorrect when entering u 1
a is incorrect when entering u 1 propagated from u 0
predicate p 0 holds
a is incorrect when exiting u 0
a input into the system is incorrect
SP-2
SP-4
DF-3-case-1

Figure A.2. Base Case 1.2. Elaboration of event *"input to v is incorrect"* with an input a used at v and the definition of a from **start** reaches u_1 via $(u \rightarrow v)$ guarded by predicate p_0. LHS: $T1_{DF}(v)$. RHS: $T1_{SP}(v)$.

- Using the SP template set:
  - First applying template SP-2 to the event *"input to u_n is incorrect"*, we get *"a is incorrect when entering u_n"* . Then applying SP-4, we get *"a is incorrect when exiting u_{n-1}"* and potentially primary event *"predicate p_{n-1} holds"* if $(u_{n-1} \to u_n)$ is a guarded edge (see Figure A.3).
  - Obviously, because of the construction of $IDD(v)$, we can see that a is never defined at any u_i, $0 < i < n$. Therefore, since a is not defined at u_{n-1}, template DF-3-case-3 can be applied to get *"a is incorrect when entering u_{n-1}"*. Applying SP-4 again, we get *"a is incorrect when exiting u_{n-2}"* and potentially primary event *"predicate p_{n-2} holds"* if $(u_{n-2} \to u_{n-1})$ is a guarded edge.
  - The two templates DF-3-case-3 and SP-4 are repeatedly applied until we reach *"a is incorrect when exiting u_0"*.
  - Finally, since $u_0 \equiv$ **start**, DF-3-case-1 can be applied and we get the fault tree $T1_{SP}(v)$ as in the right hand side of Figure A.3.

Both of the trees $T1_{DF}(v)$ and $T1_{SP}(v)$ have only one cut set, that is { *"a input to the system is incorrect"*, *"p_{k_1} holds"*,..., *"p_{k_m} holds"*}.

Case 2: *IDD*(v) has more than one element

Since we are still proving the base case of S1(v) with $L(v) = 1$, $IDD(v)$ having more than one element means that there are more than one variable defined at the **start** vertex and used at v. We provide the proof for $IDD(v)$ having two elements. The case of more than two can be proved similarly. Suppose $IDD(v) = \{(a_{u_0}, P), (a'_{u_0}, P')\}$, $u_0 \equiv$ **start**.

- $(a_{u_0}, P) \in IDD(v)$ — a is used at v and the definition of a at u_0 reaches v via the path guided by P. Suppose the path is $u_0 \to u_1 \to ... \to u_n$ where $u_n \equiv v$. And

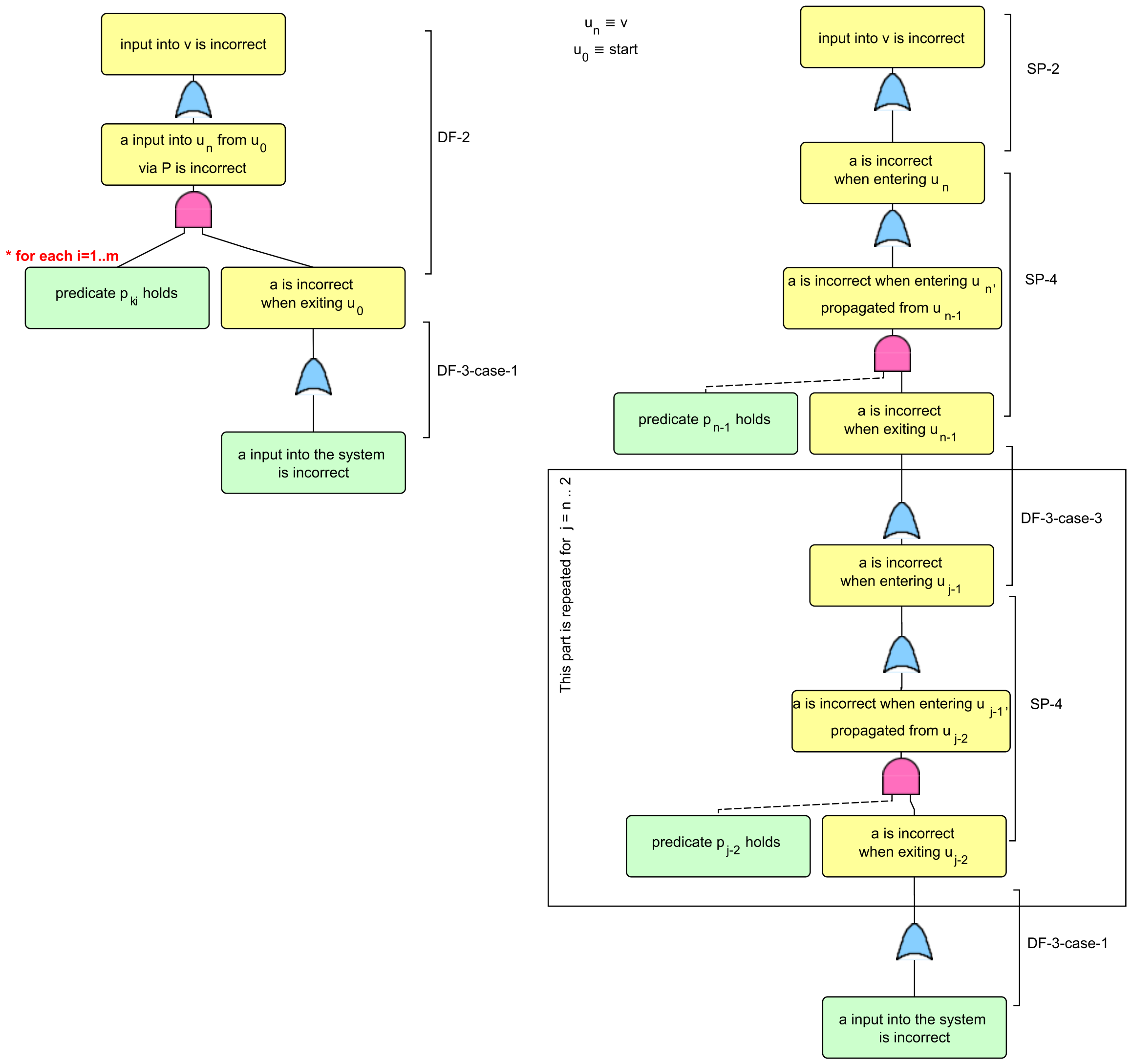

input into v is incorrect
a input into u n from u 0 via P is incorrect
* for each i=1..m
predicate p ki holds
a is incorrect when exiting u 0
a input into the system is incorrect
DF-2
DF-3-case-1
u n ≡ v
u 0 ≡ start
input into v is incorrect
SP-2
a is incorrect when entering u n
SP-4
a is incorrect when entering u n' propagated from u n-1
predicate p n-1 holds
a is incorrect when exiting u n-1
This part is repeated for j = n .. 2
DF-3-case-3
a is incorrect when entering u j-1
SP-4
a is incorrect when entering u j-1' propagated from u j-2
predicate p j-2 holds
a is incorrect when exiting u j-2
DF-3-case-1
a input into the system is incorrect

Figure A.3. Base Case 1.3. Elaboration of event *"input to v is incorrect"* with an input a used at v and the definition of a from **start** reaches v via a path guided by $P = \angle u_0, ..., u_n\rangle$ with $u_0 \equiv$ **start** and $u_n \equiv v$. LHS: $T1(v)_{DF}$. RHS: $T1(v)_{SP}$.

along that path, there are m guarded edges $(u_{k_1} \rightarrow u_{k_1+1}), ..., (u_{k_m} \rightarrow u_{k_m+1})$ with $0 \leq m \leq n$, $0 \leq k_i < n$. Let p_i denote the predicate on the guarded edge $(u_i \rightarrow u_{i+1})$.

- $(a'_{u_0}, P') \in \mathit{IDD}(v)$ — a' is used at v and the definition of a' at u_0 reaches v via the path guided by P'. Suppose the path is $u_0 \rightarrow u'_1 \rightarrow ... \rightarrow u'_{n'}$ where $u'_{n'} \equiv v$. And along that path, there are m' guarded edges $(u'_{k'_1} \rightarrow u'_{k'_1+1}), ..., (u'_{k'_{m'}} \rightarrow u'_{k'_{m'}+1})$ with $0 \leq m' \leq n'$, $0 \leq k'_i <$ n'. Let p'_i denote the predicate on the guarded edge $(u'_i \rightarrow u'_{i+1})$.

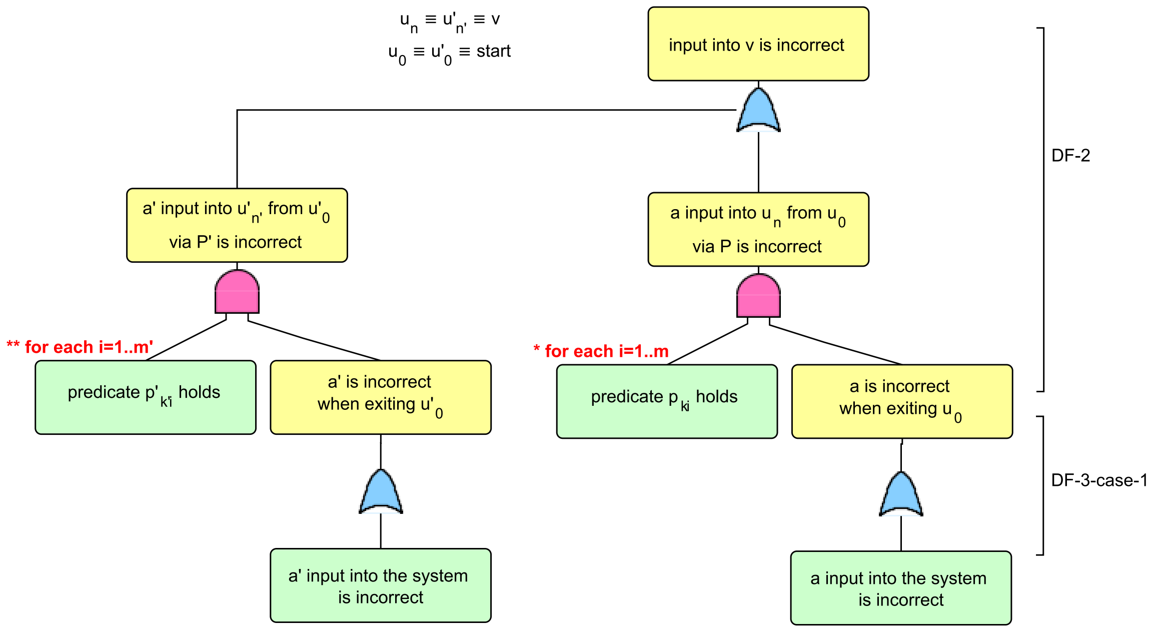
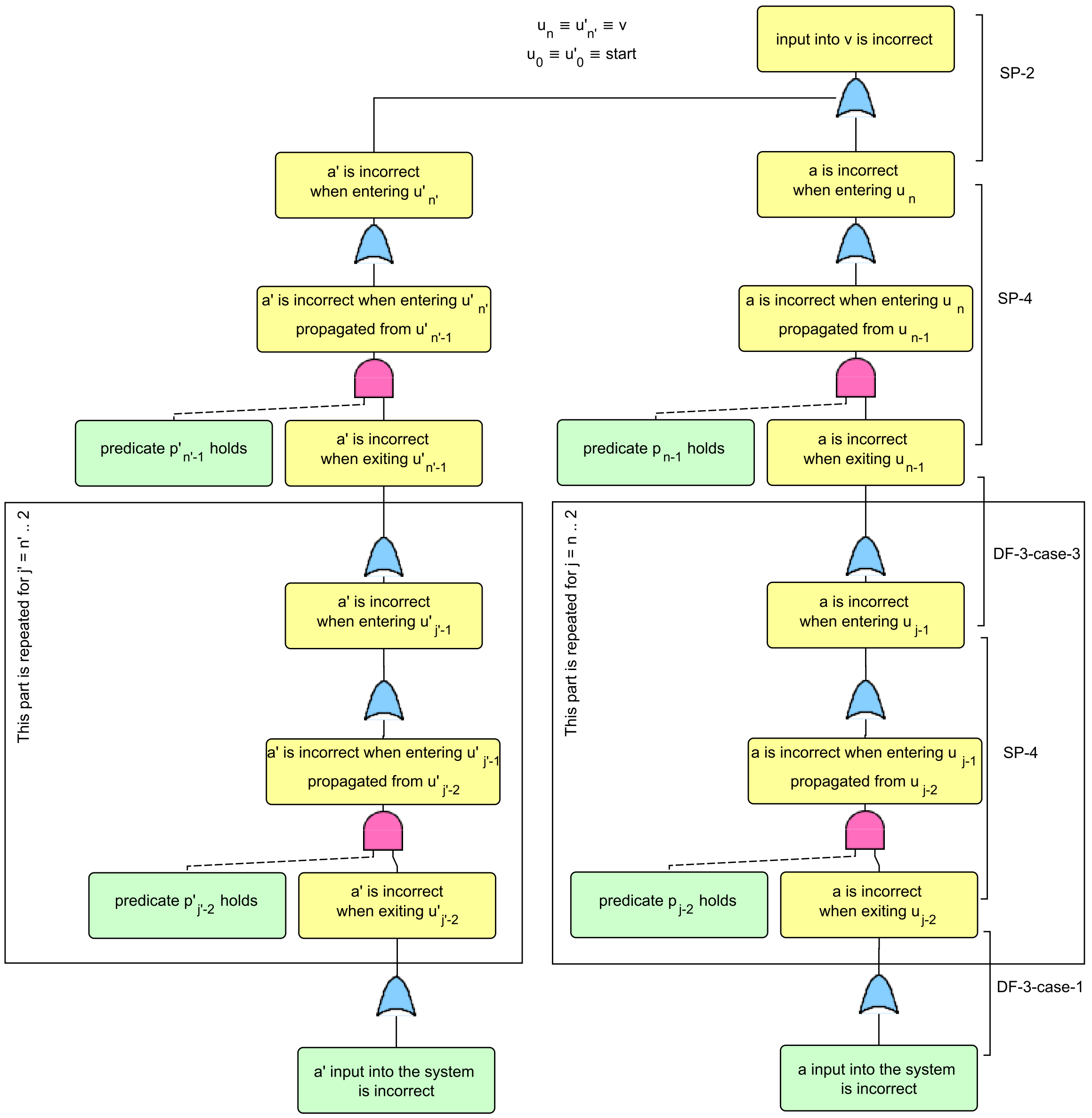

u_n ≡ u'_n' ≡ v
u_0 ≡ u'_0 ≡ start
input into v is incorrect
DF-2
a' input into u'_n' from u'_0 via P' is incorrect
a input into u_n from u_0 via P is incorrect
** for each i=1..m'
predicate p'_ki holds
a' is incorrect when exiting u'_0
* for each i=1..m
predicate p_ki holds
a is incorrect when exiting u_0
DF-3-case-1
a' input into the system is incorrect
a input into the system is incorrect

Figure A.4. $\mathit{T1}_{DF}(v)$ Of Base Case 2. Elaboration of event *"input to v is incorrect"* using the DF set, with v having two input a and a' whose definitions come from the **start** vertex via paths guided by P and P' respectively.

Figure A.4 and Figure A.5 show the elaboration of event *"input to v is incorrect"* using the DF set and SP set respectively. Both fault trees $\mathit{T1}_{DF}(v)$ and $\mathit{T1}_{SP}(v)$ have two cut sets { *"a input to the system is incorrect", "p_{k_1} holds",..., "p_{k_m} holds"*} and { *'a input to the system is incorrect", "$p'_{k'_1}$ holds",..., "$p'_{k'_{m'}}$ holds"*}. In other words $\mathit{T1}_{DF}(v) \cong \mathit{T1}_{SP}(v)$.



$u_n \equiv u'_{n'} \equiv v$
$u_0 \equiv u'_0 \equiv$ start
input into v is incorrect
SP-2
a' is incorrect when entering $u'_{n'}$
a is incorrect when entering u_n
SP-4
a' is incorrect when entering $u'_{n'}$ propagated from $u'_{n'-1}$
a is incorrect when entering u_n propagated from u_{n-1}
predicate $p'_{n'-1}$ holds
a' is incorrect when exiting $u'_{n'-1}$
predicate p_{n-1} holds
a is incorrect when exiting u_{n-1}
This part is repeated for j' = n' .. 2
This part is repeated for j = n .. 2
DF-3-case-3
a' is incorrect when entering $u'_{j'-1}$
a is incorrect when entering u_{j-1}
SP-4
a' is incorrect when entering $u'_{j'-1}$ propagated from $u'_{j'-2}$
a is incorrect when entering u_{j-1} propagated from u_{j-2}
predicate $p'_{j'-2}$ holds
a' is incorrect when exiting $u'_{j'-2}$
predicate p_{j-2} holds
a is incorrect when exiting u_{j-2}
DF-3-case-1
a' input into the system is incorrect
a input into the system is incorrect

Figure A.5. $T1_{SP}(v)$ Of Base Case 2. Elaboration of event *"input to v is incorrect"* using the SP set with v having two input a and a' whose definitions come from the **start** vertex via paths guided by P and P' respectively.

$\therefore$ S1(v) is correct for all v that have $L(v) = 1$. QED base case.

A.1.2 Proof of S1(v) - Induction case $L(v) > 1$

Assume S1(u) is correct with every vertex u that $L(u) < t$.

Let's consider S1(v) with vertex v that $L(v) = t$. Again, first we prove that S1(v) is correct when v has only one input a, and $(a_u, P) \in \mathit{IDD}(v)$ — the definition of a at u reaches v via the path guided by P, and a is used at v. Suppose the path is $u_0 \rightarrow u_1 \rightarrow \ldots \rightarrow u_n$ where $u_0 \equiv u$ and $u_n \equiv v$. And along that path, there are m guarded edges $(u_{k_1} \rightarrow u_{k_1+1}), \ldots, (u_{k_m} \rightarrow u_{k_m+1})$ with $0 \leq m \leq n$, $0 \leq k_i < n$. Let p_i denote the predicate on the guarded edge $(u_i \rightarrow u_{i+1})$.

Similar to case 1 in section A.1.1:

- Using DF set, first DF-2 is applied. Since $L(v) > 1$, u_0 is not the **start** vertex, so DF-3-case-2 is applied next, generating *"a output from u_0 is incorrect"*, then DF-1 is applied next generating *"input into u_0 is incorrect"*. Elaboration of *"input into u_0 is incorrect"* gives us the fault tree $\mathit{T1}_{DF}(u_0)$ (see Figure A.6).

- Using SP set, first SP-2 then SP-4 are applied, then DF-3-case-3 and SP-4 are repeatedly applied until we get to *"a output from u_0 is incorrect"*, then DF-1 is applied next generating *"input into u_0 is incorrect"*. Elaboration of *"input into u_0 is incorrect"* gives us the fault tree $\mathit{T1}_{SP}(u_0)$.

Since both sets use the same template DF-5 to elaborate *"execution incorrectly reaches u_0"*, and $\mathit{T1}_{DF}(u_0) \cong \mathit{T1}_{SP}(u_0)$ (because $L(u_0) = t - 1$ and the induction assumption says S1(v) is correct for all v with $L(v) < t$), we can see that the two fault trees T'_{DF} and T'_{SP} (surrounded by dashed rectangles in Figure A.6) are equivalent. Suppose the same set of cut sets they have is $\mathcal{C}'$. Then, $\mathit{T1}_{DF}(v)$ and $\mathit{T1}_{SP}(v)$ both have the same set of cut sets $\mathcal{C}$ as follows:

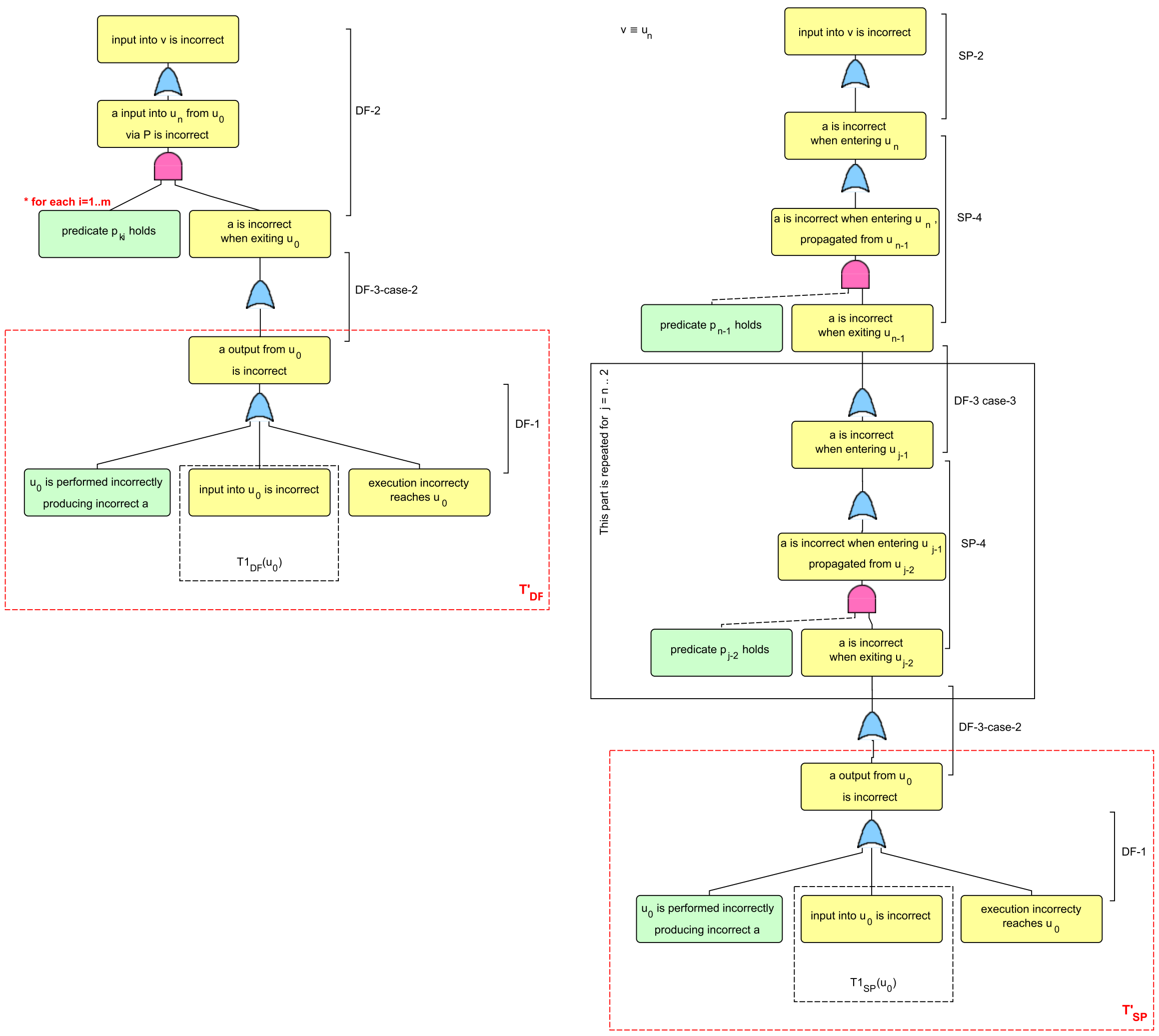

input into v is incorrect
a input into u_n from u_0 via P is incorrect
* for each i=1..m
predicate p_ki holds
a is incorrect when exiting u_0
DF-2
DF-3-case-2
a output from u_0 is incorrect
DF-1
u_0 is performed incorrectly producing incorrect a
input into u_0 is incorrect
execution incorrecty reaches u_0
T1_DF(u_0)
T'_DF
v ≡ u_n
input into v is incorrect
SP-2
a is incorrect when entering u_n
SP-4
a is incorrect when entering u_n, propagated from u_n-1
predicate p_n-1 holds
a is incorrect when exiting u_n-1
This part is repeated for j = n .. 2
DF-3 case-3
a is incorrect when entering u_j-1
SP-4
a is incorrect when entering u_j-1 propagated from u_j-2
predicate p_j-2 holds
a is incorrect when exiting u_j-2
DF-3-case-2
a output from u_0 is incorrect
DF-1
u_0 is performed incorrectly producing incorrect a
input into u_0 is incorrect
execution incorrecty reaches u_0
T1_SP(u_0)
T'_SP

Figure A.6. Induction Case. Elaboration of event *"input to v is incorrect"* with an input a used at v and the definition of a from u_0 reaches v via a path guided by P. $u_0 \neq$ **start** LHS: $T1_{DF}(v)$. RHS: $T1_{SP}(v)$.

$$\mathcal{C} = \{C \cup \{\text{“}p_{k_1} \mathit{holds}\text{”}, \ldots, \text{“}p_{k_m} \mathit{holds}\text{”}\} \mid \forall C \in \mathcal{C}'\}$$

$\therefore\ T1_{DF}(v) \cong T1_{SP}(v)$.

Similar proof can be done as in case 2 of section A.1.1 to show that S1(v) is correct when $L(v) = t$ and v has more than one input.

By induction, we have shown that S1(v) is correct for all v.

With the same reasoning, we can show that S2(v) is correct for all v.

Hence, we see that the two template sets generate equivalent fault trees. QED.

APPENDIX B

PSEUDO-CODE FOR TEMPLATES

```
input  : event e of type "b output from v is incorrect"
output: newEvents– the new intermediate events generated from applying this template

newEvents ⟵ {};
create new OR gate orGate;
e.setGate(orGate);

// case 1:  agent dependence
create new event eAgent of type "v is performed incorrectly producing incorrect b";
orGate.addInEvent(eAgent);
// this is a primary event, not added to the newEvents set

// case 2:  data dependence
create new event eData of type "input into v is incorrect";
orGate.addInEvent(eData);
add eData to newEvents;

// case 3:  control dependence
create new event eControl of type "execution incorrectly reaches v";
orGate.addInEvent(eControl);
add eControl to newEvents;
```
Algorithm B.1: DF-1 Elaborate event *"b output from v is incorrect"*.

```
input  : event e of type "input to v is incorrect"
output: newEvents– the new intermediate events generated from applying this template
newEvents ⟵ {};
create new OR gate orGate;
e.setGate(orGate);
foreach $(a_u, P)$ in IDD(v) do
    create new event eauP of type "input $a$ to v from $u$ via $P$ is incorrect";
    orGate.addInEvent(eauP);
    create new AND gate andGate;
    eauP.setGate(andGate);
    create new event eau of type "$a$ is incorrect when exiting $u$";
    add eau to newEvents;
    andGate.addInEvent(eau);
    foreach $e_i$ in $P$ do
        create new event ep of type "predicate $p(e_i)$ holds"; andGate.addInEvent(ep);
        // this event is primary in the initial fault tree, so it is not
           added to the newEvents set.
```

Algorithm B.2: DF-2 Elaborate event *"input into v is incorrect"*.

```
input  : event e of type "b is incorrect when exiting v"
output: newEvents– the new intermediate events generated from applying this template
newEvents ⟵ {};
create new OR gate orGate;
e.setGate(orGate);
if v == start vertex then
    create new event eInput of type "b input to system is incorrect";
    orGate.addInEvent(eInput);
    // this event is primary, so it is not added to the newEvents set.
else if b is output of v then
    create new event eOutput of type "b output from v is incorrect";
    orGate.addInEvent(varOutputEvent);
    add eOutput to newEvents;
else
    // this is when v is not the start vertex and b is not output of v
    create new event ePassing of type "b is incorrect when entering v";
    orGate.addInEvent(ePassing);
    add ePassing to newEvents;
```

Algorithm B.3: DF-3 Elaborate event *"b is incorrect when exiting v"*.

```
input  : event e of type "b is incorrect when entering v"
output: newEvents– the new intermediate events generated from applying this template

newEvents ⟵ {};
create new OR gate orGate;
e.setGate(orGate);
foreach $(\mathsf{b_u}, \mathsf{P})$ in IDD(b, v) do
    create new event ebuP called "definition of b reaches v from u via P is incorrect";
    orGate.addInEvent(ebuP);
    create new AND gate andGate;
    ebuP.setGate(andGate);
    create new event ebu of type "b is incorrect when exiting u"; add ebu to newEvents;
    andGate.addInEvent(ebu);
    foreach $e_i$ in P do
        create new event ep of type "predicate $p(e_i)$ holds";
        andGate.addInEvent(ep);
        // this event is primary in the initial fault tree so it is not
           added to the newEvents set.
```

Algorithm B.4: DF-4 Elaborate event *"b is incorrect when entering v"*.

```
input  : event e of type "execution incorrectly reaches v"
output: newEvents– the new intermediate events generated from applying this template
newEvents ⟵ {};
create new OR gate orGate;
e.setGate(orGate);
foreach (u → t) in CD(v) do
    create new event eutv called "execution incorrectly reach v from u through (u → t)";
    orGate.addInEvent(eutv);
    create new OR gate orGateTwo; eutv.setGate(orGateTwo);
    create new event eOne called "execution incorrectly reach u then from there reach v
    through (u → t)";
    orGateTwo.addInEvent(eOne);
    create new AND gate andGate;
    eOne.setGate(andGate);
    create new event eOneA of type "execution incorrectly reach u";
    andGate.addInEvent(eOneA);
    add eOneA to newEvents;
    create new event ep of type "predicate p(u, t) holds";
    andGate.addInEvent(ep);
    // this is primary event in the initial fault tree so it is not
       added to the newEvents set.
    create new event epIncorrect of type "predicate p(u, t) incorrectly holds";
    orGateTwo.addInEvent(epIncorrect);
    add epIncorrect to newEvents;
```

Algorithm B.5: DF-5 Elaborate event *"execution incorrectly reaches v"*.

```
input  : event e of type "predicate $p(u,t)$ incorrectly holds"
output: newEvents– the new intermediate events generated from applying this template
newEvents ⟵ {};
create new OR gate orGate;
e.setGate(orGate);
create new event epu called "$p(u,t)$ is incorrect when exiting u";
orGate.addInEvent(epu);
create new OR gate orGateTwo;
epu.setGate(orGateTwo);
foreach a used in predicate $p(u,t)$ do
    create new event eau of type "a is incorrect when exiting u";
    orGateTwo.addInEvent(eau);
    add eau to newEvents;
create new FINAL event ep "$p(u,t)$ holds";
orGate.addInEvent(ep);
```

Algorithm B.6: DF-6 Elaborate event *"predicate $p(u,t)$ incorrectly holds"*.

APPENDIX C

LITTLE-JIL FTA TEMPLATES

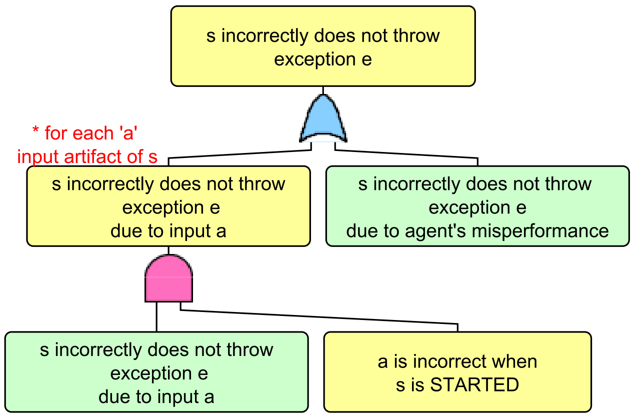

s incorrectly does not throw exception e
* for each 'a' input artifact of s
s incorrectly does not throw exception e due to input a
s incorrectly does not throw exception e due to agent's misperformance
s incorrectly does not throw exception e due to input a
a is incorrect when s is STARTED

Figure C.1. Template LS-3 for Event Type *"s incorrectly does not throw exception e"*.

C.1 Template LS-3: s incorrectly does not throw exception e

A Little-JIL step's specification can contain exceptions the step may throw during its execution. A non-leaf step may throw exceptions propagated from its sub-steps. The decision of throwing an exception from a leaf step depends totally on the agent performing the step. Then the leaf step incorrectly not throwing exception might be caused by:

- one of the inputs into the step being incorrect,
- the agent's misperformance.

Figure C.1 shows the template LS-3 to elaborate the event "s incorrectly does not throw exception e".

BIBLIOGRAPHY

[1] Allen, Frances E. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization* (New York, NY, USA, 1970), ACM, pp. 1–19.

[2] Bishop, Matt, Conboy, Heather, Phan, Huong, Simidchieva, Borislava I., Avrunin, George S., Clarke, Lori A., Osterweil, Leon J., and Peisert, Sean. Insider Threat Identification by Process Analysis.

[3] Chen, Bin. *Improving Processes Using Static Analysis Techniques.* PhD thesis, University of Massachusetts, Amherst, MA 01003, USA, Sept. 2010.

[4] Chen, Bin, Avrunin, George S., Clarke, Lori A., and Osterweil, Leon J. Automatic Fault Tree Derivation from Little-JIL Process Definitions. In *Software Process Change*, Qing Wang, Dietmar Pfahl, David M. Raffo, and Paul Wernick, Eds., no. 3966 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2006, pp. 150–158.

[5] Christov, Stefan, Avrunin, George S., and Clarke, Lori A. Online Deviation Detection for Medical Processes. pp. 395–404.

[6] Christov, Stefan, Avrunin, George S., Clarke, Lori A., Osterweil, Leon J., and Henneman, Elizabeth. A Benchmark for Evaluating Software Engineering Techniques for Improving Medical Processes.

[7] Christov, Stefan, Chen, Bin, Avrunin, George S., Clarke, Lori A., Osterweil, Leon J., Brown, David, Cassells, Lucinda, and Mertens, Wilson. Rigorously Defining and Analyzing Medical Processes: An Experience Report. In *Models in Software Engineering*, Holger Giese, Ed., no. 5002 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Sept. 2007, pp. 118–131.

[8] Clarke, Lori A., Chen, Yao, Avrunin, George S., Chen, Bin, Cobleigh, Rachel, Frederick, Kim, Henneman, Elizabeth A., and Osterweil, Leon J. Process Programming to Support Medical Safety: A Case Study on Blood Transfusion. In *Unifying the Software Process Spectrum*, Mingshu Li, Barry Boehm, and Leon J. Osterweil, Eds., no. 3840 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, May 2005, pp. 347–359. DOI: 10.1007/11608035_29.

[9] Conboy, Heather, Avrunin, George S., and Clarke, Lori A. Modal Abstraction View of Requirements for Medical Devices Used in Healthcare Processes.

[10] Conboy, Heather, Maron, Jason K., Stefan, Christov C., Avrunin, George S., Clarke, Lori A., Osterweil, Leon J., and Zenati, Marco A. Process Modelling of Aortic Cannulation in Cardiac Surgery: Toward a Smart Checklist to Mitigate the Risk of Stroke.

[11] Cooper, Keith D., Harvey, Timothy J., and Kennedy, Ken. Iterative data-flow analysis, revisited. Tech. rep., 2004.

[12] Ericson II, Clifton A. Fault Tree Analysis - A History. In *17th International System Safety Conference* (1999).

[13] Ferrante, Jeanne, Ottenstein, Karl J., and Warren, Joe D. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 319–349.

[14] Friedman, M.A. Automated software fault-tree analysis of Pascal programs. In *Reliability and Maintainability Symposium, 1993. Proceedings., Annual* (1993), pp. 458–461.

[15] Helmer, Guy, Wong, Johnny, Slagell, Mark, Honavar, Vasant, Miller, Les, and Lutz, Robyn. A Software Fault Tree Approach to Requirements Analysis of an Intrusion Detection System. *Requirements Engineering 7*, 4 (Dec. 2002), 207–220.

[16] Hyman, William A.; Johnson, Erin. Fault Tree Analysis of Clinical Alarms. *Journal of Clinical Engineering* (2008), 85–94.

[17] Kildall, Gary A. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1973), POPL '73, ACM, pp. 194–206.

[18] Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., and Wolfe, M. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1981), POPL '81, ACM, pp. 207–218.

[19] Lauer, Christoph, German, Reinhard, and Pollmer, Jens. Fault Tree Synthesis from UML Models for Reliability Analysis at Early Design Stages. *SIGSOFT Softw. Eng. Notes 36*, 1 (Jan. 2011), 1–8.

[20] Lazarus, Eric L. Change Result of Election Successfully. attack tree, 2010.

[21] Leveson, N.G., Cha, S.S., and Shimeall, T.J. Safety verification of Ada programs using software fault trees. *IEEE Software 8*, 4 (July 1991), 48–59.

[22] Mhenni, F., Nguyen, Nga, and Choley, J.-Y. Automatic fault tree generation from SysML system models. In *2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)* (July 2014), pp. 715–720.

[23] Moore, A. P., Ellison, R. J., and Linger, R. C. Attack Modeling for Information Security and Survivability. Tech. rep., Carnegie Mellon University, Software Engineering Institute, 2001.

[24] Pai, G.J., and Bechta Dugan, J. Automatic synthesis of dynamic fault trees from UML system models. In *13th International Symposium on Software Reliability Engineering, 2002. ISSRE 2003. Proceedings* (2002), pp. 243–254.

[25] Phan, Huong, Avrunin, George S., Clarke, Lori A., Leon, Osterweil J., and Bishop, Matt. A Systematic Process-Model-based Approach for Synthesizing Attacks and Evaluating Them. In *Presented as part of the 2012 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections* (Berkeley, CA, 2012), USENIX.

[26] Schneier, Bruce. Attack Trees. *Dr. Dobb's Journal* (1999).

[27] Shin, Seung Yeob, Brun, Yuriy, Osterweil, Leon J., Balasubramanian, Hari, and Henneman, Philip L. Resource Specification for Prototyping Human-Intensive Systems. In *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE)* (2015), pp. 332–346.

[28] Simidchieva, Borislava I., Engle, Sophie J., Clifford, Michael, Jones, Alicia Clay, Peisert, Sean, Bishop, Matt, Clarke, Lori A., and Osterweil, Leon J. Modeling and Analyzing Faults to Improve Election Process Robustness. In *Proceedings of the 2010 International Conference on Electronic Voting Technology/Workshop on Trustworthy Elections* (Berkeley, CA, USA, 2010), EVT/WOTE'10, USENIX Association, pp. 1–8.

[29] Simidchieva, Borislava I., and Osterweil, Leon J. Generation, Composition, and Verification of Process Families. In *SPLC '14: Proceedings of the 18th International Software Product Line Conference* (Italy, Sept. 2014), pp. 207–216.

[30] Vesely, WE, Dugan, J, Fragola, J, Minarick, J, and Railsback, J. Fault Tree Handbook with Aerospace Applications. *NASA Office of Safety and Mission Assurance* (2002). NASA HQ.

[31] Wang, Danhua, Pan, Jingui, Avrunin, George S., Clarke, Lori A., and Chen, Bin. An Automatic Failure Mode and Effect Analysis Technique for Processes Defined in the Little-JIL Process Definition Language.

[32] Ward, J.R., Lyons, M.N., Barclay, S., Anderson, J., Buckle, P., and Clarkson, P.J. Using fault tree analysis (FTA) in healthcare: a case study of repeat prescribing in primary care. In *Patient Safety Research: Shaping the European Agenda* (2007).

[33] Wise, Alexander. Little-JIL 1.5 Language Report. Tech. rep., Uinversity of Massachussetts Amherst, 2006.

[34] Zhao, Zhao, and Petriu, Dorina. {UML Model to Fault Tree Model Transformation for Dependability Analysis}. In *Proceedings of the International Conference on Computer and Information Science and Technology* (Ottawa, Ontario, Canada, May 2015).