Verification and Anomaly Detection for Event-Based Control of Manufacturing Systems

by

Lindsay Victoria Allen

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering: Systems)
in The University of Michigan
2010

Doctoral Committee:

Professor Dawn M. Tilbury, Chair
Professor Stephane Lafortune
Associate Professor Jionghua Jin
Associate Research Scientist James R. Moyne

© Lindsay Victoria Allen 2010
All Rights Reserved

This work is dedicated to Bobby and my family, for all their love and support.

ACKNOWLEDGEMENTS

I would like to thank Rackham Graduate School, the NSF Engineering Research Center for Reconfigurable Manufacturing Systems, NSF grants EEC 95-92125 and CMS 05-28287, and the NSF EAPSI Fellowship program for each providing financial support at some point during my doctoral work. This very worthwhile pursuit would not have been possible without this generosity.

I am grateful for Professor Dawn Tilbury's guidance as my PhD advisor. She pushed me to do better and be stronger than I thought I could, and yet encouraged my independence.

Numerous thanks go to my student and research colleagues, from whom I have learned so much, both academic and otherwise. The RFT students were always willing to be a sounding board when I got stuck and made work enjoyable even when things got tough. I thank James Moyne for always reminding me of the industry perspective, and Kiah Mok Goh for helping me see the commonalities in research and people, regardless of the part of the world.

Finally, I want to express my deep appreciation to Bobby, my family, and my friends. Without you, I would not have had the passion, persistence, nor sanity to reach this goal.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF TABLES vii

LIST OF FIGURES viii

ABSTRACT x

CHAPTER

I. Introduction 1

1.1 Motivation 1
1.2 Research Approaches 4
1.2.1 Verification 4
1.2.2 Detection Solution 6
1.2.3 Detection Application 7
1.3 Contributions 7
1.4 Summary 9

II. Background and Related Work 10

2.1 Discrete Event System (DES) Models 10
2.1.1 Finite State Machine (FSM) 11
2.1.2 Petri Nets 12
2.1.3 System Identification for DES 14
2.2 Logic Controllers 15
2.2.1 IEC 61499 16
2.2.2 ECA MFSM 17
2.3 Handling Faults in DES 19
2.3.1 Verification 19
2.3.2 Fault Detection and Diagnosis 20

III. Verification of Input Order Robustness 22

3.1 Related Concepts 23
3.2 Logic Controllers and Networks of Controllers 25
3.2.1 Logic Controllers 25
3.2.2 Network of Controllers 27
3.3 IOR Theory 30
3.3.1 Input Order Robustness 31
3.3.2 Pairwise Input Order Robustness of a Single Controller 33

3.3.3 Input Order Robustness of a Network of Controllers 34
3.4 IOR Verification Procedure 39
3.4.1 General Procedure 39
3.4.2 Example System for Verification 41
3.4.3 First Steps of Verification Example 44
3.4.4 ECA MFSM Verification 46
3.4.5 IEC 61499 Verification 46
3.5 Application to IEC 61499 47
3.5.1 Open Execution Semantics Issues 47
3.5.2 Application of Verification to IEC 61499 Network of Controllers . . 51
3.6 Computational Complexity 55
3.7 Conclusions 56

IV. Anomaly Detection for Event-Based Systems Without Pre-Existing Formal Models 57

4.1 Description of Small Manufacturing Cell 58
4.2 System of Processes That Interact Through Shared Resources 60
4.2.1 Intuition and Examples for SPSRs 60
4.2.2 Formal Definitions for SPSRs 62
4.3 Petri Net Models for System of Processes that Interact Through Shared Resources 64
4.4 Problem Statements 74
4.5 Model Generation 75
4.5.1 Steps for Set-Up 75
4.5.2 $\alpha+$ Algorithm 76
4.5.3 Model Generation Algorithm 80
4.5.4 Theory 87
4.6 Performance Assessment 89
4.7 Anomaly Detection 91
4.8 Application of Solution to Simulated RFT Cell 93
4.9 Conclusions 97

V. Application of Anomaly Detection to Industrial Manufacturing Line . . . 99

5.1 Description of Machining Cell 99
5.2 Initial Application of Solution 102
5.3 Inconsistencies Between Academic Assumptions and Industry Realities . . . 104
5.3.1 Observable Events to Acquire/Release Resources 105
5.3.2 String of Ordered Events 107
5.3.3 Consistent Mapping Between Event and Meaning 109
5.3.4 System Starts in Initial State for Each Event Stream 111
5.3.5 Separate, Labeled Event Streams 112
5.4 Barriers to Application to Machining Cell 113
5.5 Conclusions 114

VI. Application of Anomaly Detection to Simulated Systems 115

6.1 Multiple Bit Change (MBC) Inconsistency 115
6.1.1 Handling Combination Events in DES 118
6.1.2 MBC Algorithm 119
6.1.3 Application of MBC Decision Algorithms to Small Manufacturing Cell 122
6.1.4 Limitations of Heuristic MBC Algorithms 126

6.2 Initial State Inconsistency 127
6.2.1 Model Producing Event Stream From Unknown Initial State 127
6.2.2 Theory and Algorithms 130
6.2.3 Application of Initial State Algorithms to Small Manufacturing Cell 134
6.3 Results for Simulated RFT Cell 138
6.4 Conclusions 141

VII. Conclusions and Future Work 142

7.1 Verification Contributions 142
7.2 Anomaly Detection Contributions 143
7.3 Future Work 145

VIII. Appendix: List of Acronyms 150

Bibliography **152**

LIST OF TABLES

Table

3.1 Events for Cell 1 Controller 42

3.2 Verification of Event Pairs: P = cannot occur nearly same time, D = order should matter, S = need to verify 44

3.3 States Removed by Restrictions R1 - R3 where $\bullet$ = any valid value 45

3.4 Event Modifications for Application to IEC 61499 47

3.5 IEC 61499 Execution Open Issues 51

4.1 Events in Manufacturing Cell 59

4.2 Resource information for Manufacturing Cell 60

4.3 Comparison of Petri net formalisms that use resources 67

4.4 Comparison of Sound SWF-nets and TPs 68

4.5 Event Pair Occurrences in Example 78

4.6 Ordering Relations for Event Pairs in Example 78

4.7 Ordering Relations Due to Resources for Event Pairs in Process 1 85

4.8 Relationships from Event Log Minus Relationships Due to Resources for Process 1 85

4.9 Unique Models Generated from Algorithm for Given Log 94

4.10 Performance Results in Percentage for Each Model Generation Event Log, Where Num is the number of events in the stream and the results are expressed as Max, Min, and Mean 96

5.1 Physical and Data Events That Acquire and Release Cell's Resources With Unobservable Events in *italics* 104

5.2 Inconsistencies and Their Resolutions, Where Responsible Party is Either **A**cademia or **I**ndustry 113

6.1 Ordering Relations for Event Pairs in MBC Example 125

LIST OF FIGURES

Figure

1.1 Closed loop system consisting of plant and controller that exchange events. 2

1.2 Illustration of input order robustness error for Part 1 arriving 4

2.1 FSM that describes simple machine. 12

2.2 Petri net that describes simple machine. 14

2.3 IEC 61499 controller for Cell 1 of RFT. 16

2.4 ECC for Cell 1 FB. 17

2.5 Schematic ECA MFSM for cell controller 18

3.1 Simple network of controllers where $\pi^A \subseteq \{a, c\}$, $\pi^B \subseteq \{g_{A2}(\pi^A, x^A), b\}$ and the outputs of Component A are segmented into outputs to the environment and to Component B, $g_A(\pi^A, x^A) = g_{A1}(\pi^A, x^A) \cup g_{A2}(\pi^A, x^A)$ 29

3.2 Network of controllers with no feedback loops, which means it can be labeled using partial ordering. 30

3.3 Left: Network of controllers with two feedback loops, Right: Equivalent network of controllers where components have been combined to eliminate feedback loops . 38

3.4 Cell controller's decision-making 43

3.5 Fixture and carriage network example (based on [11]) 53

3.6 Fixture and carriage ECCs (based on [11]). 54

4.1 Illustration of manufacturing cell, where dashed lines show possible movements of the robot and milling machine, and their associated events (in italics) 58

4.2 Example S^2PR from [21] where resources represented by places $r2$, $r3$, and $r4$ . . . 65

4.3 Example S^2PR from [21] modified to have different resource usage 67

4.4 TP from process 1 of example manufacturing cell 70

4.5 TPR Model, Generated for Process 1 with NotR2, m_1 causes d_1 72

4.6 TPR Model Generated for Process 2 of Manufacturing Cell Example [$TPCR$ Model for Process 2 of Assembly Ex] 73

4.7 Example $STPR$, Whole Model From Combining Third Process 1 Model and Sole Process 2 Model 75

4.8 Model result from applying $\alpha+$ algorithm to example system 79

4.9 Models Generated for Process 1 Before Decisions Added 85

4.10 Models Generated for Process 1 86

4.11 Example $STPR$ for Cell 1 (composed of TPR for Part 1, TPR for Part 2, and $TPCR$ for Empty) with resource places Pallet Stop, Robot, Empty, M1, M2, NotM1, and NotM2. 95

5.1 Machining cell that consists of two gantries (G1, G2) operating in serial and six CNCs (M1-M6) operating in parallel 100

5.2 Data collection set-up for machining cell 101

5.3 An MBC where two bits change per event and the three options are: keep it as a unique event, or split it into one of two possible sequences 109

6.1 Small MBC Manufacturing Cell 125

6.2 Model created by model generation for MBC RFT cell 140

ABSTRACT

Many important systems can be described as discrete event systems, including a manufacturing cell and patient flow in a clinic. Faults often occur in these systems and addressing these faults is important to ensure proper functioning. There are two main ways to address faults. Faults can be prevented from ever occurring, or they can be detected at the time at which they occur. This work develops methods to address faults in event-based systems for which there is no formal, pre-existing model. A primary application is manufacturing systems, where reducing downtime is especially important and pre-existing formal models are not commonly available. There are three main contributions.

The first contribution is formalizing input order robustness - inputs occurring in different orders and yielding the same final state and set of outputs - and creating a method for its verification for logic controllers and networks of controllers. Theory is developed for a class of networks of controllers to be verified modularly, reducing the computational complexity. Input order robustness guarantees determinism of the closed-loop system.

The second contribution is an anomaly detection solution for event-based systems without a pre-existing formal model. This solution involves model generation, performance assessment, and anomaly detection itself. A new variation of Petri nets was created to model the systems in this solution that incorporates resources in a less restrictive way. The solution detects anomalies and provides information about

when the anomaly was first observed to help with debugging.

The third contribution is the identification and resolution of five inconsistencies found between typical academic assumptions and industry practice when applying the anomaly detection solution to an industrial system. Resolutions to the inconsistencies included working with industry collaborators to change logic, and developing new algorithms to incorporate into the anomaly detection solution. Through these resolutions, the anomaly detection solution was improved to make it easier to apply to industrial systems.

These three contributions for handling faults will help reduce down-time in manufacturing systems, and hence increase productivity and decrease costs.

CHAPTER I

Introduction

1.1 Motivation

Many systems that are essential parts of our functioning society can be described as discrete event systems (DES). A discrete event system has a set of discrete states ($x \in X$) that it transitions among due to the asynchronous occurrence of events ($e \in E$), where the system's next state is entirely dependent on its current state and the event. For example, the operation of a machine can be modeled as a DES. The events could include *start*, *finish*, *break*, and *repair* and the states would be IDLE, BUSY, and BROKEN, where for instance, the machine would transition from state BROKEN to state IDLE when the event *repair* occurs. Patient flow through a clinic could also be modeled as a DES. Events could include *start app't*, *doctor available*, and *get test* with possible states such as WAITING, CONSULTATION, WAITING FOR DIAGNOSIS where, for example, from state CONSULTATION the event *get test* would transition the patient to the state WAITING FOR DIAGNOSIS. One type of discrete event system is a logic control system, which consists of a plant (something physical to be controlled) and a logic controller, as illustrated in Figure 1.1. Sometimes these physical systems have faults (behave improperly) or anomalies (behave in a way other than expected), and these faults and anomalies must be

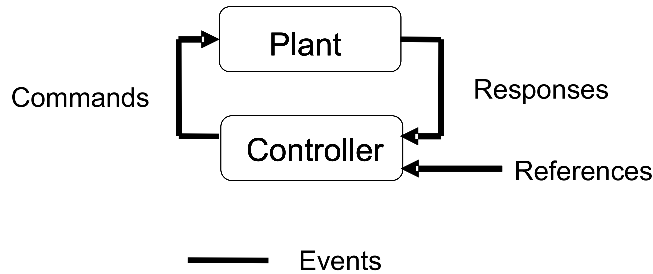


Figure 1.1: Closed loop system consisting of plant and controller that exchange events.

addressed for the systems to work well.

The Reconfigurable Factory Testbed (RFT) is a small-scale manufacturing line at the University of Michigan that is used for research and instructional purposes [44]. It experienced a reoccurring fault that served as motivation for much of this research. The system includes two processing cells, a conveyor to connect them, and an RFID system to track parts and pallets and update tables that maintain the queue for each processing cell. In operating the testbed, there were some intermittent errors that caused incorrect operation of the system and were difficult to diagnose. Correct operation of the system requires that when a pallet arrives at a processing cell, the cell receives information about which part the pallet has on it and treats the pallet appropriately. An empty pallet has a completed part loaded on it if one is waiting, waits if a part is being machined, and otherwise is simply released. A pallet with an unfinished part is unloaded if a machine is available to process it and otherwise is released. Most of the time, the system operated correctly. Occasionally and unpredictably the cell would release a pallet with an unfinished part, as if it were empty, and then treat the next pallet as if it had an unfinished part. This erroneous behavior did not trigger any fault flags but resulted in a part not being processed and another pallet being treated incorrectly.

In part because of its intermittent, infrequent nature and lack of fault response,

this error was difficult to diagnose. Eventually, the root cause was found – a controller that was not properly programmed. The controller was designed expecting some inputs to occur in a particular order, but occasionally network delays or the order in which certain control modules were executed would cause the inputs to occur in a different order, and this different order resulted in different commands to the cell about what to do with the pallets and incorrect operation. When a pallet arrived at the Cell 1 pallet stop, the conveyor controller notified the high-level controller, which queried the RFID tables to determine which part that pallet carried, and set the appropriate *LoadPartw* (*LPw*) input high (where w = 1, 2, or 0 for part 1, 2, or empty) and *PartReady* (*PR*) input high to indicate that a pallet with part w had arrived. These two inputs traveled different paths through the Cell 1 controller. *LPw* went through two control modules before reaching the main module, whereas *PR* went through a different module and then directly to the main module. The main module of Cell 1's controller expected the *LPw* to be set high and, in the same scan or shortly thereafter, *PR* to also be set high. The module's state and outputs for this sequence are shown in the top trajectory of Figure 1.2, where we assume $w =$ 1. When the error occurred, however, the *PR* input reached the main module before the *LP1* input, causing the controller module to believe that it had no information on the part present and thus assume that it was empty and release the pallet, as shown in the bottom trajectory of Figure 1.2. When *LP1* eventually arrived, the controller held this information until it next received *PR* for the subsequent pallet, and hence treated the subsequent pallet as if it were the first.

This error was addressed by modifying the controller's main module so that the inputs $\{LP1, PR\}$ could arrive in either order and produce the same outcome – treating each pallet correctly. Once the source of the error was found, it was easily

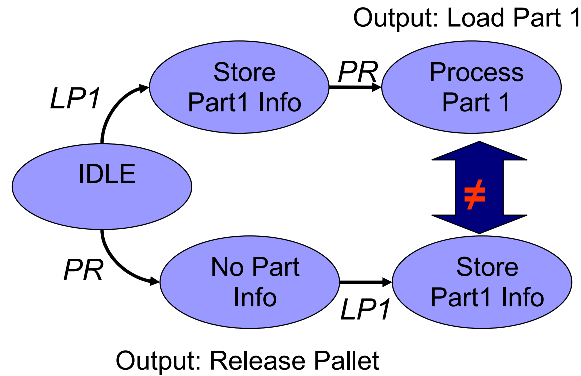


Figure 1.2: Illustration of input order robustness error for Part 1 arriving

remedied, but finding the source was a time intensive and difficult manual process. To address this problem and problems like it in systems that can be modeled as discrete event systems, two different approaches were developed – verification to prevent this type of problem from ever occurring, and anomaly detection to determine when such a problem has occurred and help determine the root cause.

1.2 Research Approaches

1.2.1 Verification

Verification checks that a system possesses a certain property or class of properties. If the system does have a certain property, then any faults associated with not having that property are guaranteed to be avoided, and if the system does not have it, then some possible faults are known and a decision can be made about whether, if possible, the system should be modified to possess the property. For discrete event systems, properties that are commonly verified include controllability, language specifications, absence of deadlock, and diagnosability. Controllability is the property that a system modeled by a DES could be controlled to meet a given specification using a controller also implemented in DES. Language specifications are restrictions on the desired behavior of a controlled system. Deadlock is the property where there are one or

more states that can be reached from which no transitions to other states are defined. Diagnosability is the property of being able to determine that a particular fault has occurred when the fault is described by an unobservable event. Lack of a particular property was identified as the root cause for the RFT cell fault.

Logic controllers receive inputs from their environment, process these inputs, and produce outputs. If the environment is non-deterministic, then two or more inputs may arrive at nearly the same time or can arrive in any order. In such cases, the behavior of the logic controller – its final state and set of outputs – may depend on the order in which the inputs arrive. A logic controller that behaves this way is not *input order robust* for this set of inputs. If a logic controller should be input order robust for a set of inputs but is not, then it can cause faults in the controlled system that are difficult to debug. The RFT cell is an example of a logic controller that was not input order robust. Note that input order robustness only requires that the set of outputs be the same, not the order of those outputs.

Input order robustness is an important property for a controller to possess because then the closed-loop behavior of the controller with its environment is deterministic, so the closed-loop behavior can be predicted, and thus checked as to whether it meets desired specifications. To the author's best knowledge, there is currently no proposed formal procedure for such verification. In Chapter III, input order robustness is formally defined and a verification method for it developed. Input order robustness verification consists of identifying which sets of inputs can occur in different orders and should have the same behavior regardless of the order, and then simulating the system to check if these different orders actually produce the same behavior. Theory is developed to support this verification method for both logic controllers and networks of logic controllers, and examples of both are illustrated. The relationship

between input order robustness verification and open execution semantics for IEC 61499, a particular logic control formalism, are also explored. By verifying that a system is input order robust, faults associated with a lack of input order robustness are prevented from ever occurring. A disadvantage of this verification approach is that it checks only input order robustness. In general, verification can only check known properties, but there can be many types of faults and anomalies.

1.2.2 Detection Solution

As another approach to addressing the type of fault found in the RFT cell, Chapter IV develops an anomaly detection solution that could also help address faults not related to input order. In discrete event systems (DES), streams of events represent the behavior of the system, and some faults that occur may be evident in the event stream. For example, there could be a fault in the interaction of a machine and controller that causes a part to not be properly processed, even if no fault alarm is triggered. Detecting and debugging these faults efficiently is important to keeping complex systems, such as those in manufacturing, running well and reducing their downtime.

One of the main challenges in this detection problem is that the RFT cell, as well as many industry systems, does not have a pre-existing formal mathematical model. Thus, this solution consists of model generation using observed event streams that do not have faults, followed by anomaly detection for observed event streams that may or may not have faults and then fault detection to determine whether the anomalies are actually faults. An anomaly is an unusual occurrence in the behavior of the system, and some anomalies may be faults, while others may be infrequent yet correct system behavior. This research proposes a solution to the anomaly detection problem based on model generation; fault detection based on these anomalies is left for future work.

Anomaly detection on its own can provide insight into possible problems in a system by highlighting where observed behavior differs from the no-fault behavior described by the models. The main disadvantage to this approach over verification is that it provides information about an anomaly (possible fault) once it has already occurred, rather than preventing it from ever happening. Detection also requires that the system is already operating.

1.2.3 Detection Application

The anomaly detection solution from Chapter IV was applied to an industrial machining cell. In the process of that application, a number of inconsistencies were found between academic assumptions and industry realities. Chapter V presents this industrial application and the academic versus industry inconsistencies, including resolutions found to address them. This set of inconsistencies is not all-inclusive but several of the inconsistencies addressed would typically be faced in most industry systems. Although a few continuing barriers prevented completion of the industrial application, the inconsistencies' resolutions have been tested through application to simulated systems that have been developed to mimic the industry conditions described in Chapter V.

1.3 Contributions

The first main contribution is input order robustness verification. Input order robustness is defined and a verification procedure developed so that the closed-loop behavior of logic control systems can be guaranteed as deterministic and prevent faults associated with the lack of this property. The theory generated to support input order robustness verification makes use of modularity to improve the verification procedure's scalability. Additionally, the role of input order robustness in mitigating

some of the problems associated with open execution semantics in the IEC 61499 standard is demonstrated.

The second main contribution is development of an anomaly detection solution for event-based systems without a pre-existing formal model, which is a significant step toward fault detection for such systems. A model generation algorithm was developed to estimate models of the system's behavior that extended an existing algorithm by creating additional model variations and incorporating resources. A variation of a resource based Petri net was created that is less restrictive to use as the modeling formalism for this model generation. The anomaly detection algorithm was created and it incorporates performance assessment of the models to improve its own performance on detecting anomalies.

The third main contribution is identification and resolution of five inconsistencies between academic assumptions that are common in DES and related areas and industry realities of the systems they are trying to model and analyze. Resolutions to some of these inconsistencies produced recommendations for industry partners in the design of data collection to make analysis using DES easier. Other resolutions required developing significant academic contributions in the form of both theory and algorithms. A heuristic algorithm was developed to decide how to split multiple bit changes, a common occurrence in programmable logic controller (PLC) data, based on the bits' relationships. A necessary condition was developed for a DES to produce a stream of events starting from an unknown state; this condition was implemented in an algorithm.

1.4 Summary

To address faults that occur in systems that can be modeled by discrete event systems (DES), two different approaches were developed. Some faults can be prevented by verifying that the logic controller or network of controllers is input order robust, and resolving any violations of that property. Alternatively, faults can be detected in the system through anomaly detection using model generation. These two approaches are associated with two of the main areas of contribution, with the third area being identifying and resolving inconsistencies between common academic assumptions and industry practice found during industry application of the anomaly detection solution.

CHAPTER II

Background and Related Work

This chapter reviews work related to input order robustness verification and anomaly detection. The informal description of discrete event system in Chapter I is formalized, examples presented, and modeling formalisms discussed in Section 2.1. Logic controllers are defined and two of their modeling formalisms, IEC 61499 and Event-Condition-Action Modular Finite State Machine (ECA MFSM), are described in Section 2.2. Section 2.3 discusses two main approaches to addressing faults – verification and detection/diagnosis.

2.1 Discrete Event System (DES) Models

Definition 1 (Event)**.** An *event* is something that occurs instantaneously and transitions a system from one state to another.

Definition 2 (Discrete event system)**.** A *discrete event system (DES)* is a discrete-state, event-driven system, that is, its state evolution depends entirely on the occurrence of asynchronous discrete events over time [9]. Each DES has a set of events E, a set of discrete states X, and a function that describes how the events cause the system's state to change.

Definition 3 (Event stream, prefix of event stream)**.** An *event stream* σ is a finite

ordered sequence of events drawn from event set E, $\sigma = e_1 e_2 ... e_m$. If an event stream σ' is equal to $e_1 ... e_k$ for some $1 \leq k \leq m$, then σ' is a prefix of σ and is denoted by $\hat{\sigma}$.

When a DES operates, it generates an event stream and a corresponding sequence of states. As mentioned in Chapter I, many systems can be modeled as a DES such as the operation of a machine, patient flow through a clinic, and traffic at an intersection.

Discrete event systems should not be confused with the similar sounding discrete-time systems. A discrete-time system is one in which values, such as inputs and outputs, are sampled at discrete times, but in general these values are from a continuous signal in contrast to events that occur asynchronously.

DES can be modeled using a number of formalisms. The idea of a DES is often best understood through an example modeled by a finite state machine (FSM), which is presented in Section 2.1.1. Another modeling formalism for DES is Petri nets. Petri nets can model a broader class of systems, often with a more concise representation, and are introduced in Section 2.1.2. Some systems that can be modeled by a DES do not have such a model available, but system identification can be used in some cases to create such a DES model (Section 2.1.3).

2.1.1 Finite State Machine (FSM)

For a finite state machine (FSM), sometimes called a finite state automata, each state is represented by a circle, generally containing the name of the state, and each event is represented by a name, where arcs connect states and each arc is labeled with the name of the event that causes that transition. More detail on this formalism is given in Chapter 2 of [8]. An example DES is illustrated in Figure 2.1 that describes

the behavior of a simple machine. This system has three discrete states – IDLE, BUSY and BROKEN – represented by the circles, and four events – *start*, *finish*, *break*, and *repair* – each of which label a transition between states. The unconnected arrow to the state IDLE indicates that this is the initial state. In this system, the machine starts in the initial state IDLE and remains in that state until the event *start* occurs, at which point the machine transitions to the state BUSY as it is busy processing a part. From BUSY, either the machine finishes processing the part (*finish* occurs) or the machine breaks (*break* occurs), which transition the system to IDLE and BROKEN respectively. From BROKEN, the machine can return to idle only when the event *repair* occurs. Although this formalism has the advantage of being easy to understand and model for smaller systems, it is less expressive and can be more difficult to use for larger systems. Another common DES modeling formalism that avoids some of these drawbacks is Petri nets, which are used in the anomaly detection method and thus are described next.

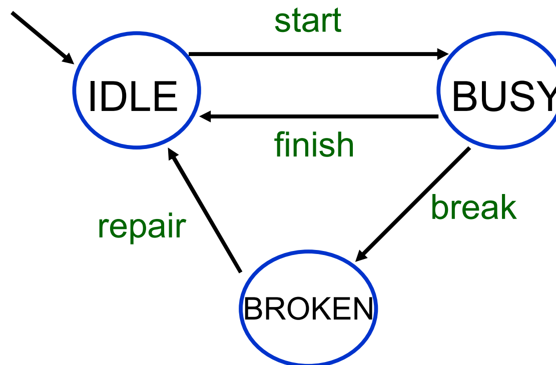


Figure 2.1: FSM that describes simple machine.

2.1.2 Petri Nets

A common and more expressive DES formalism is Petri nets, of which a brief overview is provided here. A more in-depth treatment is available in Chapter 4 of

[8].

Definition 4 (Petri net, Marked Petri net)**.** A *Petri net* N is a graph (P,T,F), where P is a finite set of places, T is a finite set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs from places to transitions and from transitions to places. A *marked Petri net* N =(P,T,F,M) is a Petri net (P,T,F) with M as a marking of the set of places P that represents the Petri net's state. The *initial marking* M_0 of a Petri net corresponds to the initial state.

The marking M is the number of tokens in each place. $M(p_i)$ refers to the number of tokens in place p_i. The notation $\bullet p$ refers to all transitions t that put a token in p when they fire ($(t, p) \in F$), whereas $p\bullet$ are the transitions that remove a token from p to fire ($(p, t) \in F$), and similarly for $\bullet t$ and $t\bullet$. A transition t is enabled when $\forall p \in P$ such that $(p, t) \in F$, $M(p) \geq 1$. When transition t fires, the Petri net has a new marking M' based on removing tokens from the places that feed t ($\bullet t$) and adding tokens to the places that receive from t ($t\bullet$). An example Petri net is illustrated in Figure 2.2 that represents the same simple machine as in Figure 2.1. If the places are ordered as [IDLE, BUSY, BROKEN] then the initial marking is [1 0 0], and the only transition that can fire is *start*. Once *start* occurs, the marking is [0 1 0] and either *finish* or *break* can fire next. For this Petri net, the places correspond directly to the states of the finite state machine representation. One of the advantages the Petri net formalism can be understood by considering if there were two such identical machines. For the Petri net, the only change required to Figure 2.2 is to add another token so that the initial state was [2 0 0]. For the finite state machine, however, there would need to be a state for each combination of the machines' statuses, i.e. one state for both machines idle, one state for one machine idle and the other busy, etc., which would make the representation much larger.

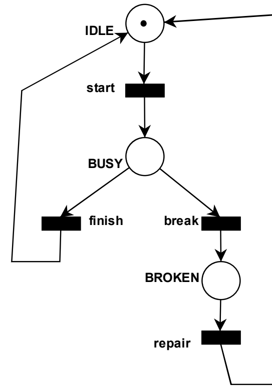


Figure 2.2: Petri net that describes simple machine.

For a Petri net, a firing sequence is a set of transitions $t_1t_2t_3...t_n$ that fire in the given order. Starting in the initial marking M_0, if there exists a firing sequence that leads the Petri net to a marking M, then M is reachable ($M \in R(N, M_0)$). The incidence matrix A of a Petri net is a $|P| \times |T|$ matrix where the $(i, j)th$ entry, $a_{(i,j)}$, is equal to the net change in tokens in place p_i when transition t_j is fired.

2.1.3 System Identification for DES

If a formal DES model is not available for a system, it may be created by observing its behavior. The resulting model can be used for fault or anomaly detection. This general approach is used in [35] [54] and in the anomaly detection solution proposed in Chapter IV. A variety of DES techniques are available for identification of DES, but none of them, including those used in [35] and [54], are applicable to the type of systems focused on for the proposed anomaly detection solution, as described in Section 4.2.1.

Some system identification techniques use linear programming. Integer programming is used to identify free labeled Petri nets, assuming that the number of transitions and maximum number of places is known, in [7]. The identification problem is solved for DES where all of their events are observable, as well as at least part of

the state, first in the case where the places are unknown and then unknown, using integer programming in [19]. An interpreted Petri net is created in [42] to describe a discrete event system where each measurable place in the interpreted Petri net is associated with a sensor. This approach, however, cannot identify behavior of a shared resource in a mutual exclusion situation and assumes sensor signals are observed instead of events, where sensor signals might provide some state information. In [35] and [54], a non-deterministic autonomous automaton is created to model the DES using input/output binary vectors. This solution assumes non-deterministic behavior, and can only handle concurrency when the concurrent activities are distinguishable through prior knowledge. Additionally, none of these DES solutions can incorporate a priori information about the system itself.

Workflow mining also provides techniques for DES system identification. A field of computer science, workflow mining is the technique of observing a workflow in order to create a model of it. One such technique is the $\alpha+$ algorithm [16] which creates a model similar to a Petri net, and is used as the basis of the model generation algorithm developed in Chapter IV. In contrast to the DES techniques, workflow mining assumes very little information about the system – only observed event streams, which may not include all possible streams that the system can generate [60] [14]. A disadvantage to the workflow mining techniques is that they are not able to incorporate any other system information besides observed event streams.

2.2 Logic Controllers

Logic controllers are the controller part of the closed-loop system illustrated in Figure 1.1. Various formalisms have been developed to implement logic controllers. One such formalism is the IEC 61499 standard which makes use of some existing,

traditional logic control formalisms used by industry. Another logic control formalism is Event-Condition-Action Modular Finite State Machine (ECA MFSM) which was developed in academia and makes use of modularity. Both of these formalisms are used in applying the verification work developed in Chapter III.

2.2.1 IEC 61499

IEC 61499 [32] is a distributed control standard developed by the International Electrotechnical Commission to provide an implementation-independent standard based largely on methods already used in industry, such as ladder logic, that are part of the IEC 61131 standard. A thorough introduction to modeling controllers using IEC 61499 is provided by [36], while [61] gives a practical introduction to using IEC 61499 controllers in industry settings and a particular IEC 61499 software tool called FBDK [23]. The primary unit of IEC 61499 is the function block (FB), as shown in Figure 2.3. A function block has two kinds of inputs and outputs – events

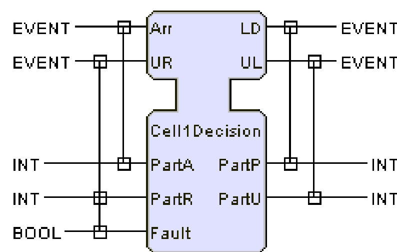


Figure 2.3: IEC 61499 controller for Cell 1 of RFT.

and data, such as Arr and PartA, respectively. Within the function block is an execution control chart (ECC), such as in Figure 2.4, and a set of algorithms written in IEC 61131 languages. When an input event ocurrs, the ECC can transition to a new state and, associated with that state, execute an algorithm that processes the data inputs and produces data outputs and an event output. A transition occurs when its guard condition (GC) is satisfied, where the guard condition is an input

event and/or any associated conditions on data and internal variables. For example, starting from initial state START, when input event Arr occurs and input data PartA = 0, the ECC transitions to state EArrival. Data are associated with events, as shown by the vertical lines in Figure 2.3, such that when an input event arrives its associated data is read and an output event is only produced once its associated data is ready.

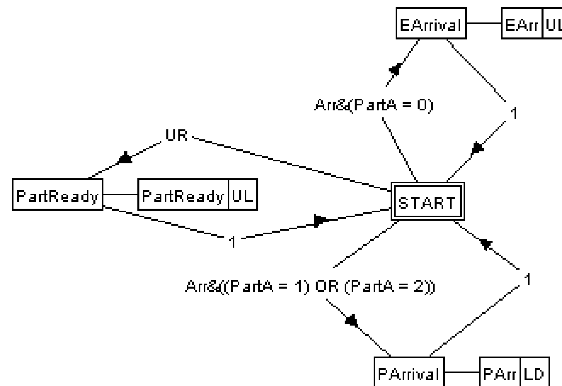


Figure 2.4: ECC for Cell 1 FB.

Basic function blocks, like that in Figure 2.3, can be combined by connecting outputs of one function block to inputs of another to form composite function blocks or networks of function blocks. Such networks can be used to control complicated systems in a modular fashion.

2.2.2 ECA MFSM

Event-Condition-Action Modular Finite State Machine (ECA MFSM) is a logic control formalism based on ECA rules that describe the control behavior [5]. An ECA rule consists of an event (E) which triggers the rule, checking certain conditions (C) and, based on those conditions, sending output events called actions (A). The event and action are input and output, respectively, while the condition is a function of the internal state. Each ECA MFSM consists of interconnected modules, as shown in

Figure 2.5, where a module is an encapsulated trigger-response finite state machine. An ECA MFSM has two types of modules – a single MAIN module and a finite number of peripheral modules, in this case four. The MAIN module consists of a set of ECA rules that represent the behavior of the controller, where there is one rule for every incoming event, whereas the peripheral modules each hold state information about the environment or controller. The MAIN module handles all external communication (input and output) and is connected to each peripheral module.

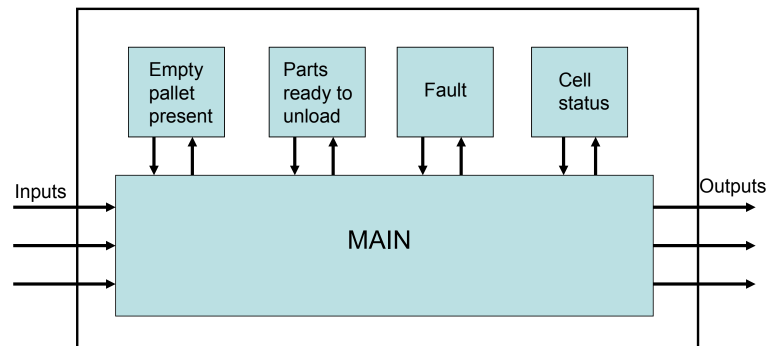


Figure 2.5: Schematic ECA MFSM for cell controller

The ECA MFSM for the RFT cell controller, shown in Figure 2.5, has one MAIN module and four peripheral modules, where the names indicate what state information they hold. The peripheral modules are: empty pallet present, parts ready to unload, fault and cell status. MAIN is connected, through its inputs and outputs, to the hardware of the cell and other controllers. An example of a rule in MAIN is when the event $Arr1$ arrives, MAIN checks the associated conditions from the peripheral modules – whether there is already a part 1 being processed (cell status) and if there is a fault (fault) – and if neither of these conditions are true, takes the action of $LD1$ (loading the part), and otherwise takes the action of Rel (releasing the part).

2.3 Handling Faults in DES

Faults and anomalies in discrete event systems can be handled in one of two primary ways. The system can be verified to possess certain properties in advance of its use to guarantee that certain types of faults and anomalies will not occur. Alternatively, if the system is operating, then fault detection and diagnosis can be used, respectively to determine whether a fault has occurred and what kind of fault. Verification has the advantage of preventing faults from ever occurring, whereas fault detection and diagnosis have the advantage of not being limited to handling faults whose root causes are related to particular properties that are known to be important and can be verified.

2.3.1 Verification

In the field of discrete event systems (DES), verification work has included checking controllers or controlled systems for particular properties, including controllability, language specifications, and diagnosability, among others [51]. Research has been done studying how to verify each of these properties in different circumstances and through different means, of which a sampling is provided here. Verification of controllability by means of counter-examples was developed in [6]. The complexity of verifying language specifications has been studied [53]. Means of checking diagnosability for partially observed systems have been created [64]. The user requirements for IEC 61499 systems can be checked through model checking [34].

There are several main techniques for DES verification. One of the most common is test and simulation, which has the benefit of simplicity but can only provide information about the specific scenarios simulated. Model checking is often used for checking controllability with respect to a specification [13]. Model checking produces

a “yes” or ‘no” answer but can have problems with state space explosion, although this issue has been somewhat addressed by the development of symbolic model checking and abstraction [39]. Traditional model checking techniques, such as described in [12], are applicable to formalisms such as automata and Kripke structures but are not applicable to formalisms that have a distinct output event structure, such as ECA MFSM and IEC 61499, which are the formalisms of focus in the verification in Chapter III. If the model checking techniques were extended to apply to such formalisms, then model checking could be used for verifying input order robustness in place of simulation in the procedure described Chapter III. Theorem proving techniques validate desired properties by building upon already known or verified properties and rely heavily on the user to perform often repetitive steps [33]. Compositional verification [17] proves a particular property for various components of the system given certain assumptions and then extends that property to the entire system.

If one had a formal model of the plant to be controlled by the logic controller, then a number of options would be available, including using model checking to verify input order robustness and guaranteeing input order robustness through synthesis of supervisory control [51]. From the current state of the art of logic control verification, no previous verification procedure exists for input order robustness – neither with nor without a formal model of the plant.

2.3.2 Fault Detection and Diagnosis

There are several existing fault detection and diagnosis solutions in the field of DES. In one such solution, observers are used to detect specific event-based faults in systems that can be modeled by finite state automata [57]. This solution assumes that the system components that comprise the plant either have pre-existing finite

state automata models or such models can be created reliably and correctly. In another solution, additional modeling can be added to an existing Petri net model of a plant to perform event-based diagnosis [27]. Partially observed Petri nets are diagnosed in [55], where faults are unobservable transitions and a belief on faults is calculated. These fault detection solutions, and many other similar ones, require the plant to be modeled in a specific mathematical formalism, such as Petri nets. Such pre-existing formal models are not usually available for large-scale industrial systems and creating them solely through an expert's knowledge is often very labor-intensive and prone to error.

CHAPTER III

Verification of Input Order Robustness

A logic controller is input order robust (IOR) if, for every set of inputs that may occur in different orders and whose order should not affect the logic controller's final state nor set of outputs, the logic controller produces the same final state and set of outputs regardless of the order of the set of inputs. This is an important property for a controller to possess because then the closed-loop behavior of the controller with its environment is deterministic, so the closed-loop behavior can be predicted, and thus checked as to whether it meets desired specifications. Communication networks are increasingly being used to transmit control information in manufacturing systems [45]. The inherent variation in network delivery time may be significant; for example, in Ethernet, the standard deviation of the network time may be close to half the communication time itself [48]. Therefore, logic controllers that rely on information transmitted via networks should be verified as input order robust for all relevant input sets to make the closed-loop system operate predictably. To the author's best knowledge, no proposed formal procedure for such verification exists.

To clarify the concept of input order robustness, it will be distinguished from other related concepts in Section 3.1. Formal definitions for logic controller and network of controllers, along with their notation, are given in Section 3.2. These

definitions serve as foundation for the theory, which is discussed in Section 3.3, and includes a formal definition of input order robustness, a lemma for pairwise checking of input order robustness, and a theorem describing a sufficient condition for the verification of networks of controllers. Section 3.4 describes the verification procedure and provides an example application of it to a system in the ECA MFSM and IEC 61499 formalisms. The relationship between input order robustness verification and the IEC 61499 formalism is the focus of Section 3.5, which discusses how input order robustness relates to some open execution semantic issues for IEC 61499 and applies the verification procedure to a network of controllers implemented in IEC 61499, where this verification involves the open execution semantic issues. Finally, computational complexity of input order robustness verification is highlighted in Section 3.6. The work presented in this chapter is from the papers [3] and [1].

3.1 Related Concepts

Input order robustness is related to the previously-defined concepts of independence and confluence, both of which address the order in which events in DES occur. Two events a and b in a DES are said to be independent [12] if, for every state s in the DES in which both a and b can occur, then 1) a is also enabled in $b(s)$ and b is enabled in $a(s)$ and 2) $a(b(s)) = b(a(s))$, where $a(s)$ is the state reached when a occurs in state s. A comparable concept to (event) independence in active database management systems is called confluence. A rule set is confluent if, from any initial database state, the final state does not depend on the order in which the rules are processed [49]. The term confluence has also been used in DES research, but with a different meaning — all control command choices can eventually lead to states with the same future solely through additional commands [18] [40].

There are two major differences between these existing concepts and input order robustness as defined in this paper. Most importantly, input order robustness takes an external perspective of a system interacting with its environment, with inputs possibly arriving in different orders. These inputs will cause changes of state and outputs, that could go to other systems or back to the environment. The environment is not explicitly modeled; rather, the focus is on validating the internal consistency of the controller. The second difference is that (event) independence and confluence were defined for standard automata models, which do not have some properties of the systems considered here (see Definition 5), such as explicit outputs and discrete variable inputs and outputs. While it may be possible to extend those concepts to systems considered here, the extensions that would be needed to consider networks of controllers are not as immediately apparent.

Input order robustness should not be confused with other concepts that may seem similar in name or meaning but are quite different in practice. Traditionally, robust control considers the situation where the actual plant may be somewhat different than the model of the plant, and shows that some desirable properties, such as stability and disturbance rejection, can be preserved. Robust control [65] generally is applied to to continuous state space controllers, rather than discrete state space controllers, such as DES. There are some similarities, however, between robustness and input order robustness. Input order robustness can be thought of as robustness to non-determinism in the environment, rather than lack of knowledge about the plant, and the property to be preserved is closed-loop determinism. A race condition in hardware design occurs when a single input causes two internal changes which then "race" to see which completes first, as that may affect the outputs of the system [63]. Some work has been done to create logic circuits that are guaranteed to be race-free

[50]. In contrast to race conditions, input order robustness is a property of logic controllers and considers the case of processing multiple inputs in different orders, rather than the case of a single input causing multiple responses.

3.2 Logic Controllers and Networks of Controllers

Input order robustness is a property for logic controllers and although they were informally defined and some modeling formalisms discussed in Section 2.2, formal definitions and notation are required to develop input order robustness theory. First logic controllers themselves are defined, then networks of controllers, which is a particular means of combining logic controllers modularly.

3.2.1 Logic Controllers

Definition 5 (Logic Controller)**.** A *logic controller* (or simply *controller*) $L = \{X, I, O, f, g\}$ is a deterministic function or program that has an internal state set $X = \{x_1, x_2, ...x_r\}$, receives input events in the set $I = \{i_1, i_2, ...i_p\}$ from the environment, and computes output events in the set $O = \{o_1, o_2, ...o_q\}$ which it sends to the environment. The controller acts upon an input event i and the logic controller's state x to deterministically change the logic controller's state $x' = f(i, x)$, and send a set of control outputs $o = g(i, x) \subseteq O$ to the environment.

Individual elements of a logic controller's set of outputs are given by an indexed o, such as o_1, but an o without a subscript is an un-ordered subset of all the possible outputs; $o = g(i, x)$ may contain more than one output, such as $o = \{o_1, o_3\}$. Occasionally, particularly for some controllers implemented in IEC 61499, the next state may also be dependent on the previous output, in which case $x' = f'(i, x, o)$. For convenience, the effect of applying multiple inputs sequentially will be expressed in shorthand, which can be understood iteratively. Given that a sequence of events s

occurs followed by another input i, $f(si, x_0) = f(i, f(s, x_0))$. For the same sequence and subsequent input, $g(si, x_0) = g(i, f(s, x_0)) \cup g(s, x_0)$. Note that $g(si, x_0)$ includes all of the outputs produced from applying the input sequence si from the initial state x_0. If $f(s_\ell, x)$ is undefined for a given $s_\ell = i_{\ell_1} i_{\ell_2} \ldots$, then $f(s_\ell s_m, x)$ is undefined for any $s_m = i_{m_1} i_{m_2} \ldots$.

This definition of logic controller is similar to a DES [9] that has explicit outputs and no time component, but differs from DES in that discrete variable inputs and outputs are allowed as well. Such logic controllers can be implemented in formalisms including ECA MFSM and IEC 61499, which are the focus of this work, as well as others such as Mealy machines and some forms of Petri nets.

Any non-empty subset of the controller's inputs I can be expressed by $\pi \subseteq I$. We define Π to be the set of all possible subsets π. Note that for inputs and outputs, capital variables are used to indicate sets, whereas lowercase variables indicate individual elements. Exceptions are $o = g(i, x)$ and π which are sets.

Logic controllers are of two main types – *event-based* and *scan-based* – although some control formalisms are a hybrid of these types, such as IEC 61499, which uses both events and variables. An *event-based logic controller* is activated when an input event arrives at the logic controller, triggering processing that can then produce output events and change the state. Inputs and outputs of event-based logic controllers are events. In contrast, a *scan-based logic controller* (sometimes called a run-to-completion controller) is one that, at pre-determined time intervals, reads its inputs and acts upon changes in the inputs' values to produce new output values and change state. One complete cycle is called a scan. Within a scan time, if any inputs change value, these changes will not be processed until the next scan. Inputs and outputs of scan-based logic controllers are discrete variables. For conciseness, in

this paper we will discuss inputs arriving and outputs being sent. This terminology applies to scan-based controllers where events are changes in inputs and outputs. The logic controller's dynamic behavior is not further specified to keep the discussion of input order robustness theory more general and not specific to a particular modeling formalism, as the internal behavior of the logic controller is not as relevant for the purposes of input order robustness.

3.2.2 Network of Controllers

In many practical scenarios, a monolithic controller is impractical because it would be too large so a distributed controller is preferred. A major advantage of a distributed controller is that it has inherent modularity, which can be used to reduce the computational complexity of verification as demonstrated in Section 3.6. A particular definition of distributed controller, called network of controllers, is presented here.

Definition 6 (Network of controllers)**.** A *network of controllers* N is a logic controller that is composed of c *component controllers* or simply *components.* Each input to N is an input to at least one of the components, and each output of N is an output from one of the components. In addition, component outputs can be connected to component inputs, so that they are internal events within N. The state set of N is the cross-product of the state sets of its components $X = X^1 \times ... \times X^c$.

For networks of controllers, a superscript index is used for component controllers since there are multiple ones to consider, and the network itself is referred to without a superscript. For example, component j has input set I^j and state x^j. The inputs of a particular component j from the environment are I^j_0 where 0 references the environment. The outputs of component k that are inputs to component j are I^j_k,

which are also called intermediate inputs/outputs because they are outputs from one component and inputs to another. When the network receives the input set π, the input set received by component j is referred to as π^j.

A network of controllers is said to have no *feedback loops* if its structure is a directed acyclic graph, where the nodes of an acyclic graph have a partial ordering function [28]. In this case, no component controller sends outputs to a component whose own output has a path, possibly through additional components, back to the original component's input. The partial ordering allows levels to be defined. The environment is Level 0 and a component is in Level m if it receives inputs only from components in Levels 0 through $m-1$ and has at least one input from Level $m-1$. Because the number of components c is finite, the highest level $\bar{m} \leq c$ is also finite ($\bar{m}+1 =$ total number of levels including the environment). Without feedback loops, we can define $L_m = \{\text{components } \ell | Level(\ell) < m\}$, which implies that $L_{\bar{m}+1} = \{\text{all components}\}$. A component j in Level m only receives intermediate inputs from components in L_m. Using this notation, the levels for the components can be found recursively starting with the environment as Level 0 and finding those components that only receive input from L_1 and labeling them as Level 1, then those components that only receive input from L_2 and labeling them as Level 2, and so on.

Several conclusions can be drawn from these definitions and this notation. First, note that all of the inputs to component j are either from the environment or from another component, and hence $I^j = \cup_{k=0}^{c}(I_k^j)$. Second, recognizing that all inputs to the network are inputs to at least one component controller, $I = \cup_{j=1}^{c} I_0^j$. Thus when a set of inputs to the network $\pi \subseteq I$ occurs, each component controller j will receive a set of inputs π^j that is the union of inputs from the environment and intermediate inputs from other components, such that $\pi^j = \cup_{k=0}^{c}(\pi_k^j)$ where the intermediate

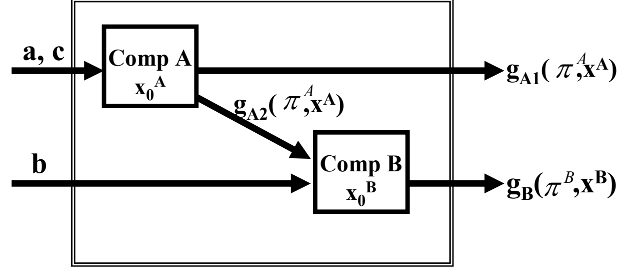


Figure 3.1: Simple network of controllers where $\pi^A \subseteq \{a, c\}$, $\pi^B \subseteq \{g_{A2}(\pi^A, x^A), b\}$ and the outputs of Component A are segmented into outputs to the environment and to Component B, $g_A(\pi^A, x^A) = g_{A1}(\pi^A, x^A) \cup g_{A2}(\pi^A, x^A)$

inputs from other components depend on the state of those components.[1] For a network of controllers without feedback loops and a component j in Level m, $I^j = \cup_{k=0}^{c}(I_k^j)$ can be reduced to $I^j = \cup_{k \in L_m}(I_k^j)$, and similarly, $\pi^j = \cup_{k=0}^{c}(\pi_k^j)$ can be reduced to $\pi^j = \cup_{k \in L_m}(\pi_k^j)$.

A simple network is illustrated in Figure 3.1, where there are two components, A and B. In this example, $I = \{a, b, c\}$ and $X = X^A \times X^B$, where x^A and x^B are the states of the component controllers. For the individual components, $I^A = I_0^A$ =$\{a, c\}$ and $I^B = I_0^B \cup I_A^B$ such that $I^B = \{b\} \cup g_{A2}(\pi^A, x^A)$ for any $x^A \in X^A$, $\pi^A \subseteq I^A$. The set of outputs is $O = g_{A1}(\pi^A, x^A) \cup g_B(\pi^B, x^B)$, where x^A, x^B can be any states in X^A, X^B respectively. For $\pi = \{a, b, c\}$, the components receive the subsets $\pi^A = \{a, c\}$ and $\pi^B = \{b, g_{A2}(\pi^A, x^A)\}$, where π^B may depend on $x^A \in X^A$. A more complicated network of controllers is shown in Figure 3.2. This network of controllers has no feedback loops, so a partial order on the components exists and the components can be partially ordered into Levels. Component A receives input only from the environment (Level 0), so it is in Level 1. Next, components B and D are Level 2 because they receive input only from Levels 0 and 1. Component E is Level 3 because it receives inputs only from Level 2. Likewise, Component F is Level 4 and

[1]This notation assumes the set of inputs to component j will be the same regardless of the order in which the inputs in π occur, which is proved for input-order robust controllers in Section 3.3.

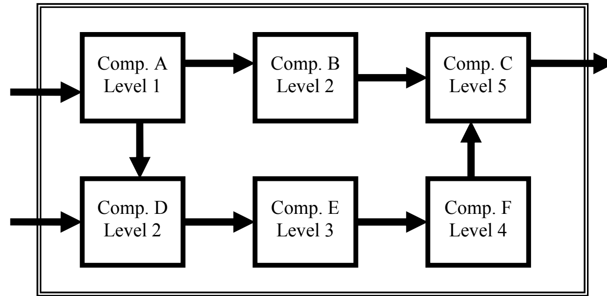


Figure 3.2: Network of controllers with no feedback loops, which means it can be labeled using partial ordering.

Component C is Level 5.

The overall behavior of a network of controllers – its interaction with the environment – can be event-based or scan-based, just like the logic controller behavior described in Section 3.2.1. Because of synchronization issues, a scan-based network of controllers can only be implemented on a single device (such as a PLC), whereas an event-based network of controllers can be on one device or spread across multiple devices (different components on different devices). For either event-based or scan-based execution, the network of controller's components may process sequentially or in parallel. Regardless, when a network receives a set of inputs, the components have to process not only the external inputs but also the intermediate inputs generated by other components during the process. Similarly to logic controllers, the dynamic behavior of networks of controllers is not further specified to keep the theory more general.

3.3 IOR Theory

An informal definition of input order robustness was provided in discussing the motivating example in Section 1.2.1; a formal definition is provided here. Additionally, a lemma shows how pairwise checking of all inputs is a sufficient condition for

input order robustness. Conditions for input order robustness of a network of controllers are proposed and proved. These theoretical results are presented without reference to a particular modeling formalism for the logic controller or network of controllers to provide broader results. For a given modeling formalism, the theory could be developed in more depth in future work. Because input order robustness is related to the arrival of discrete event inputs, it is not applicable to continuous systems – systems described by the evolution of continuous variables in continuous or discrete time – and therefore they are not discussed.

3.3.1 Input Order Robustness

Definition 7 ($\{\pi, x_0\}$-input order robust)**.** For a logic controller $L = \{X, I, O, f, g\}$, a subset of its inputs $\pi = \{i_1, \ldots, i_k\} \subseteq I$, and an initial state $x_0 \in X$, the controller L is $\{\pi, x_0\}$-*input order robust* if for any possible permutations $s_\ell = i_{\ell_1} \ldots i_{\ell_k}$ and $s_m = i_{m_1} \ldots i_{m_k}$ of the inputs in π, $g(s_\ell, x_0) = g(s_m, x_0)$ and $f(s_\ell, x_0) = f(s_m, x_0)$.

In words, a logic controller is $\{\pi, x_0\}$-input order robust if whenever it starts in initial state x_0, it produces the same set of outputs and final state after all of the inputs in π have occurred, regardless of the order in which the inputs in π occur. Note that $\{\pi, x_0\}$-input order robustness only requires that the set of outputs be the same, not the order of those outputs. Requiring that the output order, not just the set of outputs, be the same regardless of input order would be a more restrictive property, and thus a subset of the logic controllers that are $\{\pi, x_0\}$-input order robust would also have this property.

The conditions for $\{\pi, x_0\}$-input order robustness can be satisfied even if $g(s_\ell, x_0) = g(s_m, x_0) = \emptyset$ or $f(s_\ell, x_0) = f(s_m, x_0) =$ undefined. A counter-example of this property is the motivating example in Chapter I, where the set of inputs $\{LP1, PR\}$

occurring from initial state IDLE in either order results in different final states and sets of outputs, indicating that the logic controller is not $\{\{LP1, PR\}$,IDLE$\}$-input order robust.

Definition 8 (X_π)**.** For a logic controller $L = \{X, I, O, f, g\}$, and a subset of its inputs $\pi = \{i_1, \ldots, i_k\} \subseteq I$, the set of valid initial states is $X_\pi = \{x_0 \in X |\ \exists\ i \in \pi$, $f(i, x_0)$ is defined$\}$.

If $\pi = \pi_A \cup \pi_B$, then $X_\pi = X_{\pi_A} \cup X_{\pi_B}$. The set of valid initial states for π is all states in which there is a transition defined for at least one input in π. For some π it may be the case that $X_\pi = X$; however, for many controllers and possible π, X_π may be significantly smaller in size than X.

Definition 9 (π-input order robust)**.** For a logic controller $L = \{X, I, O, f, g\}$, and a subset of its inputs $\pi = \{i_1, \ldots, i_k\} \subseteq I$, the controller L is π-*input order robust* if it is $\{\pi, x_0\}$-input order robust $\forall x_0 \in X_\pi$.

A logic controller is π-input order robust if whenever it starts in an initial state x_0 in which at least one input in π could occur, it produces the same set of outputs and final state after all of the inputs in π have occurred, depending only on x_0 and not on the order in which the inputs in π occur. Note that if $f(a, x_0)$ and/or $f(b, x_0)$ is defined, then $x_0 \in X_{\{a,b\}}$. Additionally, if $f(ab, x_0)$ is also defined, but $f(ba, x_0)$ is not, then the logic controller assumes that a will occur before b and therefore is not $\{\pi, x_0\}$-input order robust, nor π-input order robust, for $\pi = \{a, b\}$. If $f(a, x_0)$ is defined, and both $f(b, x_0)$ and $f(ab, x_0)$ are undefined, then the logic controller is $\{\{a, b\}, x_0\}$-input order robust because neither ab nor ba can execute from x_0.

Definition 10 (Π_{check})**.** For a logic controller $L = \{X, I, O, f, g\}$, the set of π for which L should be verified as π-input order robust is $\Pi_{check} = \{\pi \subseteq I |$ all $i \in \pi$

can arrive in any order and their order should not affect the final state, x, or set of outputs, o}.

For some logic controllers, it may be the case that Π_{check} is every possible subset of inputs; however, in many cases there may be significantly fewer input sets in Π_{check} than in Π. If an environment produces inputs in non-deterministic orders, a controller that is π-input order robust for every $\pi \in \Pi_{check}$ will make the closed-loop system deterministic. If a set of inputs π cannot physically arrive in different orders, then it does not matter if the logic controller is π-input order robust, and hence $\pi \notin \Pi_{check}$. If the order of occurrence of a set of inputs π should affect the final state and/or output, then the controller should not be π-input order robust, and again $\pi \notin \Pi_{check}$.

3.3.2 Pairwise Input Order Robustness of a Single Controller

The theory for verifying input order robustness of a single controller is provided here, while a method for such verification is given in Section 3.4. We note that π-input order robustness can be verified in a pairwise manner.

Lemma III.1. *For a logic controller L and a subset of inputs $\pi \subseteq I$, if the logic controller is π_2-input order robust for every $\pi_2 = \{i_1, i_2\} \subseteq \pi$, then logic controller L is π-input order robust.*

Proof. Suppose $\pi = \{a, b, c\}$, and assume that L is π_2-input order robust for every subset of π with two elements, but there are two sequences of three inputs that result in different states (or outputs). If $x \in X_{\{a,b,c\}}$ then x is a valid initial state for at least one input of a, b, and c, and we can write

$$f(abc, x) = f(c, f(ab, x)) = f(c, f(ba, x)) = f(bac, x) =$$

$$f(ac, f(b, x)) = f(ca, f(b, x)) = f(bca, x)) = \dots$$

and thus by pairwise switching the order of the inputs we can achieve any permutation. This switching does not require checking whether events are enabled in a given state, for example whether ac and ca are defined in $f(b, x)$ to determine that $f(ac, f(b, x)) = f(ca, f(b, x))$, because either at least one event in the pair (a or c) is enabled and thus input order robustness holds, or neither event is enabled and thus both lead to an undefined state. Note that for a given initial state x, these state functions may evaluate to be invalid (the state undefined), but in that case they will all evaluate to be invalid because of pairwise switching, and thus still all be the same. The same result can be shown for the output functions. This result for three events can be generalized to any finite number of events [10]. □

Pairwise input order robustness, as described in Lemma III.1, is a sufficient, but not necessary condition for input order robustness of a set of size greater than two. Consider the set $\pi = \{a, b, c\}$. There may exist a logic controller with initial state x_0 for which $f(a, x_0)$, $f(b, x_0)$, and $f(c, x_0)$ are all defined and $f(s, x_0)$ are equal for all s that are permutations of π, but for which $f(ab, x_0) \neq f(ba, x_0)$. Thus, the logic controller is $\{a, b, c\}$-input order robust, but not $\{a, b\}$-input order robust. Hence, Lemma III.1 only provides a sufficient condition.

3.3.3 Input Order Robustness of a Network of Controllers

As mentioned in Section 3.2.2, a network of controllers has inherent modularity. In this section, we will show how that modularity can be leveraged to make the verification of input order robustness modular for networks of controllers without feedback loops.

We first consider those components in Level 1 (only receive input from the environment), and then proceed to higher Levels. From Section 3.2.2, for a network N

without any feedback, the set of inputs received by a component j in Level m can be expressed as $\pi^j = \cup_{k \in L_m}(\pi_k^j)$. For a component j in Level 1 this can be simplified to $\pi^j = \cup_{k \in L_1}(\pi_k^j) = \pi_0^j = \pi \cap I^j$ where π is the input received by the network N, which is the same regardless of the order in which the inputs in π arrive at the network. Thus, for a network N without any feedback and initial state $x_0 = (x_0^1, .., x_0^c)$, if it receives input set π, then each of its components j in Level 1 can be verified as $\{\pi^j, x_0^j\}$-input order robust because each π^j is known. Now consider how to verify components in Level 2.

Lemma III.2. *Let N be a network of c component controllers with no feedback loops, $\pi \subseteq I$ be a subset of the inputs to N, and $x_0 = (x_0^1, .., x_0^c)$ be the initial state of N. Let N's highest Level be $\bar{m} = 2$. If every component k in Level 1 is $\{\pi^k, x_0^k\}$-input order robust, j is a component in Level 2, and the network receives input π, then there exists a unique set π^j such that for every ordering of events in π, the set of inputs received by component j is equal to π^j.*

Proof. Suppose we are given a controller that meets the required conditions, but that there exists a component j in Level 2 such that π^j depends on the order of π. We know that $\pi^j = \pi_0^j \cup (\cup_{k \in Level\ 1}(\pi_k^j))$ and that $\pi_0^j = \pi \cap I^j$, which means that π_0^j is independent of the order of π. Thus, the dependency on the order of π must come from $\cup_{k \in Level\ 1}(\pi_k^j)$. For each $k \in$ Level 1, $\pi_k^j = g_k(\pi^k, x_0^k) \cap I^j$. However, I^j and x_0^k are fixed, and g_k is deterministic, which means that the dependency in order must be due to π^k, but that violates input order robustness of component k. Hence, such a component j cannot exist and we have shown how π^j can be calculated using the given information. $\square$

Because components in a given Level cannot influence components in lower Levels,

we can extend Lemma III.2 to networks with higher Levels by calculating π^j and verifying $\{\pi^j, x_0^j\}$-input order robustness level by level. This extension is described in Corollary III.3.

Corollary III.3. *Let N be a network of c component controllers with no feedback loops, $\pi \subseteq I$ be a subset of the inputs to N, and $x_0 = (x_0^1, ..., x_0^c)$ be the initial state of N. For a given Level $m+1 \leq \bar{m}$, if each component k in Levels $1 \ldots m$ is $\{\pi^k, x_0^k\}$-input order robust then the set of inputs π^j received by each component j in Level $m+1$ can be calculated for the initial state x_0, and does not depend on the order of π.*

Theorem III.4. *If a network of controllers N is composed of c component controllers that do not have any feedback loops among them and that are each $\{\pi^j, x_0^j\}$-input order robust, then the network is $\{\pi, x_0\}$-input order robust, where $\pi = \cup_{j=0}^{c}(\pi^j) \cap I$ and $x_0 = (x_0^1, ..., x_0^c)$.*

Proof. Using $\pi^j = \pi \cap I^j$ for components in Level 1 and Corollary III.3 for components in higher Levels, we can calculate π^j for each component j starting with components in Level 1 and working up through the higher Levels, and none of the π^j depend on the order of π, only on x_0. Because each component j is $\{\pi^j, x_0^j\}$-input order robust and will receive the same set of inputs π^j regardless of the order of π, each component's final state $x^j = f_j(\pi^j, x_0^j)$ and set of outputs $o^j = g_j(\pi^j, x_0^j)$ will be the same. Because $o \subseteq \cup_{j=0}^{c}(o^j)$ and $x = (x^1, ..., x^c)$, the set of outputs and final state of the network will be the same when the initial state is x_0 regardless of the order of π, hence the network is $\{\pi, x_0\}$-input order robust. □

Similar to the proof of verifying input order robustness for a logic controller, verification of a network of controllers occurs level by level. To verify $\{\pi, x_0\}$-input

order robustness for a network N, first confirm that there are no feedback loops among the components. Then determine $\pi^k = \pi \cap I^k$ for each component k in Level 1 and check that each component k in Level 1 is $\{\pi^k, x_0^k\}$-input order robust. In the process, determine the set of outputs produced when π^k occurs from initial state x_0^k. Next, determine $\pi^j = (\pi \cap I^j) \cup_{\{k \in Level\ 1\}} I_k^j$ for each component j in Level 2 and check that it is $\{\pi^j, x_0^j\}$-input order robust and calculate the outputs that it produces. Repeat this process for all of the levels. Thus, $\{\pi, x_0\}$-input order robustness of the network can be verified.

For illustration of verification, recall the simple network shown in Figure 3.1. We can check whether the network is $\{\pi, x_0\}$-input order robust for $\pi = \{a, b\}$ and a given x_0. This network does not have any feedback loops, and Component A is in Level 1, while Component B is in Level 2. Starting with Level 1, Component A has $\pi^A = \{a\}$, so it does not need to be verified as input order robust because it receives only one input, and thus will produce outputs $g_A(a, x_0^A) = g_{A1}(a, x_0^A) \cup g_{A2}(a, x_0^A)$. For Level 2, Component B has input set $\pi^B = \{b, g_{A2}(a, x_0^A)\}$ and we can verify that Component B is $\{\pi^B, x_0^B\}$-input order robust. Thus if Component B is $\{\pi^B, x_0^B\}$-input order robust, then the network as a whole is $\{\{a, b\}, x_0\}$-input order robust.

Because a network of controllers is also a logic controller in its own right, Definitions 8 and 9 are applicable, so a network of controllers is π-input order robust if it is $\{\pi, x_0\}$-input order robust $\forall x_0 \in X_\pi$. However, the calculation of X_π for a network of controllers may not be readily apparent. For simplicity, we assume that the components have been numbered such that components 1 through ℓ_1 are in Level 1, $\ell_1 + 1$ through ℓ_2 are in Level 2, and so on. Initially, we can assume $X_\pi = X_{\pi^1} \times ... \times X_{\pi^{\ell_1}} \times X^{\ell_1+1} \times ... \times X^c$ because we know π^k for $k \in 1...\ell_1$ (components in Level 1), and hence X_{π^k}, and for components in higher levels, we assume the

entire state space. In verifying that components in Level 1 are π^k-input order robust, we do not use the portion of the network state associated with the other component controllers, but we do calculate π^j for each component controller j in Level 2, and thus can determine the valid initial states for the Level 2 components, X_{π^j} and use them to replace X^j. Thus, in verifying π-input order robustness for a network, the set of valid initial states X_π is determined during the verification itself and does not need to be known in advance.

Note that if a network of controllers has feedback loops among its components, then it can be verified for input order robustness if the components in the feedback loops are combined so that each feedback loop becomes a single component and that single component is tested for input order robustness. The network of controllers illustrated on the left of Figure 3.3 has two feedback loops – one between Components A and B, and the other among Components C, E, and F. To verify input order robustness for this network, we would need to combine Components A and B, and combine Components C, E, and F to produce the equivalent network illustrated on the right in Figure 3.3. Exactly how the components are combined depends on the

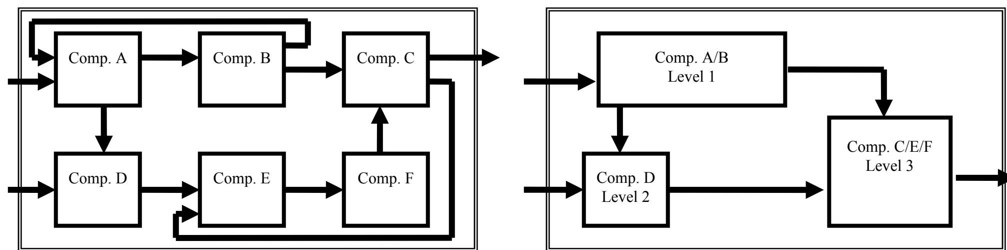


Figure 3.3: Left: Network of controllers with two feedback loops, Right: Equivalent network of controllers where components have been combined to eliminate feedback loops

formalism in which the logic controllers are implemented, but the main idea is that the combination has the same interaction with the other components in the network as the components making up the combination. It is important to note, however,

that combining components may be computationally intensive or complex depending on the number and size of components being combined.

Particular execution semantics – event-based versus scan-based and sequential versus parallel – are not assumed in the proof of Theorem III.4, which relies on absence of feedback loops and components' input order robustness to show that each component will receive the same set of inputs and therefore produce the same set of outputs. The differences between event-based and scan-based execution only limits what is considered a network of controllers, as discussed in Section 3.2.2. Parallel execution only affects when inputs are processed and outputs produced, not what those inputs and outputs are. Thus, under the assumptions of Theorem III.4, a network is input order robust regardless of whether the component controllers take turns processing (sequential) or process available inputs at the same time (parallel).

3.4 IOR Verification Procedure

3.4.1 General Procedure

π-input order robustness can be verified using simulation to check that the order of inputs in π does not affect the set of outputs nor final state, for any $x_0 \in X_\pi$. Simulation, however, cannot determine which subsets should be verified for input order robustness, $\pi \in \Pi_{check}$ nor their set of initial states X_π, and for a closed-loop system to be deterministic, the logic controller must be π-input order robust for all $\pi \in \Pi_{check}$. As such, this section details a verification procedure that consists of two portions – determining the system information (Π_{check} in Step 1, X_π for each $\pi \in \Pi_{check}$ in Step 2), and performing simulation (Step 3). Note that input pairs can be verified rather than input sets because of Lemma III.1, so Step 1 determines $\Pi_{2,check} = \{\pi \in \Pi_{check} | \, |\pi| = 2\}$.

Step 1a: Of all the possible pairs of inputs, $\{a, b\}$ where a, $b \in I$, identify those

that could arrive in either order. Based on the controller and environment, there may be some inputs that cannot arrive at the controller in close time proximity or in either order. For example, if there are two mutually exclusive processes such that one cannot be started until notification is received that the other has finished, then an input indicating that one process has just finished cannot arrive in close time proximity to another input indicating that the second process has just finished.

Step 1b: Of the remaining pairs of inputs, identify those whose order should not affect the final state and/or set of outputs, and include them in $\Pi_{2,check}$. For some input pairs, their order should matter, resulting from common physical scenarios such as: two inputs both trigger the exclusive use of the same resource; the processing of one input starts a particular process and the processing of another input prevents that process from starting until further notification; one input triggers the use of a resource that can only handle one job at a time and another input triggers the release of this resource from its current job.

The input pairs identified in Step 1 are the ones in $\Pi_{2,check}$. If a controller has p inputs, then there are $p(p-1)/2$ unique different-element pairs (for pairs $\{a, b\}$ where $a = b$, they are trivially $\{a, b\}$-input order robust). In many cases, $\Pi_{2,check}$ may have significantly fewer pairs than $p(p-1)/2$, and thus fewer pairs that need to be checked to guarantee determinism.

For each such pair of inputs $\{a, b\} \in \Pi_{2,check}$:

Step 2: Determine the initial states of the controller for which the arrival of a is not physically possible or represents an error. All other states of the controller are valid initial states for input a, $x \in X_a$. Repeat for input b. $X_{\{a,b\}} = x \in X_a \cup X_b$

Now that the system information ($\Pi_{2,check}$ and X_π for each $\pi \in \Pi_{2,check}$) is determined, the simulation is performed:

Step 3: For each $x \in X_{\{a,b\}}$, simulate the logic controller for ab and ba. Compare the final states, $f(ab, x)$ and $f(ba, x)$, and the sets of outputs, $g(ab, x)$ and $g(ba, x)$, and if they are the same, then the controller is $\{a, b\}$-input order robust.

This procedure is described in a general manner so that it is applicable to controllers implemented in a variety of formalisms, although it is especially useful for those with an explicit output event structure, such as ECA MFSM and IEC 61499. To illustrate how the procedure is applied, an example system is described in Section 3.4.2, the system information determined (Steps 1 and 2) in Section 3.4.3 and the simulation is performed for an ECA MFSM implementation and an IEC 61499 implementation in Sections 3.4.4 and 3.4.5. An example network of controllers implemented in IEC 61499 is described and verified for input order robustness later, in Section 3.5.

3.4.2 Example System for Verification

The verification procedure will be illustrated through a set of examples encountered in verifying a controller that performs some of the control for one processing cell of the Reconfigurable Factory Testbed (RFT). An RFT processing cell has one robot and two computer numerical controlled (CNC) machines that mill parts and can work in parallel. Pallets and a conveyor are used to bring unprocessed parts to the cell and remove processed parts from the cell. The robot interacts with parts on pallets only at one location – the pallet stop. There are two types of parts, called part 1 and part 2, both processed by this cell but on different CNCs.

The plant (robot, machines, machine-level controllers, rest of the RFT, etc.) and the cell controller interact through exchanging events, which are listed and described in Table 3.1. All of these events are either input to the cell controller or output from the controller, where inputs (first column) cause the cell controller to check certain

Table 3.1: Events for Cell 1 Controller

Input	**Output**
Arr0: empty pallet arrives	Rel: release pallet (w/ or w/out part)
Arr1: part 1 arrives	LD1: load part 1
Arr2: part 2 arrives	LD2: load part 2
UR1: part 1 unload req	UL1: unload part 1
UR2: part 2 unload req	UL2: unload part 2
FT: fault occurs	

conditions, and depending on those conditions, possibly produce an output (second column). The uncontrolled cell behavior is as follows: a) when a part arrives, it is either loaded into a machine for processing or released unprocessed; b) when a part is finished being processed by a machine, it either waits in the machine or is unloaded on an empty pallet and released; c) when an empty pallet (called part 0) arrives, it has a finished part unloaded on to it, waits at the pallet stop, or is released. The cell controller decides which of these possible behaviors (i.e. load a part or release it unprocessed) is correct in a given situation, where its decision-making is illustrated in Figure 3.4. When the fault event occurs, the only effect is that the fault boolean is set true. This cell controller can be implemented in different formalisms, as illustrated by a simplified example of this processing cell described here and implemented in ECA MFSM in Section 3.4.4 and in IEC 61499 in Section 3.4.5.

The general expression of state for this controller is (Empty Pallet Present (true or false) × Parts Ready to Unload (neither, one, both in either order) × Fault (true or false) × Cell Status (neither, one, or both)):

$$\begin{bmatrix} T \\ F \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 2 \\ 12 \\ 21 \end{bmatrix} \times \begin{bmatrix} T \\ F \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 2 \\ 12 \end{bmatrix}$$

```
if Arr1
    if Fault, then Rel
    else
        if Cell Status == 1 or 12, then Rel
        else, LD1
else if Arr2 … (comparable to Arr1)
else if UR1
    if ~Fault
        if Empty Pallet Present, then UL1
        else, add 1 to unload queue
else if UR2 … (comparable to UR1)
else if Arr0
    if Fault, Empty Pallet Present set true
    else
        if Parts Ready to Unload == 0
            if Cell Status == 0, then Rel
            else, Empty Pallet Present set true
        else if Parts Ready to Unload == 1 or 12, UL1
        else if Parts Ready to Unload == 2 or 21, UL2
```

Figure 3.4: Cell controller's decision-making

The Empty Pallet Present component indicates whether there is an empty pallet waiting at the cell's pallet stop. Parts Ready to Unload is a queue of the completed parts that are waiting to be unloaded, and the queue is ordered such that if Parts Ready to Unload is 12 then Part 1 will be unloaded first, whereas if it is 21 then Part 2 will be unloaded first. Fault indicates whether an external fault has been set, which affects the proper system behavior. Cell status indicates which part or parts are currently being processed (loaded, machined, and/or unloaded) in the cell, where 0 indicates no parts and 12 indicates both parts. Some combinations of the component states are not valid states because they represent incorrect behavior of the system due to conflicting or incompatible states of the various components. For example, Parts Ready to Unload cannot be 1 if Cell Status is 0. Due to these incorrect combinations, the system has 34 states instead of 80 ($= 2 \times 5 \times 2 \times 4$).

Table 3.2: Verification of Event Pairs: P = cannot occur nearly same time, D = order should matter, S = need to verify

	Arr0	Arr1	Arr2	UR1	UR2	FT
Arr0	P	P	P	S	S	D
Arr1	P	P	P	S	S	D
Arr2	P	P	P	S	S	D
UR1	S	S	S	P	D	D
UR2	S	S	S	D	P	D
FT	D	D	D	D	D	P

3.4.3 First Steps of Verification Example

Because determining the system information (Steps 1 and 2) only involves knowledge about the plant to be controlled and the proper controller behavior, it can be done the same regardless of controller implementation. In Step 1 of the verification procedure, we can eliminate some pairs of inputs from consideration $\{a, b\} \notin \Pi_{2,check}$. For example, the arrival of a part 1 and the arrival of an empty pallet {Arr1, Arr0} cannot occur at nearly the same time since only one pallet can be at the pallet stop for Cell 1 at a time. Additionally, for the events request unload part 1 and request unload part 2 {UR1, UR2} the order should affect the controller's state and outputs since both inputs request the same shared resources – the robot and an empty pallet. All the possible input pairs are listed in Table 3.2, and associated with each pair is a P, meaning they cannot occur in either order, a D indicating their order should create different results (final state and/or output set), or an S showing that the input pair is in $\Pi_{2,check}$. With 6 input events, there are $p(p-1)/2 = 15$ unique different-element pairs, which means the maximum size of $\Pi_{2,check}$ is 15. From Table 3.2, we see that there are only 6 such pairs, making the computational complexity of the verification significantly less than maximum. See Section 3.6 for further discussion of computational complexity.

Consider the input pair request unload part 1 and empty pallet arrives {UR1,

Arr0}. The valid initial states for these inputs, $X_{\{UR1,Arr0\}}$, must be determined. When UR1 occurs, the cell controller follows if-else statements for UR1 in Figure 3.4, and likewise, when Arr0 occurs, the cell controller follows the if-else statements for Arr0. Thus, regardless of which order these inputs arrive in, the controlled outcome is that the finished part 1 is loaded on to the empty pallet (UL1) if no other parts were waiting.

There are some initial states that are not valid for both inputs of this pair, and if $x \notin X_{UR1}$ and $x \notin X_{Arr0}$, then $x \notin X_{\{UR1,Arr0\}}$.

- Restriction 1 (R1): An empty pallet already waiting in the cell is not a valid initial state for the input indicating an empty pallet has arrived, as there is only one location within the cell where a pallet can wait. ($x \notin X_{Arr0}$)
- Restriction 2 (R2): Likewise, a part 1 cannot already be waiting for unloading when the input indicating that a part 1 is ready to be unloaded arrives, because only one part 1 can be processed at a time. ($x \notin X_{UR1}$)
- Restriction 3 (R3): All initial states in which there is no part 1 in the cell are invalid for this input pair because having a part 1 ready for unloading requires that a part 1 is present in the cell. ($x \notin X_{UR1}$)

The states removed by these restrictions are summarized in Table 3.3, where • means any valid value of that component state. Considering these restrictions, of the 34 states, only 25 states are in $X_{\{UR1,Arr0\}}$. The remainder of this verification example is discussed for the ECA MFSM and IEC 61499 formalisms in Sections 3.4.4 and 3.4.5.

Table 3.3: States Removed by Restrictions R1 - R3 where • = any valid value

	Empty	Unload	Fault	Status
R1 and R2	T	{1,12,21}	•	•
R1 and R3	T	•	•	{0,2}

The system information can be extracted from information from the design process. In [43], constraints on the plant – preconditions required for movements and forbidden system states – are developed during the design process, and these constraints can provide the system information. Alternatively, the system information can be generated from Relation of Operations [52] created during the design process.

3.4.4 ECA MFSM Verification

Recall the ECA MFSM formalism introduced in Section 2.2.2. The ECA MFSM for the cell controller example used for verification is the ECA MFSM illustrated in Figure 2.5 and described in Section 2.2.2. Steps 1 and 2 are completed in Section 3.4.3, and summarized in Tables 3.2 and 3.3. Starting in each $x_0 \in X_{\{UR1,Arr0\}}$, the controller is simulated for $UR1Arr0$ and for $Arr0UR1$ and the resulting final states and output sets are compared and found to be the same. Thus, the logic controller is $\{UR1, Arr0\}$-input order robust.

3.4.5 IEC 61499 Verification

The example discussed in Section 3.4.2 can be implemented as a function block, as illustrated in Figure 2.3. Because in IEC 61499 events can be associated with data, the events are generalized and associated with data that provides the specific information, as shown in Table 3.4. The state is also modified for IEC 61499, where the fault information is kept as input data instead of stored as state, and the Cell Status is stored in two internal boolean variables, Part1In and Part2In. Thus, for example, Cell Status = 2 is equivalent to Part1In = false, Part2In = true. State is therefore expressed as (Empty Pallet Present × Parts Ready to Unload × Part1In × Part2In). This example involves transitions to the ECC states EArrival and PartReady, and their associated algorithms, as shown in Fig. 2.4, when an empty

pallet arrives or a part 1 is finished, respectively. To demonstrate a case in which a controller may not be input order robust, suppose that the fault input was only added after the function block was completed, and that the designer included this additional functionality only in the PartReady algorithm, and not in the EArr algorithm.

Table 3.4: Event Modifications for Application to IEC 61499

Original Events	**IEC 61499 Event**	**IEC 61499 Data**
Arr0, Arr1, Arr2	Arr	PartA = 0, 1, 2
UR1, UR2	UR	PartR = 1,2
FT	(none)	Fault
Re, UL1, UL2	UL	PartU = 0, 1, 2
LD1, LD2	LD	PartL = 1, 2

With the example implemented as a function block, Step 3 can be performed. The controller is simulated for $x \in X_{\{UR1,Arr0\}}$, for each order $UR1Arr0$ and $Arr0UR1$, and the final states for the different orders are found to be different in the case when there is a fault because the EArr algorithm ignores the fault input. Thus this controller, with the fault input not properly included, is not $\{UR1, Arr0\}$-input order robust.

3.5 Application to IEC 61499

3.5.1 Open Execution Semantics Issues

The execution semantics are not completely specified by the IEC 61499 standard, and hence, different researchers have assumed somewhat different execution semantics in studying and modeling IEC 61499 controllers [25], [11], [24], [59] and [20]. Different IEC 61499 runtime environments often assume different execution semantics, which can lead to difficulties when a system is moved from one runtime environment to another [11]. For each open execution semantic issue, the verification is either applicable for any possible execution option or only applicable for certain possible execution options, and it is important to know which is the case for each

issue. Five such execution semantic issues are discussed herein.

The first open execution semantic issue is function block scan order. Function block scan order is one of the two major execution semantic options discussed in [24], which lists the options as not fixed and fixed. The not fixed option we interpret to mean that the function blocks are scheduled based on receiving events, i.e. if FB2 receives an event and the others are idle, then FB2 will be scheduled to execute next. The fixed option requires setting a complete ordering of all of the function blocks in advance. Either option is applicable for input order robustness verification. A well-chosen FB scan order could avoid some problems associated with lacking input order robustness (see the example in Section 3.5.2), but a poorly-chosen scan order could cause consistently incorrect behavior.

Another major issue is whether the IEC 61499 standard allows multi-tasking. Both [25] and [24] pose the question of whether the standard allows multi-tasking, i.e. multiple function blocks within a network running at the same time. [59] offers three possible options by discussing thread assignment – how function blocks are distributed among the execution threads. The three possibilities are: only one thread for the entire network (no multi-tasking allowed), one thread per function block, or a subset of the function blocks within the network are assigned to each thread. Whether multi-tasking is permitted and how threads are assigned only affect when events are processed by a function block and not what output events and data are produced. Thus, input order robustness verification of a network does not require a particular option for the issue of multi-tasking because, as noted in Section 3.3, the only requirements are that the component function blocks are input order robust and that they do not have any feedback relationships.

The next execution semantic issue considered is that of how contiguous events are

handled, meaning what happens when two or more events arrive in quick succession such that the function block must decide which to process first or has not finished processing the first event before the next event arrives. A discussion of different options for handling contiguous events and which options have been used by particular IEC 61499 software is provided in [11]. According to that discussion, the runtime environment ISaGRAF 5.0 may drop events that arrive at a function block when its ECC is still processing a previous event (no queue), whereas the runtime environment Fuber reports all incoming events to the ECC in the order in which they arrived (FIFO queue). Another possible interpretation of how contiguous events could be handled is that of a priority queue, where events are processed based on the order in which their transitions are declared in the code. This interpretation could be problematic, however, because as [59] points out, the function block may be generated graphically, making it difficult for the engineer to specify the priority. If there is no queue and some contiguous events are dropped, then the system will not be input order robust because the order of event arrival will have the severe impact of allowing one event to be processed and completely dropping the other. Using a priority queue may seem to be an easy way to guarantee input order robustness because the same processing order is always enforced for events that arrive simultaneously or during the same scan cycle. Unfortunately, violations of input order robustness may still occur with a priority queue. Consider the case where there are only two events, a and b and the priority is such that a will be processed first. If the two events can arrive simultaneously or at least very close in time, it would be possible for event b to arrive just before a but still have b processed earlier if they arrive right near the end of one scan cycle and the beginning of another. Thus, input order robustness is not applicable without an event queue, but is applicable and should be verified for

systems with an event queue, regardless of what type of queue it is.

Another open issue is which transition is fired when multiple guard conditions are satisfied. [20] discusses this issue, and based on interpreting the standard, determines that transitions are evaluated in the order in which they appear in the XML code (top to bottom) or in the graphical representation (left to right), and that the first transition evaluated to be satisfied is what will be fired first. As this order may not always be clear, the IOR verification procedure assumes that only one guard condition will be satisfied at a time. Extending the procedure to be applicable to [20]'s interpretation would likely be feasible.

Also discussed in [20] is how guard conditions consume events. For example, if an event occurs that causes a GC to be satisfied and its transition to fire, and in the new state, there is another transition's GC that is satisfied if the event still holds, will both transitions occur or does the first transition's GC consume the event and thus only it will occur? Also, what about guard conditions that do not involve events and only consist of data variables – when are these executed? Three possible options for event consumption are: a single guard condition consumes the event; a single guard condition consumes the event but other transitions whose GCs are satisfied without the event are allowed to execute; or multiple GCs can be satisfied with the one event, and transitions can continue occurring based on that event until no more GCs are satisfied. The second option is the one assumed by the verification work presented here, however it could relatively easily be extended to be applicable to the third option as well. The first option would be difficult to accommodate in IOR verification because, at the beginning of a scan cycle, before any events had arrived, there could be a satisfied GC and thus a transition occurs without an event arriving.

Table 3.5: IEC 61499 Execution Open Issues

	FB Scan Order	**Multi-tasking**	**Contiguous Events Handling**	**Multiple Guard Cond. (GC) Satisfied**	**How GC Consume Events**
IOR Ver Applicable	-Event occurrence -Fixed a priori	-None -1 thread per FB -Subset of FBs per thread	-Prioritized queue -FIFO queue	-Not allowed	-Single GC plus others without event
IOR Ver Not Applicable			-Dropped	-Priority from code*	-Single GC -Not until last satisfied*

All of these execution semantics issues and their possible options are summarized in Table 3.5, along with whether the verification is applicable or not applicable to the particular options. Options for which the current verification is not applicable, but could be feasibly modified to be applicable, are noted by an asterisk (*). Input order robustness verification is applicable for a variety of execution semantics – either function block scan order implementation, any of the multi-tasking options listed, and any type of queue for contiguous events. Additionally, the verification could easily be extended to be applicable for different options of when multiple GC are satisfied and when an event is consumed. Thus, input order robustness verification avoids many difficulties associated with verifying properties for IEC 61499 function blocks due to different execution semantics and runtime environments.

3.5.2 Application of Verification to IEC 61499 Network of Controllers

Recall that Theorem III.4 requires that the components are input order robust and do not have any feedback loops in order for the network to be input order robust. For example, any network that contains the function block described in Section 3.4.5 will not be input order robust because Arr0 and UR1 could arrive in

either order and cause different output sets from the function block, and hence the network. In IEC 61499, such networks would be applications, systems, and even composite function blocks if the composite is treated as a transparent container. These IEC 61499 networks are often self-contained with initiating blocks, such as E_RESTART, or have few inputs which are unlikely to arrive at almost the same time or in either order. Therefore, one might think that input order robustness is not a significant issue for most IEC 61499 networks, when, in fact, meeting the conditions of network input order robustness can resolve some of the issues associated with different execution semantics. An example application of input order robustness to a network of controllers is described, first performing the verification procedure and then discussing how this application relates to the open execution semantic issues.

In this example, which is based on a network from [11], there is a fixture for holding a workpiece and a carriage for transporting the workpiece, where the fixture has a clamp to hold the piece in place during processing and a part that pushes the completed piece off the workspace and onto the waiting carriage. The network to control this application is illustrated in Figure 3.5, and the execution control charts for the carriage and fixture are shown in Figure 3.6. The networks of logic controllers considered here have inputs and outputs, thus the example network considered is that of the fixture and carriage function blocks only, with the inputs EI and EI1 and the output EO (from the fixture block).

To verify $\{\pi, x_0\}$-input order robustness for this network for $\pi = \{EI, EI1\}$ and $x_0 = \{IDLE, IDLE\}$, first check whether there are any feedback loops among the logic controllers and in this example there are none. Then verify that the logic controllers, carriage and fixture, are input order robust for their corresponding π and x_0. The carriage function block is Level 1 because it only receives input from

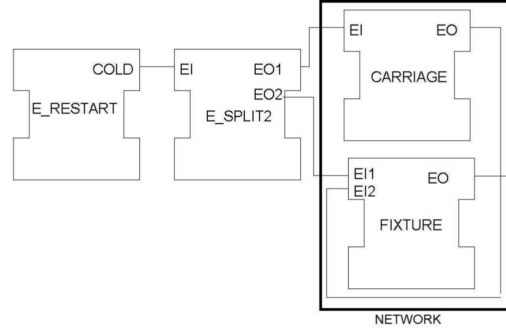


Figure 3.5: Fixture and carriage network example (based on [11])

the environment, whereas the fixture function block is Level 2 because it receives EI2 from the carriage function block. Thus, the carriage function block is verified first. For $\pi = \{EI, EI1\}$, $\pi_{carriage} = \{EI\}$ so the carriage function block is input order robust trivially.

Next, verify the fixture function block where for $\pi = \{EI, EI1\}$ and $x_0 = \{IDLE, IDLE\}$, $\pi_{fixture} = \{EI1, EI2\}$ and $x_{0fixture} = IDLE$. In Step 1a, the inputs $EI1$ and $EI2$ could occur in either order because, from knowledge of the system, it is known that they will occur at the same time and that it is unknown the order the function blocks will be executed in. In Step 1b, the inputs $EI1$ and $EI2$ should be able to occur in either order and yield the same result because $EI1$ is the instruction to process a workpiece and $EI2$ is the notification that the carriage is ready for the part, so it should not make a difference whether the workpiece or the carriage is ready first. In Step 2, we note that the $IDLE$ state is valid for $EI1$ and x_1 state is valid for $EI2$. In Step 3, $IDLE$ and x_1 are valid for at least one of the inputs, and therefore we can go ahead with checking input order robustness for $x_{0fixture} = IDLE$. The function block is simulated for both orders of $EI1$ and $EI2$ from the $IDLE$ state, and find that the final states and outputs are different. If $EI1$ occurs followed by $EI2$, the final state is the $IDLE$ state and

the outputs are those associated with running the OpenClamp and PushOut algorithms, which result in processing and releasing a workpiece. If the inputs occur in the other order, then either the final state is $IDLE$ and there are no outputs (if having $EI2$ occur in $IDLE$ stops the function block from processing) or the final state is x_1 and the output is only that from OpenClamp. Thus, the fixture function block is not $\{\{EI1, EI2\}, IDLE\}$-input order robust, and hence the network is not $\{\{EI, EI1\}, \{IDLE, IDLE\}\}$-input order robust.

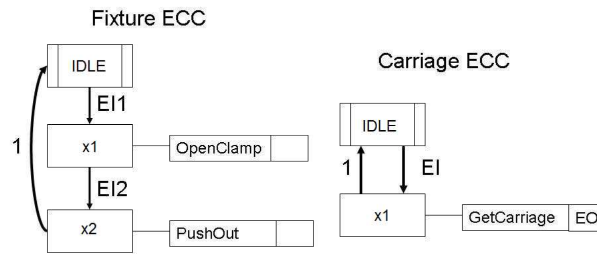


Figure 3.6: Fixture and carriage ECCs (based on [11]).

If the execution semantics have a pre-determined order for function block scanning that dictates the fixture function block be executed before the carriage function block, then for the fixture FB to be IOR, it does not need to be {EI1, EI2}-input order robust because Step 1a will identify that EI1 and EI2 will not occur in either order and thus do not need to be verified. If, however, the function block scanning has the carriage FB execute first, or the order varies from one time to another, then there may be incorrect behavior caused by the lack of IOR. In a small example, such as this one, it may be easier to identify this problem and schedule the function blocks to avoid it. For larger-scale systems, however, determining the function block scanning order that results in correct behavior may be quite difficult. If the conditions for network input order robustness are met, then different execution semantics can be used and still result in correct behavior, which means that moving the network to a

different runtime environment is less likely to cause problems.

3.6 Computational Complexity

The logic controller and network of controllers examples from Sections 3.4 and 3.5 are relatively small in size, but some of the issues associated with applying this verification to larger and more complicated systems can be highlighted through discussing the verification's computational complexity. Discussed here is the computational complexity of verifying $\{\pi, x\}$-input order robustness for all $\pi \in \Pi_{2,check}$, and their associated valid initial states $x \in X_\pi$, not finding these pairs and states. The computational complexity of verifying $\{\pi, x\}$-input order robustness where $|\pi| = 2$ is that of the two simulations (one for each input order), which is generally proportional to the size of the state space $2|X|$. The number of input pairs that need to be verified to guarantee deterministic closed-loop behavior is $|\Pi_{2,check}|$. The computational complexity of the verification is $O(|\Pi_{2,check}||X_{\{a,b\}}||X|)$ where $|X_{\{a,b\}}|$ is the maximum size of initial states for any $\pi \in \Pi_{2,check}$. The maximum number of simulations is $p(p-1)|X|$ and the estimated complexity is $O(p(p-1)|X|^2)$ where p is the number of inputs. If a system has very large values for either p or $|X|$, its input order robustness verification may not be feasible. Some such large systems, however, may be able to be described by a network of controllers and this may reduce the computational complexity to make verification feasible.

An advantage to using a network of controllers is that the computational complexity of its verification may be less than that of a comparable single logic controller. If there is a network that has c components, each of which has r states and p inputs, then the maximum computational complexity of performing its input order robustness verification is $cr^2(p^2 - p)$. In contrast, if that same network is composed

into a single logic controller, even if we assume it only has as many inputs as one of the components from which it was created (p inputs), the maximum computational complexity of performing its input order robustness verification is $r^{2c}(p^2 - p)$, which is exponentially larger than that of the network. The computational complexity of verifying a network as input order robust may be further reduced in the case that some of its components have already been verified for other networks, thus reducing the number of simulations required for the current verification.

3.7 Conclusions

In this verification research, the concept of input order robustness was formalized and a verification procedure created for both logic controllers and networks of controllers. Theory was proven for conditions under which a network of controllers can be verified modularly, and the procedure for doing so described. This verification procedure involves identifying the sets of inputs that need to be checked, performing simulations, and comparing the results. The input order robustness verification was applied to ECA MFSM and IEC 61499 logic controllers, and the effect of input order robustness on several open execution semantic issues for IEC 61499 analyzed. The computational complexity of this verification procedure was discussed. Input order robustness verification provides a means for faults associated with a lack of input order robustness to be found so they can be remedied prior to a logic controller or network of controllers being used. Thus, such verification can reduce downtime of the system due to faults.

CHAPTER IV

Anomaly Detection for Event-Based Systems Without Pre-Existing Formal Models

To help detect faults in systems without pre-existing formal models, such as the motivating RFT example, we developed an anomaly detection solution [4]. This solution is a form of supervised learning for binary classification. Given a set of data that belongs to one of two classes, a classifier (a black box decision-maker or function) is learned. The classifier can then be used to classify future data produced from the same system [56] [29]. Our proposed solution has four main steps:

1. Solution Set-Up (see Section 4.5.1)
2. Model Generation (see Section 4.5.3)
3. Performance Assessment (see Section 4.6)
4. Anomaly Detection (see Section 4.7)

The first step prepares the inputs – knowledge about the system's events and resources and a set of labeled event streams that describe some of the system's behavior. These inputs are used in the model generation step, where the labeled event streams are used to learn a set of models of the system's behavior that can act as classifiers. The third step is performance assessment, which consists of using labeled event streams to assess the models' performance on detecting anomalies. Finally,

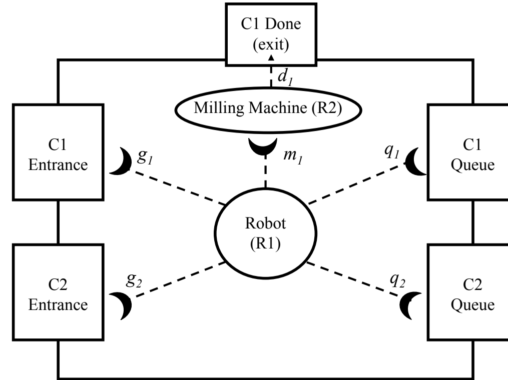


Figure 4.1: Illustration of manufacturing cell, where dashed lines show possible movements of the robot and milling machine, and their associated events (in italics)

anomaly detection itself is done in the fourth step, where unlabeled event streams are classified based on the models' agreement or disagreement with the streams. First in Section 4.1 a small manufacturing cell is described that will be used throughout this chapter as a running example. Next, the types of systems to which this solution is applicable and their modeling formalism are presented in Sections 4.2 and 4.3. The specific problems addressed by this solution are described in Section 4.4, and then the four steps of the solution will be described in detail in Sections 4.5 through 4.7. Results are presented from applying this anomaly detection solution to an abstracted version of the RFT cell in Section 4.8. Application of the solution to an industrial system is the focus of Chapter V. The work presented in this chapter is from [4].

4.1 Description of Small Manufacturing Cell

A small example system will be used to illustrate the type of system for which our anomaly detection solution is applicable and the steps of the solution. A manufacturing cell, illustrated in Figure 4.1, interacts with two different components, called $C1$ and $C2$, and performs machining. The cell has two resources – a robot (R1) to transport components, and a milling machine (R2) to mill components. When a

component $C1$ arrives at the cell, the robot will get it, and if the milling machine is available, place it in the milling machine for further processing, after which the milling machine pushes it out of the cell. If the milling machine is not available, the robot will place it in a queue for $C1$ components waiting to be processed. When a component $C2$ arrives at the cell, the robot will get it and place it in a queue for $C2$ components, where removal from this queue is outside the context of this cell. Thus, the process performed by this cell is naturally broken down into two processes, one for the handling of each component.

The cell's events are listed in Table 4.1 and illustrated in Figure 4.1. Resource information is summarized in Table 4.2, including the interaction between resources and events. The observed streams for this cell are

$$\sigma_1 = g_1 m_1 g_1 d_1 m_1 d_1 g_1 m_1 g_1 q_1 g_1 q_1 g_2 d_1 q_2 g_1 m_1 g_2 d_1 q_2$$

$$\sigma_2 = g_2 q_2 g_1 m_1 g_1 q_1 d_1 g_2 q_2 g_2 q_2 g_1 m_1 g_2 q_2 g_1 d_1 m_1 g_1 d_1$$

$$\sigma_3 = g_2 q_2 g_1 m_1 g_2 d_1 q_2 g_2 q_2 g_2 q_2 g_2 q_2 g_1 m_1 g_1 q_1 g_2 d_1 q_2$$

$$\sigma_4 = g_2 q_2 g_1 m_1 d_1 g_2 q_2 g_1 m_1 g_1 d_1 m_1 g_2 q_2 d_1 g_1 m_1 g_1 d_1 q_1$$

Streams σ_1 - σ_3 are labeled "no-fault" and σ_4 is labeled "fault."

Table 4.1: Events in Manufacturing Cell

Name	Process	Description
g_1	1	**G**et $C1$
m_1	1	Put $C1$ in **M**achine, begin processing it
d_1	1	$C1$ is **D**one
q_1	1	Put $C1$ in **Q**ueue
g_2	2	**G**et $C2$
q_2	2	Put $C2$ in **Q**ueue

A variation on this example is also considered, where the cell performs assembly in addition to machining. In the assembly cell, $C2$ is not a separate component, but rather a sub-component that is joined with $C1$ to form a final product after $C1$ has been machined. Thus $C2$ is a resource that is created, where the events that acquire

Table 4.2: Resource information for Manufacturing Cell

Name	Type	Acquire Events	Release Events
R1 (robot)	A (Always)	g_1, g_2	m_1, q_1, q_2
R2 (machine)	A (Always)	m_1	d_1

it are q_2 and d_1 and the event that creates (releases) it is g_2. When $C1$ arrives at the assembly cell, it is queued if $R2$ is unavailable, and otherwise it is machined and then waits for a $C2$ to arrive to be assembled. When $C2$ arrives at the assembly cell, it can be assembled with a $C1$ if available or can be queued.

4.2 System of Processes That Interact Through Shared Resources

To further constrain the problem's scope, a set of assumptions have been made about the type of system considered and the information assumed known about it. We consider systems of processes that interact through shared resources, or SPSRs for short. First an informal description of an SPSR and example SPSRs will be presented, then the formal definitions.

4.2.1 Intuition and Examples for SPSRs

Each process in an SPSR consists of a set of events and a set of resources that interact deterministically. The state of a process is the availability of the resources and their current use. A process may, but is not required to, execute concurrent instances of itself. It also may exhibit mutual exclusion where two events are possible, but a choice about which event to execute (or at least execute first) is made based on resource availability.

We assume that each event is associated with exactly one process, but that a resource can be associated with one or more processes. Each resource has at least one event that acquires it and at least one event that releases it. Each event may acquire and/or release any number of resources, including none, although it may not

both acquire and release the same resource. Resources may be always present (A) or created (C). An always present resource is available whenever it is not busy with another task, whereas a created resource is created and then consumed, making it unavailable until another instance is created.

We further assume that we know which process each event belongs to, which if any resources it acquires, and which if any resources it releases. An SPSR has a set of labeled event streams – streams of events recorded from the system running, where each stream has a label indicating whether it represents no-fault or fault behavior of the system. The set of labeled event streams includes only a proper subset of all the possible no-fault streams because it is assumed that not all of the system behavior has been recorded. We also assume that a formal mathematical model of the system is not available, nor any information about such a model other than the event and resource information described herein.

One example SPSR is a manufacturing cell, like the motivating RFT example, that machines more than one type of part, where each part type has a process associated with its machining, and these processes interact only through shared resources, such as material handling devices and milling machines. Each part that arrives at the cell causes a new instance of its associated process to begin. Each event is associated with the machining of a particular part type, i.e. part type 1 arrives, start machining part type 2, etc. Some resources, such as pallet stops, empty pallets, and robots are shared resources among the different processes, while others, such as CNCs, may be used by only one process. Most of the resources are type A, such as the robot and CNCs. The empty pallet, however, is a type C resource meaning that it is created (by arriving at the cell) and consumed through use (a part put on it, making it no longer empty). We know which events acquire and release each resource, such

as starting to load part type 1 acquires CNC1 and finishing unloading part type 1 releases CNC1. Another example SPSR is patient flow through a clinic, where there are two processes – one for patients with appointments and one for patients that are walk-ins – that interact through shared resources such as receptionists, nurses, physicians, equipment, and rooms. Each patient that arrives is associated with a new instance of either the appointment or walk-in process. Each event is associated with treating a particular patient, i.e. a patient with an appointment arrives, a walk-in patient has their need evaluated, etc.

4.2.2 Formal Definitions for SPSRs

The problem of interest and our solution are related to events, resources, processes, and systems of processes that interact through shared resources (SPSRs). Events, as specified in Definition 1, are discrete occurrences or messages such as OPC tags or bit changes on a PLC. As an example, the small manufacturing cell's event set (Definition 2) is $E = \{g_1, m_1, d_1, q_1, g_2, q_2\}$, and σ_1 - σ_4 are some of its event streams (Definition 3). As mentioned in the introduction to his chapter, because our anomaly detection is a form of supervised learning, we need a set of classified or labeled examples of our system's behavior, so we extend Definition 3.

Definition 11 (Labeled event stream)**.** A *labeled event stream* σ is an event stream that has an associated binary label (0 or 1). If the labeled event stream is provided for model generation, then 1 indicates that it is no-fault behavior, and 0 indicates fault behavior. If the labeled event stream is labeled by the anomaly detection solution, then 1 indicates normal behavior and 0 indicates abnormal behavior.

Definition 12 (Event log)**.** An *event log* Σ is a finite set of event streams, where all streams are generated by the same system, all are either labeled or unlabeled, and

they may be of different lengths.

In the manufacturing cell, each of the event streams σ_1 - σ_4 are labeled as either no-fault or fault, and $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$ is an event log. In addition to event definitions, we also will define resource and process in order to give a formal definition of SPSR, which was described informally in Section 4.2.1.

Definition 13 (Resource)**.** A *resource* is a physical object that is acquired by an event e_a for use and released by an event e_r when its use is complete. Resources can be of two types – either Always present (A) or Created (C). A type A resource is available whenever it is not busy with another task, whereas a type C resource is created and then consumed (the type C equivalent of released and acquired, respectively).

The manufacturing cell's resources are the robot (R1) and machine (R2), with their information about type and events that acquire and release them given in Table 4.2.

Definition 14 (Process)**.** A *process* Q consists of a set of events E and a set of resources R that interact deterministically to accomplish a goal, and in doing so, produce a stream of events σ.

The goal may be accomplished in more than one way or multiple times, leading to different event streams that can be produced. A process may, but is not required to, execute concurrent instances of itself where the maximum number of such instances may not be known in advance. Decisions about how a goal is accomplished can be determined by resource availability or randomly. A resource may be used multiple times and in different ways during the course of running the process, and multiple resources may be used together. The manufacturing cell has two processes – one for each component, $C1$ and $C2$. Two instances of the process for $C1$ may execute at

the same time because one $C1$ may be machined while another $C1$ is being picked up by the robot. The decision about what to do with a $C1$ depends on the availability of the milling machine, $R2$. A process may be split into several smaller processes, or several processes may be combined into one larger process. The example system as a whole can be considered an SPSR, where the system is the set of two processes that interact through the shared robot $R1$.

Definition 15 (System of Processes That Interact Through Shared Resources (SPSR)**).** A *system of processes that interact through shared resources*, or *SPSR*, is a set of processes Q_i $i = 1...n$ with disjoint event sets $E_i \cap E_j = \emptyset\ \forall i \neq j$ and non-disjoint resource sets R_i where $\forall i = 1...n\ \exists j \neq i$ such that $R_i \cap R_j \neq \emptyset$ and each process is connected to every other process, either directly through shared resources or indirectly through other processes. The event and resource sets for the SPSR, E_i and R_i for $i = 1...n$, are known as well as which events acquire and release each resource.

4.3 Petri Net Models for System of Processes that Interact Through Shared Resources

To generate models for systems of processes that interact through shared resources (SPSRs), a modeling formalism needs to be selected or created. Some formalisms specifically incorporate resources into processes modeled by Petri nets. In [21], a special type of Petri net called a simple sequential process (S^2P) is used to describe the behavior of flexible manufacturing systems. This definition is expanded to include special resource places (simple sequential process with resources S^2PR), where the token is removed from a resource place when the resource is put into use and the token is replaced when the resource is no longer in use. An example S^2PR is illustrated in Figure 4.2. Systems of S^2PRs, called S^3PRs, are composed of S^2PRs that interact through shared resources. In [31], these definitions were modified to create a different

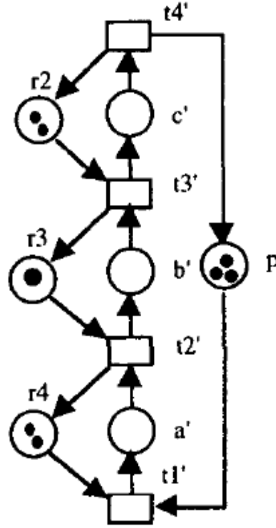


Figure 4.2: Example S^2PR from [21] where resources represented by places $r2$, $r3$, and $r4$

type of resource Petri net, one that has beginning and ending transitions rather than an idle place, such as p' in Figure 4.2. These resource Petri nets include EWF-net, $EWFR$-net, and $MEWFR$-net, which are comparable to S^2P, S^2PR and S^3PR, respectively. Another modeling formalism derived from a variant of S^3PR is the Gadara net [62] which is created to model multithreaded programs.

These existing resource-based Petri net formalisms (those from [21], [31], and [62]) cannot completely describe SPSRs as defined by Definition 15, with properties inherited from Definition 14. The definitions from [21] require that there be a process idle place (p_0, marked as p' in Figure 4.2) such that to start an instance of the process ($t1'$ to occur in Figure 4.2), a token must be drawn from this place and to end an instance ($t4'$ to occur in Figure 4.2), a token must be put back. Having such a place requires that the number of instances of a process that can occur at the same time be limited to some fixed number (the initial marking of the idle place, $M_0(p^0)$), which violates the requirement in Definition 14 that the number of concurrent instances may not be known. For example, in Figure 4.2 the process is arbitrarily limited to

having only three concurrent instances because $M_0(p') = 3$, even though there are enough resources to allow five instances. None of these existing formalisms allow resources to be created – they are present initially, unavailable when in use, and available again as soon as their use is completed, whereas an SPSR can have resources that are created. If the example SPSR included modeling the pallets on which the components leave the system (type C resources), then the existing formalisms could not model it. Negated resources are not allowed by the existing formalisms, which makes it difficult, if not impossible, for them to abide by the condition in Definition 14 that decisions may be made based on resource availability. In fact, Gadara nets specifically prohibit that if/else decisions are made based on resource availability. In the manufacturing cell example, whether $C1$ is processed depends on the availability of resource $R2$ but making that decision in a Petri net requires having a negated resource. Additionally, the formalisms from [21] and [31] incorporate resources but require that there is a one-to-one relationship between tasks and resources, meaning that each task uses exactly one resource and each resource is used by only one task, but in an SPSR resources may be used by different tasks within a process or may be used continuously through a set of tasks. A variation of the system from Figure 4.2 is shown in Figure 4.3 where the resource usage has changed to illustrate some of the expanded possibilities allowed by SPSR. These existing formalisms also require that the individual process (i.e. S^2P or EWF-net or sub-net of Gadara net) be a state machine, which means that it can exhibit competition but not concurrency (a transition feeding two non-resource places), and although excluding concurrency is not prohibited for SPSRs, it unnecessarily restricts the systems that could be considered SPSRs. Hence, in this paper new formal definitions have been created for a formalism called System of Transition Processes with Resources ($STPR$) that are

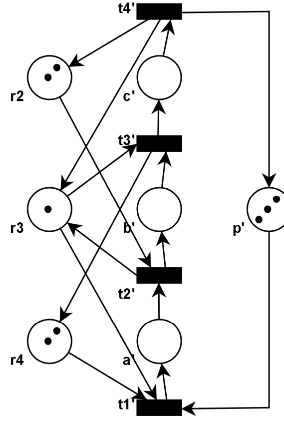


Figure 4.3: Example S^2PR from [21] modified to have different resource usage

based on those from [21], [31], and [62] but can describe SPSRs and avoid these overly restrictive properties. A summary comparison of these resource Petri net formalisms is given in Table 4.3.

Table 4.3: Comparison of Petri net formalisms that use resources

Attributes	S^3PR [21]	$MEWFR$-net [31]	Gadara nets [62]	$STPR$
Has basic, with resources, and combined versions of formalism	Yes (S^2P, S^2PR, S^3PR)	Yes (EWF-net, $EWFR$-net, $MEWFR$-net)	Yes, but not formally defined	Yes (TP, TPR and $TPCR$, $STPR$)
Initiates and terminates basic process	idle place, p^0	one beginning, one ending transition	one beginning, one ending transition	sets of beginning and ending transitions
Num. concurrent process instances	$\leq M_0(p^0)$	constrained by system	constrained by system	constrained by system
Resources can be created	No	No	No	Yes
Negated resources can be used	No	No	No	Yes
Resources per task	1	1	≥ 1, $\leq$ num. resources	≥ 1, $\leq$ num. resources
Tasks per resource for basic process	1	1	≥ 1, $\leq$ num. tasks	≥ 1, $\leq$ num. tasks
Petri net type for basic process	state machine	state machine	state machine	free choice Petri net

Another relevant Petri net formalism is the sound SWF-net from [60]. A comparison is provided between sound SWF-net and TP – the comparable model from

our definitions – because an algorithm that creates sound SWF-nets is modified in Section 4.5 to create a set of models that may include TPs. The comparison is summarized in Table 4.4. The TP formalism is different than sound SWF-nets primarily in having a set of beginning and ending transitions instead of a single input place and single output place, and in only being bounded (allowing a finite number of tokens per place), rather than safe (allowing only one or fewer tokens per place) when resources are added to make it a TPR. Due to the single input and single output places, where the input place initially has one token, SWF-nets cannot model concurrent instances of a process, and thus cannot model SPSRs.

Table 4.4: Comparison of Sound SWF-nets and TPs

Attributes	Sound SWF-net [60]	TP
Starting and ending	Input place i, output place o	Set of beginning transitions, ending transitions
Petri net type	Free choice	Free choice
Implicit places	Not allowed	Not allowed (but is in TPR)
Boundedness	1-boundedness (safeness) required	Guaranteed when restricted by resources (TPR)

Our new resource-based Petri net formalism, called $STPR$, is modularly constructed. An $STPR$ consists of processes that are described by Transition Processes with Resources (TPRs) and/or Transition Processes that Create Resources ($TPCR$s) that interact through shared resources. Each TPR is made from a Transition Process (TP) to which resources have been added. For all levels of this formalism, the set of transitions T corresponds exactly to the set of events E.

Definition 16 (Transition Process TP)**.** A *transition process* (TP) is a marked Petri Net $N = \{P, T, F, M_0\}$ where

1. the set of transitions consists of three types $T = T_B \cup T_M \cup T_E$, where $T_B = \{t \in T | \bullet t = \emptyset, t\bullet \neq \emptyset\}$ are transitions that begin an instance of the process, $T_M = \{t \in T | \bullet t \neq \emptyset, t\bullet \neq \emptyset\}$ are transitions in the middle of the process,

$T_E = \{t \in T | \bullet t \neq \emptyset, t\bullet = \emptyset\}$ are transitions that end an instance of the process, and $T_B \neq \emptyset$, $T_E \neq \emptyset$

2. $N' = \{P \cup p_0, T, F'\}$ is a strongly connected free choice Petri net where $F' = F \cup \{(p_0, t_b) | \forall t_b \in T_B\} \cup \{(t_e, p_0) | \forall t_e \in T_E\}$ and p_0 is added to test this condition by connecting the beginning and ending transitions

3. $\forall t \in T, t \bullet \cap \bullet t = \emptyset$

4. $\forall p \in P, M_0(p) = 0$

5. there are no implicit places

From item 1, the set of transitions consist of beginning transitions (T_B) that initiate an instance of the process without requiring any tokens, intermediate transitions (T_M) that both require tokens and produce tokens, and ending transitions (T_E) that end an instance of the process and do not produce tokens. Item 2 states that, with the addition of an extra place connecting the beginning and ending transitions, the net is strongly connected free choice which means that it allows both concurrency and conflict, just not at the same time, and all of the nodes are connected. Loops of length 1 (one event that can repeat itself) are prohibited by Item 3. Item 5 prohibits implicit places, which are places that do not affect the behavior of the Petri net. A TP is not a process, as defined in Definition 14, because it does not have resources. From the manufacturing cell described in Section 4.1, the associated TP for processing $C1$ is shown in Figure 4.4 where it has four events, two places, and exhibits conflict but not concurrency. If resources are added to a TP, then a TPR can be created.

Definition 17 (Transition Process with Resources (TPR))**.** A transition process with resources (TPR) is a marked Petri net $N = \{P \cup P_R, T, F, M_0\}$ such that

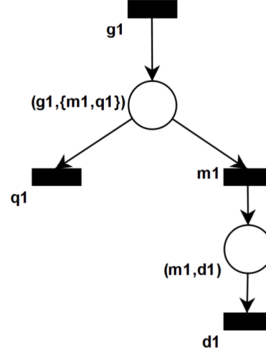


Figure 4.4: TP from process 1 of example manufacturing cell

1. The subnet generated by $N' = \{P, T, F', M'_0\}$ is a TP, where $F' = \{(p, t_1), (t_2, p) \in F | p \in P, t_1, t_2 \in T\}$ and M'_0 is M_0 restricted to the places in P
2. $P_R = P_A \cup P_C \cup P_{NA} \cup P_{NC}$
3. $\forall r \in P_A \cup P_{NC}$, $1 \leq M_0(r) < \infty$ and $\forall p \in (P \cup P_C \cup P_{NA})$, $M_0(p) = 0$
4. The four following statements are verified:
   a. $\forall r \in P_A \cup P_{NA}$ $r\bullet \neq \emptyset$, $\bullet r \neq \emptyset$ and $\bullet r \cap r\bullet = \emptyset$
   b. $\forall r \in P_C, r\bullet \neq \emptyset$, $\bullet r = \emptyset$ and $\forall r \in P_{NC}, r\bullet = \emptyset$, $\bullet r \neq \emptyset$
   c. Each $r \in P_{NA}$ is associated with a unique resource $r' \in P_A$ where the arcs for r are the exact opposite of the arcs for r' plus possibly decision arcs that go both directions between r and an event; same for each $r \in P_{NC}$ with respect to $r' \in P_C$
   d. For each $r \in P_A$, there exists a unique minimal-support p-semiflow y_r such that $||y_r|| \cap (P_A \cup P_{NA}) = r$, $||y_r|| \cap (P \cup P_C) \neq \emptyset$ and $\forall p \in P \cup P_C$, $\exists r \in P_A$ such that $p \in ||y_r||$
5. $\forall t_B \in T_B$, $t_B \bullet \cap P_R = \emptyset$, and $\forall t_E \in T_E$, $\bullet t_E \cap P_R = \emptyset$
6. $\exists$ at least one event stream $\sigma = t_1 t_2 ... t_k$, where k finite, such that when σ fires in N with initial marking M_0, the resulting marking $M_\sigma = M_0$

7. N has no dead transitions

From item 1, the subnet of the TPR that includes only the process places P is a TP. Item 2 means that resources fall into four categories: resources that are Always present or Created as mentioned in Section 4.2.1, and the negation of each, NA and NC. Type A resources are initialized with at least one token to represent their initial availability. The negation of a resource is associated with a unique resource and shows the opposite of the resource's availability – it has the exact opposite arcs (if the resource has an arc to a place, then the negation has an arc from that place), plus possibly decision arcs (an arc both to and from a given transition) as explained in 4c. These negation places are used only for making decisions, i.e. when one place feeds two transitions and we need to know which transition (event) should occur. Item 4a expresses that for each A and NA resource, at least one event releases it and at least one event acquires it, and these events cannot be the same for a given resource. For each C resource, at least one event releases it and no events acquire it, and the opposite for each NC resource, as described in Item 4b. Item 4d states that there is a p-semiflow for each for each type A resource and that this p-semiflow's support does not contain any other type A or NA resources but does contain at least one non-resource or type C place. This item guarantees that when a type A resource is acquired, it will eventually be released because the TPR is conservative with respect to the minimal supports for the type A resources. Each non-resource place and type C place is a member of at least one p-semiflow's support, also according to Item 4d, which implies that every step in the process requires a type A resource, and thus these resources limit what the process can do and how much concurrency it can exhibit. Item 5 says that no beginning transitions release resources and no ending transitions acquire resources. Item 6 states that the TPR must have at least one

event stream σ that returns the net to its original marking. Finally, according to Item 7, the TPR cannot have any dead transitions, which are transitions that cannot be reached by a marked Petri net. From Items 3, 4d, and 4e, and the property that $y^T M = y^T M_0$ for all reachable markings M, we can conclude that the TPR must be bounded. A TPR can model some processes, where process is as given by Definition 14. Non-resource places ($p \in P$) are also called process places.

An example TPR is illustrated in Figure 4.5. The TPR in Figure 4.5 comes from adding resources $R1$, $R2$, and $NotR2$ to the TP from Figure 4.4. A TPR can use type C resources (acquire them) but not create them (as reflected in Item 4b). In contrast, a Transition Process that Creates Resources $TPCR$ only creates a type C resource and does not have any process places.

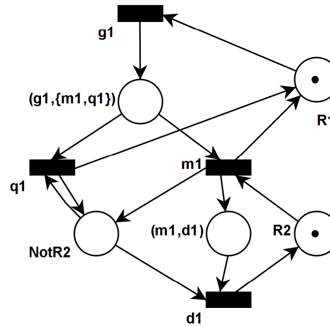


Figure 4.5: TPR Model, Generated for Process 1 with NotR2, m_1 causes d_1

Definition 18 (Transition Process that Creates Resources ($TPCR$)). A transition process that creates resources ($TPCR$) is a marked Petri net $N = (P_R, T_B \cup T_E, F, M_0)$ where all of the places are resources, $P_R = P_A \cup P_C$, and $!\exists r \in P_C$ such that $\forall b \in T_B$, $(b, r) \in F$ and $\forall e \in T_e$, $(r, e) \in F$; and $M_0(p) = 1$ only for $p \in P_A$.

A $TPCR$ can model some processes, specifically those that only create a type C resource. The assembly variation of the manufacturing cell has a $TPCR$ for the process associated with $C2$. This $TPCR$ is illustrated in Figure 4.6 where place

(g_1, q_1) is re-labeled as $C2$, as shown in brackets. Processes can be described by TPRs and $TPCR$s, and when they are combined through shared resources, describe SPSRs using $STPR$s. First we define the composition of a set of processes, then $STPR$s.

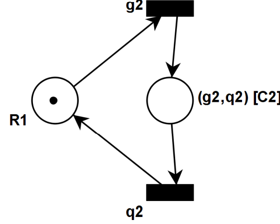


Figure 4.6: TPR Model Generated for Process 2 of Manufacturing Cell Example [$TPCR$ Model for Process 2 of Assembly Ex]

Definition 19 (Composition of TPRs and/or $TPCR$s)**.** The composition of TPRs and/or $TPCR$s $N_i = \{P_i \cup P_{Ri}, T_i = T_{Mi} \cup T_{Bi} \cup T_{Ei}, F_i, M_{0i}\}$ for $i = 1...m$ is defined as follows: 1) $P = \cup P_i$, 2) $P_R = \cup P_{Ri}$, 3) $T_B = \cup T_{Bi}$, $T_M = \cup T_{Mi}$, $T_E = \cup T_{Ei}$, $T = \cup T_i$ 4) $F = \cup F_i$, 5) $M_0(p) \geq 1 \; \forall p \in P_A \cup P_{NC}$ and $M_0(p) = 0 \; \forall p \notin (P_A \cup P_{NC})$ and 6) Items 4d and 4e from Definition 17 also hold for the composition of N_i and the resulting P and P_R

Given these requirements, the process models can be composed into a single model by taking the union of the places, transitions, and flow functions, and setting the marking based on the initial resource availability.

Definition 20 (System of TPR ($STPR$))**.** A system of TPR ($STPR$) is defined recursively as follows:

1. A TPR or $TPCR$ is an $STPR$.
2. Let $\{N_i\}$ $i \in \{1, 2, ...m\}$ $m < \infty$, where $N_i = \{P_i \cup P_{Ri}, T_i = T_{Mi} \cup T_{Bi} \cup T_{Ei}, F_i, M_{0i}\}$, be a set of TPRs and/or $TPCR$s such that

a. $P_i \cap P_j = \emptyset$ for $i \neq j$; for every $i \in \{1...m\}$ $\exists$ at least one $j \neq i$ such that $P_{Ri} \cap P_{Rj} \neq \emptyset$; each process i is connected to every other process j, either directly through shared resources or indirectly through another process

b. for all $i \neq j$ $T_i \cap T_j = \emptyset$

c. $\forall r \in P_{Ci} \cup P_{NCi}$, $\exists i \in \{1, 2, ...m\}$ such that $\bullet r \cap T_i \neq \emptyset$ and $r \bullet \cap T_i \neq \emptyset$

then the net $N = \{P \cup P_R, T_B \cup T_E \cup T, F, M_0\}$ resulting from the composition of N_i for $i = 1...m$ is an $STPR$.

A TPR or $TPCR$ can be an $STPR$ on its own, as stated in Item 1. Item 2 lists requirements for the process models that make up the $STPR$. None of the process models share any process places, but they are all connected by sharing resources (Item 2a). None of the TPR and $TPCR$ models share any transitions (Item 2b). For every type C and type NC place, there exists a process model that creates and consumes the resource (Item 2c). Given the requirements of Item 2, the TPRs and/or $TPCR$s can be composed into an $STPR$. These resource-based Petri net formalisms, from TP through $STPR$, are used in our model generation algorithm. The TPRs for the example manufacturing cell are combined into an $STPR$ and illustrated in Figure 4.7. For the example assembly cell, the only difference would be in re-naming place (g_2, q_2) to $C2$ and adding an arc from the place for $C2$ to the transition d_1, as a sub-component would be required to finish processing a $C1$ component.

4.4 Problem Statements

With the definitions available, the problems addressed by this anomaly detection solution can be formalized.

Problem 1. Given an SPSR as in Definition 15, including its processes' events T_i, resources R_i, and the mapping between resources and the events that acquire and

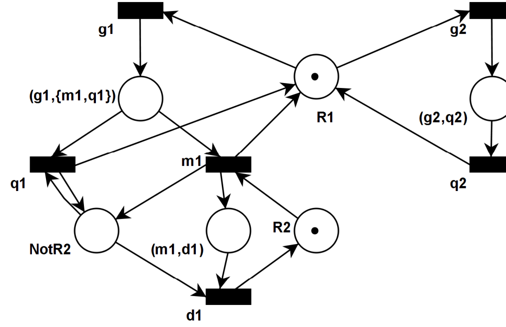


Figure 4.7: Example $STPR$, Whole Model From Combining Third Process 1 Model and Sole Process 2 Model

release them, T_{aR_i} and T_{eR_i} for $i = 1...$ number of processes, and at least one event log Σ whose event streams are all labeled as no-fault, generate a set of $STPR$ models that describe the behavior of the SPSR.

Problem 2. Given a set of $STPR$ models, a labeled event log Σ_N of no-fault streams, possibly a labeled event log Σ_A of fault streams, and an unlabeled event stream σ, provide a metric indicating how well the $STPR$ models' behavior matches that of Σ_N and Σ_A and use the results of this metric and how well the $STPR$ models' behavior matches the unlabeled event stream σ to determine whether σ likely contains an anomaly.

4.5 Model Generation

4.5.1 Steps for Set-Up

Given the definitions, Problems 1 and 2, and the proposed solution from the introduction of Chapter IV, we can now describe the four steps for the set-up of our solution:

S1) Using knowledge about the system, list all processes, events, and the association between them

S2) Using knowledge about the system, list all resources and, for each resource, its

type, which events acquire it, and which events release it

S3) Collect event streams from the running system where each stream starts from the system's initial state and is labeled as no-fault or fault (1 or 0, respectively)

S4) Create three event logs: put all of the fault streams into one log (to use in performance assessment), and randomly divide the no-fault streams into two separate logs (one for model generation, another for performance assessment), such that neither of these logs is empty.

The set-up is applied to the manufacturing cell described in Section 4.1.

S1) The process and event information is listed in Table 4.1 and illustrated in Figure 4.1.

S2) The resource information is given in Table 4.2.

S3) The event streams available are σ_1, σ_2, σ_3, and σ_4.

S4) The event streams are split into logs $\Sigma_1 = \{\sigma_1, \sigma_2\}$ for model generation, and $\Sigma_2 = \sigma_3$ and $\Sigma_3 = \sigma_4$ are the no-fault and fault, respectively, for performance assessment.

4.5.2 $\alpha+$ Algorithm

The $\alpha+$ algorithm, developed in [60] and [16], is a workflow mining algorithm that creates a sound SWF-net model based on an observed event log. Although the $\alpha+$ algorithm was developed to create sound SWF-nets, parts of this algorithm can also be used in our model generation algorithm to create a set of models, some of which may be TPs, that are used to create whole process models for anomaly detection purposes. The $\alpha+$ algorithm calculates statistics about the pairs of events that occur sequentially in the event log and then uses these statistics to develop event relationships and create a model. The main steps of the $\alpha+$ algorithm are:

α1) Determine the events for the given log, Σ, and the events that occur first and last in the streams in Σ

α2) Calculate the ordering relations between event pairs using Definition 21, which is modified from [60] and [16] to fit the notation used here

Definition 21 (Ordering Relations for Event Pairs)**.** Given two events, a and $b \in T$, that occur in the event log Σ, the *ordering relation* for the event pair a and b is

- $a > b$ if and only if $ab \in \sigma \in \Sigma$
- $a \Delta b$ if and only if $aba \in \sigma \in \Sigma$
- $a \diamond b$ if and only if $a \Delta b$ and $b \Delta a$
- $a \rightarrow b$ if and only if $a > b$ and $b \not> a$
- $a \sharp b$ if and only if $a \not> b$ and $b \not> a$
- $a || b$ if and only if $a > b$ and $b > a$

Intuitively, $a \diamond b$ implies that a and b are in a two event loop, $a \rightarrow b$ implies that a causes b, $a \sharp b$ implies that a and b have no causal relationship, and $a || b$ implies that a and b can occur in either order.

α3) Using the event pair ordering relations, create places – if $a \diamond b$, then create one place connecting a to b and another place connecting b to a; if $a \rightarrow b$ then create one place connecting a to b.

α4) Combine places as much as possible – two places can be combined if all of the source events for the two places have a $\rightarrow$ or $\diamond$ relationship with all of the sink events of the two places, and none of the source events are related to one another ($\sharp$) and likewise none of the sink events are related to one another ($\sharp$).

α5) Add initial place and final place – add a place that is initially marked and is connected to the initial events (those that occur first in a stream), and an

unmarked final place that is connected to the final events (those that occur last in a stream).

The basics of the $\alpha+$ algorithm can be illustrated through the manufacturing cell example described in Section 4.1.

$\alpha1$) For the given log $\Sigma_1 = \{\sigma_1, \sigma_2\}$, the events are listed in Table 4.1, the events that occur first in the streams are g_1 and g_2, and the events that occur last in the streams are q_2 and d_1.

$\alpha2$) The ordering relations for the event pairs are determined by first finding the event pair occurrences, which are totaled in Table 4.5. For example, the entry for (m_1,g_1) is 4, indicating that in the event streams in Σ_1, there were four occurrences of m_1 followed by g_1. Based on these event pair occurrences, the ordering relations using Definition 21 are illustrated in Table 4.6, where the row event is first and the column event is second such that $(i, j)th$ entry is the relationship from event i to event j.

Table 4.5: Event Pair Occurrences in Example

	g_1	m_1	d_1	q_1	g_2	q_2
g_1	0	5	3	3	0	0
m_1	4	0	1	0	2	0
d_1	1	2	0	0	1	2
q_1	1	0	1	0	1	0
g_2	0	0	2	0	0	4
q_2	4	0	0	0	1	0

Table 4.6: Ordering Relations for Event Pairs in Example

	g_1	m_1	d_1	q_1	g_2	q_2
g_1	$\sharp$	$\|\|$	$\|\|$	$\diamond$	$\sharp$	$\leftarrow$
m_1	$\|\|$	$\sharp$	$\|\|$	$\sharp$	$\rightarrow$	$\sharp$
d_1	$\|\|$	$\|\|$	$\sharp$	$\leftarrow$	$\|\|$	$\rightarrow$
q_1	$\diamond$	$\sharp$	$\rightarrow$	$\sharp$	$\rightarrow$	$\sharp$
g_2	$\sharp$	$\leftarrow$	$\|\|$	$\leftarrow$	$\sharp$	$\diamond$
q_2	$\rightarrow$	$\sharp$	$\leftarrow$	$\sharp$	$\diamond$	$\sharp$

α3) The places created from the $\diamond$ and $>$ relationships are: (g_1,q_1), (q_1,g_1), (m_1,g_2), (d_1,q_2), (q_1,d_1), (q_1,g_2), (g_2,q_2), (q_2,g_1), and (q_2,g_2)

α4) Through combining, the places become (g_1,q_1), (d_1,q_2), $(\{m_1,q_1,q_2\},g_2)$, $(\{q_1, q_2\},g_1)$, (q_1,d_1), and (g_2,q_2)

α5) The initial place, *input*, is added with initial marking of 1 and connected to the initial events g_1 and g_2; the final place, *output*, is added and has connections from the final events d_1 and q_2

The resulting Petri net model is illustrated in Figure 4.8. This model does not reflect the system's behavior, as it cannot even produce the event streams from which it was created. As mentioned in Section 4.3, the sound SWF-nets produced by the α+ algorithm assume a single input place and a single output place which means that the process will occur exactly once and then terminate. In the example system, however, multiple parts may be processed and those processes interwoven (i.e. a part 1 being machined while a part 2 is picked up). Also this algorithm cannot use any prior knowledge of the system such as information about the resources. Thus, the α+ algorithm cannot be directly used to generate models for this type of system.

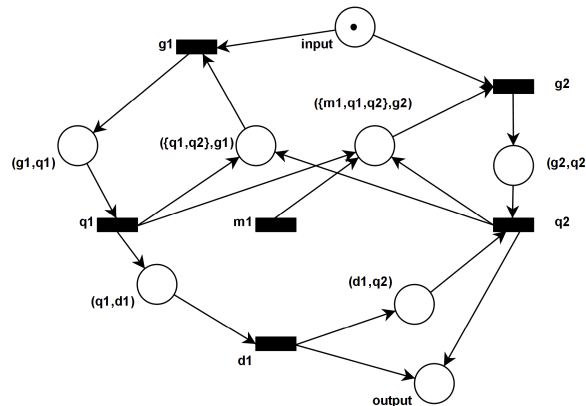


Figure 4.8: Model result from applying α+ algorithm to example system

4.5.3 Model Generation Algorithm

Given the information from the set-up in Section 4.5.1, the model generation solution produces a set of $STPR$s that approximate the system's behavior, along with a set of event statistics about the event pairs that appear in the event streams used as input. The event statistics can later be used to determine how certain a particular relationship in a model is, i.e. whether a behavior has been observed many times or only a few times when deciding which places to create.

The model generation, described in Algorithm 1, uses Steps $\alpha 2$–$\alpha 4$ of the $\alpha+$ algorithm. In Step M1, we determine the event relationships (Step $\alpha 2$), as given by Definition 21. In Steps M2 and M3, we create a set of models for each process. If the process is a $TPCR$, then a single $TPCR$ model is made in Step M2 using just the information about acquiring and releasing resources. Otherwise, a set of TPR models is made for the process in Step M3, which is the bulk of the algorithm.

In Step M3.a, the event relationships are determined (using Step $\alpha 2$) for the events in the process. These relationships may be different than those from Step M1 because the event log is projected onto the process' event set. Because of this, alternate process event relationships are determined in Step M3.b that take into account the relationships from Step M1. Steps $\alpha 3$ and $\alpha 4$ of the $\alpha+$ algorithm are used to create a set of models – one based on the relationships from M3.a (the basic model) and, if there are alternate relationships from M3.b, one or more based on those (some variations) – in Step M3.c.

In Step M3.d, variations of the models from Step M3.c are made where the event relationships due to resources are subtracted from the event relationships based on the log. Using only the information about which events acquire and release each resource, a set of event relationships that abide by Definition 21 can be created.

Model Generation Algorithm:

Given: information stated available in Problem 1

- M1) Find relationships among events in whole process event log (α2)
- M2) For each *TPCR* process, create *TPCR* model based on resource information
- M3) For each *TPR* process,
  - M3.a) Determine the process-specific event relationships (α2)
  - M3.b) Find alternate process-specific event relationships based on the whole process event relationships
  - M3.c) Create all models (α3- α4) possible from the original and alternate process-specific relationships
  - M3.d) Create versions of the models (α3- α4) thus far where the event relationships due to resource use have been subtracted
  - M3.e) Find implicit relationships, create models (α3- α4) with them
  - M3.f) For each process model, add the resource information to model
  - M3.g) For each process model, determine whether any decisions in process are made based solely on resources, and if so, add use of negated resources and their associated connections
  - M3.h) Check whether each process model is a *TPR*
- M4) Create all possible whole models from combinations of the process models (*TPR*s and *TPCR*s)
- M5) Combine the resource places of the processes so that the resources are properly shared; makes them *STPR*s

Outputs: set of deterministic whole models (*STPR*s), frequency statistics of the event pairs used to determine the relationships among events

Algorithm 1: Model Generation Algorithm Including Its Use of the $\alpha+$ Algorithm

These relationships due to the resources can then be subtracted from the event relationships based on the log, using Definition 22, to create other variations of the the event relationships.

Definition 22 (Subtracting Event Relationships)**.** Given two events, a and $b \in T$, that occur in the event log Σ, and two different event relationships have been determined for this pair, then subtracting one relationship from the other is defined as:

- Given $\rightarrow$ and subtract $\rightarrow$ results in $\sharp$
- Given $\leftarrow$ and subtract $\leftarrow$ results in $\sharp$
- Given $||$ and subtract $\rightarrow$ results in $\leftarrow$
- Given $||$ and subtract $\leftarrow$ results in $\rightarrow$
- Given $||$ and subtract $||$ results in

  - $\sharp$ if both a and b have causal relationships
  - $\sharp$ if both a and b are beginning events (a, $b \in T_B$)
  - $\sharp$ if a has a causal relationship and b is a beginning event, or vice versa
  - $\rightarrow$ if b does not have a causal relationship nor is a beginning event but a either has a causal relationship or is a beginning event, $\leftarrow$ if vice versa
  - $\diamond$ if neither a nor b have causal relationships nor are beginning events
- For any other subtraction combination, the result is the same as the original relationship.

Most of this subtraction definition is intuitive. For example, if event log has $a||b$ then there are occurrences of ab and ba, but if the resources have $a \rightarrow b$ then they could account for the ab occurrences, leaving only the ba occurrences which means that $a \leftarrow b$ (or equivalently, $b \rightarrow a$). When subtracting $||$ from $||$, the result depends on whether the two events involved have causal relationships and are beginning events ($e \in T_B$) because each event that is not a beginning event should have at least one other event that causes it. The interaction of resources with events can both hide possibly correct causal relationships as well as suggest possibly incorrect causal relationships which is why this model variation is created.

In Step M3.e, all implicit relationships whose inclusion would reduce the number of places in the models made thus far are found, and models are made that include these relationships. An implicit relationship is one that does not affect the system behavior. For example, if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$ is an implicit relationship. Additionally, if there were two places (a, b) and $(d, \{b, c\})$ and there was the implicit relationship $a \rightarrow c$, then the two places could be combined into a single place $(\{a, d\}, \{b, c\})$ if $a \sharp d$.

Once this set of models is created for a process, additional modifications are made to each model. The resources are added to each model in Step M3.f – the places that represent each resource and the arcs that connect them with the events that acquire and release them. With the resource places added, the algorithm considers whether any of the resource places supersede existing places, as given in Definition 23.

Definition 23 (Superseding place)**.** Given two places, p_1 and p_2, in an $STPR$ N, p_1 is said to *supersede* p_2 if $\bullet p_2 \subset \bullet p_1$ and $p_2 \bullet \subseteq p_1 \bullet$, or $\bullet p_2 \subseteq \bullet p_1$ and $p_2 \bullet \subset p_1 \bullet$.

If a resource place p_R supersedes an existing place p in the model, then the existing place p is removed from the model. In Step M3.g, the algorithm determines whether any decisions about which event to execute are made based solely on resource availability, and if there are such decisions, adds the necessary negated resource place and connections to make this decision. This step is accomplished by determining the resources used at each point in the event streams, and checking if there is a resource whose availability always correlates to making a particular decision between conflicting events (one place leading to more than one event). If there is such a resource, then the negation of this resource is added, along with arcs both to and from the event that should execute when the resource is unavailable. If removing an existing place because it is superseded causes a non-source event ($e \notin T_B$) to not have any causal relationships, then variations of the model are created where each causal relationship that was removed for that event due to superseding is included. Finally, Step M3.h checks whether the model is a TPR. One common reason the model may not be a TPR is that it does not have a process place for each intermediate transition (event), which makes the underlying TP not strongly connected. In most cases, if the process does not create a resource, at least one or more of the models created is a TPR. All of the models, TPR and not TPR, are used in the remainder of the

algorithm.

After making the process models, whole models are created from making all possible combinations of the process models (TPRs and $TPCR$s) in Step M4. For each whole model, the resource places of the processes are combined so that the processes properly share the resources, making each whole model a $STPR$ in Step M5. This model generation algorithm is implemented as a set of Matlab programs.

The manufacturing cell example is again used, this time to illustrate our model generation algorithm.

- Step M1: The event relationships from Σ are summarized in Table 4.6.
- Step M2: Skipped because both Processes 1 and 2 are TPRs, not $TPCR$s
- Step M3: A set of TPRs is created for each Process, 1 and 2, which we will illustrate for Process 1.
- Step M3.a: The process-specific relationships for Process 1 are the same as the shaded part of Table 4.6.
- Step M3.b: Because the process-specific relationships are the same as those for the whole process, there are no alternate process-specific relationship.
- Step M3.c: One process model is created that has three places, each of which connect a pair of events – (g_1, q_1), (q_1, g_1) and (q_1, d_1).
- Step M3.d: The relationships due to the resources used by this process (R1 and R2) are summarized in Table 4.7. For example, because R1 is released by m_1 and acquired by g_1 and R2 does not interact with either of these events, the relationship between the resources m_1 and g_1 is $m_1 \rightarrow g_1$. Subtracting the resource relationships from the relationships based on the event log results in the set of relationships shown in Table 4.8. These relationships produce

another version of the Process 1 model, which has two places: $(g_1, \{m_1, q_1\})$ and $(\{m_1, q_1\}, d_1)$.

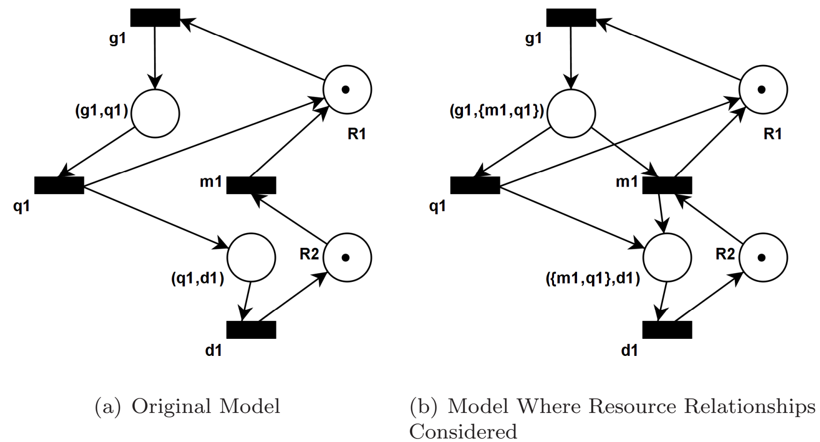
Table 4.7: Ordering Relations Due to Resources for Event Pairs in Process 1

	g_1	m_1	d_1	q_1
g_1	$\sharp$	$\leftarrow$	$\sharp$	$\leftarrow$
m_1	$\rightarrow$	$\sharp$	$\leftarrow$	$\sharp$
d_1	$\sharp$	$\rightarrow$	$\sharp$	$\sharp$
q_1	$\rightarrow$	$\sharp$	$\sharp$	$\sharp$

Table 4.8: Relationships from Event Log Minus Relationships Due to Resources for Process 1

	g_1	m_1	d_1	q_1
g_1	$\sharp$	$\rightarrow$	$\|\|$	$\rightarrow$
m_1	$\leftarrow$	$\sharp$	$\rightarrow$	$\sharp$
d_1	$\|\|$	$\leftarrow$	$\sharp$	$\leftarrow$
q_1	$\sharp$	$\sharp$	$\rightarrow$	$\sharp$

- Step M3.e: No implicit places are found that reduce the number of places in either Process 1 model.
- Step M3.f: The resource information for R1 and R2 from Table 4.2 is added to each of the Process 1 models, resulting in the two models in Figure 4.9.

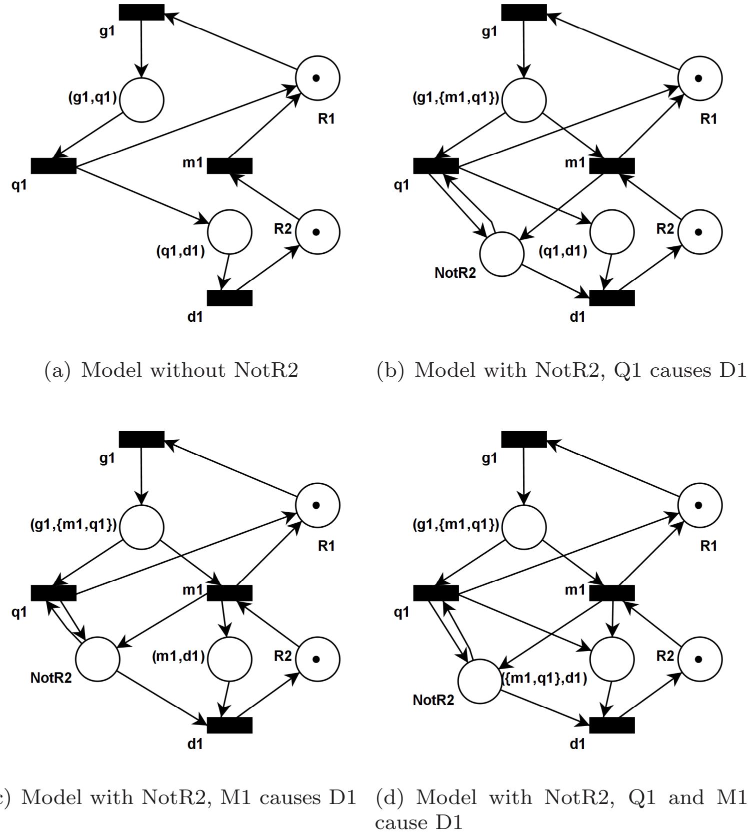


(a) Original Model (b) Model Where Resource Relationships Considered

Figure 4.9: Models Generated for Process 1 Before Decisions Added

- Step M3.g: The first Process 1 model (Figure 4.9 (a)) does not have any decisions made based on resource availability, but the second model (Figure 4.9 (b)) does

– if R2 is available when g_1 has occurred, then m_1 will occur, otherwise q_1 will occur. In words, if the machine is available to process a $C1$, then it will process it, otherwise it will put $C1$ in the queue. Thus the negated resource, NotR2, is included with this model. The NotR2 resource place supersedes one of the existing places in Figure 4.9 (b), which means all causal relationships for d_1 are removed. Thus, variations of this model are created that include one of the removed possible causal relationships: $q_1 \rightarrow d_1$, $m_1 \rightarrow d_1$, and $\{q_1, m_1\} \rightarrow d_1$. These four Process 1 models (the first model and the three variations of the second model) are shown in Figure 4.10. The first model is not a TPR, because the associated sub-net TP is not strongly connected since m_1 is only connected to resource places. The other three models are TPRs.



(a) Model without NotR2

(b) Model with NotR2, Q1 causes D1

(c) Model with NotR2, M1 causes D1

(d) Model with NotR2, Q1 and M1 cause D1

Figure 4.10: Models Generated for Process 1

- Step M2 for Process 2: This same procedure is repeated for Process 2, which results in a single model, shown in Figure 4.6 where the place is (g_2, q_2) because we are considering the manufacturing cell, not the assembly cell.
- Step M4: The resulting models of the entire process are created, where an example of these, made by combining the third Process 1 model with the sole Process 2 model, is shown in Figure 4.7.
- Step M5: The shared resource places of the process models (in this case only R1) are combined in each whole model in Step 4, resulting in the final set of whole models.

4.5.4 Theory

Under ideal conditions, a model generation algorithm should create at least one model that is identical to the underlying model that produced the event data.

Creating an $STPR$ through Algorithm 1 is a deterministic procedure, and thus will be the same as the underlying model if the underlying model is a $STPR$ and if the process models from which the $STPR$ is created, TPRs and/or $TPCR$s, are the same as the underlying process models. Each $TPCR$ model is created based solely on the resource information, and each TPR is created from adding the resources to a TP, so if the resource information is correct and the underlying TP is also correct, then each of the process models created (TPR or $TPCR$) will be the same as the underlying process models. Thus, if the resource information provided is correct, then the only thing that needs to hold for the $STPR$ created to be the same as the underlying $STPR$ model is for our model generation algorithm to create the correct TPs.

In Steps M3.a - M3.e of Algorithm 1, a set of models are created using the $\alpha+$ al-

gorithm, where some variations are made on the event ordering relationships (Steps M3.b, M3.d-e) so that several models may be created. With some additional requirements, we give a theorem for a case in which the TP model is guaranteed to be re-created. First we define a loop-complete event log, based on the loop-complete workflow log (Definition 3.1) of [16].

Definition 24 (Proper event log, complete event log, loop-complete event log)**.** Let $N = (P, T, F, M_0)$ be a TP. Σ is a *proper event log* of N if and only if $\Sigma \in T^*$, every stream $\sigma \in \Sigma$ is a firing sequence of N that starts and ends in N's initial state (marking) M_0. Σ is a *complete event log* of N if for any proper event log Σ' of N, Σ''s event relationships are a (possibly proper) subset of Σ's event relationships, and for any $e \in \Sigma$ there is a $\sigma \in \Sigma$ such that $e \in \sigma$. Σ is a *loop-complete event log* of N if and only if it is a complete event log and for any two-event loops that are possible, then there is a $\sigma \in \Sigma$ such that σ shows that two-event loop.

Using this definition, we can give our theorem and proof.

Theorem IV.1 (Re-creation of TP)**.** *Let $N = \{P, T, F, M_0\}$ be a TP, Σ be a loop-complete event log of N, and N' be a modified version of N: $N' = \{P', T, F', M_0'\}$ where $P' = \{P \cup p_{input} \cup p_{output}\}$, $F' = \{F, \{(p_{input}, t_b)\ \forall t_b \in T_B\}, \{(t_e, p_{output})\ \forall t_e \in T_E\}\}$ and $M_0 = [M_0 10]^T$. If N' is*

1. *safe – for all reachable markings M of N', each place has no more than one token*
2. *live – for every reachable marking M of N' and transition $e \in T$, there is a state M_e reachable from M such that e is enabled*
3. *has no transitions that are connected by multiple places in the same direction i.e. $\nexists e_1, e_2 \in T$ such that $| \bullet e_1 \cap e_2 \bullet | > 1$*

then the model generated in Steps M3.a and M3.c of Algorithm 1 is identical to N.

Proof. A similar result for sound SWF-nets is given in [16], where Theorem 3.7 states "Let $N = (P, T, F)$ be a sound one-loop-free SWF-net and let W be a loop-complete workflow log of N. Then α(W) = N modulo renaming of places." This theorem relies on Theorem 3.6, also from [16], as well as a number of theorems and results from [60]: Theorems 4.1, 4.5, 4.6, 4.8, and 4.10 and Property 4.4. A comparison of sound SWF-nets and TPs is provided in Table 4.4. All of the properties of sound SWF-nets and the event log used in these theorems and their proofs are either already required for TPs or are explicitly required as additional conditions for N' in this theorem. □

Given these restrictions, the TP can be properly re-created. The variations of the TP created in our model generation algorithm are made so that when some of these restrictions are relaxed, such as not having all of the event streams to make up a loop-complete event log, one of these variations may still be the correct underlying TP.

4.6 Performance Assessment

The next step after model generation is performance assessment. To determine the models' performance, a performance assessment algorithm is used which determines and updates the performance but does not change the models themselves (Algorithm 2). The inputs to this algorithm are the whole models, a labeled event stream that has not only an overall label but also a label for each event in the stream, and the current performance of the models. If each event in a stream is labeled as no-fault, then the entire stream is labeled no-fault, and if at least one event in a stream is labeled fault, then the entire stream is labeled fault. For each model, the algorithm determines

whether the model accepts each event in the event stream based on whether the event could occur in the model in its current state. If a model accepts an event, then the model's state is updated to indicate that event has occurred and the event is labeled normal, whereas if the model does not accept the event, then its state is not changed and the event is labeled anomalous. The algorithm compares the label assigned by the model with the event label. For each event, if the model labeled it correctly (no-fault as normal or fault as anomalous), then the model's performance is updated to be better, and if incorrectly, then its performance is updated to be worse. In the current implementation of the algorithm, a correct labeling of an event adds a value of 1 to the model's performance and an incorrect labeling subtracts 1. Thus, performance of a model is the overall sum of the number of events correctly labeled minus the sum of the number of events incorrectly labeled. This performance can be normalized to percent of events labeled correctly.

This type of performance assessment is typical for supervised learning, where some data is used to develop the learning algorithm (in this case, the models) and other data is used to assess its performance [56]. The particular metric used here, adding one for correct labels and subtracting one for incorrect labels, is sufficient for this anomaly detection solution, but that does not mean it is the best metric when the solution is applied to industry systems. Other metrics that could be used for such performance assessment include the percentage of event streams or events correctly classified [56], and the squared or absolute error between the given classification and the learned classification of events [29]. The output of the algorithm is the model's updated performance using our simple metric.

This performance assessment is illustrated through the example manufacturing cell. Using the models produced in Section 4.5, and the two event logs for per-

Performance Assessment Algorithm:
Given: information available for Problem 2, current performance of whole models (optional, can be null)
For each whole model
    For each event in labeled event stream
        Determine whether the model accepts the event, and if agrees with event label
        If the model and label agree, increase the model's performance by 1
        Else, decrease the model's performance by 1
Outputs: updated performance of whole models

Algorithm 2: Performance Assessment Algorithm

formance assessment Σ_2 and Σ_3, from Section 4.5.1, the performance assessment algorithm was run. Each event stream, $\Sigma_2 = \sigma_3$ and $\Sigma_3 = \sigma_4$, had 20 events meaning the best possible performance would be 40 (all labeled correctly) and the worst would be -40 (all labeled incorrectly). The first two models had a performance of 8, which translates to labeling 60% of the events correctly, and the last two models had a performance of 40, which means they labeled 100% of the events correctly.

4.7 Anomaly Detection

Once the models have been generated and their performance determined, the anomaly detection algorithm can be run (Algorithm 3) to label an unlabeled stream. For each event in the unlabeled stream, this algorithm adds the performances of the models that label the event as normal and subtracts the performances of the models that label it anomalous, and if this value is greater than a threshold (currently using 0), then the event is labeled as normal, otherwise it is labeled anomalous. The models' states are updated in the same way as in Algorithm 2. The output of the algorithm is a label of normal or anomalous for each event in the stream, where an anomalous label indicates that an event is anomalous. A stream is labeled as normal only if none of the events in the stream are labeled as anomalous, otherwise the stream is labeled anomalous. The idea of comparing observed behavior to a model to determine possible faults has been used previously [47] [30], but these

approaches have used one model known to be correct, rather than a set of models whose performance has been determined.

By incorporating performance assessment, the anomaly detection algorithm can make improvements online. If a new set of event streams labeled as no-fault becomes available, Algorithm 2 can be run again using those streams to update the models' performance and thus lead to better anomaly detection. If all of the models have poor performance on these new labeled event streams, then an offline improvement to the anomaly detection can be made by running the model generation algorithm again but this time including all of the new, no-fault event streams. Thus the new models generated will better incorporate the behavior represented by these new event streams.

Anomaly Detection Algorithm:

Given: information available for Problem 2, performance of whole models

1) For each event in unlabeled stream
   a. Determine whether each model accepts this event, and if a model accepts this event, then update the model's marking to reflect the event occurring
   b. Sum the performance of the models that accept the event and subtract the performance of the models that do not; if the result is greater than 0, label the event "normal" and otherwise label it "anomalous"
2) The entire stream is labeled "normal" if all of the events in it are labeled "normal"; otherwise it is labeled "anomalous"

Outputs: whole models with updated marking, label for event stream and each event in it

Algorithm 3: Anomaly Detection Algorithm

Returning to the example manufacturing cell, the anomaly detection algorithm was run on the generated models with their performances for an unlabeled event stream that had not been introduced before,

$$\sigma_5 = g_1 m_1 g_2 d_1 q_2 g_1 m_1 g_1 q_1 g_2 d_1 q_2 g_1 m_1 g_2 q_2 g_1 d_1 m_1 g_1$$

that was no-fault behavior. The weighted voting of the models correctly identified the entire stream as normal. This stream was then modified to include some fault behavior and run through the anomaly detection algorithm to see if it could detect

the anomalies associated with the fault. The stream was changed so that the second to last event was switched from m_1 to q_1, which in this sequence would mean that a part 1 was put in the queue in spite of R2 being available, and thus fault behavior. The results from the anomaly detection algorithm indicated that all events were normal except the one changed, which was anomalous, thus correctly finding the fault. Another anomalous stream was created by taking the initial stream and deleting the first occurrence of d_1. In this case, the anomaly detection algorithm results identified the 6th and 7th events in that stream (m_1 and g_1, which were the 7th and 8th in the original) as anomalous. Although these events were not actually the problem, the anomaly was manifested there since d_1 would represent the end of processing a part 1 and m_1 should not be able to occur again until d_1 has occurred. The anomaly labeling disappeared after these two events because then an entire cycle (g_1, m_1, and d_1) of processing a part 1 had been dropped and the system seemed normal again. This result indicates that the anomalous events found by our algorithm may not always be the actual anomaly or fault.

4.8 Application of Solution to Simulated RFT Cell

In this section, model generation and performance assessment are further illustrated with a larger example. This example system is an abstracted version of a cell of the Reconfigurable Factory Testbed (RFT), the system that motivated this research. There are two types of parts, called part 1 and part 2. The abstracted version of this cell has the same basic behavior as the actual cell but with only 20 events.

The process that occurs in the machining cell can be divided into three processes: 1) process part 1, 2) process part 2, and 3) arrival/release of part 0 (empty pallet).

Table 4.9: Unique Models Generated from Algorithm for Given Log

Logs	Part 1 Models	Part 2 Models	Part 0 Models	Whole Models
Σ_1 = One 10000 event stream	4	4	1	16
Σ_2 = One 500 event stream	6	6	1	36
Σ_3 = Five 100 event streams	8	10	1	80

Each of the first two processes has 9 events and 4 resources – robot, CNC, pallet stop, and empty pallet. The third process has 2 events and 2 resources – pallet stop and empty pallet. The robot, CNC, and pallet stop are type A resources, while the empty pallet is a type C resource. The pallet stop and empty pallet resources are shared among all three processes, while the robot is shared between the first two processes, and each of the CNCs is used only by a single process.

The event log Σ for model generation of this example was created from simulating a model of the abstracted cell. Three different model generation logs were created – Σ_1 has a single event stream with 10000 events, Σ_2 has a single event stream with 500 events, and Σ_3 has 5 streams each with 100 events for a total of 500 events. The event logs (both no-fault and fault streams, Σ_4 and Σ_5 respectively) used for performance assessment were created to specifically highlight the variety of behavior allowed by the cell. Most of these event streams are short, on average only 6 events, because each stream illustrates one particular behavior of the cell. These event logs have a total of 10 event streams, 5 of which are no-fault (in Σ_4) and 5 of which are fault (in Σ_5), with 60 events among the 10 streams.

The model generation algorithm was run for the machining cell for each of its three event logs. Each run of the algorithm results in a set of whole models – a Part 1 model, a Part 2 model, and a Part 0 model – that interact through their shared resources. The results of these runs are summarized in Table 4.9.

For a given model generation event log – Σ_1, Σ_2 or Σ_3 – approximately the same

number of models are generated for the Part 1 process as for the Part 2 process, which makes sense because the Part 1 and Part 2 processes are identical other than naming. The event log with a set of short streams, Σ_3, yields more models than either log with one long stream, Σ_1 or Σ_2, because the event relationships are not as well known and thus more model variations are created. Only one Part 0 model is generated for each event log because the Part 0 process is a $TPCR$. An example of one of the whole models generated using both Σ_1 and Σ_2 is illustrated in Figure 4.11.

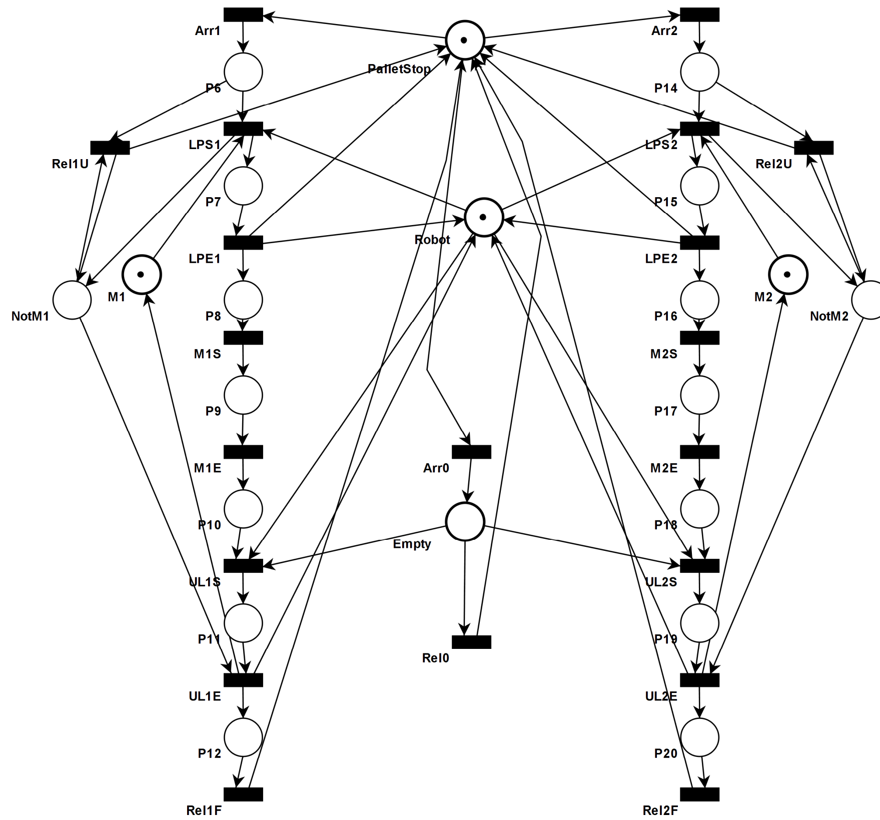


Figure 4.11: Example $STPR$ for Cell 1 (composed of TPR for Part 1, TPR for Part 2, and $TPCR$ for Empty) with resource places Pallet Stop, Robot, Empty, M1, M2, NotM1, and NotM2.

The models' performance is assessed using Σ_4 and Σ_5, and the normalized results of that assessment are summarized in Table 4.10. For each stream, the number of events in it is listed. Of the models generated for each event log, the maximum,

minimum, and mean performance for each event stream is given in percentage correct. In general, the performance of the models generated from one longer event stream, Σ_1 and Σ_2, is better than that from the set of short event streams, Σ_3. The best maximum performance is the total number of events in the two training logs, Σ_4 and Σ_5, which is 100%. Perfect performance was achieved by at least one model generated from Σ_1 and at least one model generated from Σ_2, but not by any models generated from Σ_3. These results illustrate that an event log with a single long stream, such as Σ_2, may perform better than an event log with the same number of events split into more streams, such as Σ_3. We observed that even though Σ_1 has more events than Σ_2, the models generated based on Σ_1 do not yield better mean performance than those based on Σ_2. The max performance for the models created based on Σ_1 and Σ_2, however, are the same and because anomaly detection is based on the models' labeling but weighted by their performance, the max performance of a set of models can be more important than the mean performance. The $STPR$ in Figure 4.11 is the one with the best performance and exactly matches the model that was used to simulate the abstracted cell and create the event logs used for model generation.

Table 4.10: Performance Results in Percentage for Each Model Generation Event Log, Where Num is the number of events in the stream and the results are expressed as Max, Min, and Mean

	σ	All	1	2	3	4	5	6	7	8	9	10
	Num	60	9	5	10	2	6	6	3	2	6	11
Σ_1 (16 models)	Max	100	100	100	100	100	100	100	100	100	100	100
	Min	13	11	0	10	50	17	17	0	50	17	9
	Mean	55	47	50	55	75	58	58	50	75	58	55
Σ_2 (36 models)	Max	100	100	100	100	100	100	100	100	100	100	100
	Min	13	11	0	10	50	17	17	0	50	17	9
	Mean	63	59	63	62	83	73	56	50	83	73	59
Σ_3 (80 models)	Max	87	56	100	90	100	83	100	100	100	83	91
	Min	13	11	0	10	50	17	17	0	50	17	9
	Mean	35	17	12	38	58	25	65	60	58	25	39

To see how well the anomaly detection solution scales with system size, the solution was applied to a simulated system similar to the previous example, but much

larger in size. This system consists of five cells, each like the cell in the previous example, which collectively machine six parts that are assembled into two final products in a final assembly cell. There are 17 processes, one for each part in each machining cell ($3 * 5 = 15$) plus one for each final assembly (2). The system has 104 events and 37 resources. A simulated model of this example was used to generate a set of 20 no-fault streams each with 5000 events. These streams were used for model generation.

The model generation algorithm was able to create models for each process, where the models for the machining cells were similar to those produced for the previous example. Combining these process models into all possible whole models, however, was not possible because it resulted in too many whole models. Each process had between 1 and 12 models, and all possible combinations would yield more than 200 million models.

This problem is due in part to the number of models per process, and in part due to the number of processes. One means of reducing the number of models per process, and thus the number of whole models, is to do performance assessment of the models of a given process and discard all those that perform below a certain level. The idea behind this approach is that if a process model performs very poorly at predicting just the process' behavior, then it will also contribute to poor performance of a whole model.

4.9 Conclusions

Taking the approach of handling faults once they occur, this research developed an anomaly detection solution for event-based systems without pre-existing formal models. For the model generation step, an algorithm was created that builds upon

the $\alpha+$ algorithm [16] to create a set of Petri net models that explicitly incorporate resources to describe the underlying no-fault behavior of the system. These models' performance in detecting anomalies is assessed using labeled event streams, and anomaly detection is performed on unlabeled event streams on the basis of these models and their previous performance. A variation of Petri nets that incorporate resources in a less rigid manner, called $STPR$s, was developed for use with the model generation algorithm. The applicability of the solution was demonstrated on a simulated version of the motivating RFT example.

CHAPTER V

Application of Anomaly Detection to Industrial Manufacturing Line

The anomaly detection solution from Chapter IV was designed to be applicable to industrial event-based systems, which is why it assumes there is no pre-existing formal model and only limited information available about the system. Thus, a significant test of the applicability of the anomaly detection solution was to apply it to a real system, in this case an industrial machining cell used by Ford Motor Co. The cell is described in Section 5.1, and in Section 5.2, we describe a number of inconsistencies that were found between the assumptions that were made in the solution approach and the realities that we found on the plant floor. These inconsistencies are presented, along with their resolutions, in Section 5.3. A couple of persistent barriers prevented completion of the industrial application (Section 5.4), so the resolutions of these inconsistencies have been tested through application to simulated systems that have been created to mimic the industry conditions (see Chapter VI). The work presented in this chapter appeared in [2].

5.1 Description of Machining Cell

The Ford machining cell under investigation takes partially machined parts and performs further machining on them. Illustrated in Figure 5.1, the cell has two

gantries (G1-G2) that operate in serial and six CNCs (M1-M6) that operate in parallel. Parts arrive at the entry (point 1), where the first gantry (G1) picks them up one at a time and puts two parts at the hand-off location (point 2). Once two parts are available at the hand-off location, the second gantry (G2) picks up the two parts together and, if necessary, waits until a CNC (M1-M6) has finished processing its parts and requests an unload. Then the second gantry (G2) moves to that CNC, unloads the processed pair of parts and loads its new unprocessed parts. While G2 departs to take the pair of processed parts to the exit (point 3) if they are good, or to the reject station (point 4) if they are not, the CNC begins to process its new pair of parts. This completes one cycle of the process.

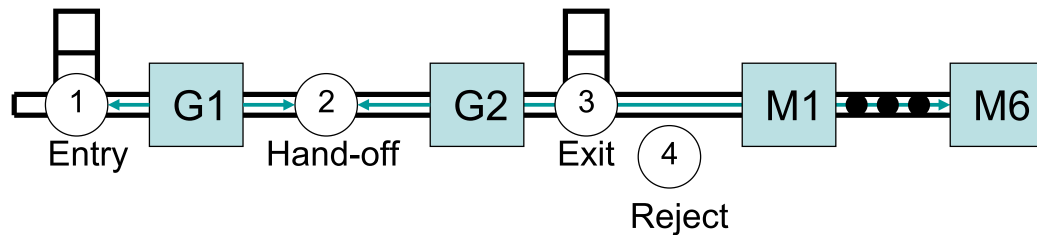


Figure 5.1: Machining cell that consists of two gantries (G1, G2) operating in serial and six CNCs (M1-M6) operating in parallel

The machining cell collects data from each of its eight machines (two gantries and six CNCs). The data collection is illustrated in Figure 5.2, where there is a PLC for each machine that sends data to the IT system. Each PLC has driving logic written by the machine supplier that provides some of its status information to a special function block (FB) designed by Ford to standardize how machines interact with the data collection system. At each point where data is passed, it is filtered and processed.

Each PLC can report up to 40 words of data, although only the first 20 of them are currently used. This data is reported to IT only when certain key bit changes

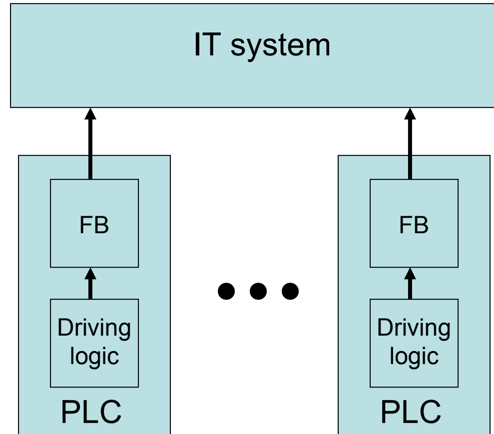


Figure 5.2: Data collection set-up for machining cell

occur. Some words are used for numbers, such as the last cycle time, part ID, and transaction/message counter. Other words are broken down into individual bits. Words 4 and 5 have bits that express the machine's status, such as the start and end of the cycle (Cycle End, Cycle Start), the type of cycle (Good, Bad, Non-Continuous, Dry), whether there is currently any abnormal status (Manual Intervention, Bypass, In Fault, etc.), and whether it is interacting with another machine (Wait Aux), starved, or blocked. Words 14-20 are also split into bits that express fault and warning codes, where these codes are specific to the particular type of machine. Some of the most important bits for understanding normal operation are Cycle End, Wait Aux, Blocked and Starved. Cycle End is generally pulsed high when a machine has completed one cycle, whether that is a CNC having finished processing a pair of parts or G2 dropping off a pair of parts at the exit or reject. Wait Aux is generally high for a machine when it is waiting for another machine, such as when G1 is waiting to drop off a part at the hand-off location because G2 is currently there. This bit is also used for some secondary purposes, such as during routine tests to check the machine's functionality. Blocked and Starved are used when a machine cannot perform further operation because it is prevented by the operation just downstream

or just upstream. For example, a CNC is blocked when it is done processing parts and has to wait for the finished parts to be unloaded, and it is starved when it is waiting for raw parts to process.

In summary, the information known about this machining cell is its physical set-up, including its machines, its data structure, and design information about how the data bits are supposed to be used. Information not available for this cell includes a formal DES model of its correct operation and a complete log of all possible no-fault event streams.

This machining cell generally operates correctly. Occasionally and unpredictably, however, the second gantry (G2) will wait holding unprocessed parts for a minute or two as if there are no CNCs available to unload and reload, even though one or more CNCs are actually Blocked, indicating that they are ready to unload and reload. Eventually, G2 will move to one of the waiting CNCs and the cell will resume operation, but it is not clear what is causing this fault nor why it resolves itself. The fault needs to be debugged so that the underlying issue can be fixed and production time will no longer be lost.

5.2 Initial Application of Solution

To apply the anomaly detection solution requires event streams (data) observed from the cell. A data set from this cell includes the data messages sent by each machine in the cell to the IT system over some period of time. Two such data sets were provided for this application. The first data set was approximately 6.5 days long, during which time there were over 190,000 messages from the machines to the IT system and more than 9600 parts were processed through the cell. The second data set was 2 days long, during which time there were approximately 83,000

messages and over 2000 parts processed.

Prior to applying this solution, information known about the cell must be interpreted to satisfy the information that the solution requires about the system. More than one interpretation is possible for some of the information. The cell's entire behavior could be considered as a one process, or there could be one process per CNC and one process for both gantries, or one process for each of the eight machines. The resources for the cell are the two gantries six CNCs, and the buffer.

Because the problem with the cell is in the interaction among the machines, the bits that should be included as events are some of those from Words 4 and 5, which relate to the machine's status. The most important bits to include in a model for this problem are Cycle End, Good Cycle, Wait Aux, Starved, and Blocked. The association between events and processes is evident due to each event being a bit change in a machine's PLC, so each event is associated with the process to which its machine belongs.

The last set of information to determine for the cell is which events are associated with acquiring and releasing each resource. A description of physical events associated with acquiring and releasing each resource is given in Table 5.1, where BF is the buffer between G1 and G2, and only one CNC is listed because all six operate the same way. The data events in this table will be discussed in Section 5.3.1. G1 is acquired when it goes to pick up a raw part and released when it finishes moving that part. G2 is used for two different purposes – to load raw parts and to unload processed parts. In loading raw parts, G2 is acquired when it picks up a pair of raw parts and released when it finishes loading those parts. In unloading processed parts, G2 is acquired when it begins to unload the pair of parts and released when it drops off the pair at the Exit or Reject locations. To apply the solution, these physical

events that acquire and release the resources should be associated with particular bit changes in the data streams. However, we found that was not the case.

Table 5.1: Physical and Data Events That Acquire and Release Cell's Resources With Unobservable Events in *italics*

Res	Type	Events That Acquire	Events That Release
G1	Physical	*Starts picking up part from Entry*	*Finishes placing part at Hand-off*
	Data	Starts picking up 3 pairs from Entry (G1 Good Cycle fall)	Finishes placing 3 pairs at Hand-off (G1 Good Cycle rise)
G2	Physical	*Starts picking up pair of parts from Hand-off*	Finishes loading pair into CNC
	Data	Starts interacting with CNC (CNC Wait Aux rise)	(CNC Wait Aux fall)
G2	Physical	Starts unloading pair of parts from CNC	Drops off pair at Exit or Reject
	Data	(CNC Wait Aux rise)	(G1 Cycle End rise)
CNC	Physical	*Starts loading pair of raw parts*	Finishes unloading pair of processed parts
	Data	Raw part begun to be processed (CNC Wait Aux fall)	(CNC Cycle End rise)
BF	Physical	*Receives pair of raw parts from G1*	*Pair of raw parts removed by G2*
	Data	Finishes placing 3 pairs at Hand-off (G1 Good Cycle rise)	Starts interacting with CNC (CNC Wait Aux rise)

This inconsistency between what the solution assumes from an academic viewpoint and what was actually true for this industrial system was only one of several such inconsistencies discovered while applying this anomaly detection solution. Five such inconsistencies were identified that must be resolved to complete the application.

5.3 Inconsistencies Between Academic Assumptions and Industry Realities

Five inconsistencies between academic assumptions made by the anomaly detection solution and realities of this particular industry system were identified. This set of inconsistencies is not all-inclusive, but the academic assumptions and industry situations are both widespread, which means that these inconsistencies are relevant not only for the application of this anomaly detection solution to this machining cell, but for application of other academic solutions to other industry systems. The first

academic assumption, made by any DES that explicitly uses resources, is that the events associated with acquiring and releasing each resource are observable; however, in an industry system these events may not be recorded at the level at which data is collected. Another assumption is that the system produces a string of ordered events, required for any DES, but many industry systems' data is produced by PLCs which may have multiple bits change within a single message. A consistent mapping between events and meaning is assumed for modeling, whereas in any real system there may be inconsistency in how events are used. Workflow mining, introduced in Section 2.1.3, assumes that each event stream starts from the system's initial state, but industry systems often run continuously and, especially if they have any parallelism, may not be in their initial state often. Finally, workflow mining also assumes that separate, labeled (no-fault or fault) event streams are available from the system even though industry systems often run continuously and may not have a means by which to label their streams. These inconsistencies are discussed in detail and ideas for their resolution presented in the subsequent sections.

5.3.1 Observable Events to Acquire/Release Resources

Some types of DES, such as the resource-based Petri nets proposed in [21] and [31] and used in this anomaly detection solution, explicitly include resources in their models. In these types of DES, resources serve an important role in determining what behavior is allowed, and providing a modular structure by having sub-models interact solely through the resources. For these DES, it is assumed that there are specific observable events associated with acquiring and releasing each resource. In industry systems, however, not all such events may be observable. For example, in the machining cell, the second gantry (G2) picks up a pair of unprocessed parts from the Hand-off location, but this physical event does not have a corresponding event

that is recorded in the data. Because the gantry receives a command from its PLC to pick up the parts, the event is registered in the PLC and thus observable at some level of the data, but this event was not originally chosen for upload to the IT system. Another example is that the first gantry (G1) picks up one pair of unprocessed parts at a time and puts them at the Hand-off location, but these physical events only have corresponding events recorded in the data for every third pair, because the capacity of the pallet on which the raw parts arrive is three pairs.

This inconsistency between theory and practice is difficult to address because it is fundamentally about missing information in the form of certain events being unobservable. The ideal way to address this inconsistency is for the desired events to be sent to the IT system and recorded, but this would require changing the PLC programming. If the ideal solution is not possible, then one approximate solution is to find events that can serve as proxy for the unobservable events. A proxy event is one that is used in place of the unobservable event. A good proxy event should always occur shortly before or after the unobservable event, and never at any other times. If the proxy occurs shortly after the unobservable event, that may be preferable because then it is assured that the unobservable event has definitely occurred. Also it is best if the proxy event is not associated with acquiring or releasing any other resources because then the the two resource interactions will be incorrectly associated.

In the machining cell, one of the unobservable events is that for acquiring the G2 resource. The best proxy events available for this unobservable event are when a CNC starts to interact with G2 to unload and load parts. This set of proxy events always occurs after the G2 resource has been acquired (G2 has picked up raw parts) and is not associated with acquiring any other resources. The difficulty with using this set of proxy events is that it can mask the gantry waiting problem with the

machining cell. Because there is no event to directly indicate when G2 has parts and is waiting, one can only know when it has already reached a CNC and by that point, the problem has resolved. Thus, by using this set of proxy events, one only knows when many of the CNCs are blocked, which could indicate the problem but may also occur in other situations too. Thus, the use of proxy events is a less than ideal solution.

In the machining cell, of the five events that acquire resources, four are unobservable, and of the five events that release resources, two are unobservable, as shown in Table 5.1. Based on this information, making such events observable is not standard practice. One recommendation for future design of industry systems would be to make all events that acquire and release resources observable so that problems like the gantry waiting can be more easily found and debugged. Because the use of proxy events was insufficient and this inconsistency was preventing the successful application of the anomaly detection solution to this system, our Ford collaborators were willing to invest the time and effort required to change the PLC programming such that all of the events that acquire and/or release resources are sent to the IT system and recorded.

5.3.2 String of Ordered Events

This anomaly detection solution assumes a string of ordered, isolated events, a common assumption in DES. At first glance, it may seem that this assumption can hold for the machining cell example by making each bit change (rise and fall) a separate event. Between subsequent PLC messages, however, there may be multiple bit changes (MBCs) which makes the correlation between bits and events unclear. An MBC event is one in which multiple bits have changed between one PLC message and the next. Some causes of MBCs may be removed, such as by requiring that every

bit change causes a PLC message to be generated. However, if multiple bits change within one PLC scan, MBCs cannot be prevented.

If MBCs cannot be prevented, one may wonder if they are infrequent enough that they could be treated as noise or errors in the data. How often MBCs occur depends on how the PLC is programmed, but to give an example illustration, the occurrence of MBCs in data from the machining cell application was studied. Of all of the events, 35% were MBC events, indicating that they are very prevalent and cannot be easily dismissed. MBC events also account for the majority of unique events – using 18 bits, there are 35 unique single bit change (SBC) events (note that there would be 18*2=36 SBC events, but one of the bits only reported rising and not falling, thus eliminating one event), but more than 250 unique MBC events.

Possible ways to address MBCs lie along a continuum. At one extreme, each MBC is treated as a unique event, and at the other, each MBC is split into a sequence of SBC events. If each MBC is treated as a unique event, then the number of unique events in the system may be significantly larger, as illustrated with the sample data where the number of unique events would jump from 35 to 285. If each MBC event is split into a sequence of SBC events, it is unclear how the order of the SBC events should be decided and there may be some MBC events that truly represent unique events. For a simple, two bit change example, the possible options for how to handle it are illustrated in Figure 5.3. Instead of choosing either extreme, the choice can be made for each MBC. For example, if the bits in a particular MBC occur sequentially in the data and have a causal relationship, then the MBC can likely be split into SBC events whose order is determined by that causal relationship. If they occur sequentially in either order, then their relationships to the events directly before and directly after the MBC can be considered. Alternatively, if the bits in an MBC

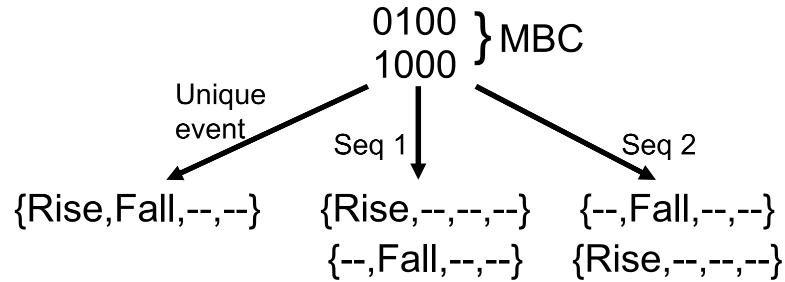


Figure 5.3: An MBC where two bits change per event and the three options are: keep it as a unique event, or split it into one of two possible sequences

never occur sequentially, then the MBC should probably be kept as a unique event. A heuristic decision algorithm was developed to handle MBCs – decide whether to split them, and if so the order of the constituent SBCs. More detail on this algorithm is provided in Chapter VI, where it is applied to a simulated system. In summary, a recommendation to industry would be to have all the most important bits cause a PLC message to be generated whenever they change to reduce the occurrence of MBCs, and a recommendation to academia is to develop algorithms that can handle MBCs.

5.3.3 Consistent Mapping Between Event and Meaning

This anomaly detection solution makes the relatively common academic assumption that the system has a consistent mapping between events and their meaning, based on design documents that dictate how bits should be used. In the example application, though, the PLCs are programmed in a way that results in occasional occurrences of inconsistent mapping in the data, in the form of bits being used inconsistently with the design.

This machining cell's original data had some infrequent instances of such inconsistent mapping. Some instances of this issue in the data are apparent from understanding which physical events are associated with bits. For example, the Cycle

End bit for G2 pulses high each time that G2 places a pair of processed parts at the exit (point 3) or reject pile (point 4). Occasionally in the data, this bit pulses high twice in a row even though it is not physically possible for G2 to drop off a pair of processed parts and then drop off another pair without picking up parts in between. Other instances of inconsistent mapping are due to bits being used inconsistently with their intention, such as the Wait Aux bit being used for occurrences other than interacting with another machine.

The most complete manner of addressing this inconsistency is to find all errors in programming and use of bits other than intended, and to remedy these in the programming and implementation of the PLCs and data collection systems. Another possible way to address this inconsistency is to use knowledge of the system, such as it not being realistic that G2 has Cycle End pulse high twice in a row, to pre-process the data to eliminate the occurrences that do not match the mapping. Both of these approaches were used to address this inconsistency. Some re-programming was done of the PLCs, in part because it was already necessary to resolve the first inconsistency. A simple script was also created that uses knowledge of the system and some of its known inconsistencies to pre-process the data. Using system knowledge, we determined certain patterns in the data that represent inconsistent mapping, such as G2's Cycle End pulsing high twice in a row, or a CNC's Wait Aux rising and falling without Cycle End rising in between. The script pre-processes the data by taking these patterns as inputs, searching the data for the patterns and removing each instance of the pattern. The script can also take replacement patterns as input, if a particular pattern should be replaced rather than removed. The main limitation to both of these approaches is that they can only resolve known issues with the mapping between bit and meaning. Thus if additional mapping issues are found

or are created in the PLC re-programming, they must be identified in order to be remedied.

5.3.4 System Starts in Initial State for Each Event Stream

The anomaly detection solution uses observed event streams to generate models, compares other labeled event streams to these models to assess their performance, and then compares an unlabeled event stream to the models to determine whether it is likely an anomaly. In the second and third of these steps, the solution assumes that each stream of data starts in the system's initial state, which corresponds to the typical workflow mining assumption of starting at the beginning of the process. In the industry machining cell, however, the system is running continuously and has a serial-parallel configuration, both of which contribute to the the system not often being in its initial state. Therefore, only using streams that start from the initial state is not realistic. This inconsistency needed to be resolved from the academic side.

The models generated by our solution do not depend on the observed event streams starting in the system's initial state. The performance assessment and anomaly detection steps, however, require comparing an observed stream to what the models expect, which does require that the stream and models start in the same state. For this comparison, the initial state of the event stream must be known so the model can start in that same state. This state may not be known exactly, but a lower bound for it can be calculated based on the events that occurred in the stream and an upper bound can be calculated based on the shared resources. Thus, a necessary condition for a model to accept an event stream is that this lower bound be less than or equal to the upper bound. The theory for this necessary condition as well as its algorithm implementation are developed in Chapter VI.

5.3.5 Separate, Labeled Event Streams

Workflow mining techniques are incorporated into the model generation portion of the solution, and such techniques assume that separate, labeled event streams of data are available for the system to be modeled, where the labeling indicates whether the stream represents no-fault or fault behavior. The no-fault streams are used to create a set of models of the system, and both no-fault and fault streams are used to assess the models' performance to see how well they match the system behavior. Instead, the machining cell produces a continuous, unlabeled event stream because the system is generally running all the time and has no mechanism for labeling.

The continuous industrial event stream can be split based on different criteria. For this anomaly detection solution, an algorithm was created that implements several splitting mechanisms and they were tried out on the machining cell example. An event stream can be split based on certain key events, or after such events occur a particular number of times. Another possibility is that the stream can be split into streams of a particular size. Trying both approaches for the machining cell data revealed that splitting based on key events makes anomalies more likely to be missed if they tend to occur around that key event, whereas splitting into streams of a particular size provides better results.

These split streams can be labeled by a system expert who can determine whether the stream represents no-fault or fault behavior. Alternatively, if the fault behavior is associated with particular symptoms or patterns in the data, then a program can be written to label the event streams automatically based on whether it finds the given symptoms. With the problem in the machining cell, the fault behavior is when G2 has raw parts and is waiting to service a CNC, while at least one CNC has completed parts that it is waiting to have unloaded and new parts loaded. If this

scenario can be associated with a given combination of bit values (symptoms), then the event streams can be labeled via a program that keeps track of the bit values. This automated labeling based on symptoms was added to the splitting algorithm.

5.4 Barriers to Application to Machining Cell

To complete application of the anomaly detection solution to the machining cell, resolutions were found for the five inconsistencies identified. Some of these inconsistencies were addressed by the Ford engineers through changing logic in the machine controllers, while others were addressed by incorporating additional algorithms into the anomaly detection solution. These resolutions are summarized in Table 5.2 along with the responsible party (**A**cademia or **I**ndustry).

Table 5.2: Inconsistencies and Their Resolutions, Where Responsible Party is Either **A**cademia or **I**ndustry

	Inconsistency	**Resp.**	**Resolution**
1	Events that acquire/release resources vs. not all such events recorded	I	Changed logic to make available
2	Ordered string of events vs. multiple bit changes per message	A	Algorithm created to decide whether split or keep unique
3	Consistent mapping between event and meaning vs. inconsistent mapping	I, A	Changed logic to address some, created algorithm to filter some
4	Event streams start in system's initial state vs. start in a variety of states	A	Determined necessary condition for feasibility of stream, implemented in algorithm
5	Separate, labeled event streams vs. continuous, unlabeled event stream	A	Algorithm created to split streams and label automatically based on symptoms

Resolution of the first and third inconsistencies required logic changes, but the machining cell is running production parts and therefore could not have such changes made. A similar machining cell (a gantry that serves a set of CNCs) was identified on which such logic changes could be made, and after these changes, more data was collected. With this new data and the additional new algorithms in the solution, application of the solution to the data was attempted. There still exist two main barriers to successful completion of this application. The first barrier is that the ma-

chining cells for which the logic changes could be made are only running occasionally for short test runs, and thus the quantity of data is much smaller – fewer than 600 parts. With this small amount of data, there are some two-event sequences that can occur but do not in the data, yielding incorrect models. The second barrier is that some new issues with mapping between event and meaning arose in the new data, in large part because the resolution to this inconsistency only works on a case-by-case basis. Some of the newly found mapping issues also will require logic changes, such as some bits that are not being updated real time and thus causing incorrect event relationships.

5.5 Conclusions

The research on anomaly detection for event-based systems without pre-existing formal models was advanced through the application of the anomaly detection solution to an industrial system – a machining cell at Ford. Five inconsistencies between common academic assumptions made by this solution and industry practice for the machining cell were identified. A resolution to each inconsistency was developed – either a required change to the logic of the machining cell or the creation of algorithms to incorporate into the solution (several of which are discussed in detail in Chapter VI). Further issues were found in the application of the anomaly detection solution, including the inconsistency resolutions, to the Ford machining cell. These inconsistencies can provide insight into the gap between academia and industry and the resolutions give ideas about how to bridge at least part of that gap.

CHAPTER VI

Application of Anomaly Detection to Simulated Systems

Because of the barriers described in Section 5.4, the algorithms developed to address some of the inconsistencies could not be tested on industrial data. As an alternative, some simulated systems were modified to exhibit the inconsistencies that had academic resolutions. First the background on handling multiple bit changes (MBCs) and the decision algorithm developed for them is presented, along with an example application to the small manufacturing cell from Chapter V. Next the initial state inconsistency is addressed – existing work is discussed, the theory is developed, algorithms described, and example application to the small manufacturing cell is presented, thus completing the full application to this cell in the process. Finally, the entire anomaly detection solution, including the academic resolutions to inconsistencies, is applied to an abstracted, simulated version of the RFT cell, first described in Chapter III.

6.1 Multiple Bit Change (MBC) Inconsistency

To address the multiple bit change inconsistency described in Section 5.3.2, a heuristic decision algorithm was developed to pre-process the event logs prior to their use for all steps of the anomaly detection solution. Prior to describing the existing work, the algorithm, and its application, some foundation is laid. First,

recall the ideas of events, event sets, event streams, labeled event streams, and event logs (Definitions 1, 2, 3, 11, 12), then some of these ideas will be used and/or built upon.

Definition 25 (Combination event)**.** A *combination event* e_C is a finite set of events $\{e_1, ..., e_k\}$ $1 < k < \infty$, all drawn from the same event set E, whose relative order is not known. All possible combination events drawn from an event set E are designated by the set E_C. The events that are part of combination event e_C are called its *constituent events*. A *multiple bit change (MBC) event* is one type of combination event where the events' order is not known because the events occur in one scan.

Definition 26 (Basic event)**.** A *basic event* e is a single event drawn from event set E and can be a constituent event for combination events. A *single bit change (SBC) event* is a type of basic event, and can be a constituent event for an MBC event.

Definition 27 (Unique events for an event log)**.** The *unique events for an event log* U_Σ are the events $u \in E \cup E_C$ that occur in the event log Σ.

These definitions are necessary because the original definitions related to events assumed strings of ordered events rather than the possibility of combination events. Due to combination events, an event can be represented in two ways, and this leads to two corresponding representations for event streams and for event logs.

Definition 28 (Index representation of unique event, row representation of unique event)**.** An *index representation of a unique event* $u \in U_\Sigma$ is an integer between 1 and $|U_\Sigma|$ that corresponds to that particular unique event. A *row representation of a unique event* $u \in U_\Sigma$ is a row of size $(E \cap U_\Sigma)$ where each element is 1 if the corresponding $e \in (E \cap U_\Sigma)$ is a constituent event of u, and 0 otherwise. An *index representation of an event stream* is an event stream whose events are represented

by indices, and a *row representation of an event stream* is an event stream whose events are represented by rows. Similarly, an *index representation of an event log* and a *row representation of an event log* are event logs whose event streams have the corresponding representation.

For example, consider a set of unique events for an event log U_Σ that has four basic events. The index representation of $u \in U_\Sigma$ would be an integer between 1 and the number of unique events (basic plus combination). The row representation of $u \in U_\Sigma$ would be [1000], [0100], [0010] or [0001] if u is a basic event and otherwise, if u is a combination event, it could be [1001] or [0111], for example, depending on which combination events are part of U_Σ. Suppose these are the only events in the log, $U_\Sigma = \{[1000], [0100], [0010], [0001], [1001], [0111]\}$. The index representation is more compact and can more easily be associated to other aspects of the Petri net formalism, such as the incidence matrix. An example event stream $\sigma \in \Sigma$ in index representation would be $\sigma = 1\ 4\ 6\ 3$. The row representation, however, allows basic and combination events to be easily distinguished and the constituent events of combination events to be easily identified. In row representation, this same event stream σ is expressed as $\sigma =$ [1000] [0001] [0111] [0010]. The unique event u_5 can be expressed either as $u_5 =$ [1001] or $u_5 = \{u_1, u_4\}$. The set of basic events is split into two classes.

Definition 29 (Stand-alone event, accompanying event)**.** A *stand-alone event* is a basic event that can occur on its own, whereas an *accompanying event* is a basic event that only occurs as part of a combination event. Thus, $E = E_{SA} \cup E_A$. A *trigger event* is a stand-alone PLC bit event, and a *non-trigger event* is is an accompanying PLC bit event.

With these definitions and associated notation, existing work and our MBC algorithms can now be described.

6.1.1 Handling Combination Events in DES

Previous work has been done in discrete event systems to handle combination events, in which they have been called simultaneous events. In [38], the vector DES (VDES) formalism was extended to include simultaneous events and it was shown how to create a non-deterministic controller that allows maximum concurrency in the controlled system while still enforcing desired specifications. Supervisory control of concurrent discrete event systems under partial observation to meet a specification was developed in [58], with the motivation being concurrent operation of multiple sub-systems. In both of these cases, the simultaneous events are generated by the controller in circumstances where such simultaneity will not cause a specification to be violated. In contrast, when describing the behavior of a system with simultaneous events from a PLC, it is not known whether these events are intended to be simultaneous or sequential, and if sequential, then their proper order.

Some research related to handling simultaneous events has specifically considered using events from PLCs. In [22], the simultaneity inherent in PLC data is discussed, but again from the control rather than identification perspective. This work takes the view that the scan-based nature of PLC execution makes simultaneous events unavoidable, and thus the supervisor must be insensitive to the interleaving of events which may occur simultaneously. Essentially, the issue of simultaneous events is handled by assuring that they can be treated as sequential events whose order does not matter. In contrast, the system identification in [35] and [54] assumes that the difference between subsequent PLC messages (I/O vectors) are events, thus assuming that each simultaneous (combination) event is a unique event instead of a sequence

of individual events.

This existing work illustrates the two main options of how to handle combination (simultaneous) events from a PLC – treat each combination event as a sequence of basic events or treat each combination event as a unique event. Each of these options, however, has its drawbacks. If each combination event is treated as a sequence of basic events, then the order of those events must be determined for the purpose of modeling the system behavior. If each combination event is treated as a unique event, then the number of events in the system's model is much larger and may become overly complicated. Instead of using either of these options exclusively, our work develops a heuristic algorithm to determine which is the best option for a given combination event.

6.1.2 MBC Algorithm

This algorithm makes a decision for each MBC event in the event log to either leave it as a unique event or split it into its constituent SBC events (and also determine their order). For a given MBC, this decision is based on the constituent SBC events, including the relationships among them, whether any of them do not trigger a message, and their relationships with the events that occur just before and just after this MBC in the given event log. This algorithm is implemented as a set of three algorithms that are nested within one another.

The highest level algorithm, called MBC Decision Algorithm, implements this decision-making for an event log. This algorithm takes as inputs an event log Σ in row form and the set of non-trigger events, E_A. The output of the algorithm is Σ', an updated version of Σ after applying the algorithm and changing to index representation. The first step (D1.1) is to determine the unique events in the event log U_Σ and the relationships among these events, as defined in Definition 21. Next,

an updated event log Σ' is created by going through each event e in each event stream σ of the event log Σ (D1.2). If e is an SBC, then it is included in Σ' as is. If e is an MBC, then Algorithm 5 (Decision for Particular MBC Algorithm) is called to determine whether e should be split, and if so, in what order, yielding σ_{new} which may be a single event or stream of events. After this process is completed for every e in every σ of Σ, then the unique events are determined for the updated event log $U_{\Sigma'}$ (D1.3) because some MBC events in Σ may never occur in Σ' due to being split. Finally in Step D1.4, using $U_{\Sigma'}$, Σ' is changed to index representation. This algorithm is described in Algorithm 4.

MBC Decision Algorithm:
Given: event log Σ in row form and E_A (events that do not trigger a message)

- D1.1) Determine the unique events U_Σ and their relationships in Σ
- D1.2) Update Σ to Σ' by making a decision for each MBC (Σ' initialize as empty)
  - a. For each event stream σ in Σ, create updated stream σ' (initialized as empty)
    - i. For each event e in σ, create updated event or event sequence
      1. If event is SBC ($e \in E$), then $\sigma' = \sigma' e$
      2. If event is MBC ($e \in E_C$), call **Decision For Particular MBC Algorithm** for e which will give result σ_{new}; update $\sigma' = \sigma' \sigma_{new}$
    - ii. Update $\Sigma' = \{ \Sigma', \sigma' \}$ to include σ'
- D1.3) Determine the unique events $U_{\Sigma'}$ in the updated event log Σ'
- D1.4) Change Σ' from row to index representation, using indices of $U_{\Sigma'}$

Outputs: updated event log Σ' in index representation

Algorithm 4: High level heuristic algorithm that decides how to split multiple bit change (MBC) events

The middle level algorithm, called Decision For Particular MBC Algorithm, has the main purpose of checking the number of bit changes in a given MBC, and treating those cases appropriately. The inputs for this algorithm are an MBC event $e \in E_C$, the events just before and just after this event in a particular event stream U_Σ, relationships among these events, and E_A. The output is σ_{new}, which may be one or a sequence of events in which each constituent SBC of e is included exactly once,

either on its own or as part of an MBC event. Steps D2.1 through D2.3 are the mutually exclusive cases of what to do if the MBC has two, three, or more than three constituent SBC events, respectively. In D2.1 the low level Algorithm, called Decision For Two-BC Algorithm, is called on the 2-BC (two-bit-change) event e to produce σ_{2new} which is set as $\sigma_{new} = \sigma_{2new}$. Step D2.2 checks whether the 3-BC event e can be split into a 2-BC event e_{2BC} and an SBC event e_{SBC}, and if so, calls Decision For Two-BC Algorithm on e_{2BC} and combines its result σ_{2new} with e_{SBC}. If it cannot be split this way, then $\sigma_{new} = e_a e_b e_c$ where e_a, e_b and e_c are the constituent events of e and they appear in this order in U_Σ. If e has more than three constituent events, Step D2.3 splits it into its associated SBC events and their order is based on the unique events, likewise to the case of three constituent events when they cannot be split into a 2-BC event and an SBC event. This algorithm is described in Algorithm 5.

The purpose of the low level algorithm, called Decision For 2-BC Algorithm, is to make the MBC splitting decision for 2-BC events. The inputs to this algorithm are the 2-BC itself $e = \{e_a, e_b\}$, the events that occur just before and just after this 2-BC event, U_Σ and the relationships among its events, E_A, and whether e_a and e_b are in the same process. The output is σ_{2new}, which can either be $\{e_a, e_b\}$, $e_a e_b$ or $e_b e_a$. There are three cases based on the input. If the events have no relationship ($e_a \sharp e_b$), both trigger messages ($e_a, e_b \notin E_A$) and are in the same process, then the case is D3.1 and $\sigma_{2new} = \{e_a, e_b\}$. If the e_a and e_b have a causal relationship or one is a member of E_A, then case D3.2 applies and $\sigma_{2new} = e_a e_b$ or $e_b e_a$ depending on the direction of the causal relationship or which event is in E_A. The final case, D3.3, is if $e_a || e_b$ (they can occur in either order) or are in different processes, and neither is in E_A, in which case their relationships with the previous and next events are

Decision For Particular MBC Algorithm:
Given: an MBC event $e \in E_C$, the event just before it e_{-1}, the event just after it e_{+1}, the unique events U_Σ , the relationships among the events in U_Σ, and the events that do not trigger a message (E_A)

D2.1) If e has two SBC events, call **Decision For Two-BC Algorithm** and result σ_{2new} from that algorithm will be the result σ_{new} from this algorithm

D2.2) Else if e has three SBC events

a. Determine each possible 2-BC and SBC combination
b. For each such combination, check if this 2-BC exists in U_Σ
c. If there exists a combination for which the 2-BC exists in U_Σ
   i. If (one of these) 2-BC(s) has a causal relationship with its SBC event, then choose it; if multiple such cases, choose randomly among these
   ii. Otherwise, choose randomly among these 2-BCs in U_Σ
   iii. Call **Decision for Two-BC Algorithm** on this 2-BC event to get σ_{2new}, which is either the 2-BC event or its SBC events in a particular order
   iv. Result of this algorithm $\sigma_{new} = \sigma_{2new}$SBC or σ_{new} = SBCσ_{2new}, depending on the relationship between the SBC and 2-BC
d. Else, σ_{new} is the constituent SBC events in the order in which they occur in U_Σ

D2.3) Else if e has more than three SBC events, σ_{new} is the constituent SBC events in the order in which they occur in U_Σ

Outputs: σ_{new}, which may be one or a sequence of events up to the number of basic events in the input event e

Algorithm 5: Middle level MBC algorithm that handles the cases of different numbers of bit-changes: 2-BC, 3-BC, >3-BC

considered to determine whether $\sigma_{2new} = e_a e_b$ or $e_b e_a$. This algorithm is described in Algorithm 6.

6.1.3 Application of MBC Decision Algorithms to Small Manufacturing Cell

This MBC decision algorithm is illustrated through applying it to a variation of the small manufacturing cell described in Section 4.1 that has been altered to include some multiple bit changes. This variation, called small MBC manufacturing cell, differs from the original in two ways – it has an added event that does not trigger a message, and its event streams are altered to reflect this non-trigger event as well as allow some of the other events to occur within the same message. This

Decision For 2-BC Algorithm:
Given: a 2-BC event $e = \{e_a,e_b\}$, the event just before it e_{-1}, the event just after it e_{+1}, U_Σ , the relationships among the events in U_Σ, the events that do not trigger a message (E_A), and whether e_a and e_b are in the same process

D3.1) If the relationship between e_a and e_b is none, e_a and e_b are in the same process, and e_a, $e_b \notin E_A$, then $\sigma_{2new} = e = \{e_a,e_b\}$

D3.2) If e_a causes e_b or $e_a \in E_A$, $\sigma_{2new} = e_a e_b$ (or vice versa)

D3.3) If e_a and e_b can occur in either order, use their relationships among e_a, e_b, e_{-1}, and e_{+1} to decide whether $\sigma_{2new} = e_a e_b$ or $\sigma_{2new} = e_b e_a$

Outputs: the event or sequence of events, σ_{2new}

Algorithm 6: Low level MBC algorithm that handles the case of 2-BC events

added event, e_1, indicates when the machining of the part 1 has ended, which makes sense that it might not trigger a message because it does not acquire nor release any resources since the machine (resource 2) is not available again until the piece is pushed out (associated with the event d_1). Hence, $E_A = e_1$. Events g_1 and d_1 (getting a new part 1 and being done with a completed part 1) were set to be able to occur in the same message. Based on these changes, events for this cell included (in row representation): [0001001] and [1010000] where the events are expressed as $[g_1, m_1, d_1, q_1, g_2, q_2, e_1]$.

The MBC Decision Algorithm was applied to a no-fault event stream consisting of 5000 events created by this cell for use in model generation.

- Step D1.1: There are 11 unique events: all of the SBCs except that associated with e_1 and the following MBC events – $\{g_1, e_1\}$, $\{q_1, e_1\}$, $\{d_1, e_1\}$, $\{g_1, d_1\}$, and $\{g_1, d_1, e_1\}$. The ordering relationships among these events are expressed in Table 6.1. Note that the event e_1 is included in the table even though it does not occur in the log – it is included to highlight that its relationship with all of the other events is $\sharp$ because these relationships are considered in the next step.
- Step D1.2: This log has only one stream, and this step will be demonstrated using a few example MBC events from this stream. First consider the stream

$\sigma_1 = [1000000][0001001][0010000]$, where the middle event is an MBC. Calling Algorithm 5 notes that there are two constituent SBC events (4 and 7, or q_1 and e_1) and thus calls Algorithm 6. Because e_1 does not trigger a message ($e_1 \in E_A$), this event is handled by Step D3.2, which causes the event to be split into the sequence of events $\sigma_{new1} = [0000001][0001000]$ because the non-trigger event e_1 must have occurred first. Another example stream is $\sigma_2 = [0001000][1010000][0100000]$, where the middle event is an MBC. Calling Algorithm 5 notes that there are two constituent SBC events (1 and 3, or g_1 and d_1) and thus, calls Algorithm 6. Because $g_1||d_1$, this event is handled by Step D3.3. Looking at the previous and next events (q_1 and m_1, respectively), g_1 has relationship $||$ with each of them, $q_1 \rightarrow d_1$ and $d_1 \rightarrow m_1$, so there is not a preference for one order over the other, and so the MBC event is split and the order randomly chosen.

- Step D1.3: For this updated log Σ', the unique events $U_{\Sigma'}$ are exactly the SBCs (basic events) because every MBC was split.
- Step D1.4: Σ' is changed from row representation to index representation. For example, the two streams considered become $\sigma_1 = g_1e_1q_1d_1$ and $\sigma_2 = q_1g_1d_1m_1$, respectively.

When Algorithm 4 was applied to this event stream (event log, but it has only one stream), it created a new event stream whose unique events were all of the SBC events and no MBC events, which is correct in this case because all of the SBC events should stand alone and the only non-trigger event was known by the algorithm. To evaluate the ordering decisions, the model generation algorithm (Algorithm 1) was applied to the resulting new event stream to check if the event relationships were correct, which requires the order of the constituent SBCs to be correct. There were

Table 6.1: Ordering Relations for Event Pairs in MBC Example

	g_1	m_1	d_1	q_1	g_2	q_2	e_1	g_1, e_1	q_1, e_1	d_1, e_1	g_1, d_1	g_1, d_1, e_1
g_1	$\sharp$	$\parallel$	$\parallel$	$\parallel$	$\sharp$	$\leftarrow$	$\sharp$	$\sharp$	$\parallel$	$\parallel$	$\sharp$	$\sharp$
m_1	$\parallel$	$\sharp$	$\leftarrow$	$\sharp$	$\rightarrow$	$\sharp$	$\sharp$	$\rightarrow$	$\sharp$	$\parallel$	$\leftarrow$	$\parallel$
d_1	$\parallel$	$\rightarrow$	$\sharp$	$\leftarrow$	$\parallel$	$\parallel$	$\sharp$	$\leftarrow$	$\leftarrow$	$\sharp$	$\sharp$	$\sharp$
q_1	$\parallel$	$\sharp$	$\rightarrow$	$\sharp$	$\rightarrow$	$\sharp$	$\sharp$	$\parallel$	$\sharp$	$\rightarrow$	$\rightarrow$	$\rightarrow$
g_2	$\sharp$	$\leftarrow$	$\parallel$	$\leftarrow$	$\sharp$	$\parallel$	$\sharp$	$\sharp$	$\leftarrow$	$\parallel$	$\sharp$	$\sharp$
q_2	$\rightarrow$	$\sharp$	$\parallel$	$\sharp$	$\parallel$	$\sharp$	$\sharp$	$\rightarrow$	$\sharp$	$\parallel$	$\rightarrow$	$\rightarrow$
e_1	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$
g_1, e_1	$\sharp$	$\leftarrow$	$\rightarrow$	$\parallel$	$\sharp$	$\leftarrow$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$
q_1, e_1	$\parallel$	$\sharp$	$\rightarrow$	$\sharp$	$\rightarrow$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\rightarrow$	$\sharp$
d_1, e_1	$\parallel$	$\parallel$	$\sharp$	$\leftarrow$	$\parallel$	$\parallel$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$
g_1, d_1	$\sharp$	$\rightarrow$	$\sharp$	$\leftarrow$	$\sharp$	$\leftarrow$	$\sharp$	$\sharp$	$\leftarrow$	$\sharp$	$\sharp$	$\sharp$
g_1, d_1, e_1	$\sharp$	$\parallel$	$\sharp$	$\leftarrow$	$\sharp$	$\leftarrow$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$	$\sharp$

two models generated, one of which was identical to the underlying model shown in Figure 6.1 and the other was similar to the underlying model but had no causal relationship between g_1 and m_1 because they can occur in either order and the effect of the resource R1 on the event relationships was not considered. Thus, the MBC Decision Algorithm worked well for this application.

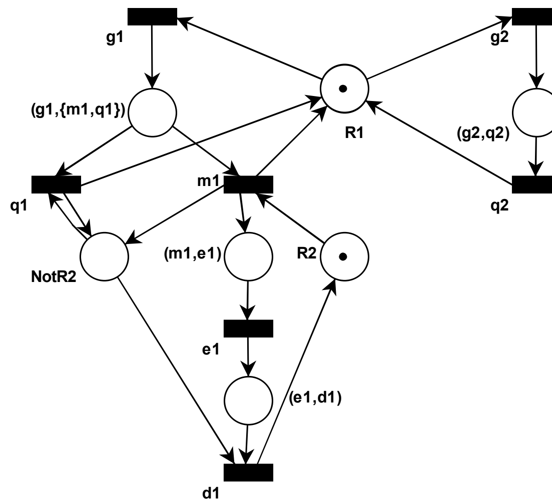


Figure 6.1: Small MBC Manufacturing Cell

6.1.4 Limitations of Heuristic MBC Algorithms

Another application of the MBC Decision Algorithm is included in Section 6.3, but because these are only two examples and this is a heuristic algorithm, some of its limitations are discussed here. One of the main limitations of the MBC Decision Algorithm is inherited from the anomaly detection solution – if there is not sufficient data, then the event relationships may be incorrect, which would cause the algorithm to possibly make poor MBC decisions. The other main limitations are related to cases in which the MBC has more than two constituent events.

Making decisions for higher-bit change events in a rigorous way is much more difficult because there are more options. For a 2-BC, there are three options – two SBC orders, and leave as unique. In contrast, for a 3-BC there are 13 possible options – there are six possible order of SBCs, six possible orders and combinations for a 2-BC and a SBC, and one option to keep as a 3-BC. Thus, the limitations of this algorithm for 3-BCs and higher are more significant. For 3-BC and higher events, this algorithm will not keep them as unique events largely because that could allow for too many events, but this limitation is lessened somewhat because, from examination of the data, higher-bit changes are more uncommon. For 3-BC events, if no two of its bits are present in the data as a 2-BC event, then this algorithm splits the MBC and sets its order to the order in which the bits appear in a message (i.e. bit 1, then bit 2, etc.). Also for 3-BC events, some factors that affect how the 3-BC should be split are not considered, such as if its split into a 2-BC event and an SBC, then only the relationship between these are considered even though the 2-BC may be split into its SBCs.

6.2 Initial State Inconsistency

To resolve the initial state inconsistency described in Section 5.3.4, the problem that had to be addressed was: given an event stream and an $STPR$ model, determine whether there exists a sequence of states (markings) of the model such it could produce this event stream. Note that this is only a necessary condition for a model to produce an event stream, and not a sufficient condition. Thus, solving this problem will provide detection of some, but not necessarily all, anomalies. To address this inconsistency, existing work is examined, some theory and a set of algorithms are developed, and the algorithms are applied to the small manufacturing cell to demonstrate their use.

6.2.1 Model Producing Event Stream From Unknown Initial State

It is common in academic work to assume that all observed event streams start from the system's initial state. For example, in the area of discrete event systems (DES), system identification often assumes that the system's behavior is cyclic and that the observed event streams start and end in the initial state [35], [54], [42]. In workflow mining, an area of computer science that also involves system identification of some forms of DES, a similar assumption is made [60] [16]. Many industry manufacturing systems run continuously, however, and thus observed event streams may not start from the system's initial state.

Several existing approaches were considered to determine whether an $STPR$ could have generated an event stream, starting from an unknown initial state. The set of reachable states could be calculated [41] and then searched for sequences of valid markings that would allow the event stream to occur. If at least one such sequence of markings existed, then the model would accept that event stream, but if no such

sequence existed, then the model would not accept the event stream. A major concern about this approach is the computational complexity of calculating the set of reachable states, which in general takes exponential time [46], searching through it, and keeping track of the different possible marking sequences thus far.

Another approach considered was also searching the reachable markings for a sequence that allows the event stream, but calculating the set of reachable markings in a simpler way. For some classes of Petri nets, the set of reachable markings can be described concisely without reference to such a graph. One classification scheme for Petri nets has asymmetric choice nets as its broadest category.

Definition 30 (Asymmetric choice net)**.** An asymmetric choice net (AC net) is a Petri net N where if two places, p_1 and p_2, feed the same transition ($p_1 \bullet \cap p_2 \bullet \neq \emptyset$), then either the transitions fed by p_1 are a subset of those fed by p_2 or vice versa (either $p_1 \bullet \subseteq p_2 \bullet$ or $p_2 \bullet \subseteq p_1 \bullet$) [46].

Another classification scheme for Petri nets has generalized trap-circuit nets (TCC nets) and generalized siphon-circuit nets (SCC nets) as its broadest categories.

Definition 31 (TCC (SCC) nets)**.** TCC nets (SCC nets) are Petri nets where each directed circuit contains a trap (siphon) [46].

The type of Petri net used by this anomaly detection solution, $STPR$s, however, are not AC nets because in $STPR$s there can be a choice between two events (p feeds more than one transition, $t_1, t_2 \in p\bullet$) where one of these events t_1 acquires a resource p_R ($t_1 \in p_R\bullet$) but this resource also is acquired by another transition t_3 ($t_1 \in p \cap p_R$, $p\bullet \not\subseteq p_R$ and $p_R\bullet \not\subseteq p$). $STPR$s also are neither TCC nets nor SCC nets because some fundamental directed circuits, such as those for resources used in multiple cases in a process or in multiple processes, do not contain traps but

instead must combine all circuits for a resource to make a trap. An $STPR$ can be a single TPR, which means that likewise TPRs are neither AC nets nor TCC/SCC nets so the reachability cannot be determined for processes (TPRs) and then built up to systems ($STPR$s). Reachability cannot be applied to TPs because they have null initial marking and hence no reachable states. Thus, the existing results for determining the set of reachable markings for Petri nets of these classes are not applicable in solving this initial state problem.

Because the state from which an observed event stream starts is unknown, and the set of reachable states cannot be easily described for $STPR$s, state estimation is necessary. Considering that Petri nets are the modeling formalism being used, state estimation is marking estimation. Some work in marking estimation, such as [15], assumes the initial marking is known and estimates the current marking, where the uncertainty is due to silent transitions. Other research estimates the initial marking through observation of an event stream. In [26], a lower bound on the initial marking of a Petri net is determined through observing a stream of events, where the lower bound is updated after each observed event. In [37], the minimum initial marking is calculated for the case in which transition labels are observed and the uncertainty comes from the labels and transitions not having a one-to-one marking. In both of these approaches, only a lower bound on the initial marking is determined, not a lower bound on each of the markings associated with the event stream occurring, and hence these approaches cannot be used directly. The idea of a lower bound on the initial marking is used in developing the theory in the next section, although extended to a lower bound on all the markings for producing an event stream.

6.2.2 Theory and Algorithms

The approach selected to address the initial state inconsistency determines a sequence of lower bounds on the markings necessary for the given model to have produced the given event stream and checks whether these lower bound markings violate the upper bound marking restrictions caused by the conservation of resources. If the lower bound of the markings does not exceed the upper bound marking restrictions, then the model is said to accept the event stream (label as normal), whereas if the any of these lower bounds exceeds the upper bound, the model is said to not allow the event stream (label as anomalous) where the first event for which the upper bound is violated is indicated as the first anomaly. If an anomaly is detected, its effect is ignored for the remainder of the stream, and if other events that cause a violation of the upper bound are found, they are labeled as anomalies as well and their effect ignored. First, the theory for the lower bound markings is developed, then for the upper bound marking restrictions based on the resources, and then their combination.

Given an event stream and a model, a set of lower bound markings can be calculated for that model to have generated that event stream. The theory builds up from the simplest case – a single event.

Lemma VI.1. *Given a Petri net and an event e that can occur in that Petri net, the lower bound on the model's marking prior to the event e occurring is $M_{LB,0} = \bullet e$ and the lower bound after the event e has occurred is $M_{LB,1} = e\bullet$.*

This lemma is evident based on the definition of a Petri net. For an event e to occur in a Petri net, it must be that $M_0 \geq \bullet e$ so that e is enabled. Likewise, when an event e occurs it will necessarily produce $e\bullet$, and thus $M_1 \geq e\bullet$.

Lemma VI.2. *Given a Petri net, a stream of events $\sigma = e_1...e_m$ where e_i $\forall i = 1...m$ can occur in the Petri net, and a lower bound on the markings $M_{LB,0}...M_{LB,m-1}$ based on events $e_1...e_{m-1}$, the lower bound on the model's markings when e_m is included are $M'_{LB,0}...M'_{LB,m-1}M'_{LB,m}$, where*

- $M'_{LB,i} = M_{LB,i} + NE_m$ *for* $i = 1...m-1$
- $NE_m = max((\bullet e_m - M_{LB,m-1}), 0)$
- $M'_{LB,m} = M'_{LB,m-1} - \bullet e_m + e_m \bullet$

and thus for $i = 1...m$ $M_{LB,i} \leq M'_{LB,i} \leq M_i$.

NE_m is the set of tokens required for e_m ($\bullet e_m$) that are not explained by the previous state ($M_{LB,m-1}$), and thus need to be added to the previous lower bounds ($M_{LB,0}...M_{LB,m-1}$) to update them ($M'_{LB,0}...M'_{LB,m-1}$) so that e_m could occur. The lower bound marking for $M'_{LB,m}$ is then simply the previous marking $M'_{LB,m-1}$ with the effect of e_m included. This lemma was coded into Algorithm 7. The lower bound markings associated with a stream of events can be calculated iteratively using Lemma VI.1 to initialize the lower bound markings based on the first event and Lemma VI.2 to update the lower bound markings for each subsequent event.

Calculate Lower Bound Marking Algorithm:

Given: previous lower bound (prevLB, matrix where each column is lower bound of a marking, first column is lower bound of initial state), current event in stream (t_c), the places that feed the current event ($\bullet t_c$), and the places that are fed by the current event ($t_c\bullet$)

LB.1) Calculate the change required for t_c: chngReq = $\bullet t_c$ – prevLB(prevState)

LB.2) Calculate what is required for t_c to occur that has not already been explained: notExpln = max(chngReq, 0)

LB.3) Update prevLB by adding notExpln to each state in it

LB.4) Make newest entry to lower bound as newState = prevLB(prevState) - $\bullet t_c$ + $t_c\bullet$

LB.5) Create newLB as concatenation of updated prevLB and newState

Output: new lower bound (newLB)

Algorithm 7: Algorithm that updates the lower bound markings due to a stream of events, given the most current event

Next, the the upper bound marking restrictions based on the resources are determined. The general idea with these restrictions is that an $STPR$ model has at least some type A resources and because these resources are limited to known quantities (i.e. number of robots in system), they impose restrictions on the markings. For example, a single robot is used for exactly one thing at a time and this must be reflected in the marking – a token associated with the robot is in exactly one place. Because only a lower bound on the markings is known, this constraint which should be an equality is instead relaxed to an inequality.

Theorem VI.3. *Given an $STPR$ (N, M_0), the set of reachable markings $R(N, M_0)$ is upper bounded by the resource constraint equations $\Sigma_{i=1}^{||P||} y_j M(p_i) = c_j$ for every p-semiflow y_j associated with a type A resource ($j = 1$...number type A resources), where $c_j = \Sigma_{i=1}^{||P||} y_j M_0(p_i)$.*

Proof. As specified in Definition 20, in an $STPR$ each type A resource is associated with a p-semiflow. One important property of p-semiflows is that $M^T y = M_0^T y$ for any given initial marking M_0 and any $M \in R(N, M_0)$ [46]. For a given initial marking M_0 and p-semiflow y_j, $M^T y_j = \Sigma_{i=1}^{||P||} y_j M(p_i)$ and $M_0^T y_j$ is equal to constant c_j, which means that $\Sigma_{i=1}^{||P||} y_j M(p_i) = c_j$. Thus from [8], the $STPR$ N is conservative with respect to each p-semiflow y_j, and each of these p-semiflows enforces a restriction on the marking expressed by the previous equation. □

The resource conservation constraints developed in Theorem VI.3 are implemented in Algorithm 8. Each resource constraint inequality is of the form

$$y^1 M(p_1) + ... + y^{||P||} M(p_{||P||}) \leq max \tag{6.1}$$

where

$$max = y^1 M_0(p_1) + ... + y^{||P||} M_0(p_{||P||}) \tag{6.2}$$

where $y^1...y^{||P||}$ are the elements of a p-semiflow y and also the coefficients of the inequality, and $y^1 M_0(p_1) + ... + y^{||P||} M_0(p_{||P||})$ is a constant and the maximum value of the inequality. These lower and upper bound constraints can be used together to create a necessary condition for a model to accept an event stream.

Calculate Resource Constraints Algorithm:
Given: model, events
- RC.1) Calculate the p-invariants as solutions to $A^T y = 0$
- RC.2) Determine which p-invariants are p-semiflows (have all non-negative values)
- RC.3) For each p-semiflow, create a resource constraint
  - a. Max value for constraint is p-semiflow times initial marking of model
  - b. Coefficients for constraint are p-semiflow

Outputs: resource constraints for model

Algorithm 8: Algorithm that calculates resource constraint equations based on the model

Theorem VI.4. *If there exists a prefix $\hat{\sigma} = e_1...e_j$ of an event stream $\sigma = e_1...e_m$, $j \leq m$, for which any marking in its sequence of lower bound markings ($M_{LB,0}...M_{LB,j}$) exceeds any of the resource conservation constraints of a model, then that model could not have generated that event stream.*

Proof. Using proof by contradiction, suppose the "if" conditions hold but the "then" conditions do not. In other words, there is a violation of at least one of the resource conservation constraints, but the model could generate the event stream and would create a valid sequence of markings associated with it. Let σ be an event stream and $\hat{\sigma}$ be a prefix of σ such that for at least one $M_{LB,i}$ $i = 0...j$, there is at least one p-semiflow y_k such that $M_{LB,i}^T y_k > M_0^T y_k$. Because y_k is a p-semiflow, and M_0 is a marking, $y_k \geq 0$ and $M_0 \geq 0$, hence $M_0^T y_k \geq 0$. Similarly, $M_{LB,i} \geq 0$. From Lemma VI.2, $M_i \geq M_{LB,i}$ for each element. Combining this information implies $M_i^T y_k > M_0^T y_k$, which means that this marking is not valid, and hence a contradiction. □

This necessary condition is checked in Algorithm 9, which makes use of Algorithms

7 and 8. This condition is only necessary – there may be some event streams the model cannot generate that do not violate these constraints, and thus will not be found by checking this condition. In contrast to the other approaches, however, this approach is valid for $STPR$ models and avoids the computational complexity of creating reachability graphs. The performance assessment and anomaly detection algorithms (Algorithms 2 and 3) were updated so that whether a model accepts an event stream is determined using Algorithm 9.

Find Anomaly in Stream Algorithm:
Given: (unlabeled) event stream ($e_1 e_2 \ldots e_m$), resource constraints, e• and e• for each event e for a particular model

FA.1) Initialize previous lower bound (prevLB) to [•e_1 e_1•]
FA.2) For each event, $e_2 \ldots e_m$
  a. Call **Calculate Lower Bound Marking Estimate Algorithm** to update the lower bound (newLB)
  b. For each resource constraint
    i. Calculate the left-hand-side (LHS) of resource conservation constraint using newLB and this resource constraint's coefficients
    ii. Compare the LHS to the max from the resource constraint and if the LHS exceeds the max for at least one place, then anomaly present and location recorded
  c. If this event was not anomalous, then update prevLB to newLB

Outputs: whether anomaly is present, and if so, at what location(s) in event stream

Algorithm 9: Algorithm that checks whether a model accepts an event stream (does not find an anomaly)

6.2.3 Application of Initial State Algorithms to Small Manufacturing Cell

These algorithms to address the initial state inconsistency are illustrated through applying them to the small MBC manufacturing cell with event streams that do not necessarily start in the initial state. A set of 35 event streams was generated from this manufacturing cell – 25 no-fault and 10 fault – to use for performance assessment and anomaly detection. These event streams start from a variety of states and have MBCs, which are first addressed by applying the heuristic decision algorithm described in Section 6.1.2. They are randomly assigned for either performance

assessment or anomaly detection, both of which use the Find Anomaly in Stream Algorithm to test the necessary condition for a model to have produced the given event stream. Thus, the Find Anomaly in Stream Algorithm will be demonstrated for a particular stream and results of the performance assessment and subsequent anomaly detection presented.

The sample event stream is $g_2q_2e_1q_1$, and we will check whether the model from Figure 6.1 finds an anomaly in this stream. First the resource constraints must be generated for this model using Algorithm 8.

- Step RC.1: The model's incidence matrix is

$$A = \begin{bmatrix} 0 & 0 & -1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 \\ 1 & -1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ -1 & 1 & 0 & 1 & -1 & 1 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

  where the places (rows) are ordered $[\ \text{P1 P2 P3 P4 R1 R2 NotR2}\ ]^T$ and the events (columns) are ordered as in Table 4.1, and based on that, its p-invariants

are

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

- Step RC.2: The p-semiflows are $y_1 = [0011100]^T$ and $y_2 = [1100010]^T$.
- Step RC.3 for first p-semiflow: Max value for constraint is $M_0^T y_1 = [0000110] * [0011100]^T = 1$, and coefficients are p-semiflow itself $[0011100]^T$.
- Step RC.3 for second p-semiflow: Max value for constraint is $M_0^T y_2 = 1$, and coefficients are p-semiflow itself $[1100010]^T$.

Next these resource constraints and $g_2 q_2 e_1 q_1$ are used as input to Algorithm 9.

- Step FA.1:

$$prevLB = [\bullet g_2 \; g_2 \bullet] = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

- Step FA.2a for second event (q_2): use Algorithm 7 to update lower bound
  - Step LB.1: chngReq $= \bullet q_2 - prevLB(:, 2) = [0000000]^T$
  - Step LB.2: notExpln $=$ max(chngReq, 0) $= [0000000]^T$

– Step LB.3: updated prevLB = prevLB + notExpln

– Step LB.4: newState = prevLB(prevState) $- \bullet q_2 + q_2 \bullet$ = $[0001000]^T - [0001000]^T + [0000100]^T = [0000100]^T$

– Step LB.5: $newLB =$

$$[updatedprevLB\ newState] = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

- Step FA.2b for second event (q_2): for each of the three states in $newLB$

$$[0000100]^T, [0001000]^T, [0000100]^T$$

  checked $M^T y_1 \leq 1$ and $M^T y_2 \leq 1$ for each state M.

- Step FA2.c for second event (q_2): both constraints held, so the model may be able produce the stream $g_2 q_2$. Note that the uncertainty is because the algorithms check a necessary, but not sufficient, condition for a model to produce an event stream from an unknown initial state.

- Repeat Step FA.2 for third and fourth events (e_1 and q_1), and find that for fourth event, the constraint for resource 1 (R1) is violated and thus the model may be able to produce the stream $g_2 q_2 e_1$, but cannot produce the stream $g_2 q_2 e_1 q_1$.

Using Algorithm 9 to check whether a model finds an anomaly in a stream, the updated performance assessment and anomaly detection algorithms were run. The

performance assessment was run on 16 event streams with a total of 312 events, yielding a performance of 302 for the underlying model and 296 for the other model. The anomaly detection was run on 19 event streams with a total of 410 events, resulting in 100% of the event streams labeled correctly and 98% of the individual events labeled correctly. Having all of the event streams labeled correctly despite some events being labeled incorrectly indicates that the incorrect event labels were associated with identifying which event(s) were anomalous within an anomalous stream.

6.3 Results for Simulated RFT Cell

To demonstrate the use of the full anomaly detection solution, including the resolutions to the inconsistences, the solution is applied to a version of the RFT cell from Chapter IV that has been modified to have $M1S$ and $M2S$ as non-trigger events, $Arr1$ and $M1E$ possibly occur in the same message, $Arr2$ and $M2E$ possibly occur in the same message, and event streams not necessarily start from the initial state. A single long event stream with just under 5000 events is used for model generation; 17 event streams, of which 4 are fault, ranging from just a few events to almost 40 events are used for performance assessment; and 18 event streams, of which 6 are fault, ranging from just a few events to around 30 events are used for anomaly detection. The no-fault event streams used for performance assessment and anomaly detection were created from longer event streams using the splitting algorithm described in Section 5.3.5, although the labeling had to be done by hand because the faults streams were made to have a variety of faults and thus did not have specific symptoms to use for labeling. Thus, the separate, labeled versus continuous, unlabeled stream inconsistency was addressed. As mentioned in Chapter V, the inconsistencies of not all resource events being observable and having an incon-

sistent mapping were addressed by industry rather than academia so they are not considered here.

To implement the full anomaly detection solution required the following steps:

1. MBC Decision Algorithm for all event streams
2. Model Generation Algorithm applied to event log of no-fault event streams
3. Performance Assessment for mix of no-fault and fault streams, using necessary condition for model accepting event stream
4. Anomaly Detection for unlabeled streams, using necessary condition for model accepting event stream

After applying Algorithm 4, all the events were SBCs. Model generation produced three different models for Process 1, similar models for Process 2, and one model for Process 0, resulting in nine models of the system. The underlying model was among the ones created and is shown in Figure 4.11. Another model created is illustrated in Figure 6.2. This model is identical to the underlying one except it has two additional places – P21 and P22 which connect RelU to Arr and ULS – that prevent any events from Process 1 or Process 2 occurring. The third type of model created is similar, but the places P21 and P22 only connects Rel1U to UL1S and Rel2U to UL2S, respectively.

Applying performance assessment to these models yields perfect performance (328 out of 328) for all of the models. On first impression, this result seems erroneous because clearly some of the models have unnecessary places that overly restrict the system's behavior. The performance assessment checks only if the necessary condition is met (there exists a sequence of states in the model from which the event stream could have been produced) and this necessary condition restricts the marking of places that represent or use resources, and these places do not. Additionally,

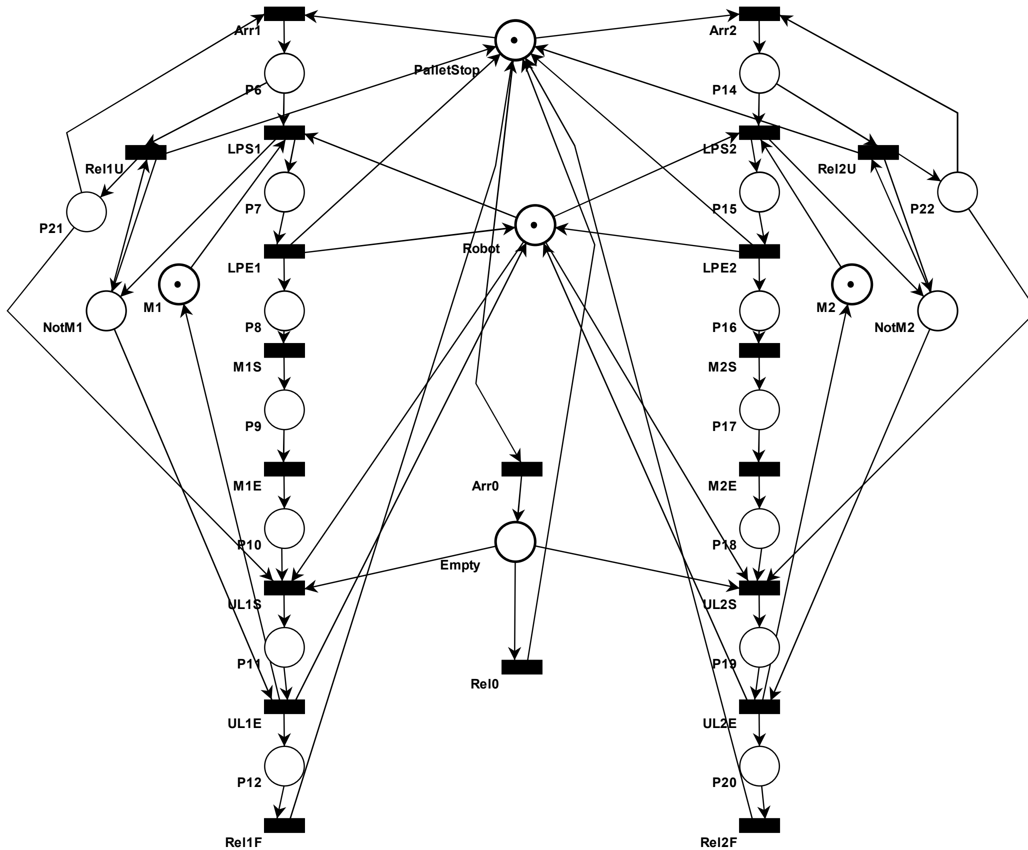


Figure 6.2: Model created by model generation for MBC RFT cell

this necessary condition does not use the fact that all places other than type A and type NC resources should not be initially marked. Thus, the performance assessment on these incorrect models is identical to that of the underlying model because under this necessary condition, the incorrect models could have as many tokens in these additional places as necessary to allow the event stream to occur. Although the necessary condition does not always distinguish between models that reflect the underlying model of the system and those that do not, the necessary condition can recognize the anomalies, which is the goal in this solution. Applying the anomaly detection to these models, which have identical performance, results in 100% of the anomaly detection event streams labeled correctly and 94% of the individual events labeled correctly. Hence, even though some of the models do not represent the sys-

tem's actual behavior, under the necessary condition used to check whether a model accepts an event stream, these models still perform well and contribute to excellent performance.

6.4 Conclusions

The research associated with both the anomaly detection solution developed in Chapter IV and the academia-industry gap identified in Chapter V are advanced by the complete resolution of two inconsistencies: (1) the string of ordered events versus multiple bit changes, and (2) the event streams start in initial state versus other states. The first inconsistency was resolved through the development of a heuristic decision algorithm that makes a decision for each multiple bit change (MBC) event about whether to split it and if so, the order of the resulting events. This decision is made based on the relationships among the MBC's constituent SBC events, their relationships with the previous and next events, and other factors. The second inconsistency was addressed by developing the theory and corresponding algorithms for a necessary condition on an $STPR$ model to generate a stream of events. Through resolving these two inconsistencies, the anomaly detection solution is more easily applied to not only the Ford machining cell but also other similar industrial systems. Since these inconsistencies may arise in the application of other academic solutions to other industrial systems, their resolutions can aid in these applications too.

CHAPTER VII

Conclusions and Future Work

This dissertation addressed problems inherent in many important systems that can be described as discrete event systems. Keeping these systems functioning well requires addressing faults that occur in them. There are two main ways to address such faults – they can be prevented from ever occurring through verification or they can be detected and resolved at the time at which they occur. This research developed means to address faults in event-based systems for which there is no formal, pre-existing model. This limitation was motivated by industrial manufacturing systems for which formal models are not commonly available and may be difficult to accurately develop, but for which unexpected downtime due to faults is very costly. The main contributions of this research, as well as its future work, fall into two categories – those related to verification and those related to anomaly detection.

7.1 Verification Contributions

The first contribution for verification is the identification and definition of the input order robustness property. A procedure for verifying input order robustness was developed for both logic controllers and networks of controllers. If a controller is input order robust, the closed-loop system is guaranteed to be deterministic. Theory was developed for a network of controllers using modular techniques, allowing the input

order robustness verification to be applied to large-scale systems. The application of input order robustness verification to systems implemented in IEC 61499 illustrated how some of the open execution semantics of the standard could be overcome if the controller is input order robust. This a step toward different IEC 61499 tools being used interchangeably and producing consistent results.

7.2 Anomaly Detection Contributions

An anomaly detection solution for event-based systems without pre-existing formal models was developed that provides insights into anomalous behavior while only requiring information that is likely available for industry systems. As part of this solution, a model generation algorithm was created that expands upon the $\alpha+$ algorithm [16] to explicitly incorporate resources and generates variations of the $\alpha+$ model that may provide for better anomaly detection performance. Theory was developed for conditions under which the model generation algorithm is guaranteed to produce a model identical to the underlying model of the system, which provides some surety of the model generation's results. A variation on Petri nets that include resources, $STPR$s, was defined based on removing some restrictions imposed by pre-existing formalisms that prevent the modeling of some common behavior, such as a task using more than one resource.

In the process of applying this anomaly detection solution to an industry system (machining line), five inconsistencies were found between common academic assumptions made by the solution and the industry practice. Resolutions were developed and implemented for each of these five inconsistencies. The inconsistency of all events that acquire/release resources being observable versus some such events being filtered at lower levels of data collection was addressed by our industry partner following our

recommendation to change the system's logic so that these events were recorded. Similarly, the inconsistency of having a consistent bit-meaning mapping versus an inconsistent mapping was addressed on a case by case basis by our industry partner changing logic and our pre-processing the data to filter out known mapping issues. An algorithm was created that splits and labels continuous unlabeled event streams to address the inconsistency of separate, labeled event streams versus a continuous, unlabeled event stream, where the labeling is based on finding occurrences of symptoms that indicate a particular anomaly. To address the ordered stream of events versus multiple bit changes per message inconsistency, a heuristic decision algorithm was developed that decides how to treat each multiple-bit-change, and this algorithm addresses two-bit-changes thoroughly. Finally, the inconsistency of each event stream starting in the initial state versus most event streams not starting in the initial state was addressed through developing theory for a necessary condition under which an $STPR$ model will generate an event stream that may not start in the initial state, and implementing this theory in an algorithm. The heuristic decision and necessary condition algorithms have the most significant impact on applying the anomaly detection solution to industry systems because such systems often have PLCs, which generate multiple-bit-change messages, and are not often starting in or returning to their initial state. Additionally, because both the academic assumptions and industry practice are not limited to this solution and industry system, respectively, these inconsistencies and their resolutions can aid in the application of other solutions to new industry systems.

7.3 Future Work

One direction of future work for input order robustness verification is further developing its applicability for systems implemented in the IEC 61499 standard. An advantage of IEC 61499 is that some commonly used function blocks can be developed and re-used to aid in design and reduce verification, because they are verified once and do not need to be verified each time they are used. Some such common function blocks could be verified for input order robustness so that when they are used in networks of controllers, they do not need to be verified each time. The verification is applicable to only some of the execution semantics options, as described in Table 3.5, but could possibly be extended to include additional options.

Alternative approaches for achieving input order robustness is another area of future work. Although the simulations required for the verification are automated, determining the sets of inputs to be checked is not, as that is dependent on the design information available. In Section 3.4.3, several approaches were mentioned for how the system information, including the sets of inputs to be checked, can be determined based on how the design information is structured. If a particular approach for design information were selected, the input order robustness verification could be more fully automated, making it easier to use without expert knowledge of the system. Additionally, if a logic controller is found to be not input order robust, then a method to modify the logic controller to enforce that property would mean that the problem could not only be identified but also resolved automatically. Taking a different perspective, instead of verifying input order robustness, the possibility could be explored of developing conditions on a logic controller that would guarantee input order robustness. Then these conditions could be incorporated into the design

so that the logic controller would be input order robust to begin with and would not need to be verified and then potentially modified. Another related area of future work would be to develop verification for the more restrictive property that is like input order robustness but where the order of outputs, not just the set of outputs, must be the same regardless of input order.

Another area of future work, already briefly mentioned in Chapter I, is extending anomaly detection to fault detection. An anomaly is an unusual behavior whereas a fault is an incorrect behavior. Generally, unless a system operates very poorly, faults are also anomalies, and hence finding anomalies can find possible faults, but will also find some anomalous, yet no-fault, behavior. One idea for such an extension is to solicit feedback from the operator about whether detected anomalies are no-fault or fault behavior, and incorporate this feedback to modify the models so they evolve to reflect more no-fault behavior.

To further the goal of this anomaly detection solution being useful for industry, an area of future work that would have more industry impact is making the solution ready to install on a manufacturing plant's network with an easy to understand interface. Several tasks would be necessary to bring the anomaly detection solution to this stage. Implemented in Matlab scripts, the solution has some error handling built in, but that would need to be expanded so that less human interaction is required. For example, single-event loops are not allowed for $STPR$s and in the solution's current implementation, if a single-event loop is found the program terminates with some error information and a human then has to figure out why the single-event loop occurred and what to do next. The method for interaction between the anomaly detection solution and operator would need to be developed, including the format in which information is provided to the operator, what information the operator can

provide to the solution (i.e., whether an anomaly was actually a fault), and which tasks the operator can perform on the solution, such as having it re-do the offline calculations or changing which bits the model includes. Given this operator interaction, the algorithms' code would need to be packaged as an executable complete with a GUI. The scalability of the anomaly detection solution would also need to be further studied, beyond the example presented in Section 4.8, and improved, perhaps through doing performance assessment on the process models and creating a performance threshold for which process models are combined into models of the whole system. The results of the anomaly detection solution may be significantly affected by incorrectly labeled event streams used for model generation, and to a lesser extent performance assessment, because then some faulty behavior may be treated as normal and incorporated into the models or their performance. Means to reduce this sensitivity to incorrect labeling could be explored, including using thresholds in determining event relationships, i.e. a and b are only causal if ab occurs at least x times and ba occurs fewer than y times where x is significantly larger than y.

In Sections 4.6 and 4.7, simple metrics are used to assess model performance and detect anomalies based on models' votes. Although these metrics were sufficient to yield good results in the simulated systems (see Chapter VI), alternative metrics could be explored to improve performance, particularly if there are other types of systems for which these metrics do not yield good results. For example, the penalty for false negatives could be different than the penalty for false positives, depending on which incorrect labeling is more problematic. Models' performance could instead be based on its labeling of entire event streams, rather than of individual events within those streams, especially because sometimes the anomaly is detected later in the stream than where it actually occurred. Alternative voting methods could

include only counting the votes of the top few models or weighting their votes based on the logarithm of their performance, so that the better performing models have a disproportionately strong vote. Many other performance metrics could be tried from the area of model assessment for statistical learning [29], and voting methods from decision-making theory [56].

Although steps have been taken to reduce the computational complexity of our anomaly detection solution, such as considering systems that can be modularly decomposed, a formal complexity analysis could be done and the result compared to other system identification and fault detection solutions. One of the challenges in doing such an analysis is that the some factors that affect the computational complexity, particularly of the model generation step, are difficult to quantify. For example, the number of events in a process is assumed to be known, but how tightly connected those events are (i.e. whether each event relates to only a few others or most of the others) may be unknown and, even if known, hard to quantify.

The anomaly detection solution is based on certain assumptions about the system and, implicitly, assumptions about how the solution itself should work. Relaxing some of these assumptions and exploring their implications could be a very productive area of future work. If the system does not have a pre-existing formal model, but some or all of its processes do have such formal models, then the anomaly detection solution could be modified to create models for processes that need them and only verify the pre-existing models for processes that have them. The solution also assumes that the no-fault behavior is modeled and faults are anomalous in comparison to this no-fault (normal) behavior, but another option is that the solution could model fault behavior and faults would be behavior that is consistent with these models. If there is sufficient data (event streams) available of fault behavior to create such models,

they could be used instead of or in addition to the models of no-fault behavior in order to detect possible faults.

CHAPTER VIII

Appendix: List of Acronyms

- CNC: computer numerical controlled (machine); used for drilling, milling, etc. in manufacturing systems
- DES: discrete event systems; for general overview, see [9]
- ECA MFSM: event-condition-action modular finite state machine; type of discrete event system modeling formalism; see [5]
- ECC: execution control chart for an IEC 61499 function block; see [32]
- EWF-net: enhanced workflow net; a special type of Petri net; see [31]
- $EWFR$-net: enhanced workflow net with resources; a special type of Petri net that incorporates resources and is based on $EWF - net$; see [31]
- $MEWFR$-net: merged enhanced workflow net with resources; a special type of Petri net that incorporates resources and is based on combining $EWFR$-nets; see [31]
- FB: function block, two uses; 1) the primary component of IEC 61499 modeling, see [32]; 2) part of the programmed logic in the PLCs used by Ford, see Section 5.5
- FIFO: first in first out; generally refers to type of queue

- FSM: finite state machine; a type of discrete event system modeling formalism; see [9]
- GC: guard condition; refers to what is required for a transition to occur in IEC 61499 function block; see [32]
- IEC 61499: distributed control standard; see [32]
- IOR: input order robustness; property described and verified in Section 3.7
- MBC: multiple bit change; refers to the difference between two PLC messages or an event; contrast with SBC; introduced in Section 5.5 and discussed in detail in Section 6.4
- OPC: OLE process control; generally refers to OPC tags, which are like global variables used for communication of control commands and responses in some systems, especially manufacturing systems
- PLC: programmable logic controller
- RFID: radio frequency identification; RFID tags are used to track objects, such as parts and pallets in a manufacturing system
- RFT: Reconfigurable Factory Testbed; small manufacturing testbed at University of Michigan [44]
- S^2P: simple sequential process; a special type of Petri net; see [21]
- S^2PR: simple sequential process with resources; a special type of Petri net that incorporates resources and is based on S^2P; see [21]
- S^3PR: system of S^2PR; a special type of Petri net that incorporates resources and is based on combining S^2PRs; see [21]
- SBC: single bit change; refers to the difference between two PLC messages or an event; contrast with MBC; introduced in Section 6.4

- SPSR: system of processes that interact through shared resources; see Definition 15
- STPR: system of TPR; a special type of Petri net that incorporates resources and is based on combining TPRs and possibly $TPCR$s; see Definition 20
- SWF-net: structured workflow net; a special type of Petri net; see [60]
- TP: transition process; a special type of Petri net; see Definition 16
- TPCR: transition process that creates resources; a special type of Petri net that incorporates resources; see Definition 18
- TPR: transition process with resources; a special type of Petri net that incorporates resources and is based on TP; see Definition 17
- x-BC: x bits change in the same PLC scan; 1-BC is a SBC and x-BC where $x > 1$ is a MBC; introduced in Section 5.5
- XML: extensible markup language; a text-based data format; for more information, see `http://en.wikipedia.org/wiki/XML`

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] L. V. Allen, K. M. Goh, & D. M. Tilbury. Closed-Loop Determinism for Non-Deterministic Environments: Verification for IEC 61499 Logic Controllers. *Proceedings of the 5th IEEE Conference on Automation Science and Engineering*, August 2009.

[2] L. V. Allen & D. M. Tilbury. Event-Based Fault Detection of Manufacturing Cell: Data Inconsistencies Between Academic Assumptions and Industry Practice. *Proceedings of the 6th IEEE Conference on Automation Science and Engineering*, August 2010.

[3] L.V. Allen, K. M. Goh, & D. M. Tilbury. *Input Order Robustness: Guaranteeing Closed-Loop Determinism for Non-Deterministic Environments.* Submitted for journal publication May 2009.

[4] L.V. Allen & D.M. Tilbury. *Anomaly Detection Using Model Generation for Event-Based Systems Without a Pre-Existing Formal Model.* Submitted for journal publication July 2010.

[5] E. E. Almeida, J. E. Luntz, & D. M. Tilbury. Event-condition-action systems for reconfigurable logic control. *IEEE Transactions on Automation Science and Engineering*, 4: 167-181, 2007.

[6] B. A. Brandin, R. Malik, & P. Malik. Incremental Verification and Synthesis of Discrete-Event Systems Guided by Counter Examples. *IEEE Transactions on Control Systems Technology*, 12:387-401, 2004.

[7] M.P. Cabasino, A. Giua, & C. Seatzu. "Identification of Petri Nets from Knowledge of Their Language." *Discrete Event Dynamic Systems*, **17**: 447-474, 2007.

[8] C.G. Cassandras & S. Lafortune. *Introduction to Discrete Event Systems.* Massachusetts: Kluwer Academic Publisher, 1999.

[9] C. G. Cassandras & S. Lafortune. *Introduction to Discrete Event Systems – 2nd Ed.* Springer, 2007.

[10] P. J. Cameron. *Permutation Groups.* Cambridge University Press: Cambridge, UK, 1999.

[11] G. Cengic, O. Ljungkrantz, & K. Akesson. Formal Modeling of Function Block Applications Running in IEC 61499 Execution Runtime. *11th IEEE Conference on Emerging Technologies and Factory Automation*, 1269-1276, 2006.

[12] E. M. Clarke, O. Grumberg, & D. A. Peled. *Model checking.* MIT Press: Cambridge, MA, 1999.

[13] E. Clarke, O. Grumber, S. Jha, Y. Lu, & H. Veith. Progress on the State Explosion Problem in Model Checking. In R. Wilhelm, editor, *Informatics: 10 Years Back, 10 Years Ahead*, vol. 2000 of *Lecture Notes in Computer Science*, 176-194, 2001.

[14] J.E. Cook & A.L. Wolf. "Discovering Models of Software Processes from Event-Based Data." *ACM Transactions on Software Engineering and Methodology* **7**: 215-249, 1998.

[15] D. Corona, A. Giua, & C. Seatzu. Marking Estimation of Petri Nets with Silent Transitions. *Proceedings IEEE 43rd International Conference on Decision and Control*, The Bahamas, 2004.

[16] A. K. A. de Medeiros, B. F. van Dongen, W. M. P. van der Aalst, & A. J. M. M. Weijters. "Process Mining: Extending the α-algorithm to Mine Short Loops." Technical report: `http://alexandria.tue.nl/repository/books/576199.pdf`

[17] W. deRoever. The Need for Compositional Proof Systems: A Survey. In W. deRoever, H. Langmaack, & A. Pnueli, editors, *Compositionality: The Significant Difference: International Symposium, COMPOS'97* vol. 1536 of *Lecture Notes in Computer Science*, 1-22, 1998.

[18] P. Dietrich, R. Malik, W. M. Wonham, & B. A. Brandin. Implementation Considerations in Supervisory Control. In *Synthesis and Control of Discrete Event Systems*, edited by B. Caillaud, P. Darondeau, L. Lavagno, & X. Xie, p. 185-201. Kluwer Academic Publishers, 2002.

[19] M. Dotoli, M.P. Fanti, & A. M. Mangini. Real time identification of discrete event systems using Petri nets. *Automatica* **44**: 1209-1291, 2008.

[20] V. Dubinin & V. Vyatkin. Towards a Formal Semantic Model of IEC 61499 Function Blocks. *INDIN* '06.

[21] J. Ezpeleta, J. M. Colom, & J. Martinez. A Petri Net Based Deadlock Prevention Policy for Flexible Manufacturing Systems. *IEEE Transactions on Robotics and Automation*, **11**, 173-184, 1995.

[22] M. Fabian & A. Hellgren. PLC-based Implementation of Supervisory Control of Discrete Event Systems. *Proceedings of the 37th IEEE Conference on Decision and Control*, 3305-3310, 1998.

[23] FBDK tool, holobloc.com

[24] L. Ferrarini & C. Veber. Implementation approaches for the execution model of IEC 61499 applications. *INDIN* '04.

[25] G. Frey & T. Hussain. Modeling Techniques for Distributed Control Systems based on the IEC 61499 Standard – Current Approaches and Open Problems. *WODES* '06.

[26] A. Giua. Petri Net State Estimators Based on Event Observation. *Proceedings of the 36th Conference on Decision and Control*, San Diego California, 4086-4091, 1997.

[27] C.N. Hadjicostis & G.C. Verghese. "Monitoring Discrete Event Systems Using Petri Net Embeddings." *Lecture Notes in Computer Science*, **1639**: 188-207, 1999.

[28] F. Harary, R. Z. Norman, & D. Cartwright. Chapeter 10 of *Structural Models: An Introduction to the Theory of Directed Graphs*. Jon Wiley & Sons, Inc.: New York, 1965.

[29] T. Hastie, R. Tibshirani, & J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York, NY: Springer Science+Business Media Inc. 2001.

[30] L.E. Holloway & B.H. Krogh. Fault detection and diagnosis in manufacturing systems: a behavioral model approach. *Proceedings of Rensselaer's Second International Conference on Computer Integrated Manufacturing*, 252-259, 1990.

[31] H. Hu, Z. Li, & A. Wang. Mining of Flexible Manufacturing System Using Work Event Logs and Petri Nets. *Lecture Notes in Computer Science*, 380-387, 2006.

[32] IEC. *IEC 61499-1: Function blocks part 1: Architecture* International Electrotechnical Commission, Tech. Rep., 2005.

[33] P. Inverardi & C. Priami. Automatic Verification of Distributed Systems: The Process Algebra Approach. *Formal Methods in System Design*, 8:7-38, 1996.

[34] M. Khalgui. NCES-based modelling and CTL-based verification of reconfigurable embedded control systems. *Computers in Industry*, **61**: 198-212, 2010.

[35] S. Klein, L. Litz, & J-J. Lesage. Fault Detection of Discrete Event Systems Using an Identification Approach. *Proceedings of the 16th IFAC World Congress*, 2005.

[36] R. Lewis. *Modelling control systems using IEC 61499: Applying function blocks to distributed systems.* Institution of Electrical Engineers: London, UK ('01).

[37] L. Li & C. N. Hadjicostis. Minimum Initial Marking Estimation in Labeled Petri Nets. *2009 American Control Conference*, St. Louis Missouri, 5000-5005, 2009.

[38] Y. Li & W. M. Wonham. Concurrent Vector Discrete-Event Systems. *IEEE Transactions on Automatic Control* **40**: 628-638, 1995.

[39] K. Loeis, M.B. Younis & G. Frey. Application of Symbolic and Bounded Model Checking to the Verification of Logic Control Systems. *10th IEEE Conference on Emerging Technologies and Factory Automation*, 247-250, 2005.

[40] P. Malik. *From Supervisory Control to Nonblocking Controllers for Discrete Event Systems.* PhD thesis, University of Kaiserslautern, Kaiserslautern, Germany 2003.

[41] E. W. Mayr. An Algorithm for the General Petri Net Reachability Problem. *Proceedings of the 13th Annual ACM Symposium*, 1981.

[42] M. E. Meda-Campana & E. Lopez-Mellado. "Incremental synthesis of Petri net models for identification of discrete event systems." *Proceedings of the 41st IEEE Conference on Decision and Control*, 805-810, December 2002.

[43] T. Meftah, H. Gueguen, N. Bouteille, & V. Boutin. Constraint specification of the control logic of automated manufacturing systems. *10th IEEE Conference on Emerging Technologies and Factory Automation*, 599-605, 2005.

[44] J. Moyne, J. Korsakas, & D. Tilbury. Reconfigurable Factory Testbed (RFT): A Distributed Testbed for Reconfigurable Manufacturing Systems. *Proceedings of 2004 Japan-USA Symposium on Flexible Automation*, 1-8, 2004.

[45] J. R. Moyne & D. M. Tilbury. The Emergence of Industrial Control Networks for Manufacturing Control, Diagnostics, and Safety Data. *IEEE Proceedings*, 95:1 29-47, 2007.

[46] T. Murata. "Petri Nets: Properties, Analysis and Applications." *Proceedings of the IEEE*, **77**(4): 541-580, 1989.

[47] D.N. Pandalai & L.E. Holloway. Template languages for fault monitoring of timed discrete event processes. *IEEE Transactions on Automatic Control* **45**, 868-882, 2000.

[48] J. T. Parrott, J. R. Moyne & D. M. Tilbury. Experimental Determination of Network Quality of Service in Ethernet: UDP, OPC, and VPN. *Proceedings of the American Control Conference*, 2006.

[49] N. W. Paton, editor. *Active rules in database systems.* Springer: New York, NY, 1999.

[50] C. Piguet. Logic Synthesis of Race-Free Asynchronous CMOS Circuits. *IEEE Journal of Solid-State Circuits*, 26:371-380, 1991.

[51] P. J. G. Ramadge & W. M. Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE*, 77:81-98, 1989.

[52] J. Richardsson & M. Fabian. Modeling the control of a flexible manufacturing cell for automatic verification and control program generation. *International Journal of Flexible Manufacturing Systems*, 18:191-208, 2006.

[53] K. Rohloff & S. Lafortune. On the Computational Complexity of the Verification of Modular Discrete-Event Systems. *41st IEEE Conference on Decision and Control*, 16-21, 2002.

[54] M. Roth, J. J. Lesage, & L. Litz. “Distributed identification of concurrent discrete event systems for fault detection purposes.” *European Control Conference*, August 2009.

[55] Y. Ru & C.N. Hadjicostis. Fault Diagnosis in Discrete Event Systems Modeled by Partially Observed Petri Nets. *Discrete Event Dynamic Systems*, **19**: 551-575, 2009.

[56] S. Russell & P. Norvig. *Artificial Intelligence: A Modern Approach*. New Jersey: Pearson Education Inc., 2003, Chapter 18.

[57] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, & D.C. Tenketzis. Failure Diagnosis Using Discrete-Event Models. *IEEE Transactions on Control Systems Technology*, **4**: 105-124, 1996.

[58] S. Takai & T. Ushio. Supervisory Control of a Class of Concurrent Discrete Event Systems Under Partial Observation. *Discrete Event Dynamic Systems: Theory and Applications* **15**: 7-32, 2005.

[59] K. Thramboulidis & G. Doukas. IEC61499 Execution Model Semantics. *Innovative Algorithms and Techniques in Automation, IET* ’07.

[60] W. van der Aalst, T. Weijters, & L. Maruster. “Workflow Mining: Discovering Process Models from Event Logs.” *IEEE Transactions on Knowledge and Data Engineering*, **16**: 1128-1142, 2004.

[61] V. Vyatkin. *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*. O3neida and Instrumentation Society of America (2007).

[62] Y. Wang, H. Liao, S. Reveliotis, T. Kelly, S. Mahlke, & S. Lafortune. “Gadara Nets: Modeling and Analyzing Lock Allocation for Deadlock Avoidance in Multithreaded Software.” *Joint 48th IEEE Conference on Decision and Control*, Shanghai, P. R. China, 2009.

[63] J. Webster (ed). Asynchronous Logic Design. *Wiley Encylopedia of Electrical and Electronics Engineering*, John Wiley & Sons, 1999.

[64] T. Yoo & S. Lafortune. Polynomial-Time Verification of Diagnosability of Partially Observed Discrete-Event Systems. *IEEE Transactions on Automatic Control*, **47**: 1491-1495, 2002.

[65] K. Zhou, J. C. Doyle, & K. Glover. *Robust and Optimal Control*. Prentice Hall: Upper Saddle River, New Jersey (1996).